Isotonic Regression by Dynamic Programming

Günter Rote
Institut für Informatik, Freie Universität Berlin, Takustraße 9, 14195 Berlin, Germany
rote@inf.fu-berlin.de
https://orcid.org/0000-0002-0351-5945

Abstract

For a given sequence of numbers, we want to find a monotonically increasing sequence of the same length that best approximates it in the sense of minimizing the weighted sum of absolute values of the differences. A conceptually easy dynamic programming approach leads to an algorithm with running time $O(n \log n)$. While other algorithms with the same running time are known, our algorithm is very simple. The only auxiliary data structure that it requires is a priority queue. The approach extends to other error measures.

2012 ACM Subject Classification Theory of computation → Dynamic programming, Theory of computation → Computational geometry, Mathematics of computing → Regression analysis

Keywords and phrases Convex functions, dynamic programming, convex hull, isotonic regression

Digital Object Identifier 10.4230/OASIcs.SOSA.2019.1

1 Problem Statement: Weighted Isotonic L_1 Regression

Weighted isotonic L_1 regression (or weighted isotonic median regression) is the following problem:

> Approximate a given sequence of numbers $a = (a_1, \dots, a_n)$ with weights $w_i > 0$ by an increasing sequence
>
> $$z_1 \le z_2 \le \dots \le z_n, \tag{1}$$
>
> minimizing the weighted L_1-error
>
> $$\sum_{i=1}^{n} w_i \cdot |z_i - a_i|. \tag{2}$$

If the input sequence $(a_1, \dots, a_n)$ has decreasing sections, the optimum solution values z_i tend to cluster together in *runs* or *level sets* of equal values $z_j = z_{j+1} = \dots = z_k$, see Figure 1. This common value z is the weighted median of the corresponding elements $a_j, a_{j+1}, \dots, a_k$, because this is the value that minimizes $\sum_{i=j}^{k} w_i|z - a_i|$.

Isotonic regression has applications in many fields, including statistics and production planning. The problem has been studied for a long time, see [2] for an early monograph, and there is a large literature that treats many variations of the problem. In particular, there are algorithms that solve the weighted L_1 regression problem in $O(n \log n)$ time [1, 8]. These algorithms will be reviewed in Section 11.

2 Our Algorithm

We present a new algorithm that is based on the dynamic programming method. Our approach differs somewhat from standard use of this technique, as the intermediate objects of our dynamic programming recursion are real functions, and thus infinite objects.

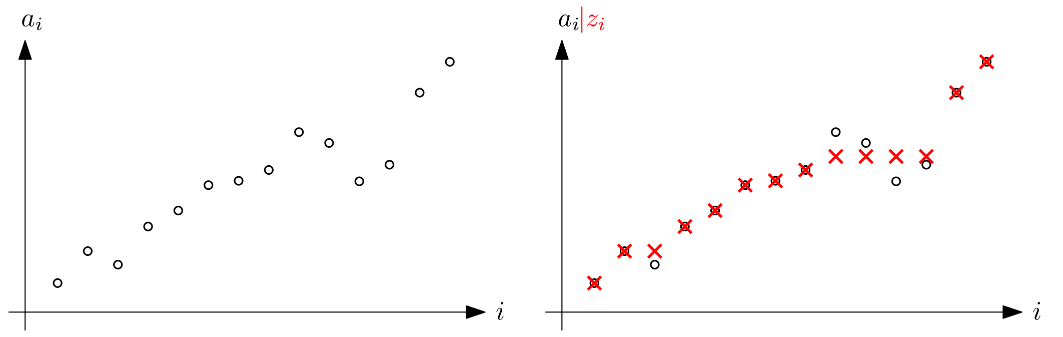

Figure 1 The original data sequence $(a_1, \dots, a_n)$ is shown on the left. The crosses on the right form a monotone approximation $(z_1, \dots, z_n)$. It contains two runs that are longer than a single element.

This is not a revolutionary idea. Historically, the concept of dynamic programming and Bellman's optimality principle applies also to continuous processes such as rocket flight. And while the notion of "dynamic programming" has made an independent career as an algorithmic design principle in computer science, it continues to be used in optimal control and in discrete as well as continuous optimization.

Computer scientists, on the other hand, tend to shun continuous and infinite structures. They need not be afraid: The functions that arise in our problem turn out to be piecewise linear functions, and they can be treated as peaceful discrete objects. In the programming contest literature, this approach is known under the name "convexity dynamic programming", see Section 12.

With the proper choice of representation, our approach leads to a simple algorithm with running time $O(n \log n)$. The most sophisticated data structure that is needed for an efficient implementation is a standard priority queue. Other merits of our algorithm are discussed in Section 11.1. Despite its simplicity, the algorithm would be mysterious without the conceptual background of its design (see Algorithm 3).

3 The Dynamic Programming Setup

We consider the subproblems

$$f_k(x) := \min\Big\{ \sum_{i=1}^{k} w_i \cdot |z_i - a_i| : z_1 \le z_2 \le \dots \le z_k = x \Big\} \tag{3}$$

for $k = 1, \dots, n$ and a real parameter x. We get the following straightforward dynamic programming recursion, including $k = 0$ as the base case:

$$\begin{aligned} f_k(x) &:= \min\{ f_{k-1}(z) : z \le x \} + w_k \cdot |x - a_k| \quad (k = 1, \dots, n;\ x \in \mathbb{R}) \\ f_0(x) &:= 0 \qquad (x \in \mathbb{R}) \end{aligned} \tag{4}$$

The following sections develop the details of how to implement this recursion.

4 The Functions f_k

▶ **Lemma 1.**

(a) *Each function f_k is a piecewise linear convex function, for $0 \le k \le n$.*

(b) *The breakpoints are located at a subset of the points $a_1, a_2, \dots, a_k$.*

(c) *The leftmost piece has slope $-\sum_{i=1}^{k} w_i$. The rightmost piece has slope w_k.*

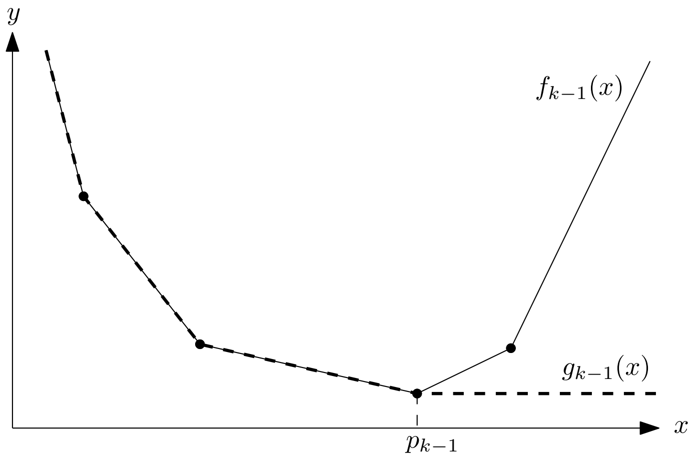

Figure 2 Constructing g_{k-1} from f_{k-1}.

Proof. These properties are easily established by induction. The base cases ($k = 0$ and $k = 1$) are obvious. We denote by

$$g_{k-1}(x) := \min\{\, f_{k-1}(z) : z \le x \,\}$$

the intermediate function in the transition from f_{k-1} to f_k.

Let $k \ge 2$, and let us assume by induction that all properties of the lemma hold for f_{k-1}. The function f_{k-1} is first monotonically decreasing to a minimum and then monotonically increasing. We denote by p_{k-1} the (not necessarily unique) position where the minimum occurs. The optimum of $f_{k-1}(z)$ under the constraint $z \le x$ depends on the position of x relative to p_{k-1}: If $x \le p_{k-1}$ then $z = x$ is the optimum choice, and $g_{k-1}(x) = f_{k-1}(x)$. If $x \ge p_{k-1}$ then the optimum choice is $z = p_{k-1}$, and $g_{k-1}(x) = f_{k-1}(p_{k-1})$, see Figure 2.

As a consequence of this, we get the following relation between the values z^*_{k-1} and z^*_k in the optimal solution:

$$z^*_{k-1} = \min\{z^*_k, p_{k-1}\} \tag{5}$$

This will be useful for recovering the optimal regression after its objective function value, i.e., the regression error, has been obtained.

In summary, *the function g_{k-1} has the same decreasing pieces as f_{k-1} but the increasing pieces are replaced by a horizontal piece of constant value $f_{k-1}(p_{k-1})$.*

Finally, to obtain f_k, we add the piecewise linear function $w_k \cdot |x - a_k|$ to g_{k-1}. It is now easy to see that f_k has the claimed properties of the lemma. ◀

5 Representing Piecewise Linear Functions

The most natural representation for a continuous piecewise linear function f would be a sorted list of breakpoints x_i with their function values $f(x_i)$, plus the slopes of the two unbounded pieces on the left and on the right. However, looking back at the discussion of the previous paragraph, the recursion (4) involves the addition of a piecewise linear function to another. The natural representation would then require all slopes to be updated.

We therefore prefer to maintain slope *differences* rather than the slopes themselves. We represent a piecewise linear function as a list of *breakpoints*, see Figure 3. Each breakpoint has a *position* – the x-value where it is located – and a *value* – the slope difference between

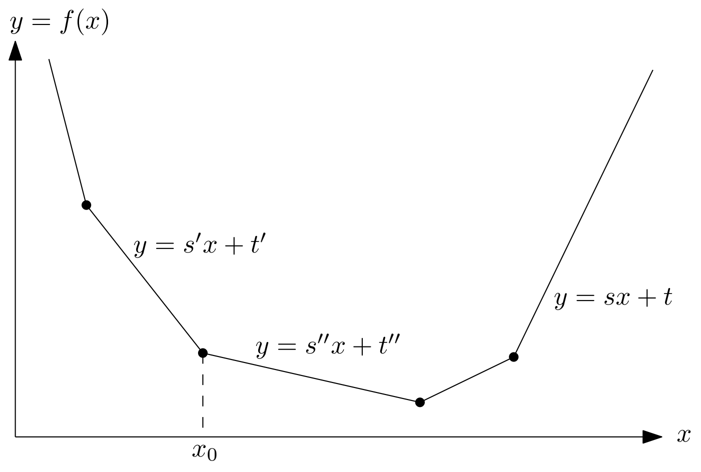

Figure 3 A piecewise linear function with four breakpoints. There is a breakpoint at position x_0 with value $s'' - s'$.

the right and the left adjacent pieces. The breakpoints are naturally ordered by position, but for the time being, we leave it unspecified whether we want to store them as a sorted list or in some other data structure. The function is convex if all breakpoints have nonnegative values.

The breakpoint data determine the function f only up to addition of an arbitrary linear function. We must specify two further parameters. Since the transition from f_{k-1} to g_{k-1} involves inspections and modifications at the right end of the function, it is most convenient to take the slope s and the intercept t of the *rightmost* linear piece $y = sx + t$.

This determines f uniquely: We proceed from right to left, and across each breakpoint, the value of the breakpoint gives us the slope $\hat{s}$ of next linear piece $y = \hat{s}x + \hat{t}$, and continuity of f allows us to fix the intercept $\hat{t}$.

Two functions are *added* by combining the list of breakpoints and adding the (s, t) parameters. If several breakpoints have the same position, they might be merged into one breakpoint, adding their values. However, this would require equal breakpoints to be found, and is not necessary; our algorithm will handle equal breakpoints just as well.

6 Carrying out the Recursion (4)

Recall from Section 4 that the function g_{k-1} has the same decreasing pieces as f_{k-1} but the increasing pieces are replaced by a horizontal piece of constant value $f_{k-1}(p_{k-1})$ and slope 0.

Algorithm 1 performs this transformation. It removes the increasing pieces from the right end of f_{k-1} one by one.

In representing the functions, we have to deal only with the *slope* s of the rightmost piece; the intercept t is not needed. Since the leftmost slope is negative, by Lemma 1c, the while-loop will terminate, and the list of breakpoints will never become empty. If f_{k-1} has a horizontal piece, the algorithm will arbitrarily choose the leftmost minimum p_{k-1}.

Finally, to obtain f_k, we must add the function $w_k \cdot |x - a_k|$ to g_{k-1}: This amounts to creating an additional breakpoint of value $2w_k$ at position a_k, and adding w_k to s.

Algorithm 1: Converting f_{k-1} to g_{k-1}.

```
Input: List of breakpoints of $f_{k-1}$ and rightmost slope $s$
Result: Updated list of breakpoints of $g_{k-1}$ and rightmost slope $s$; position $p_{k-1}$ of
        the (leftmost) minimum of $f_{k-1}$
Let $B$ be the rightmost breakpoint;
while $s - B.value \geq 0$ do  // next-to-last piece is not decreasing
    $s := s - B.value$; // remove the last piece
    Delete $B$ from the list of breakpoints;
    Let $B$ be the rightmost remaining breakpoint;
$p_{k-1} := B.position$; // $p_{k-1}$ is the position of the minimum of $f_{k-1}$.
$B.value := B.value - s$; // make the rightmost piece horizontal
$s := 0$;
```

7 The Weighted Regression Algorithm

We see that the algorithm only needs to access the rightmost breakpoint, and potentially delete it. A new breakpoint is inserted for each new data point a_k. This calls for a (max-)priority queue for storing the breakpoints, using *position* as the key. We use the standard priority queue operations *insert*, *findmax* (taking constant time), and *deletemax*.

Algorithm 2 shows the complete algorithm that we can now put together. The organization is slightly different from the recursion (4): an iteration of the main loop starts from g_{k-1}, turns it into f_k, and then into g_k. We start with the function $g_0(x) = 0$. The algorithm records the minimum position p_k for each function f_k. In the last loop iteration, the minimum of f_n is found as part of the construction of g_n.

Finally, the optimum solution values z_i are computed in a simple loop according to (5), starting with the minimum $z_n = p_n$ of the function f_n.

The variable s is always 0 at the beginning of the loop. Hence we can simplify the program. Also, it is advantageous to switch to the negative variable $\bar{s} \equiv -s$, because this turns all remaining subtractions into additions and makes these operations more transparent. The final computation of the optimal z_i values needs no change. The modified Algorithm 3 is shown below. Its inner loop can now be interpreted as follows: the variable $\bar{s}$ is initialized with the value $-w_k$; it accumulates and deletes the values from the top of the queue until the total value becomes positive.

Figure 4 shows the algorithm at work. The values in the queue Q are shown at selected times as if it were an ordered list with the highest keys a_i at the top. The first two iterations are not very interesting: A breakpoint of value $2w_k$ is inserted and, since it is at the top of Q, it is immediately reduced to w_k. The iteration $k = 3$ is shown in more detail: After $2w_3$ is inserted, $\bar{s}$ starts at $-w_3$. $\bar{s}$ is combined with the value w_2 at the top of the list, but the result, $w_2 - w_3$, is still negative. So it is combined with $2w_3$ to give $w_3 + w_2$, which is positive, and the iteration is completed.

8 Runtime Analysis

In total, n elements are inserted in the queue Q. Each iteration of the while-loop removes an element from Q, and therefore the overall number of executions of the while-loop is bounded by n. With a heap data structure for Q, each operation *deletemax* or *insert* can be carried out in $O(\log n)$ time. The *findmax* operation takes only constant time. The priority queue is not affected by the manipulation of the *values*, since it is ordered by *position*. Hence, the overall running time is $O(n \log n)$.

Algorithm 2: Dynamic Programming Algorithm for weighted isotonic L_1 regression.

```
$Q := \emptyset$; // priority queue of breakpoints ordered by the key $position$
$s := 0$;
for $k := 1, \ldots, n$ do
    // We start from $g_{k-1}$.
    $Q.insert$(new breakpoint $B$ with $B.position := a_k$, $B.value := 2w_k$);
    $s := s + w_k$; // We have computed $f_k$.
    $B := Q.findmax$;
    while $s - B.value \geq 0$ do
        $s := s - B.value$;
        $Q.deletemax$;
        $B := Q.findmax$;
    $p_k := B.position$; // record the position of the minimum
    $B.value := B.value - s$;
    $s := 0$; // We have computed $g_k$.
// compute the optimal solution $z_1, \ldots, z_n$:
$z_n := p_n$;
for $k := n-1, n-2, \ldots, 1$ do
    $z_k := \min\{z_{k+1}, p_k\}$;
```

Algorithm 3: Dynamic Programming Algorithm for weighted isotonic L_1 regression, simplified version.

```
$Q := \emptyset$; // priority queue of breakpoints ordered by the key $position$
for $k := 1, \ldots, n$ do
    $Q.insert$(new breakpoint $B$ with $B.position := a_k$, $B.value := 2w_k$);
    $\bar{s} := -w_k$;
    $B := Q.findmax$;
    while $\bar{s} + B.value \leq 0$ do
        $\bar{s} := \bar{s} + B.value$;
        $Q.deletemax$;
        $B := Q.findmax$;
    $B.value := \bar{s} + B.value$;
    $p_k := B.position$; // record the position of the minimum
// The solution $z_1, \ldots, z_n$ is computed in the same way as in Algorithm 2.
```

▶ **Theorem 2.** *Algorithm 3 (the Dynamic Programming Algorithm) solves the weighted isotonic L_1 regression problem in $O(n \log n)$ time and $O(n)$ space.*

9 Unweighted Regression

In the unweighted case ($w_i \equiv 1$), some steps can be simplified: The variable $\bar{s}$ is always -1 before the while-loop. Thus, it can be eliminated and the while-loop turned into an if-loop. Breakpoints have value 1 or 2. Algorithm 4 shows the simplified version. It is possible to

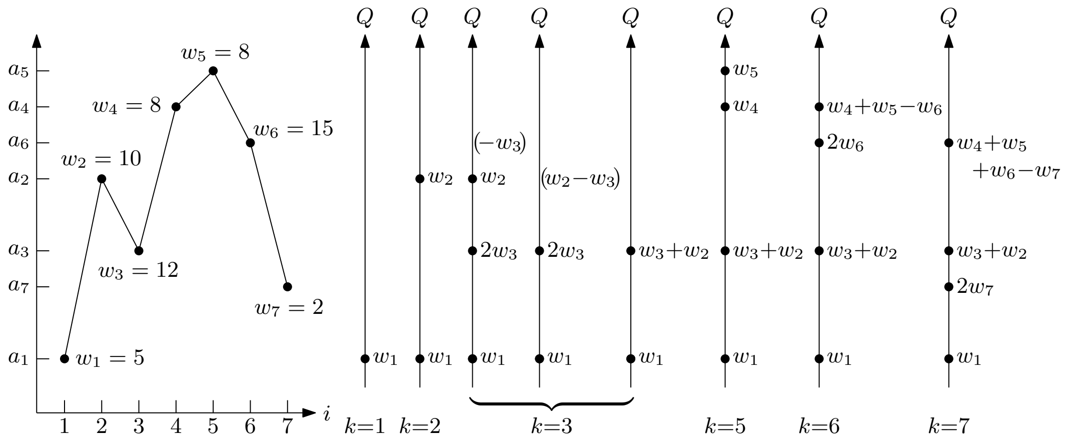

Figure 4 Running Algorithm 3 on an example.

Algorithm 4: Unweighted isotonic L_1 regression by dynamic programming.

```
$Q := \emptyset$; // priority queue of breakpoints ordered by the key *position*
**for** $k := 1, \dots, n$ **do**
    $Q.insert$(new breakpoint $B$ with $B.position := a_k$, $B.value := 2$);
    $B := Q.findmax$;
    **if** $B.value = 1$ **then**
        $Q.deletemax$;
        $B := Q.findmax$;
    **else**
        $B.value := 1$;
    $p_k := B.position$;
// The solution $z_1, \dots, z_n$ is computed in the same way as in Algorithm 2.
```

write an even simpler algorithm by eliminating the *value* attribute altogether: Instead of a breakpoint of value 2, we insert two (unweighted) breakpoints at the same position. The resulting main loop has no if-statements and needs only five lines.

10 Other Error Measures: Weighted Isotonic L_2 Regression

Instead of the L_1-error, one can consider other objective functions, where the absolute value is replaced by a different convex function h:

$$\sum_{i=1}^{n} w_i \cdot h(z_i - a_i). \tag{6}$$

More generally, one can allow a separate error measure h_i for each data point:

$$\sum_{i=1}^{n} h_i(z_i). \tag{7}$$

In this setting, there is no need for a_i and w_i, because these data can be incorporated in h_i. The most commonly considered case is the (squared) L_2-error:

$$E_2 = \sum_{i=1}^{n} w_i \cdot (z_i - a_i)^2 \tag{8}$$

It is straightforward to extend our approach to this objective function.

Exercises.

1. Show that the functions f_k defined in analogy to (3) for the L_2-case are piecewise quadratic convex functions. Explore their further properties, in analogy to Lemma 1.
2. Design an appropriate efficient data structure for representing this class of functions.
3. Show how to solve the dynamic programming recursion in $O(n)$ overall time, using only a stack as a data structure.
4. Explain which properties of the objective function, $h(z-a) = |z-a|$ versus $h(z-a) = (z-a)^2$, are responsible for the difference between the runtime of $O(n \log n)$ versus $O(n)$.
5. Compare your algorithm to the Incremental PAV Algorithm of Stout [8, Fig. 7 in connection with Fig. 5] and find out whether the two algorithms carry out essentially the same calculations.
6. Adapt the algorithm to the L_4 error, $h(x) = x^4$.
7. Show that the algorithm can be extended to handle arbitrary piecewise polynomial functions h_i in (7), provided they are convex.
8. Prove that the solution is unique if the functions h_i are strictly convex.

See Appendix A for answers.

11 Other Algorithms

The most popular approach for the isotonic regression problem is the algorithm *Pool Adjacent Violators* (PAV), see for example [2] or [8]. This classical method, which has often been rediscovered, starts by combining adjacent values that are not monotone ($a_i > a_{i+1}$) into pairs, and it further combines runs into larger runs as long as the weighted medians of adjacent groups are out of order. This algorithm extends to quite general settings like (7), provided that the functions h_i are convex.

Ahuja and Orlin [1] gave the first $O(n \log n)$ algorithm for weighted isotonic L_1 regression. It is based on the PAV principle but uses scaling for speedup. The core of this *Scaling PAV* algorithm is a procedure to turn a solution for the data $\lfloor a_i/2 \rfloor$ into a solution for the original data a_i in linear time (assuming that the a_i are integral). To get a running time that is independent of the range of values, the algorithm replaces the given values a_i by $1, \dots, n$ while keeping their relative order fixed.

Stout [8] has given a direct implementation of the PAV approach. I will refer to his algorithm as the *Incremental PAV* algorithm, because it adds the elements one at a time and completes the necessary PAV updates before looking at the next element. It requires mergeable trees (for example, AVL trees or 2-3-trees, see [5]), to achieve a running time of $O(n \log n)$.

Another natural approach is to model the problem as a minimum-cost network flow problem, see Figure 5. (I could not track down a specific source for this in the literature.) The unknown approximation values z_i are flow values along a path. Each inequality $z_i \le z_{i+1}$ becomes a flow conservation constraint, with an additional entering arc taking the slack. Since flow is nonnegative, we have to assume that all $a_i \ge 0$, which is no loss of generality. In

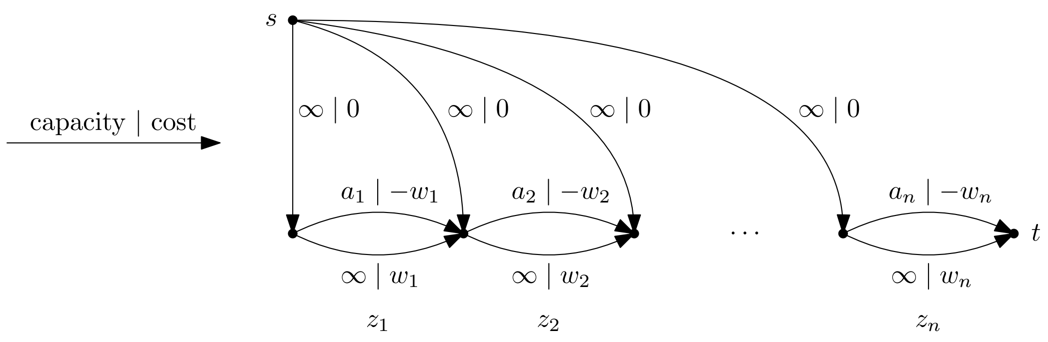

Figure 5 Minimum-cost network flow from s to t.

order to model the piecewise linear costs, each flow z_i is distributed over two parallel edges: a cheaper edge with bounded capacity and a more expensive edge with unbounded capacity. We denote by z_i the combined flow of these two edges. Then the cost of a flow $(z_1, \dots, z_n)$ in this network is

$$\begin{aligned} c(z_1,\dots,z_n) &= \sum_{i=1}^{n} \bigl[-w_i \min\{z_i, a_i\} + w_i \max\{z_i - a_i, 0\}\bigr] \\ &= \sum_{i=1}^{n} \bigl[-w_i(\min\{z_i - a_i, 0\} + a_i) + w_i \max\{z_i - a_i, 0\}\bigr] \\ &= \sum_{i=1}^{n} \bigl[w_i \max\{a_i - z_i, 0\} + w_i \max\{z_i - a_i, 0\} - w_i a_i\bigr] \\ &= \sum_{i=1}^{n} w_i \max\{a_i - z_i, z_i - a_i\} - \sum_{i=1}^{n} w_i a_i, \end{aligned}$$

which differs from the original approximation error (2) just by the constant $\sum_{i=1}^{n} w_i a_i$.

This network is series-parallel, and hence the minimum-cost flow can be found in $O(n \log n)$ time by an algorithm of Booth and Tarjan [4]. However, this algorithm also relies on mergeable trees, and moreover, it needs $O(n \log^* n)$ space to recover the optimum solution. So this approach is not preferable to Stout's algorithm. The algorithm follows the dynamic programming paradigm, and thus, in spirit, it is closer to the algorithm of this paper. We suspect that a closer study of the algorithm for the specialized network structure of Figure 5, instead of applying it out of the box, might have led to the discovery of our Dynamic Programming Algorithm.

Ahuja and Orlin [1] mistakenly credit [6] for an earlier $O(n \log n)$ time algorithm, but that paper has only an algorithm with $O(n \log^2 n)$ runtime.

11.1 Comparison

11.1.1 Simplicity

The Incremental PAV Algorithm of Stout [8] involves tree data structures (for example, AVL trees or 2-3-trees) augmented with weight information, and needs the nonstandard merging operation: two trees of size m and n with $m \le n$ have to be merged in $O(m \log \frac{n}{m}) = O(\log \binom{m+n}{n})$ time.

If might be instructive to compare the algorithms on the example of Figure 4. After item 6 is added, items 4–6 form a run $z_4 = z_5 = z_6$. In the Incremental PAV Algorithm, the three elements have been combined into a mergeable tree, in order to compute their weighted median. Our Dynamic Programming Algorithm, by contrast, has somehow taken note of the run by forming the sum $w_4 + w_5 - w_6$. On the other hand, it still keeps $2w_6$ as a separate item. Thus, it is probably difficult to explain one algorithm in terms of the other, and the distinction between the two algorithms is more fundamental.

The Scaling PAV Algorithm of Ahuja and Orlin [1], on the other hand, needs no data structures beyond arrays and linked lists. The algorithm itself, however, is not so simple. Moreover, it requires an initial sort of the elements.

The Dynamic Programming Algorithm is very simple and requires just a priority queue, in which each element is inserted once and retrieved at most once. Thus, the priority queue has to perform the same *insert* operations and fewer *deletemax* operations than would be required for heapsort; one might expect that the Dynamic Programming Algorithm is finished while the Scaling PAV Algorithm is still busy in its sorting phase.

11.1.2 Incremental Computation (Prefix Regression)

The Dynamic Programming Algorithm processes data as they arrive, producing the solution for the first k items after reading them. (To have the objective function value (2) always ready, the algorithm must be extended to deal with the intercept t in addition to the slope s. The most convenient way to do this is to maintain the value $f_k(p_k)$.)

Stout [8] has called this problem the *prefix isotonic regression* problem: solving the regression problem for all prefixes of the input. His Incremental PAV Algorithm solves this problem readily in $O(n \log n)$ time. Stout [8, p. 295] notes that this is optimal because prefix isotonic regression can be used for sorting. He uses it as a subroutine for the *unimodal* regression problem. Our algorithm can also be used for this purpose.

Ahuja and Orlin's Scaling PAV Algorithm is not suitable for incremental computation.

11.1.3 Numerical Precision

The common solution value $z_i = z_{i+1} = \cdots = z_k$ of each run is the weighted median. Thus, any algorithm that solves the problem necessarily has to compare sums of the form $\sum_{i \in I} w_i$ in order to compute weighted medians.

In our algorithm, the k-th iteration inserts a new entry of value $2w_k$ into the queue. In addition, the starting value $\bar{s} = -w_k$ is added with some entries from the top of the queue. Everything that is calculated is built from the ground set of $2n$ elements $2w_i$ and $-w_i$ by adding subsets of these elements together in some hierarchical order. All results that are ever computed are therefore of the form $2w_i$ or of the form $\sum_{i=1}^{n} e_i w_i$, where $e_i \in \{0, 1, -1\}$, and the values of the latter form are compared with 0.

The term $+w_i$ in these expressions is formed by adding $2w_i$ and $-w_i$. This incurs a slight loss of precision of 1 bit in terms of weights, when compared with the calculations that any algorithm must necessarily perform for solving the problem.

11.1.4 Data Sensitivity

Another question is how the algorithm responds to input sequences that are almost sorted. This is a natural assumption in statistical applications, where the data "ought" to be monotone but is distorted by noise.

The Incremental PAV method will have an advantage, since values a_i that are in the correct order with respect to their neighbors and are approximated by themselves ($z_i = a_i$) will be looked at only once. Moreover, runs will usually be short, and in the $O(\log n)$ bound on the tree operations, the parameter n can be replaced by the run length.

The Scaling PAV Algorithm of Ahuja and Orlin [1], on the other hand, is completely insensitive to the data: in addition to sorting, it will always perform $\Theta(\log n)$ linear-time sweeps over the data.

The Dynamic Programming Algorithm might potentially benefit from almost sorted data. At least in the case when the input comes in truly sorted order, the algorithm will never call *deletemax*. To take advantage of almost sorted data, one would need a priority queue where it is cheaper to retrieve (by *findmax*) and delete elements that have been inserted recently.

12 Convexity Dynamic Programming

One referee has pointed out that the technique that we advocate is known in the programming contest literature under the name “convexity dp” (for “convexity dynamic programming”). In fact, the unweighted problem (Section 9) has become quite a standard problem in programming contests: It was used as problem *SEQUENCE* in the 2004 Balkan Olympiad for Informatics[1], and was even solved during the contest by one high school student, Filip Wolski, with an $O(n \log n)$ solution. Numerous programs that use just a few lines of code and implement the algorithm of Section 9 can be seen at the *Codeforces* programming contest platform[2].

Another problem of the same flavor, which can also be solved using convexity, has been posed in 2009 under the name *CCROSSX – Cross Mountain Climb Extreme*[3]. Here, the approximating sequence is not required to be increasing, but it is restricted to have a bounded difference between successive elements: $|z_i - z_{i+1}| \leq d$, for some given bound d.

We are grateful to the referee for the pointers to the programming contest community.

References

1. R. K. Ahuja and J. B. Orlin. A fast scaling algorithm for minimizing separable convex functions subject to chain constraints. *Operations Research*, 49:784–789, 2001. `doi:10.1287/opre.49.5.784.10601`.
2. R. E. Barlow, D. J. Bartholomew, J. M. Bremner, and H. D. Brunk. *Statistical Inference under Order Restrictions*. Wiley, 1972.
3. R. E. Barlow and H. D. Brunk. The Isotonic Regression Problem and its Dual. *Journal of the American Statistical Association*, 67(337):140–147, 1972. `doi:10.1080/01621459.1972.10481216`.
4. Heather Booth and Robert Endre Tarjan. Finding the minimum-cost maximum flow in a series-parallel network. *J. Algorithms*, 15:416–446, 1993. `doi:10.1006/jagm.1993.1048`.
5. M. R. Brown and R. E. Tarjan. A fast merging algorithm. *J. Assoc. Comp. Mach.*, 26:211–226, 1979. `doi:10.1145/322123.322127`.
6. P. M. Pardalos, G. L. Xue, and Y. Li. Efficient computation of an isotonic median regression. *Applied Math. Letters*, 8(2):67–70, March 1995. `doi:10.1016/0893-9659(95)00013-G`.
7. Franco P. Preparata and Michael Ian Shamos. *Computational Geometry. An Introduction.* Springer, 1985.
8. Quentin F. Stout. Unimodal regression via prefix isotonic regression. *Comp. Stat. and Data Anal.*, 53:289–297, 2008. `doi:10.1016/j.csda.2008.08.005`.

[1] `http://www.boi2004.lv/Uzd_diena1.pdf`
[2] `https://codeforces.com/contest/13/status/C`
[3] `https://www.spoj.com/problems/CCROSSX/`

A Weighted Isotonic L_2 Regression

We state without proof the properties of the functions f_k that arise for the L_2 objective function.

▶ **Lemma 3.**

(a) *f_k is a piecewise quadratic convex function.*

(b) *The derivative $f'_k(x)$ is a piecewise linear increasing and concave function. In particular, it is continuous.*

(c) *The leftmost piece of $f'_k(x)$ has slope $2\sum_{i=1}^{k} w_i$. The rightmost piece has slope $2w_k$.*

In contrast to the L_1 case (Lemma 1), the breakpoints are not restricted to the points a_i.

The transition from f_{k-1} to g_{k-1} requires the elimination of the increasing part at the right rim. It is easier to carry this out at the level of the derivative: To go from f'_{k-1} to g'_{k-1}, we eliminate the positive part and replace it by the constant 0 function, see Figure 6. Afterwards, in order to produce f'_k, we increment g_{k-1} by the derivative of $w_k(x-a_k)^2$, which is the linear function $2w_k(x-a_k)$.

If we represent the quadratic pieces of the form $bx^2 - 2cx + d$ by triplets (b, c, d), we can arrange things so that the algorithm performs almost the same calculations as the Incremental PAV Algorithm of Stout [8, Fig. 7].

Like for the case of the L_1 norm (Section 5), the necessity to add functions suggests that we store differences $(\Delta b, \Delta c, \Delta d)$ between successive pieces instead of the values themselves. In fact, the differences Δd of the constant terms are redundant, because two adjacent quadratic pieces must touch at some point where they have a common tangent. In other words, the graph of the difference function $\Delta b \cdot x^2 - 2\Delta c \cdot x + \Delta d$ must touch the x-axis, and therefore $\Delta d = \Delta c^2/\Delta b$. The fact that Δd is irrelevant is also evident from the fact that f_k, its derivative being f'_k (Figure 6), is determined by f'_k up to one parameter, the integration constant. The d_i coefficients don't show up in f'_k, and therefore their differences play no role in determining f_k.

When cutting away the positive branch of the function f'_{k-1}, the algorithm must decide whether the rightmost piece, $y = 2b_1x - 2c_1$, should be completely eliminated, see Figure 6. This is the case if its intersection X with the second piece $y = 2b_2x - 2c_2$ lies to the right of the intersection with the x-axis. As shown in the illustration, this amounts to the comparison $\frac{\Delta c}{\Delta b} \ge \frac{c_1}{b_1}$.

Now, the (b, c) and $(\Delta b, \Delta c)$ values are initially created from the quadratic functions $w_kx^2 - 2w_ka_k + a_k^2 = bx^2 - 2cx + d$, and thus $(b, c) = (w_k, w_ka_k)$ During the algorithm, adjacent $(\Delta b, \Delta c)$ values are pooled together, and they become expressions of the form $\Delta b = \sum_{i=j}^{k} w_i$, $\Delta c = \sum_{i=j}^{k} w_ia_i$. These are the same as the quantities *sumw* and *sumwy* that are maintained in [8, Fig. 7]. The test $\frac{\Delta c}{\Delta b} \ge \frac{c_1}{b_1}$ is nothing but a comparison of weighted averages. Thus, indeed, the core of the Dynamic Programming Algorithm performs the same calculations as Stout's Incremental PAV Algorithm.

There are, however, some slight differences.

(i) The Incremental PAV Algorithm updates the optimal solution value E_2 directly in one step, after all positive parts of f'_k have been eliminated. For this purpose, it maintains the variables *sumwy2* $= \sum_{i=j}^{k} w_ia_i^2$. In our Algorithm 3, the update of the objective function is not detailed, but it would be most natural to update it with each iteration of the while-loop.

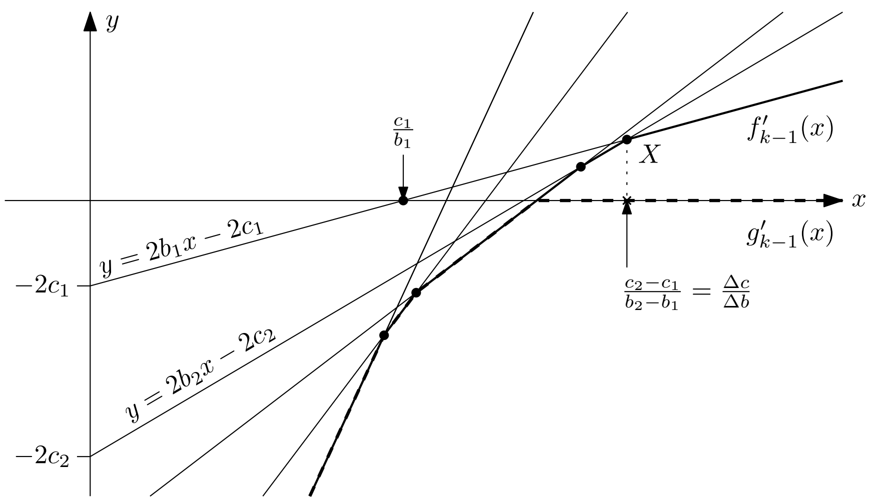

Figure 6 Transforming f'_{k-1} into g'_{k-1}.

(ii) Another difference is the determination of the optimal solution z. Stout's Incremental PAV Algorithm is straightforward: it simply sets each variable z_i to the weighted average of its run. The Dynamic Programming Algorithm, on the other hand, computes the solution by the formula (5) and achieves the same result in an indirect way, see the last part of Algorithm 2.

The reason why the L_1-regression requires $O(n \log n)$ time and the L_2-regression does not (Question 4 in Section 10) is that the absolute value function in the L_1 objective inserts breakpoints of its own, whereas the L_2 objective function is smooth, and breakpoints are created only at the right end when replacing the increasing part of f_k by a flat part.

A.1 Geometric Interpretation: Lower Envelope and Lower Convex Hull

If we look at the area below the graphs f_k and g_k, the algorithm can be interpreted geometrically as an alternating succession of the following two operations.

- Intersect the area with the negative halfplane.
- Apply an affine transformation (in particular, a shearing transformation).

We can eliminate the affine transformations and carry out all intersection operations in the original coordinate system. The problem reduces to an intersection of lower halfplanes, which result from applying the appropriate shearing transformation to the negative halfplanes, as shown in Figure 7.

▶ **Theorem 4.** *The weighted isotonic L_2 regression problem of minimizing* (8) *subject to the monotonicity constraints* (1) *can be solved by constructing the lower envelope of $n+1$ lines $F_0, F_1, \dots, F_n$, which are given by*

$$F_k\colon\ y = 2\sum_{i=1}^{k} w_i a_i - \Bigl(2\sum_{i=1}^{k} w_i\Bigr)x.$$

If two lines F_j and F_k form adjacent edges on the lower envelope, then the optimum solution has a run $z_{j+1} = z_{j+2} = \cdots = z_k$ of values that are equal to the x-coordinate of the intersection point between F_j and F_k.

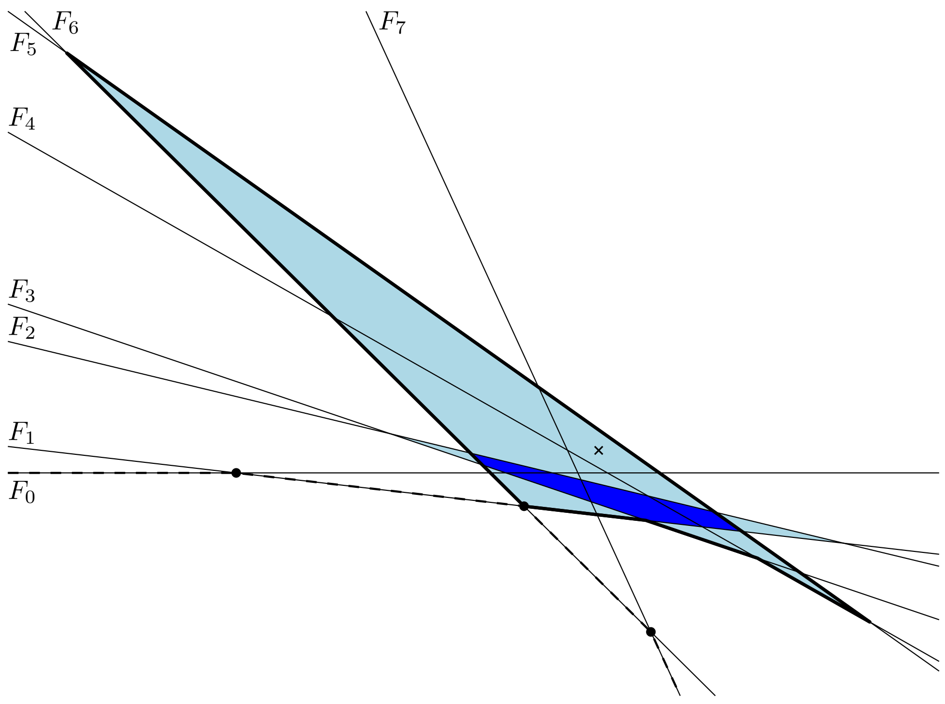

Figure 7 The L_2 isotonic regression problem as the lower envelope of lines. The approximation error E_2 equals the shaded area, with the dark shaded area counted twice.

Moreover, the approximation error E_2 of the lines can be expressed as a weighted sum of certain face areas in the line arrangement. For a face in the arrangement, we record a sequence of $n + 1$ pluses and minuses, depending on whether the face lies above F_k (+) or below F_k (−). For example, the face marked by a cross in Figure 7 has the sequence $+{+}{+}{+}{+}{-}{+}{+}$. If there are r runs of consecutive pluses, then the area of this face is counted with multiplicity $\max\{r - 1, 0\}$.

This is of course not the best way to compute the approximation error E_2. It can easily be computed in linear time by substituting the optimal solution into (8). The program of Stout [8, Fig. 7] computes the approximation error incrementally in a more direct way.

The above formula arose by working out how the objective function changes when k is incremented. It turns out that one has to add a certain integral, which, in terms of the arrangement of the lines F_i, equals the area that lies (i) above F_k, (ii) below F_{k-1}, and (iii) above the lower envelope of $F_0, F_1, \dots, F_{k-1}$. Figure 7 shows this area for $k = 6$ with a heavy outline. (Bear in mind that the line F_k or F_{k-1}, respectively, corresponds to the x-axis in the original setting of Figure 6.) Pursuing this further, one can show that the approximation error can be expressed as the sum of at most $n - 1$ triangle areas. This can be translated into the weighted sum of face areas that was given above.

Using a well-known geometric duality transform, the lower-envelope problem turns into the problem of computing the lower convex hull (or *greatest convex minorant*) of a set of points: The line $y = ax + b$ becomes the point $(-a, b)$ and the point (u, v) becomes the line $y = ux + v$. This duality preserves incidences and above/below relations. After canceling the factor 2 from the point coordinates, we arrive at the following result, see Figure 8.

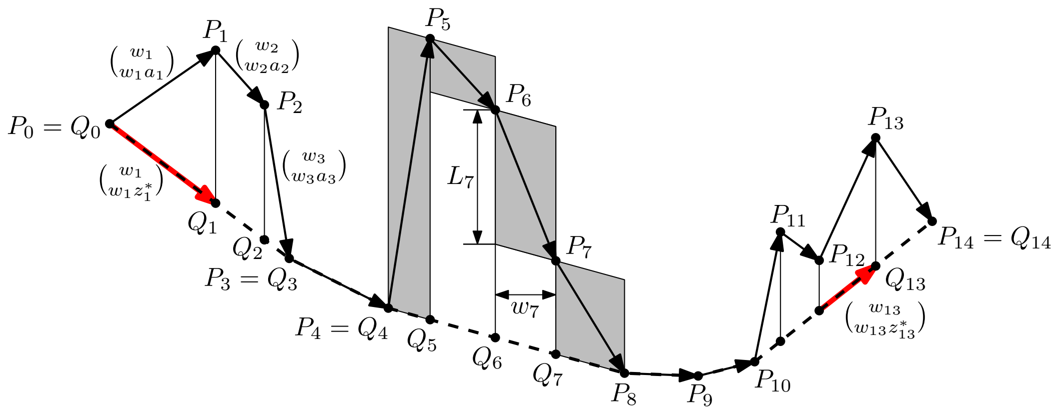

Figure 8 The regression problem as a lower convex hull. Some points Q_i on the lower hull are marked, together with two selected vectors $Q_{i-1}Q_i$; z_i^* denotes the convex hull solution.

▶ **Theorem 5.** *Consider a polygonal chain $P_0P_1 \dots P_n$ whose segments are given by the vectors $P_i - P_{i-1} = \binom{w_i}{w_i a_i}$.*

The weighted isotonic L_2 regression problem of minimizing (8) subject to the monotonicity constraints (1) can be solved by constructing the lower convex hull of the chain $P_0P_1 \dots P_n$.

If P_jP_k is an edge of the lower convex hull, then the optimum solution has a run $z_{j+1} = z_{j+2} = \dots = z_k$ of values that are equal to the slope of the edge P_jP_k.

This surprising connection between an isotonic regression problem and a basic computational geometry problem has been known for a long time [2, Section 1.2, mentioning earlier sources in Section 1.6], see also [7, Section 4.2.2]. Barlow and Brunk [3, Section 4.1, the *taut-string solution*], derived it as an instance of convex-programmining duality: The polygon $P_0P_1 \dots P_n$ is called the *cumulative sum diagram.* The set of lower envelopes $Q_0 \dots Q_n$ of $P_0 \dots P_n$ with endpoints $Q_0 = P_0$ and $Q_n = P_n$ forms a cone which is dual to the cone of increasing functions (1). From this, Theorem 5 (and the stronger inequality (11) below) follows by easy duality arguments [3, Theorem 2.1, in particular (2.4)]. A streamlined self-contained proof of Theorem 5, which does not depend on this background, is contained in the monograph of Barlow, Bartholomew, Bremner, and Brunk [2, Theorem 1.1]. We reproduce this proof in Appendix C. It appears as a clever algebraic manipulation, exploiting the fact that the optimum solution is unchanged ($z_i = z_{i+1}$) whenever the lower hull passes below P_i, see the complementarity condition (12) in Appendix C.

Since the vertices P_i are given in sorted order, the lower convex hull can be computed in linear time. The standard incremental algorithm for this task becomes the same as the Dynamic Programming Algorithm or Stout's Incremental PAV method for this case.

The approximation error E_2 is not so easy to figure out in this representation. One might be tempted to believe that it is area of the pockets between the polygonal chain and its convex hull, but this is not true: This area depends locally only linearly on the data, while the error function is quadratic. Here is an attempt at a geometric interpretation of the objective function: Enclose each edge $P_{i-1}P_i$ in a vertical parallelogram, with two sides parallel to the convex hull edge under P_{i-1} and P_i. Figure 8 shows a few of these parallelograms. If such a parallelogram has horizontal width w_i and vertical edges of length L_i, it contributes $L_i^2 w_i$ to the objective function. The objective function is the total contribution of all n edges. This is of course not a deep statement; we just measure the squared deviation of each step a_i from the average of each run, i.e., z_i.

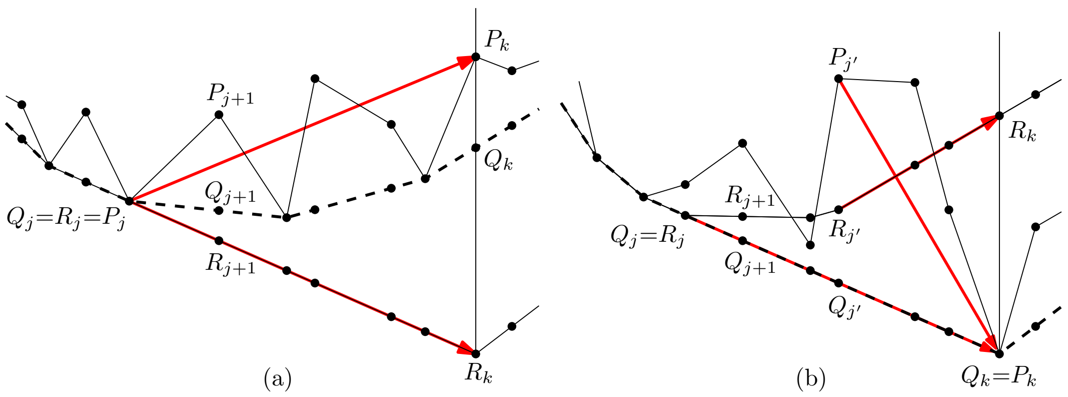

Figure 9 (a) Case 1. $z_{j+1} < z^*_{j+1}$. (b) Case 2. $z_{j+1} > z^*_{j+1}$. The segments whose slopes are compared in the proof are highlighted.

B Direct Proof of Theorem 5

For completeness, and for comparison, we give an independent elementary proof of the correspondence between isotonic L_2 regression and the convex hull (Theorem 5). This pedestrian proof does not assume any optimization background, and I hope it gives some more direct geometric insight into the correspondence.

The only tool that we need is the elementary fact that the best L_2-approximation by a run of equal values $z_j = z_{j+1} = \cdots = z_k = z$ is the weighted average:

▶ **Proposition 6.** *Let* $\mu = \sum_{i=j}^k w_i a_i / \sum_{i=j}^k w_i$ *be the weighted average of the sequence* $a_j, \ldots, a_k$. *Then*

$$\sum_{i=j}^{k} w_i(z-a_i)^2 = \sum_{i=j}^{k} w_i(\mu-a_i)^2 + \sum_{i=j}^{k} w_i(z-\mu)^2 \tag{9}$$

In particular, if we regard the left-hand side of (9) *as a function of* z, *it is a quadratic function with a unique minimum at* $z = \mu$.

If we start the polygonal chain at $P_0 = \binom{0}{0}$, the coordinates of the points $P_i = \binom{W_i}{A_i}$ are the partial sums $W_k = \sum_{i=1}^k w_i$ and the weighted partial sums $A_k = \sum_{i=1}^k w_i a_i$, for $0 \le k \le n$. The crucial observation is that the weighted average of a subsequence $a_j, \ldots, a_k$, which plays a prominent role in Proposition 6, shows up as the slope of the segment $P_{j-1}P_k$.

We denote the claimed optimal solution by z^*_i, and we set $Z^*_k = \sum_{i=1}^k w_i z^*_i$. Then the points $Q_i = \binom{W_i}{Z^*_i}$ form the lower convex hull $Q_0Q_1 \ldots Q_n$: The vector $Q_i - Q_{i-1} = \binom{w_i}{w_i z^*_i}$ has slope z^*_i, and Q_i is the point where the vertical line through P_i intersects the lower hull, see Figure 8. Accordingly, $Z^*_i \le A_i$ for $i = 0, \ldots, n$. The endpoints are fixed to $Q_0 = P_0 = \binom{0}{0}$ and $Q_n = P_n$, and thus, $Z^*_0 = A_0 = 0$ and $Z^*_n = A_n$.

For comparison, we consider an arbitrary increasing sequence z_i. We define accordingly $Z_k = \sum_{i=1}^k w_i z_i$ and the points $R_i = \binom{W_i}{Z_i}$, forming the polygonal chain $R_0R_1 \ldots R_n$.

We will now show that a sequence $(z_1, \ldots, z_n) \ne (z^*_1, \ldots, z^*_n)$ cannot be optimal. The idea is to identify a subsequence of equal consecutive values and to show that they can be jointly modified to improve the objective function, while keeping the sequence increasing.

Let us suppose that the two sequence agree up to z_j. We distinguish whether z_{j+1} is smaller or larger than z^*_{j+1}.

Case 1. $z_{j+1} < z^*_{j+1}$, and accordingly, R_{j+1} lies below Q_{j+1}, see Figure 9a. Let us take the maximum $k \le n$ such that $z_{j+1} = z_{j+2} = \cdots = z_k$. In other words, R_k is the next vertex after R_j. The slope of $R_jR_{j+1}\ldots R_k$ is the common value $z = z_{j+1} = \cdots = z_k$. Since $R_j \ldots R_k$ forms a straight line segment and $Q_j \ldots Q_k$ is convex, Q_k must lie above R_k, and therefore P_k must also lie above R_k. Q_j is a vertex of the lower hull because $z^*_{j+1} > z_{j+1} \ge z_j = z^*_j$. Thus, $Q_j = R_j = P_j$. The weighted average of $a_{j+1}, \ldots, a_k$, which is the slope of the segment P_jP_k, is therefore larger than z (the slope of $R_j \ldots R_k$). It follows from Proposition 6 that the objective function can be improved by increasing the common value z of $z_{j+1} = z_{j+2} = \cdots = z_k$. By the maximal choice of k, some small increase of z is always possible without violating the monotonicity of the sequence.

Case 2. $z_{j+1} > z^*_{j+1}$, and accordingly, R_{j+1} lies above Q_{j+1}, see Figure 9b. Then R_j must be a vertex of the convex chain $R_0 \ldots R_n$, because $z_{j+1} > z^*_{j+1} \ge z^*_j = z_j$.
We choose the largest $k \le n$ such that $z^*_j = z^*_{j+1} = \cdots = z^*_k$. In other words, Q_k is the next vertex after Q_j on the lower hull, and thus $Q_k = P_k$. Now we choose the smallest $j' \ge 0$ such that $z_{j'+1} = z_{j'+2} = \cdots = z_k$. In other words, we extend the segment $R_{k-1}R_k$ to the left until we hit the previous vertex $R_{j'}$ of the convex chain. Since R_j is a vertex, as just observed, we have $j' \ge j$. Since $P_{j'}$ lies on or above the segment Q_jQ_k,

$$\text{slope}(P_{j'}P_k) = \text{slope}(P_{j'}Q_k) \le \text{slope}(Q_jQ_k). \tag{10}$$

On the other hand, $R_jR_{j+1}\ldots R_k$ branches off upwards from the segment Q_jQ_k and forms a convex chain with increasing slopes. Thus, the slope of $R_{j'}R_k$ is strictly larger than the slope of Q_jQ_k. With (10), we conclude that the slope of $R_{j'}R_k$ (the common value $z := z_{j'+1} = z_{j'+2} = \cdots = z_k$) is larger than the slope of the segment $P_{j'}P_k$ (the weighted average of $a_{j'+1}, \ldots, a_k$). Proposition 6 implies that the objective function can be improved by decreasing z. By the minimal choice of j', some small decrease of z is possible without violating the monotonicity of the sequence.

We have thus shown that a sequence different from $(z^*_1, \ldots, z^*_n)$ cannot be optimal. From here, there are two ways to conclude the proof of Theorem 5. (a) By observing that the objective function E_2 is continuous and goes to infinity when the argument $(z_1, \ldots, z_n)$ becomes unbounded, we establish that an optimum solution exists. Since $(z^*_1, \ldots, z^*_n)$ is the only solution that is not excluded by our arguments, it must be the optimum solution.

(b) If a constructive proof is preferred, one can extend the above argument with little additional effort to a procedure that transforms any solution $R_0 \ldots R_n$ into the lower hull $Q_0 \ldots Q_n$ in $O(n)$ steps without increasing the approximation error. ◀

C The proof of Barlow, Bartholomew, Bremner, and Brunk (1972)

For the convenience of the reader, and for comparison, we reproduce the short and elegant proof of Theorem 5 from Barlow, Bartholomew, Bremner, and Brunk [2, Theorem 1.1, pp. 12–13], translated to our notation, with a few more details and and with a minor correction and improvement. They proved the following inequality:

$$\sum_{i=1}^{n} w_i \cdot (z_i - a_i)^2 \ge \sum_{i=1}^{n} w_i \cdot (z^*_i - a_i)^2 + \sum_{i=1}^{n} w_i \cdot (z_i - z^*_i)^2 \tag{11}$$

To establish the optimality of z^*, we would only need to compare the L_2-errors of an arbitrary increasing sequence z against z^* (the first two sums). The inequality (11) gives a stronger statement than needed for this, because of the additional quadratic term on the right, which measures the deviation between z and z^*. With this term, uniqueness of the optimal solution follows directly.

We use the setup with the polygons $P_0 \dots P_n$ and $Q_0 \dots Q_n$ and their coordinates $P_i = \binom{W_i}{A_i}$ and $Q_i = \binom{W_i}{Z_i^*}$, which were introduced in Appendix B for the first proof. We will need one more observation: If the lower hull lies strictly below P_i, then the hull edge passes straight through Q_i. In other words:

$$Z_i^* < A_i \implies z_i^* = z_{i+1}^* \tag{12}$$

Let us now go into the calculation: The inequality (11) has the form $\sum(u_i + v_i)^2 \geq \sum u_i^2 + \sum v_i^2$, and the difference between the left side and the right side is $2\sum u_i v_i$. In our case, this is

$$D = 2\sum_{i=1}^{n} w_i(z_i^* - a_i)(z_i - z_i^*) = 2\sum_{i=1}^{n}(z_i^* - z_i)(w_i a_i - w_i z_i^*).$$

We want to show that this is nonnegative. We apply the partial summation formula

$$\sum_{i=1}^{n} g_i(H_i - H_{i-1}) = \sum_{i=1}^{n-1}(g_i - g_{i+1})H_i + g_n H_n - g_1 H_0 \tag{13}$$

with $g_i = z_i^* - z_i$ and $H_i - H_{i-1} = w_i a_i - w_i z_i^*$, and therefore $H_i = A_i - Z_i^*$. In our case, $H_0 = A_0 = Z_0^* = 0$, and $H_n = 0$ because $Z_n^* = A_n$. Thus, the two boundary terms in (13) vanish, and we get

$$\begin{aligned} D &= 2\sum_{i=1}^{n-1}\big[(z_i^* - z_i) - (z_{i+1}^* - z_{i+1})\big](A_i - Z_i^*) \\ &= 2\sum_{i=1}^{n-1}(z_{i+1} - z_i)(A_i - Z_i^*) - 2\sum_{i=1}^{n-1}(z_{i+1}^* - z_i^*)(A_i - Z_i^*). \end{aligned}$$

The first sum is nonnegative because $z_{i+1} \geq z_i$ and $A_i \geq Z_i^*$. The second sum is 0 because of the complementarity relation (12). ◀