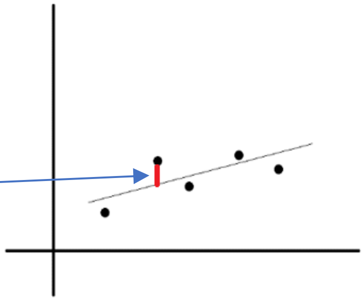
Due 11/15/23

CSCI 58000
Program 4

Introduction – read carefully

Consider a set P of n points $p_1, p_2, \ldots, p_n$ plotted on a two-dimensional plane. Point p_i has coordinates (x_i, y_i), and we will assume that P is ordered by increasing x-values. An important problem in data analysis is "curve-fitting", that is, finding a function that, when plotted, seems to follow the pattern of the points. This problem deals with segmented linear curve-fitting, that is, trying to closely fit the set P with one or more straight line segments.

Here a single straight line seems a pretty good fit for the given set of points, where the straight line has the equation y = ax + b for some constants a and b. *Least squares curve-fitting* involves measuring the differences in the y-values for a given x-value x_k between the point (x_k, y_k) and the line. Because we don't care whether the point falls above or below the line, we square that difference so that the result for a given point is never negative. Summing over n points, the error is therefore



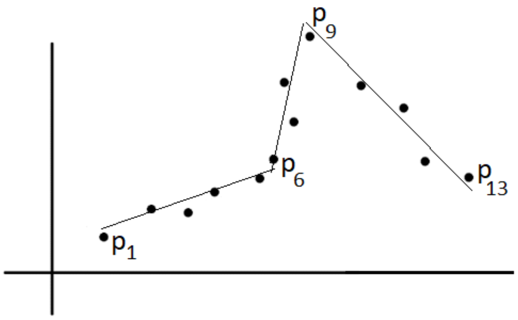
$$\text{err}(1, n) = \sum_{k=1}^{n} (y_k - (ax_k + b))^2 \qquad (1)$$

In this formula, the point coordinates are known, so the values of *a* and *b*, which determine the straight line, are the unknowns. This error expression is a function of two variables, and we want to find the *a* and *b* values that minimize the error. These values are (see the last page):

$$a = \frac{n\sum_k x_k y_k - \sum_k x_k \sum_k y_k}{n\sum_k x_k^2 - (\sum_k x_k)^2} \qquad (2)$$

$$b = \frac{\sum_k y_k - a\sum_k x_k}{n} \qquad (3)$$

where the summations again range from 1 to n. This solves the problem for curve-fitting with a single straight line.



But – what if the points look something like this? Here it seems that a single straight line would have a huge error, but perhaps 3 line segments would work pretty well, a segment from p_1 to p_6, one from p_6 to p_9, and one from p_9 to p_{13}. However, we are only guessing at partitioning P into these 3 segments because we're looking at the graph, but we would not ordinarily have such a visual cue. So we can't predetermine the optimum number of segments in the partition. Of course, for any set P, an extreme case would be to fit a line through each pair of adjacent points; these lines would fit the points perfectly (minimum error in each case) but that's probably way too many segments. So we need to figure out how to partition P into a relatively small number of segments whose corresponding lines have minimal error value.

For a given partition, its *penalty* involves two things:

(i) The number of segments in the partition times a fixed given cost factor $C > 0$.
(ii) The sum of the error values of the optimal line through each segment.

We want to find a partition with a minimum penalty, but the parts contributing to a penalty are somewhat contradictory; increase the number of segments and part (i) of the penalty increases while part (ii) decreases. The constant C, which is added to the error of each segment in the partition, is an input value and its value can be set higher or lower to help control the penalty contribution of additional lines.

Any given partition can be described by the "breakpoints" that occur, as in p_1, p_6, p_9, and $p_{13}$for the example given earlier. And for any given partition (and value of C), one can compute the penalty in linear time. Therefore a completely brute-force solution to finding an optimal partition of P is to try all possible partitions and find the one with minimum penalty. The partition is determined by the breakpoints, but p_1 and p_n are always breakpoints, so we want all possible subsequences of p_2-p_{n-1}, of which there are 2^{n-2} - exponential solution, not good!

We can use dynamic programming to solve this problem (called the *Segmented Least Squares* problem). Some notation:

i) If a segment of P is $p_i, p_2, . . . p_j$, then **err(i, j)** is the minimum error of the straight line through this segment; this is given by equations (1), (2), (3) where the summation limits are *i* and *j* rather than 1 and *n*.

ii) **OPT(j)** denotes the optimum solution for the points $p_1, p_2, \ldots p_j$. We ultimately want to find OPT(*n*), which will be a recursive process. Define OPT(0) to have the value 0 (there is no point p_0) and define OPT(1) to have the value 0 (there is no segment from p_1 to p_1). The trivial OPT(0) and OPT(1) values are also needed to make the recursion work.

A recursive formula for the optimal solution OPT(*j*), where $j > 1$, is

$$\text{OPT}(j) = \text{err}(i, j) + C + \text{OPT}(i) \qquad (4)$$

Here we assume that $p_i, p_{i+1}, \ldots, p_j$ is a single segment with error err(i, j) and cost factor C.

We have to try formula (4) for all possible values of *i*, that is, all possible "breakpoints" *i*, $1 \le i < j$ to create the last segment of $p_1, . . . p_j$. (We don't need $i = j$ because there is no segment from p_j to p_j.) But even for a specific value of *i*, we still need to find the optimum solution for the smaller segment of P from 1 to *i*. This suggests using a linear table structure, that is, an array M[0, 1 . . . *n*], where M[*j*] = OPT(*j*) for $j = 0, 1, . . . n$, and walking through the table from left to right (similar to what was done in the rod-cutting problem), ultimately finding M[*n*].

The general picture looks something like this: $p_1\overbrace{\ldots\ldots p_i}^{M[i]}\overbrace{\ldots p_j}^{err(i,j)}\ldots\ldots\ldots\ldots p_n$.

Therefore pseudocode for this part of the algorithm is

```
M[0] = M[1] = 0
for (j =2..n)
      M[j] = $min_{1 \le i < j}$ (err(i, j) + C + M[i])
```

You have to keep track of which value of i gives the minimum because this represents a breakpoint.

But before you do the computation given here, you must compute (and store for future reference) err(i, j) for all possible pairs i ≤ j. To do this, just fill in the upper triangular part of an n x n matrix, including the main diagonal, thus there are $O(n^2)$ such pairs, and for a given (i, j), err(i,j) can be computed in O(n) time, so altogether computing the *err* matrix requires $O(n^3)$ work. Filling in the OPT M table (the above pseudocode) requires a for loop of order n, and each value of j requires finding the minimum for the i values between 1 and j-1, which is O(n), so in total the pseudocode part is $O(n^2)$. Therefore the total work for this algorithm is $O(n^3)$, obviously an improvement over the brute-force approach.

Segmented Least Squares Example – read carefully

P consists of 6 points, as follows:

1.4	2.3
1.5	2.6
1.7	3.1
2.2	3.5
2.7	2.8
2.84	2.2

First we need to compute err(i,j) for all possible pairs i ≤ j. Here is a 6 x 6 array where the work has already been done for you. Because i ≤ j, the lower triangular part of the matrix is not used. If i = j, e(i, j) = 0 because there is no segment from i to j, so the main diagonal has all 0 values. If i = j – 1, that is, j = i + 1, then i and j are adjacent points that exactly determine a straight line curve fit so the error is 0, therefore the next diagonal also has all 0 values.

err	1	2	3	4	5	6
1	0	0	0.0007	0.0801	0.6567	1.2143
2	x	0	0	0.0371	0.4405	0.8624
3	x	x	0	0	0.2017	0.4570
4	x	x	x	0	0	0.0601
5	x	x	x	x	0	0
6	x	x	x	x	x	0

Here I strongly suggest that you use Excel to compute the err(1,4) value just so you understand how the calculations work.

And here is the completed OPT table, which also shows the corresponding index value (the minimum i that produced that OPT value). The segment penalty value is C = 1. Again, I strongly suggest you compute by hand, let's say, OPT(4) to understand how the calculations work.

j	0	1	2	3	4	5	6
OPT(j)	0	0	1	1.0007	1.0801	1.6567	2.1402
index	x	x	1	1	1	1	4

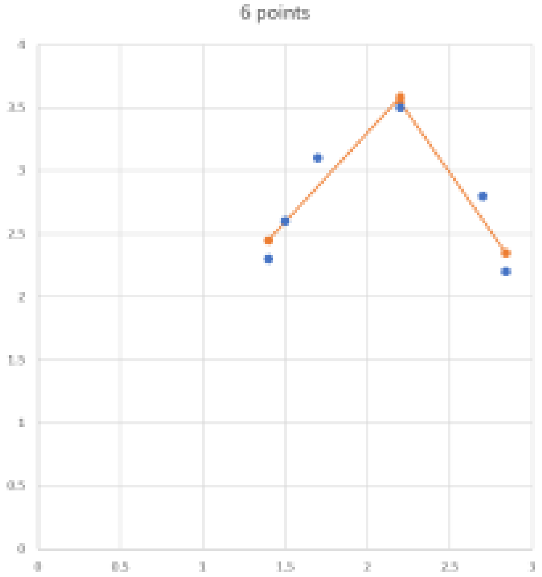
So, according to the index values (reading them from back to front), the segments in the optimal solution are p4-p6 and p1-p4. And the corresponding lines for those segments have the following equations:

line for segment 4 through 6 is y = -1.86396x + 7.64234
line for segment 1 through 4 is y = 1.42105x + 0.459211

The actual program output is highlighted here

```
j  OPT(j)  breakpoint i
0    0    -858993460
1    0    -858993460
2    1    1
3    1.00071     1
4    1.08013     1
5    1.65675     1
6    2.14021     4
 line for segment 4 through 6 is y = -1.86396x + 7.64234
 line for segment 1 through 4 is y = 1.42105x + 0.459211
```

Finally, here is an Excel graph of the points and the resulting line segments



(50 pts) Program 4 Assignment

Implement the Segmented Least Squares algorithm and submit your code file(s) on Canvas. Use C = 1 as the segment penalty value. Your program will be run on a new data file called *points.txt* (same format as the example, one point per line); you can assume a maximum data file size of 16 (or you could use a vector). Meanwhile you have enough information to use the Example data file as a test case. Your program output should look like the highlighted lines on the previous page.

Note: If you find that you cannot complete this program, you can write out just your completed *err* array for maximum 20 pts credit.

Derivation of Equations (2) and (3)

The equation

$$\text{err}(1, n) = \sum_{k=1}^{n}(y_k - (ax_k + b))^2 \qquad \text{(i)}$$

represents the error of curve-fitting a sequence of n points using a straight line of the form y = ax + b. The point coordinates are known, so this is a function of two variables *a* and *b*.

In a function of one variable, something like $f(x) = 3x^2 + 2x - 7$, you would find the x coordinate of a maximum or minimum point by differentiating with respect to x and setting the derivative (the slope of the tangent line) to 0. Because we want to minimize err(1,n), we take partial derivatives of the err function (i) with respect to the two variables *a* and *b* and set the resulting expressions equal to 0.

$$\frac{\partial err}{\partial a} = \sum_{k=1}^{n} 2\,(y_k - (ax_k + b))(-x_k) = -2\sum_{k=1}^{n}(x_k y_k - ax_k^2 - \; bx_k) \;\equiv\; 0$$

so that

$$\sum_{k=1}^{n}(x_k\, y_k - ax_k^2 - bx_k) = 0$$

or

$$a\sum_{k=1}^{n} x_k^2 + b\sum_{k=1}^{n} x_k \;=\; \sum_{k=1}^{n} x_k y_k \qquad \text{(ii)}$$

$$\frac{\partial err}{\partial b} = \sum_{k=1}^{n} 2(y_k - (ax_k + b))(-1) = \; -2\sum_{k=1}^{n}(y_k - ax_k - b) \;\equiv 0$$

so that

$$\sum_{k=1}^{n}(y_k - ax_k - \text{b}) = 0$$

or

$$a\sum_{k=1}^{n} x_k + nb = \; \sum_{k=1}^{n} y_k \qquad \text{(iii)}$$

Now equations (ii) and (iii) form a system of linear equations with unknowns *a* and *b*. To solve this system of equations, we solve equation (iii) for *b*:

$$b = \frac{\sum_{k=1}^{n} y_k - a\sum_{k=1}^{n} x_k}{n}$$

This agrees with equation (3).

Now we take this value for b and substitute it into equation (ii), then solve for a.

$$a\sum_{k=1}^{n} x_k^2 + \left(\frac{\sum_{k=1}^{n} y_k - a\sum_{k=1}^{n} x_k}{n}\right)\sum_{k=1}^{n} x_k = \sum_{k=1}^{n} x_k y_k$$

$$na\sum_{k=1}^{n} x_k^2 + \; \sum_{k=1}^{n} y_k \sum_{k=1}^{n} x_k - a(\sum_{k=1}^{n} x_k)^2 = n\sum_{k=1}^{n} x_k\, y_k$$

$$a\big(n\sum_{k=1}^{n} x_k^2 - (\sum_{k=1}^{n} x_k)^2\big) = n\sum_{k=1}^{n} x_k\, y_k - \sum_{k=1}^{n} x_k \sum_{k=1}^{n} y_k$$

$$a = \frac{n\sum_{k=1}^{n} x_k y_k - \sum_{k=1}^{n} x_k \sum_{k=1}^{n} y_k}{n\sum_{k=1}^{n} x_k^2 - \left(\sum_{k=1}^{n} x_k\right)^2}$$

This agrees with equation (2).