Jeremy Grifski
CSE 5521
11/21/19

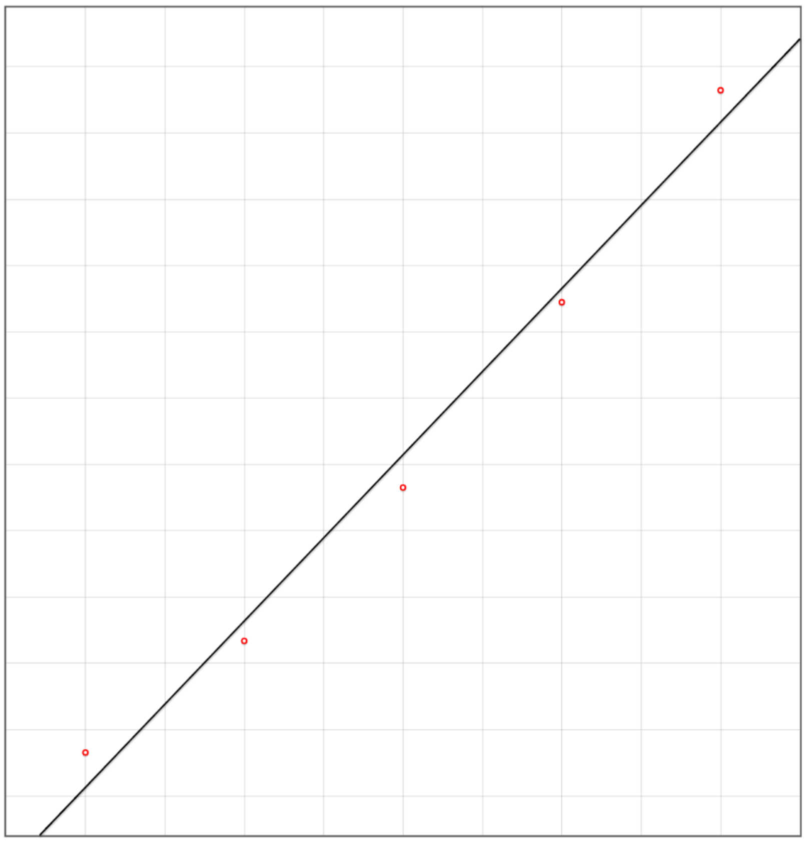
Homework 10

1. Complete the function calc_linLSQ_line(): Use linear least squares to estimate the parameters (a and b) for the following model:

$$f(x, a, b) = a\,x + b$$

Calculate the sum of squared error using your results. Also, create a plot of the model function using the parameter values you found (the "Save As Img" button can be used for this). Include these in your report. (4 pts)

<u>First Order</u>
a = 1.2547000000000004
b = -0.5629500000000007
sse = 0.22131790000000037



2. Similar to problem 1, complete the function calc_linLSQ_poly(): Use linear least squares to estimate the parameters for various polynomials, such as:

$$f(x, a, b, c) = a\ x \wedge 2 + b\ x + c$$
$$f(x, a, b, c, d) = a\ x \wedge 3 + b\ x \wedge 2 + c\ x + d$$
$$f(x, a, b, c, d, e) = a\ x \wedge 4 + b\ x \wedge 3 + c\ x \wedge 2 + d\ x + e$$

Create a plot and calculate SSE as before, for your report. Compare the plots and errors to your results from problem 1. Which of these 4 models (including problem 1) do you think best fits the data? Why? (7 pts)

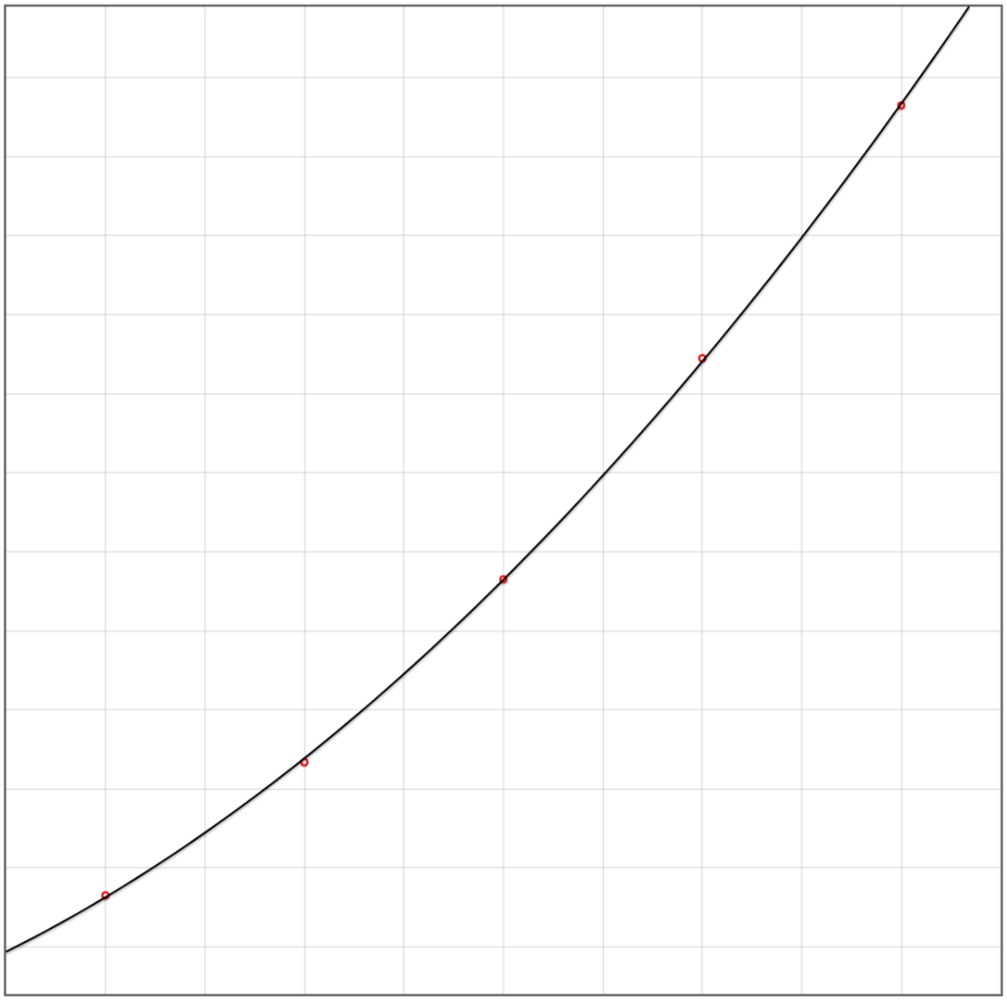
Second Order

a = 0.12535714285714403

b = 0.6279142857142741

c = -0.03018214285714671

sse = 0.0013161142857142784



Third Order

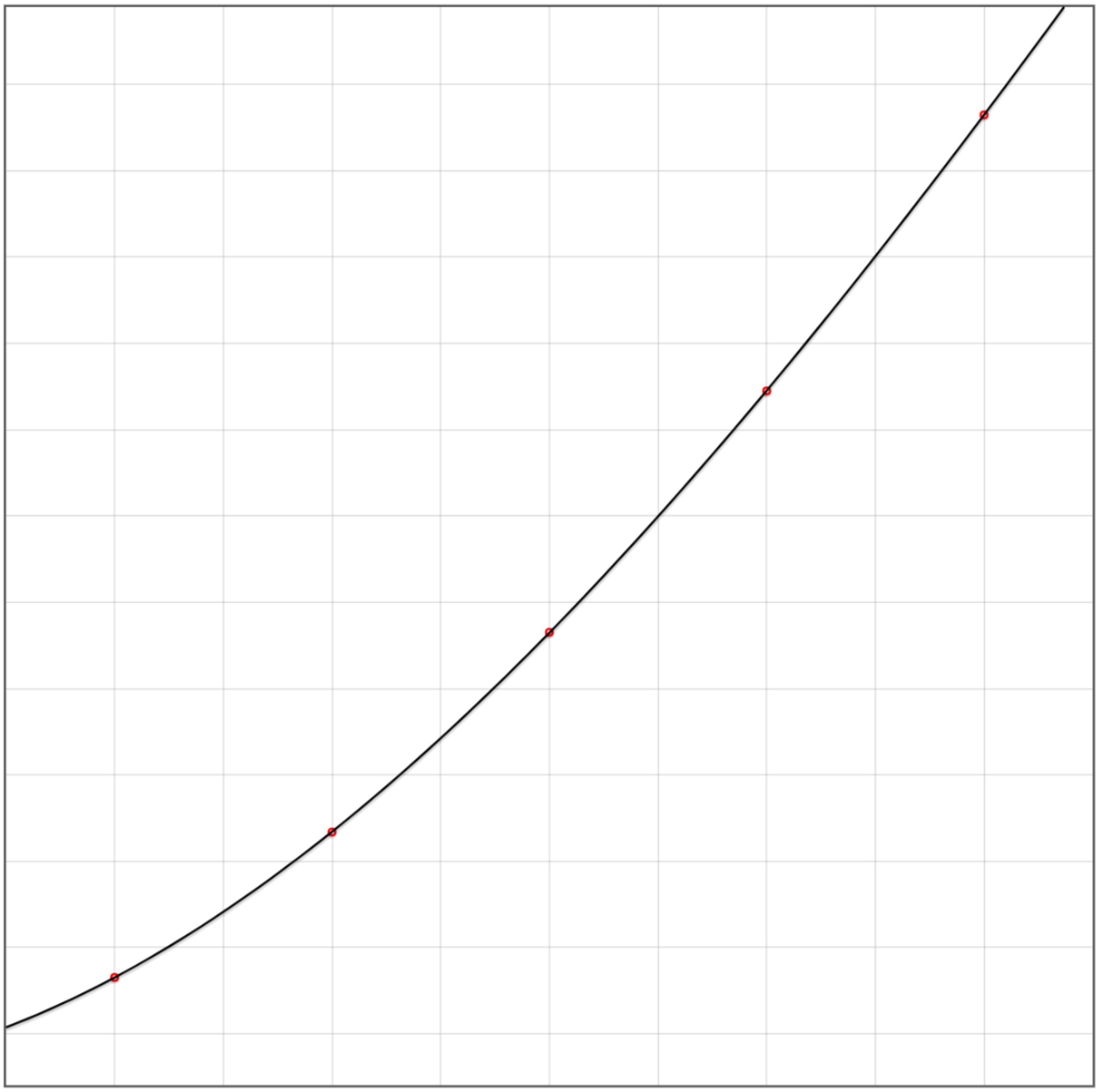
a = -0.00950000000001533

b = 0.19660714285725422

c = 0.4820892857143928

d = 0.03750535714299619

sse = 0.000016514285714287O9



Fourth Order

a = 0.0014166666665633154

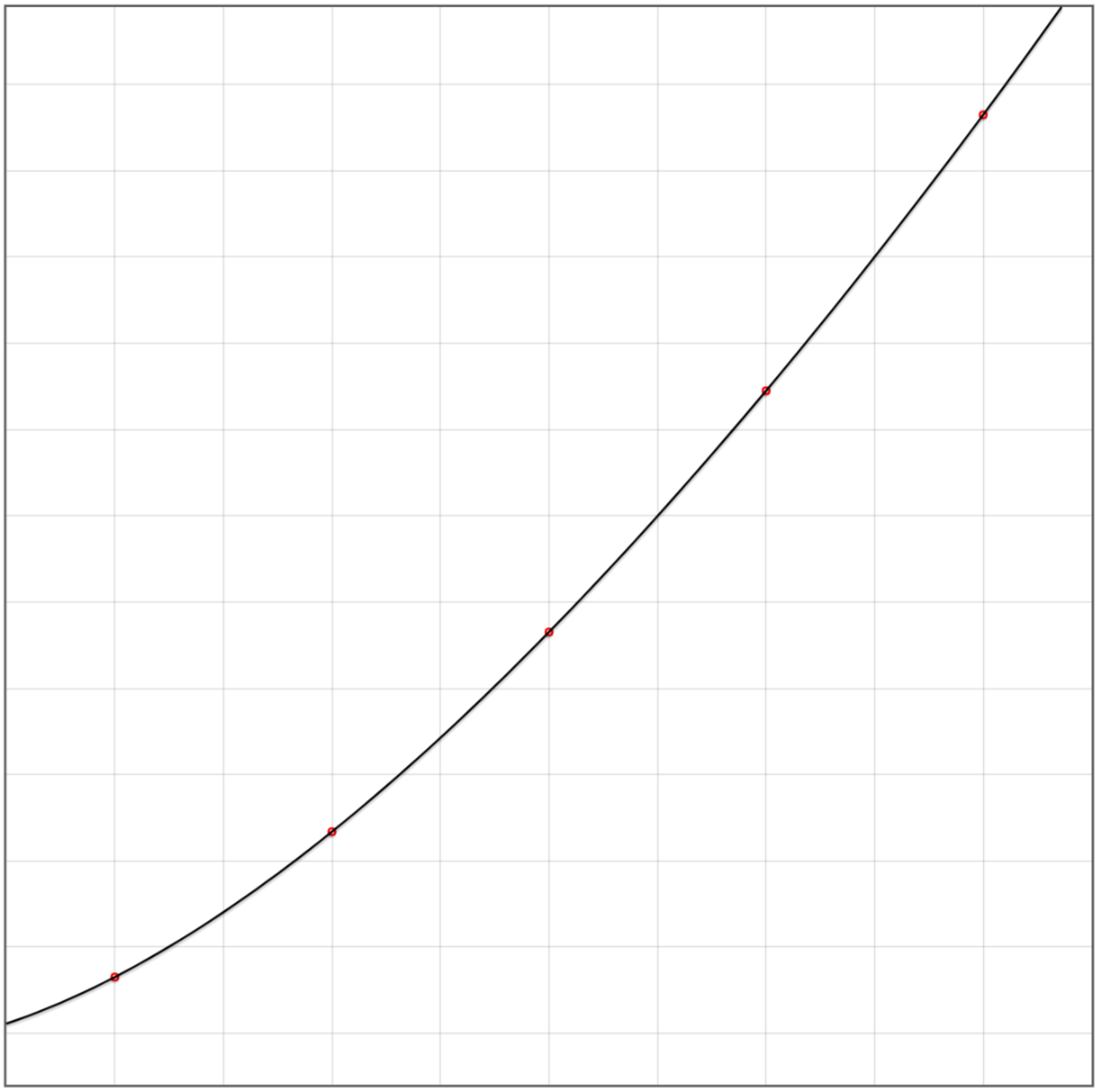
b = -0.023666666664953873

c = 0.24345833333173417

d = 0.42491666667529593

e = 0.056546874997625896

sse = 1.9420303619057346e-20



Of the four plots, the fourth order function has the best fit according to sum of squares error (SSE). Of course, this make sense because higher dimensional functions work like taylor series expansion to approximate a curve. In other words, you get more control over the shape of the curve with each additional dimension.

3. Derive the function for the Jacobian matrix for the following non-linear model:

$$f(x, \mathbf{p}) = a\, x \mathbin{\char`\^} b + c\, x + d$$

   In addition to including this derivation in your report, implement this in the calc_jacobian() function. (2 pts)

   df(x, p) / da = x^b
   df(x, p) / db = a * x^b * log(x)
   df(x, p) / dc = x
   df(x, p) / dd = 1

4. Complete the function calc_nonlinLSQ_gaussnewton(): Use Gauss-Newton non-linear least squares to estimate the parameters for the function from (3).

   Use the following for your initial guess:
   a=0.5, b=2, c=0.5, d=0.5
   Stop after 10 iterations. (Note, these are the default values in the template.)

   As with problems 1-2, create a plot of the resulting model function. Also, calculate the sum of squared error after each iteration. Do you think this model fits the data better than previous ones? Why or why not? (5 pts)

<u>Nonlinear Solution</u>

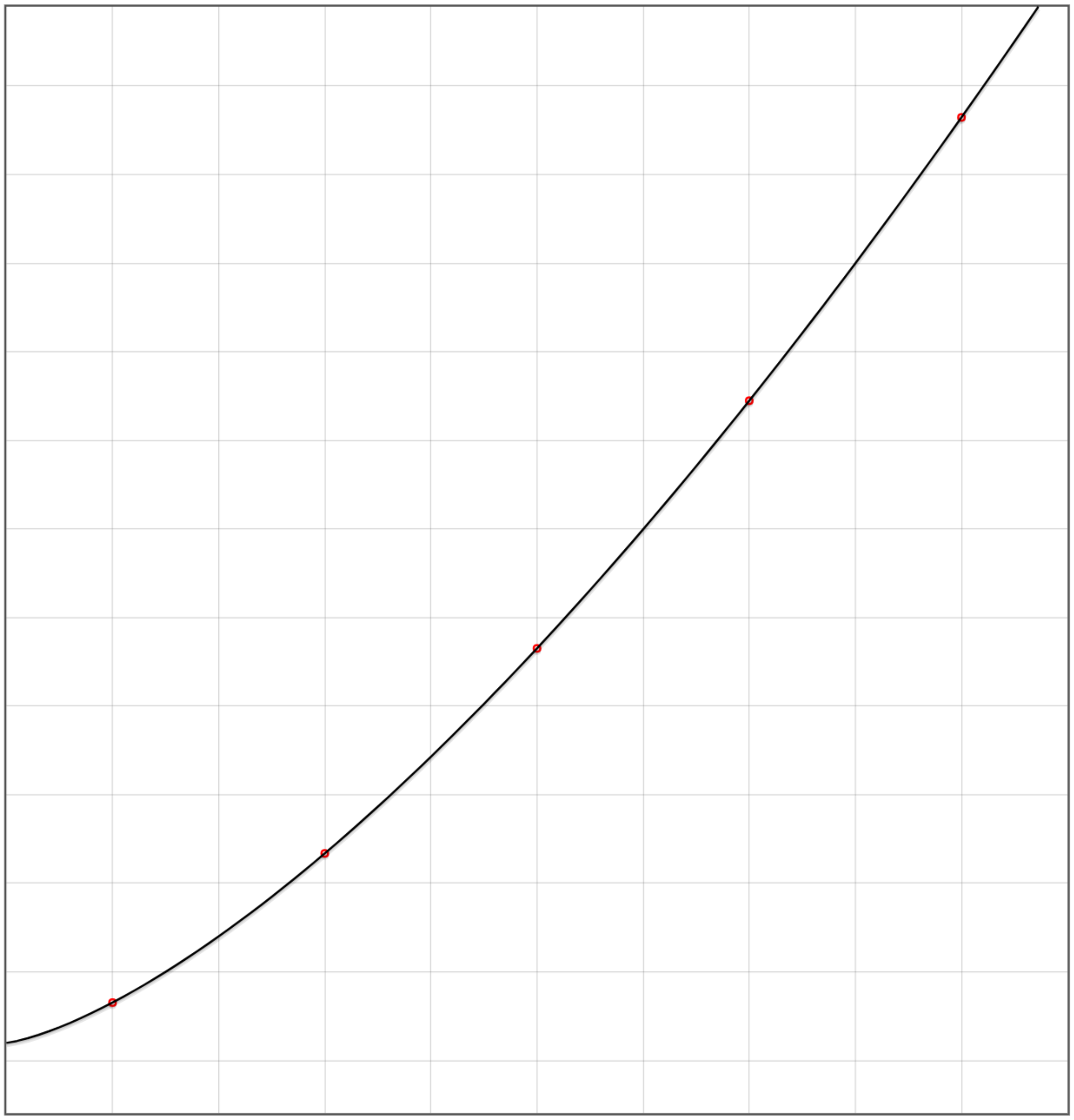
a = 0.4938455104742545

b = 1.5037635080613851

c = 0.10730333930639983

d = 0.09923420477446322

sse= 2.2090783285673085e-7



Considering the final SSE for this curve, I'd have to say that the fourth order polynomial does a better job. That said, the error is so small that I can't really tell a difference between the two curves. However, I do think this curve would be better for mitigating issues related to overfitting. After all, it appears that the SSE converges without driving to zero.

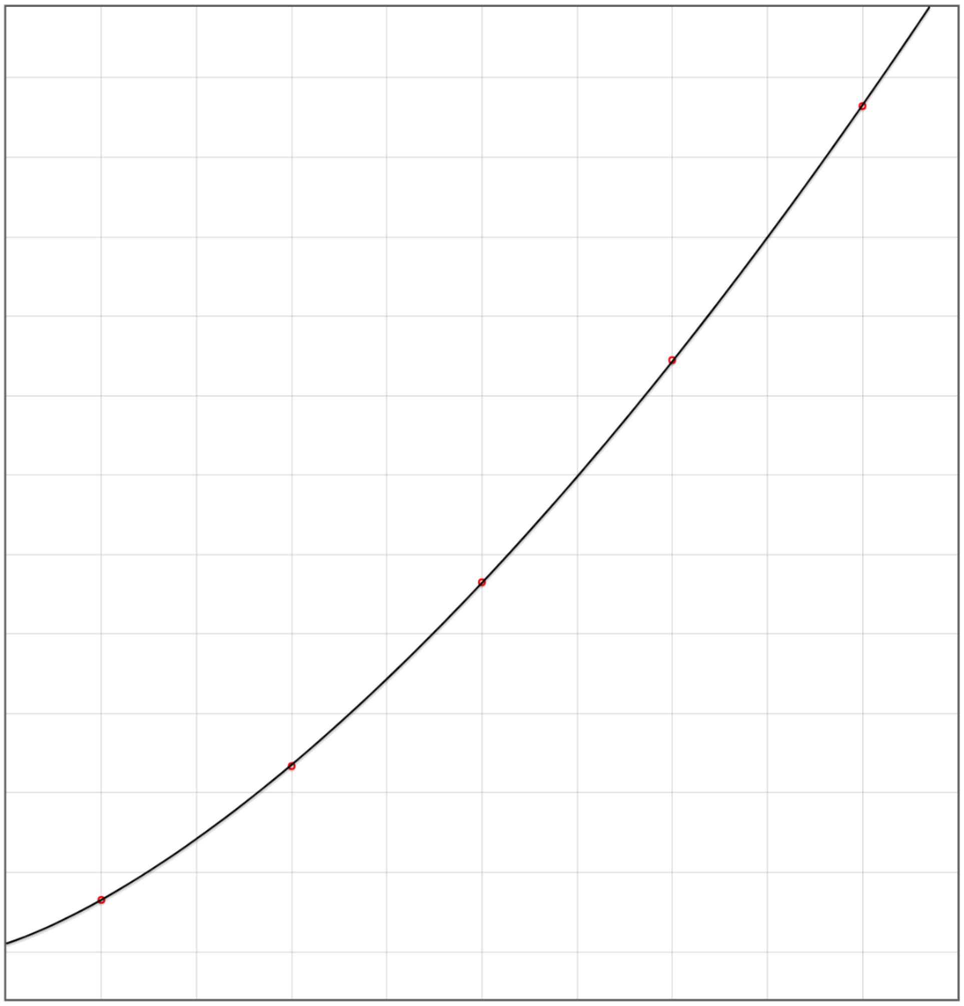
5. Complete the function calc_nonlinLSQ_gradientdescent(): Use Gradient Descent non-linear least squares to estimate the parameters for the function from (3).

   Use the same initial guess as (4). Use a learning rate of 0.001 and stop after 5000 iterations.

   As with problem 4, create a plot and find the sum of squared error after each iteration. How does this algorithm compare to (4)?

   Try different values for learning rate. Can you achieve a better convergence rate (lower error or less iterations for same error)? What are your observations on how the algorithm behaves with different values? (5 pts)

Gradient Descent
a = 0.2869086366665941
b = 1.6822635934297665
c = 0.37200221825057966
d = 0.052247386352797826
sse = 0.00020903103089654364



Honestly, this algorithm isn't great. It takes a ton of iterations to get a decent error with the learning rate we used. In addition, it's possible it settles on some local minimum, so we never get a perfect fit.

If I used a larger step, say .01, then the resulting function is terrible (sse = 0.1575108817063168). Meanwhile, a smaller learning rate, say .0001, gets bad results as well (sse = 0.027784956282841375). However, if I very slightly modify our learning rate, say .00011, then we get slightly better results (sse = 0.000171507172768426).

If I try particularly large learning rates, say > .01, the solution crashes as NANs and Infinities are generated. If I try particularly small learning rates, say < .0001, convergence is extremely slow.

Source Code

```
/**
 * Solves for p given A and b.
 *
 * @param {Array} A the A matrix
 * @param {Array} b the b matrix
 * @returns p
 */
function solveForParams(A, b) {
  let ATranspose = numeric.transpose(A)
  let AProduct = numeric.dot(ATranspose, A)
  let AProductInverse = numeric.inv(AProduct)
  let AProductInverseTranspose = numeric.dot(AProductInverse, ATranspose)
  let p = numeric.dot(AProductInverseTranspose, b)
  return p
}

/**
 * Perform linear least squares for line equation: y=a*x+b
 *
 * @param {Array} data a list of data points
 * @returns parameter array p, where p[0]=b and p[1]=a
 */
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])); // Extra
ct x (dependent) values
  let y = squeeze_to_vector(numeric.getBlock(data, [0, 1], [N - 1, 1])); // Extra
ct 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) {
    A[i][0] = 1;
    A[i][1] = x[i];
    b[i] = [y[i]];
  }

  let p = solveForParams(A, b)

  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 - b[i][0]), 2)
  }
  helper_log_write("SSE=" + sse);

  return p;
}

/**
 * Perform linear least squares for polynomial
 * (example: for quadratic a*x^2+b*x+c, order=2, p[0]=c, p[1]=b, p[2]=a)
 *
 * @param {Array} data list of data points
 * @param {Number} order the order of the polynomial
 * @returns parameter array p, where p[0] is the constant term and p[order] holds
 the highest order coefficient
 */
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])); // Extra
ct x (dependent) values
  let y = squeeze_to_vector(numeric.getBlock(data, [0, 1], [N - 1, 1])); // Extra
ct 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; j++) {
      A[i][j] = Math.pow(x[i], j)
    }

    b[i] = [y[i]];
  }

  let p = solveForParams(A, b)

  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 - b[i][0]), 2)
  }
  helper_log_write("SSE=" + sse);
  return p;
}

/**
 * Calculate jacobian matrix for a*x^b+c*x+d
 *
 * @param {Array} data list of data points
 * @param {Array} p list of parameters, where p[0]=d,...,p[3]=a
 * @returns Jacobian matrix
 */
function calc_jacobian(data, p) {
  let N = numeric.dim(data)[0];
  let x = squeeze_to_vector(numeric.getBlock(data, [0, 0], [N - 1, 0])); // Extra
ct x (dependent) values

  let J = numeric.rep([N, 4], 0);
  for (let i = 0; i < N; ++i) {
    J[i][3] = Math.pow(x[i], p[2]); // df(x, p) / da = x ^ b
    J[i][2] = p[3] * Math.pow(x[i], p[2]) * Math.log(x[i]);  // df(x, p) / db = a
 * x^b * log(x)
    J[i][1] = x[i]; // df(x, p) / dc = x
    J[i][0] = 1; // 1
  }

  return J;
}

/**
 * Perform Gauss-Newton non-linear least squares on polynomial a*x^b+c*x+d
 *
 * @param {Array} data list of data points
 * @param {Array} initial_p list of initial parameters
 * @param {Number} max_iterations number of iterations to perform before stopping
 * @returns final parameter array p, where p[0]=d,...,p[3]=a
 */
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])); //Extrac
t x (dependent) values
  let y = squeeze_to_vector(numeric.getBlock(data, [0, 1], [N - 1, 1])); //Extrac
t 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) {
      sse += Math.pow(dy[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);

    //Step 3: Calculate change in guess
    let dp = solveForParams(J, dy);

    //Step 4: Make new guess
    p = numeric.add(p, dp);
  }
  return p;
}

/**
 * Perform Gradient Descent non-linear least squares on polynomial a*x^b+c*x+d
 *
 * @param {Array} data a list of data points
 * @param {Array} initial_p contains initial guess for parameter values
 * @param {Number} max_iterations number of iterations to perform before stopping
 * @param {Number} learning_rate the learning rate (alpha) value to use
 * @returns parameter array p, where p[0]=d,...,p[3]=a
 */
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])); //Extrac
t x (dependent) values
  let y = squeeze_to_vector(numeric.getBlock(data, [0, 1], [N - 1, 1])); //Extrac
t y (target) values

  let p = initial_p.slice(0);
  let dy = numeric.rep([N], 0);
  for (let iter = 0; iter <= max_iterations; ++iter) {
    // Compute dy
    for (let i = 0; i < N; ++i) {
      dy[i] = y[i] - eval_nonlin_func(x[i], p);
    }

    // Compute sse
    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

    // Step 1: Compute gradient
    let J = calc_jacobian(data, p);
    let grad = numeric.dot(numeric.mul(numeric.transpose(J), -2), dy) // -
2 Jt(p) dy(p)

    // Step 2: Update parameters
    p = numeric.add(p, numeric.mul(grad, -learning_rate))
  }
  return p;
}
```