LAB 7 - LEAST SQUARES II

1. Finding nonlinear models

Recall that during last week's lab we saw how to find the equation of a line or parabola which best fits a given set of data. While these techniques are very useful in practice, often the data we are hoping to study does not lie on or near a line or parabola, but follows a more complicated relationship. Fortunately, the techniques we've studied can still be applied to find more complicated functions that match our data.

Consider the following situation. You are on one end of a long cable, through which a repeating electrical signal is being transmitted. You record the signal by measuring and logging its amplitude every 10 ms. The data you obtain is stored in the file `Lab7data.csv`. Each row in `Lab7data.csv` contains the result of a single measurement, and consists of the time t_j of the observation (in milliseconds) and the amplitude A_j of the signal (in millivolts) at time t_j.

Time (milliseconds)	Amplitude (millivolts)
0.00	0.429086
0.01	0.454173
0.02	0.433519
0.03	0.518289
0.04	0.40911
0.05	0.462929
⋮	⋮
6.27	0.557813
6.28	0.469532

Table 1. A sample of the data contained in `Lab7data.csv`.

- Import the data from `Lab7data.csv` into a NumPy array called `signal_data`. (See the instructions on how to do this in the notebook for this lab.)

Problem 1. (1) Using the array `signal_data`, create a NumPy vector `T` which consists of the data in the *time* column in `signal_data`, and another NumPy vector `Y` which consists of the data in the *amplitude* column of `signal_data`.

Note: `T` and `Y` should both be vectors. You can check that they are the correct shape by running the commands `T.shape` and `Y.shape`. In both cases you should get the result `(629,)`, and the array values should match Table 1.

(2) Using the arrays `T` and `Y` plot the points in `signal_data`.

- Does the data look linear? Does it look parabolic? Can you think of a function which might model the data better than a straight line or parabola?

As mentioned above, the signal we have measured is periodic. The period is 6.29 milliseconds, which means that the signal repeats itself every 6.29 milliseconds. Because of this, linear functions and quadratics will not fit our data well in the long run, since these functions are not periodic.

One nice family of periodic functions is the set of trigonometric functions of the form

$$f(t) = B\sin(mt) \quad \text{and} \quad g(t) = B\cos(mt).$$

Recall that in a function of this form, the value m will determine the period of the sine and cosine wave (in other words, how fast it oscillates), while the value B will determine the amplitude of the wave. Because sine and cosine functions are periodic, we might hope to add a number of sine and cosine waves together, with different periods and amplitudes, to try to find a function which matches our data.

To be more precise, let $f_n(t)$ denote a function of the form

$$f_n(t) = a_0 + a_1\cos(t) + b_1\sin(t) + a_2\cos(2t) + b_2\sin(2t) + \cdots + a_n\cos(nt) + b_n\sin(nt)$$

where each of the coefficients $a_0, a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ is a real-valued scalar. We will try to find functions of this form which match our data.

We will begin by fixing $n = 2$ and try to approximate the data using a function of the form

$$f_2(t) = a_0 + a_1\cos(t) + b_1\sin(t) + a_2\cos(2t) + b_2\sin(2t).$$

Similar to last week, we will use the method of least squares to determine the best choice for the coefficients a_0, a_1, a_2, b_1, and b_2 to make $f_2(t)$ most closely fit the data.

In our data set we have 629 data points $(t_0, A_0), (t_1, A_1), \ldots, (t_{628}, A_{628})$ from our observations. We are looking for values of a_0, a_1, a_2, b_1, and b_2 so that for each time t_j we have

$$A_j = a_0 + a_1\cos(t_j) + b_1\sin(t_j) + a_2\cos(2t_j) + b_2\sin(2t_j).$$

Each of these 629 equations can be expressed in a single vector equation

$$\begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ \vdots \\ A_{628} \end{bmatrix} = \begin{bmatrix} a_0 + a_1\cos(t_0) + b_1\sin(t_0) + a_2\cos(2t_0) + b_2\sin(2t_0) \\ a_0 + a_1\cos(t_1) + b_1\sin(t_1) + a_2\cos(2t_1) + b_2\sin(2t_1) \\ a_0 + a_1\cos(t_2) + b_1\sin(t_2) + a_2\cos(2t_2) + b_2\sin(2t_2) \\ \vdots \\ a_0 + a_1\cos(t_{628}) + b_1\sin(t_{628}) + a_2\cos(2t_{628}) + b_2\sin(2t_{628}) \end{bmatrix}.$$

As we did last week, we can rewrite this vector equation as a matrix equation:

$$\begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ \vdots \\ A_{628} \end{bmatrix} = \begin{bmatrix} 1 & \cos(t_0) & \sin(t_0) & \cos(2t_0) & \sin(2t_0) \\ 1 & \cos(t_1) & \sin(t_1) & \cos(2t_1) & \sin(2t_1) \\ 1 & \cos(t_2) & \sin(t_2) & \cos(2t_2) & \sin(2t_2) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \cos(t_{628}) & \sin(t_{628}) & \cos(2t_{628}) & \sin(2t_{628}) \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ b_1 \\ a_2 \\ b_2 \end{bmatrix}.$$

If we define the vectors Y and β by

$$Y = \begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ \vdots \\ A_{628} \end{bmatrix}, \qquad \beta = \begin{bmatrix} a_0 \\ a_1 \\ b_1 \\ a_2 \\ b_2 \end{bmatrix},$$

and define the 629×5 matrix X by

$$X = \begin{bmatrix} 1 & \cos(t_0) & \sin(t_0) & \cos(2t_0) & \sin(2t_0) \\ 1 & \cos(t_1) & \sin(t_1) & \cos(2t_1) & \sin(2t_1) \\ 1 & \cos(t_2) & \sin(t_2) & \cos(2t_2) & \sin(2t_2) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \cos(t_{628}) & \sin(t_{628}) & \cos(2t_{628}) & \sin(2t_{628}) \end{bmatrix}, \tag{1}$$

then we can write our equation more compactly as $X\beta = Y$. Finding a vector β which satisfies this equation would give us the coefficients for a function $f_2(t)$ whose graph exactly passes through each of the data points (t_j, A_j).

Problem 2. We will use the following steps to define a function `row_func(t,n)` which takes two values, `t` and `n` as input, and returns the list

```
[1,cos(t),sin(t),cos(2*t),sin(2*t), ... ,cos(n*t),sin(n*t)]
```

which will give us the rows of the matrix we want.

(1) In your function first create the row as a list without the `1` at the beginning. In other words, create the list `L=[cos(t),sin(t), ... ,cos(n*t),sin(n*t)]` first.

To create the list `L` we can use double list comprehension. Remember that if `g` and `h` are functions, then the command

```
[f(k) for k in range(1,4) for f in [g,h]]
```

will create the list

```
[g(1),h(1),g(2),h(2),g(3),h(3)]
```

Modify this code and include it in your function to create the list `L`. (Remember, to call cos and sin in NumPy we use `np.cos` and `np.sin` respectively.)

(2) After creating the list `L` in our function, we can add `1` to the beginning by using the command `L.insert(0,1)`. This inserts a `1` in the `0` position of `L`.

For example, when you call `row_func(2,5)` your function should return the list

```
[1,
 -0.4161468365471424,
 0.9092974268256817,
 -0.6536436208636119,
 -0.7568024953079282,
 0.960170286650366,
 -0.27941549819892586,
 -0.14550003380861354,
 0.9893582466233818,
 -0.8390715290764524,
 -0.5440211108893698]
```

Problem 3. Using the function `row_func(t,n)`, create a function `design_matrix(n)` which takes as input an integer `n`, and returns the NumPy array whose jth row is

```
[1,np.cos(t_j),np.sin(t_j), ... ,np.cos(n*t_j),np.sin(n*t_j)]
```

where `t_j` is the jth time value in `T`. In other words, `design_matrix(n)` should produce a matrix whose jth row is `row_func(t_j,n)`.

Hint: Since we already have a function to create the individual rows of the design matrix, we can create the matrix by using yet another list comprehension with the function `row_func(t,n)` (remember, a matrix can be thought of as a list of lists). Once we have created the matrix, don't forget to convert it to a NumPy array before passing it to the `return` statement.

The output of `design_matrix(n)` will be too large to check by hand (it will be a matrix with `629` rows, and `2n+1` columns). You can check that the shape is correct though. For example, `design_matrix(10).shape` should return a value of

```
(629, 21)
```

We can also check some of the rows give correct values. For example, we can check that the 101th row of `design_matrix(4)` is correct, by entering `design_matrix(4)[100,:]`. It should return the row

```
array([ 1.        ,  0.54030231, 0.84147098, -0.41614684, 0.90929743,
       -0.9899925 , 0.14112001, -0.65364362, -0.7568025 ])
```

Problem 4. When `n=2` the output of `design_matrix(2)` should be the matrix X from equation (1). Save the output of the function `design_matrix(2)` as the variable `X2`.

As no exact solution to $X\beta = Y$ exists, there are no coefficients which will make our function $f_2(t)$ perfectly match the data. Since we can't find an exact solution to $X\beta = Y$, we can instead look for a *closest approximate* solution to $X\beta = Y$, and thus find a function $f_2(t)$ which *closely approximates* our data. A best approximation to a solution of $X\beta = Y$ is again found by computing the normal equation

$$X^T X\beta = X^T Y$$

and finding a solution. Note that if $X^T X$ is invertible then this solution will be unique, and is given by $\widehat{\beta} = (X^T X)^{-1} X^T Y$ (recall that the normal equations are always guaranteed to have a solution, but it will not always be unique).

Still considering the case when `n=2`, by plugging the values for $A_0, A_1, \ldots, A_{628}$ into Y, and the values for $t_0, t_1, \ldots, t_{628}$ into X, we obtain

$$X^T X = \begin{bmatrix} 629.00 & 0.68147 & -0.0010854 & 0.68147 & -0.0021707 \\ 0.68147 & 314.84 & -0.0010854 & 0.68147 & -0.0021707 \\ -0.0010854 & -0.0010854 & 314.16 & -0.0010854 & -2.6262 \times 10^{-6} \\ 0.68147 & 0.68147 & -0.0010854 & 314.84 & -0.0021708 \\ -0.0021707 & -0.0021707 & -2.6262 \times 10^{-6} & -0.0021708 & 314.16 \end{bmatrix}$$

and

$$X^TY = \begin{bmatrix} 286.95 \\ 57.659 \\ 28.4805 \\ -4.48684 \\ -12.9247 \end{bmatrix}.$$

Thus the solution $\widehat{\beta}$ to the normal equation $X^TX\beta = X^TY$ is

$$\widehat{\beta} = (X^TX)^{-1}X^TY = \begin{bmatrix} a_0 \\ a_1 \\ b_1 \\ a_2 \\ b_2 \end{bmatrix} = \begin{bmatrix} 0.456019 \\ 0.182184 \\ 0.0906583 \\ -0.0156325 \\ -0.0411362 \end{bmatrix}. \tag{2}$$

Problem 5. (1) Using the matrix `X2` and the vector `Y`, compute the coefficient matrix of the normal equations X^TX and save it as the variable `normal_coef2`. Then compute the right-hand side of the normal equations X^TY and save it as the variable `normal_vector2`.

(2) Solve the normal equations $X^TX\beta = X^TY$. Save your answer as the variable `beta2` and compare your answer to the vector in equation (2). As in Lab 6 and elsewhere in this lab, use the function `np.linalg.solve` with the matrix `normal_coef2` and vector `normal_vect2` to solve for `beta2`.

Hint: You can check your answers by computing $X^TX\widehat{\beta} - X^TY$ in a practice notebook. You should get an array whose values are very close to zero (they won't be exactly zero, due to round-off error).

The entries from the vector $\widehat{\beta}$ give us the coefficients for our function

$$f_2(t) = 0.456 + 0.182\cos(t) + 0.0907\sin(t) - 0.0156\cos(2t) - 0.0411\sin(2t).$$

To define the function $f_2(t)$ in our notebook, we can take the values of $\widehat{\beta}$, and manually type them in as the coefficients of the sin and cos functions. There is an easier way to do this, however, which will be much more convenient when we work with more complicated functions. Note that we can write $f_2(t)$ as a dot product

$$f_2(t) = \begin{bmatrix} 0.456019 \\ 0.182184 \\ 0.0906583 \\ -0.0156325 \\ -0.0411362 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \cos(t) \\ \sin(t) \\ \cos(2t) \\ \sin(2t) \end{bmatrix}$$

Notice that the first vector is saved as the variable `beta2`, and the second vector is the output of the function `row_func(t,2)`.

Problem 6. Define the function $f_2(t)$ in your code as `f2(t)`. In other words, `f2(t)` should return the same value as $f_2(t)$.

Hint: For any `t`, your function should take the vectors `beta2` and `row_func(t,2)` and return their dot product. Notice that you will need to convert the list `row_func(t,2)` to a NumPy array before taking the dot product with `beta2`.

For example, `f2(0.75)` should return

```
0.6089782426454243
```

If we graph the function $f_2(t)$ and compare it to our data (see Figure 1) we see that although the curve loosely follows the shape of the data, there are many features of the data, including several large spikes, that it doesn't seem to capture. Our approximation seems less than ideal.

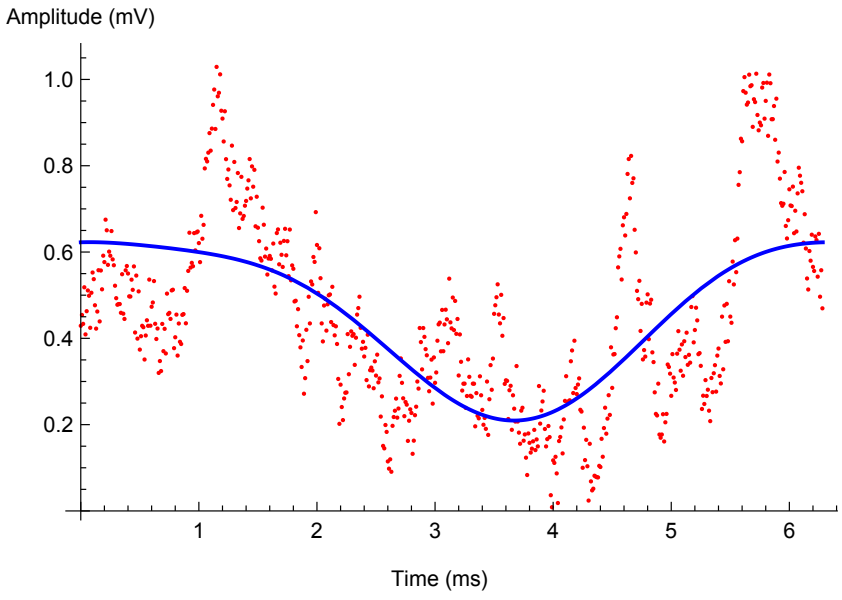

FIGURE 1. The graph of $f_2(t)$ with the data points.

Problem 7. Create the plot in Figure 1 in your notebook, by executing

```
vf2=np.vectorize(f2)      # This creates a new function, vf2, which is
                          # the same as f2 except for the fact that is
                          # can take a NumPy array as input, instead of
                          # individual values.

plt.plot(T,Y,'r.')        # Creating the plot of the data points in Y
                          # using small red dots.

plt.plot(T,vf2(T),'b-')   # Creating the plot of the function f2 with a
                          # blue line.  Here we use the function vf2
                          # instead of f2, since we are plugging in a
                          # NumPy array T.

plt.show()                # Display the plots.
```

We can measure the error of our approximation by computing the value of the *mean squared error*, which is defined in our situation to be

$$MSE = \tfrac{1}{629}\|X\widehat{\beta} - Y\|^2.$$

Indeed, recall that $\widehat{\beta}$ is our approximation to a solution of $X\beta = Y$. Therefore, the closer $X\widehat{\beta}$ is to Y, and hence the smaller $\|X\widehat{\beta} - Y\|$ is, the better the approximation will be. Dividing by 629 (the number of data points we have) gives us a measure of roughly how far on average we may expect each individual data point to stray from the graph of $f_2(t)$.

Problem 8. Compute the mean squared error $\frac{1}{629}\|X\widehat{\beta} - Y\|^2$ of our least squares solution. Save the value of the mean squared error as a variable `MSE2`.

Hint: Given a NumPy vector `a`, we can compute $\|\mathtt{a}\|$ using the command `np.linalg.norm(a)`.

We should note that we can use our approximating function $f_2(t)$ to estimate the amplitude of the signal at times we did not perform measurements. For example, by plugging in $t = 0.105$ we obtain the estimate

$$f_2(0.105) = 0.6228369323878735$$

for the value of the amplitude of the signal at time $t = 0.105$.

2. Improving our approximations

One way we might try to improve our approximation is to use more terms when defining our approximating function

$$f_n(t) = a_0 + a_1\cos(t) + b_1\sin(t) + a_2\cos(2t) + b_2\sin(2t) + \cdots + a_n\cos(nt) + b_n\sin(nt)$$

In our initial attempt, we took $n = 2$ and only added terms up to $\cos(2t)$ and $\sin(2t)$. If we include more terms, say up to $\sin(10t)$, or even $\sin(100t)$, will it improve our approximation?

We will now perform the same steps, but with larger values of n. Perform the following calculations in your practice notebook. If you include the following calculations in the notebook you submit for grading then the website may possibly time-out.

- Set up the matrix X and vector Y as above, this time using $n = 10$ (your vector Y will be the same as before, but X will now have 21 columns). Save X as the NumPy array `X10`.
- Find the unique solution $\widehat{\beta} = (X^TX)^{-1}X^TY$ to the corresponding normal equations, and save its value in the NumPy vector `beta10`. Use `np.linalg.solve` to solve for `beta10`.
- Compute the mean squared error and save it as the variable `MSE10`. Is the mean squared error smaller than our earlier approximation?
- Use your least squares solution `beta10` to define the approximating function `f10`, and graph it along with the data. Does the approximation look better than our initial approximation?
- Use the function `f10` to predict the amplitude of the signal at time `t=0.105`. Save this value as the variable `pred10`.

Problem 9. In the lab notebook you will submit for grading, do the following:

(1) Copy and paste the value you obtained in your practice notebook for `MSE10`, and save it as a variable `MSE10` in your lab notebook.

(2) Copy and paste the value you obtained in your practice notebook for `pred10`, and save it as a variable `pred10` in your lab notebook.

It is important that you copy and paste the actual decimal value of these variables, and not the formulas you used to compute them.

Problem 10. Repeat the steps in Problem 9, this time using $n = 100$. Does the mean squared error improve from your approximation in Problem 9?

You should again perform all of the actual computations in your practice notebook. The only variables you should have saved in your lab notebook should be `MSE100` and `pred100`, which should hold the decimal values you paste in from your practice notebook.

3. Model accuracy vs. computation time, and overfitting

From our experiments above it seems apparent that by using more and more terms in our approximating function $f_n(t)$, we obtain functions that fit the data better and better. Why not take n to be very large then, say $n = 1000$ or even $n = 10000$, so that we obtain extremely accurate approximations?

For one, the difficulty of performing calculations grows as we include more and more terms. If $n = 10000$, our matrix X would have 20001 columns, and finding our least squares solution would require solving a system of 20001 equations in 20001 variables. Even after finding the least squares solution, plugging values into $f_{10000}(t)$ and evaluating the function becomes much more time intensive. Indeed, every time we want to compute $f_{10000}(t)$ for a given t-value we must evaluate 20000 different trig functions to do so. While these computations can certainly be accomplished using modern computers, they take longer for larger n values. Thus the trade-off between the accuracy of the approximation and the time it takes to find and evaluate it must be carefully considered.

Perhaps more importantly, sometimes we want to make sure our model *isn't too accurate*! Indeed, any real-world data sets contain some amount of error, or *noise*. This noise is due to the fact that we can never measure anything in the real world with 100% accuracy.

As we take larger values of n and include more terms in our function $f_n(t)$, we make $f_n(t)$ more complex. The more complex $f_n(t)$ is, the more it will be able to closely approximate the values in our data. When n gets too large, however, we run the risk of $f_n(t)$ beginning to approximate the noise in our data, instead of just important features in the data. When this happens the function becomes less effective at estimating the amplitude of the signal away from our data points, even though the mean squared error may become very small. This phenomenon is known as *overfitting*, and is a very real concern whenever we try to model real world data. One method to avoid overfitting in our case would be to keep n relatively small, thereby limiting the complexity of the approximating function $f_n(t)$ and reducing its ability to model noise in the data. There are a number of other techniques that are commonly used to reduce overfitting, which are beyond the scope of this lab.

Problem 11. In your practice notebook, repeat the steps from problems 9 and 10 using $n = 1000$. Do your calculations seem to take longer? Do you run into other issues with your calculations? Does the value of `pred1000` even make sense if you interpret it as the amplitude of an electrical signal?

You should again perform all of the actual computations in your practice notebook. The only variables you should have saved in your lab notebook should be `MSE1000` and `pred1000`, which should hold the decimal values you paste in from your practice notebook.

Hint: Instead of plotting `plt.plot(T,vf1000(T),'b-')`, try replacing it with

```
T_fine=np.linspace(0,6.28,3000)
plt.plot(T_fine,vf1000(T_fine),'b-')
```

This will divide the interval `[0,6.28]` into 3000 subintervals, which it will use to graph the function `vf1000(t)`. This may give you a better idea of what the function `vf1000(t)` looks like.