Efficient Second-Order Shape-Constrained Function Fitting[*]

David Durfee[†] Yu Gao[†] Anup B. Rao[‡] Sebastian Wild[§]

May 30, 2019

We give an algorithm to compute a one-dimensional shape-constrained function that best fits given data in weighted-L_∞ norm. We give a *single* algorithm that works for a variety of commonly studied shape constraints including monotonicity, Lipschitz-continuity and convexity, and more generally, any shape constraint expressible by bounds on first- and/or second-order differences. Our algorithm computes an approximation with additive error ε in $O\big(n \log \frac{U}{\varepsilon}\big)$ time, where U captures the range of input values. We also give a simple greedy algorithm that runs in $O(n)$ time for the special case of unweighted L_∞ convex regression. These are the first (near-)linear-time algorithms for second-order-constrained function fitting. To achieve these results, we use a novel geometric interpretation of the underlying dynamic programming problem. We further show that a generalization of the corresponding problems to directed acyclic graphs (DAGs) is as difficult as linear programming.

1. Introduction

We consider the fundamental problem of finding a function f that approximates a given set of data points $(x_1, y_1), \ldots, (x_n, y_n)$ in the plane with smallest possible error, i.e., $f(x_i)$ shall be close to y_i (formalized below), subject to shape constraints on the allowable functions f, such as being increasing and/or concave. More specifically, we present a new algorithm that can handle arbitrary constraints on the (discrete) first- and second-order derivatives of f.

When we only require f to be weakly increasing, the problem is known as isotonic regression, a classic problem in statistics; (see, e.g., [13] for history and applications). It has more recently also found uses in machine learning [17, 16, 12].

In certain applications, further shape restrictions are integral part of the model: For example, microeconomic theory suggests that production functions are weakly increasing and concave (modeling diminishing marginal returns); similar reasoning applies to utility functions. Restricting f to functions with bounded derivative (Lipschitz-continuous functions) is desirable to avoid overfitting [16]. All these shape restrictions can be expressed by inequalities for

[*]The first author is supported in part by National Science Foundation Grant 1718533. The last author is supported by the Natural Sciences and Engineering Research Council of Canada and the Canada Research Chairs Programme.

[†]Georgia Institute of Technology · {ddurfee,ygao380}@gatech.edu

[‡]Adobe Research · anuprao@adobe.com

[§]University of Waterloo · wild@uwaterloo.ca

first and second derivatives of f; their discretized equivalents are hence amenable to our new method. Shape restrictions that we cannot directly handle are studied in [28] (f is piecewise constant and the number of breakpoints is to be minimized) and [26] (unimodal f). For a more comprehensive survey of shape-constrained function-fitting problems and their applications, see [14, §1]. Motivated by these applications, the problems have been studied in statistics (as a form of nonparametric regression), investigating, e.g., their consistency as estimators and their rate of convergence [13, 14, 4].

While fast algorithms for isotonic-regression variants have been designed [27], both [22] and [3] list shape constraints beyond monotonicity as important challenges. For example, fitting (multidimensional) convex functions is mostly done via quadratic or linear programming solvers [24]. In his PhD thesis, Balzs writes that current "methods are computationally too expensive for practical use, [so] their analysis is used for the design of a heuristic training algorithm which is empirically evaluated" [4, p. 1].

This lack of efficient algorithms motivated the present work. Despite a few limitations discussed below (implying that we do not yet solve Balzs' problem), we give the first *near-linear-time* algorithms for any function-fitting problem with second-order shape constraints (such as convexity). We use dynamic programming (DP) with a novel geometric encoding for the "states". Simpler versions of such geometric DP variants were used for isotonic regression [25] and are well-known in the competitive programming community; incorporating second-order constraints efficiently is our main innovation.

Problem definition. Given the vectors $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$ and $\boldsymbol{y} \in \mathbb{R}^n$, an error norm d and shape constraints (formalized below), compute $\boldsymbol{f} = (f_1, \ldots, f_n)$ satisfying the shape constraints with minimal $d(\boldsymbol{f}, \boldsymbol{y})$, i.e., we represent f via its values $f_i = f(x_i)$ at the given points. d is usually an L_p norm, $d(\boldsymbol{x}, \boldsymbol{y}) = \left(\sum_i |x_i - y_i|^p\right)^{1/p}$; least squares ($p = 2$) dominate in statistics, but more general error functions have been studied for isotonic regression [23, 19, 22, 3]. We will consider the *weighted* L_∞ norm, i.e., $d(\boldsymbol{f}, \boldsymbol{y}) = \max_{i \in [n]} w_i |f_i - y_i|$, where $[n] = \{1, \ldots, n\}$ and $\boldsymbol{w} \in \mathbb{R}^n_{\geq 0}$ is a given vector of weights.

Since we are dealing with discretized functions (a vector $\boldsymbol{f}$), restrictions for derivatives f' and f'' have to be discretized, as well. We define local slope and curvature as

$$f'_i \;=\; \frac{f_i - f_{i-1}}{x_i - x_{i-1}}, \quad (i \in [2..n]), \quad \text{and} \quad f''_i \;=\; \frac{f'_i - f'_{i-1}}{x_i - x_{i-1}}, \quad (i \in [3..n]);$$

the shape constraints are then given in the form of vectors $\boldsymbol{f'^-}, \boldsymbol{f'^+}, \boldsymbol{f''^-}, \boldsymbol{f''^+}$ of bounds for the first- and second-order differences, i.e., we define the set of feasible answers as $F = \big\{\boldsymbol{f} \in \mathbb{R}^n \;\big|\; \boldsymbol{f'^-} \leq \boldsymbol{f'} \leq \boldsymbol{f'^+} \wedge \boldsymbol{f''^-} \leq \boldsymbol{f''} \leq \boldsymbol{f''^+}\big\}$ where inequalities on vectors mean the inequality on all components. The *weighted-L_∞ function-fitting problem with second-order shape constraints* is then to find

$$\boldsymbol{f^*} \;=\; \operatorname*{arg\,min}_{\boldsymbol{f} \in F} \left(\max_i \, w_i \cdot |f_i - y_i| \right). \tag{1}$$

Often, we only need a lower resp. upper bound; we can achieve that by allowing $-\infty$ and $+\infty$ entries in $f'^{\pm}_i$ and $f''^{\pm}_i$. For example, setting $\boldsymbol{f''^-} = 0$, $\boldsymbol{f'^-} = \boldsymbol{f''^-} = +\infty$ and $\boldsymbol{f'}- = -\infty$, we can enforce a convex function/vector. We also consider the decision-version of the problem: given a bound L, decide if there is an $\boldsymbol{f} \in F$ with $\max_i w_i |f_i - y_i| \leq L$, and if so, report one.

Contributions. Our main result is a *single* $O(n)$-time algorithm for the decision problem of function fitting with second-order constraints; see Theorem 1.2 for the precise statement.

With binary search, this readily yields an additive ε-approximation for (1), and thus weighted L_∞ isotonic regression, convex regression and Lipschitz convex regression, in $O\big(n \log \frac{U}{\varepsilon}\big)$ time (Theorem 1.4), where $U = (\max_i w_i) \cdot (\max_i y_i - \min_i y_i)$. In the appendix, we give a simple greedy algorithm (see Theorem A.1) for *unweighted* ($\boldsymbol{w} = 1$) L_∞ convex regression that runs in $O(n)$ time. Finally, we show that a generalization of the problem to DAGs (where the applied first- and second-order difference constraints are restricted by the graph), is as hard as linear programming, see Appendix D.

Related work. Stout [27] surveys algorithms for various versions of isotonic regression; they achieve near-linear or even linear time for many error metrics. He also considers the generalization to any partial order (instead of the total order corresponding to weakly increasing functions). A related task is to fit a piecewise-constant function (with a prescribed number of jumps) to given data. [9, 10] solve this problem for L_∞ in optimal $O(n \log n)$ time. Since the geometric constraints are much easier than in our case, a simple greedy algorithm suffices to solve the decision version.

For more restricted shapes, less is known. [26] gives a $O(n \log n)$ solution for unimodal regression. [1] gives an $O(n \log n)$ algorithm for unweighted L_2 Lipschitz isotonic regression and a $O(n \operatorname{poly}(\log n))$ time algorithm for Lipschitz unimodal regression. [24] describes (multidimensional) L_2 convex regression algorithms based quadratic programming. Fefferman [8] studied a closely related problem of smooth interpolation of data in Euclidean space minimizing a certain norm defined on the derivatives of the function. His setup is much more general, but his algorithm cannot find arbitrarily good interpolations (ε is fixed for the algorithm). All fast algorithms above consider classes defined by constraints on the *first* derivative only, not the second derivative as needed for convexity. To our knowledge, the fastest prior solution for any convex regression problem is solving a linear program, which will imply super-linear time.

We use a geometric interpretation of dynamic-programming states and represent them implicitly. The work closest in spirit to ours is a recent article by Rote [25]; establishing the transformation of states is much more complicated in the presently studied problem, though. Implicitly representing a series of more complicated objects using data structures has been used in geometric and graph algorithms, such as multiple-source shortest paths [18] and shortest paths in polygons [5, 21, 7]. The only other work (we know of) that interprets dynamic programming geometrically is [28].

There is a rich literature on methods for speeding up dynamic programming [29, 30, 6, 11]. They involve a variety of powerful techniques such as monotonicity of transition points, quadrangle inequalities, and Monge matrix searching [2], many of which have found applications in other settings. The focus of these methods is to reduce the (average) number of transitions that a state is involved in, often from $O(n)$ to $O(1)$. Therefore, their running times are lower bounded by the number of states in the dynamic programs.

1.1. Results

We formally state our theorem for the decision problem here; results for shape-constrained function fitting are obtained as corollaries. For our algorithm, the discrete derivatives (as defined above) are inconvenient because they involve the x-distance between points. We therefore *normalize* all x-distances to 1 (s. t. $x_i = i$); for the second-order constraints, this normalization makes the introduction of an additional parameter necessary, the scaling factors α_i (see below).

Definition 1.1 (1st/2nd-diff-constrained vectors): *Let n-dimensional vectors $\boldsymbol{x}^- \le \boldsymbol{x}^+$ (value bounds), $\boldsymbol{y}^- \le \boldsymbol{y}^+$ (difference bounds), $\boldsymbol{z}^- \le \boldsymbol{z}^+$ (second-order difference bounds), and*

$\boldsymbol{\alpha} > 0$ *be given. We define* $\mathcal{S} \subset \mathbb{R}^n$ *to be the set of all* $\boldsymbol{b} \in \mathbb{R}^n$ *that satisfy the following constraints:*

$$\begin{array}{llr} \forall i \in [1..n] & x_i^- \le b_i \le x_i^+ & \textit{(value constraints)} \\ \forall i \in [2..n] & y_i^- \le b_i - b_{i-1} \le y_i^+ & \textit{(first-order constr.)} \\ \forall i \in [3..n] & z_i^- \le (b_i - b_{i-1}) - \alpha_i(b_{i-1} - b_{i-2}) \le z_i^+ & \textit{(second-order constr.)} \end{array}$$

Moreover, we consider the "truncated problems" $\mathcal{S}_k$*, where* $\mathcal{S}_k$ *is the set of all* $\boldsymbol{b} \in \mathbb{R}^n$ *that satisfy the constraints up to* k *(instead of* n*).*

A visualization of an example is shown in Figure 1. We can encode an instance $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{f'^{\pm}}, \boldsymbol{f''^{\pm}})$ of the decision version of the weighted-L_∞ function-fitting problem with second-order constraints as 1st/2nd-diff-constrained vectors by setting

$$x_i^{\pm} = y_i \pm L/w_i, \qquad y_i^{\pm} = f'^{\pm} \cdot (x_i - x_{i-1}),$$
$$z_i^{\pm} = f''^{\pm} \cdot (x_i - x_{i-1})^2, \qquad \alpha_i = \frac{x_i - x_{i-1}}{x_{i-1} - x_{i-2}}.$$

So, our goal is to efficiently compute some $\boldsymbol{b} \in \mathcal{S}$ or determine that $\mathcal{S} = \emptyset$. Our core technical result is a linear-time algorithm for this task:

Theorem 1.2 (1st/2nd-diff-constrained decision): *With the notation of Definition 1.1, in* $O(n)$ *time, we can compute* $\boldsymbol{b} \in \mathcal{S}$ *or determine that* $\mathcal{S} = \emptyset$*.*

Section 2 will be devoted to the proof. To simplify the presentation, we will assume throughout that $\boldsymbol{x}^+$, $\boldsymbol{x}^-$, $\boldsymbol{y}^+$, $\boldsymbol{y}^-$, $\boldsymbol{z}^+$, $\boldsymbol{z}^-$ are bounded.[1] For the optimization version of the problem, Equation (1), we consider approximate solutions in the following sense.

Definition 1.3 (ε-approximation): *We call* $\boldsymbol{f} \in F$ *an* ε*-approximate solution to the weighted* L_∞ *function-fitting problem if it satisfies*

$$\max_i w_i |f_i - y_i| \le \min_{\boldsymbol{g} \in F} \left(\max_i w_i |g_i - y_i| \right) + \varepsilon.$$

By a simple binary search on L, we can find approximate solutions.

Theorem 1.4 (Main result): *There exists an algorithm that computes an* ε*-approximate solution to the weighted-*L_∞ *convex regression problem that runs in* $O(n \log \frac{U}{\varepsilon})$ *time, where* $U = (\max_i w_i)(\max_i y_i - \min_i y_i)$*. The same holds true for isotonic regression, Lipschitz isotonic regression, convex isotonic regression.*

Proof: We will argue for the case of convex regression here, other cases are similar. Abbreviate $L(\boldsymbol{f}) = \max_i w_i |f_i - y_i|$. For a given L, the decision version of convex regression can be solved in $O(n)$ time using Theorem 1.2. That is, in $O(n)$ time, we can either find $\boldsymbol{f} \in F$ such that $L(\boldsymbol{f}) \le L$ or conclude that for all $\boldsymbol{f} \in F$, $L(\boldsymbol{f}) > L$. If we know an L_0 for which there exists $\boldsymbol{f} \in F$ with $L(\boldsymbol{f}) \le L_0$, then we can do a binary search for L_c in $[0, L_0]$. We can easily find such an L_0 for the convex case: Let $\boldsymbol{f} = \min y_j$ be constant (hence convex). For this $\boldsymbol{f}$, we have $L(\boldsymbol{f}) \le (\max_j w_j)(\max_j y_j - \min_j y_j)$. Therefore, we can take $L_0 = (\max_j w_j)(\max_j y_j - \min_j y_j)$ and the result immediately follows. □

[1] Some problems are stated with $\pm\infty$ values, but we can always replace unbounded values in the algorithms with an (input-specific) sufficiently large finite number.

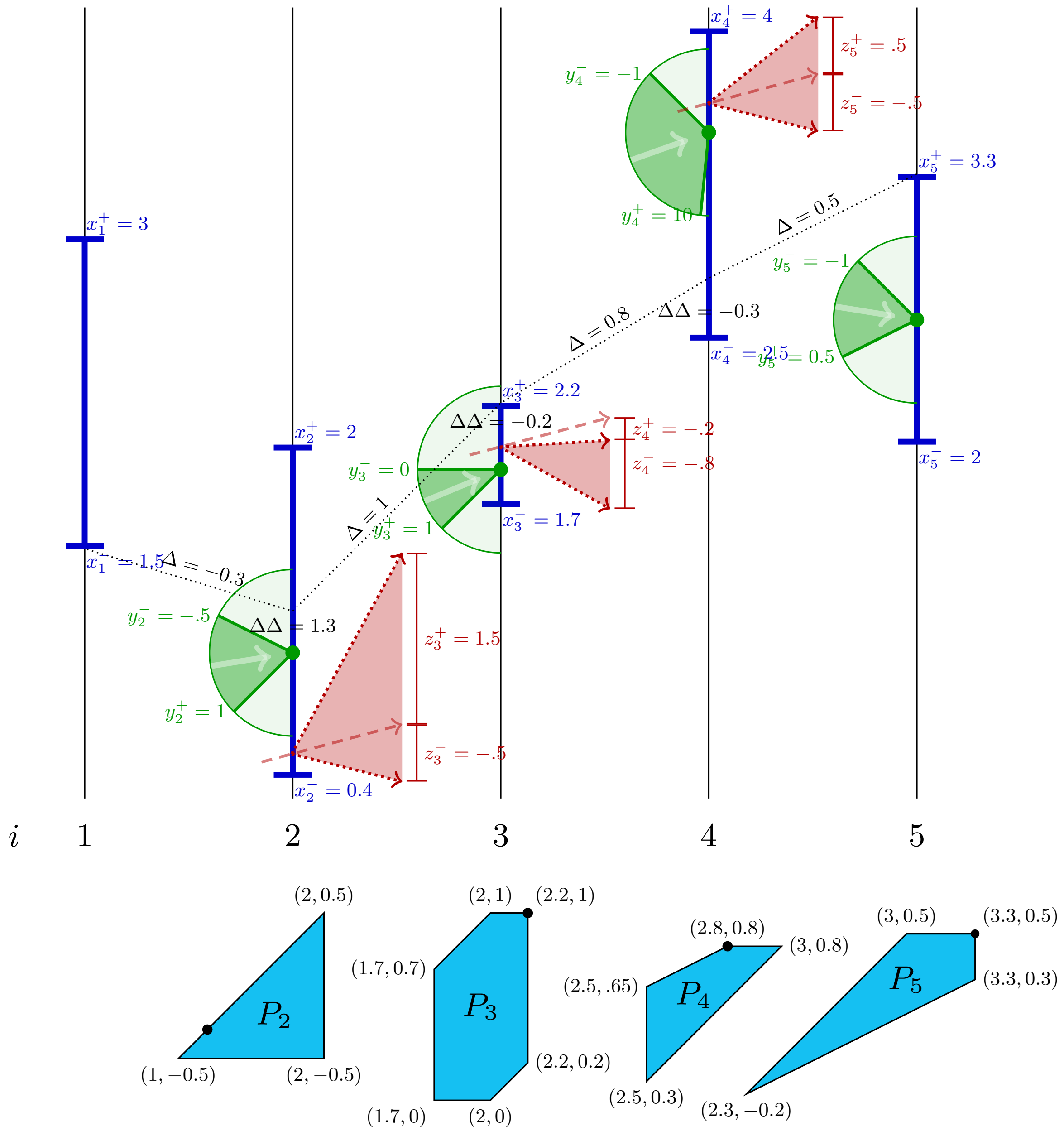

Figure 1: Exemplary input for the 1st/2nd-diff-constrained decision problem with $\boldsymbol{\alpha} = 1$. Value constraints are illustrated as blue bars. First-order constraints are shown as green circles, indicating the allowable incoming angles/slopes; the green dot and the circle can be moved up and down within the blue range. Finally, second-order constraints are given as red triangles, in which the minimal and maximal allowable change in slope is shown (dotted red), based off an exemplary incoming slope (dashed red). The thin dotted line shows $\boldsymbol{b} = (1.7, 1.2, 2.2, 2.8, 3.3) \in \mathcal{S}$.
Below the visualization of the instance, we show the set of pairs $(b_i, b_i - b_{i-1})$ for $\boldsymbol{b} \in \mathcal{S}_i$, i.e., solutions of the truncated problem; the specific solution is shown as a dot. These sets are the *feasibility polygons* P_i (defined in Section 2.1) that play a vital role in our algorithm. Given all P_i, one can easily construct a solution backwards, starting from any point in P_5.

We note that for the specific case of *unweighted* convex function fitting, there is a simpler linear-time greedy algorithm; we give more details on that in Appendix A. This algorithm was the initial motivation for studying this problem and for the geometric approach we use. For more general settings, in particular second-order differences that are allowed to be both positive and negative, the greedy approach does not work; our generic algorithm, however, is almost as simple and efficient.

2. First- and second-order difference-constrained vectors

In this section, we present our main algorithm and prove Theorem 1.2. In Section 2.1, we give an overview and introduce the feasibility polygons P_i. Section 2.2 shows how P_i can be inductively computed from P_{i-1} via a geometric transformation. We finally show how this transformation can be computed efficiently, culminating in the proof of Theorem 1.2, in Section 2.3. Two proofs are deferred to Appendix B and C.

2.1. Overview of the algorithm

Recall that the problem we want to solve, in order to prove Theorem 1.2, is finding a feasible point $\boldsymbol{b}$ in $\mathcal{S}$ from Definition 1.1. Our algorithm will use dynamic programming (DP) where each state is associated with the feasible b_i in the truncated problem. We will iteratively determine all b_i such that b_i is the ith entry of some $\boldsymbol{b} \in \mathcal{S}_i$.

Feasible b_i have to respect the first- and second-order difference constraints. To check those, we also need to know the possible pairs (b_{i-1}, b_{i-2}) of $(i-1)$th and $(i-2)$th entries for some $\boldsymbol{b} \in \mathcal{S}_{i-1}$, so the states have to maintain more information than the b_i alone. It will be instrumental to *rewrite* this pair as $(b_{i-1}, b_{i-1} - b_{i-2})$, the combination of valid values b_{i-1} and valid *slopes* at which we entered b_{i-1} for a solution in $\mathcal{S}_{i-1}$. From that, we can determine the valid slopes at which we can *leave* b_{i-1} using our shape constraints. We thus define the *feasibility polygons*

$$P_i = \big\{(x,y) \bigm| \exists \boldsymbol{b} \in \mathcal{S}_i : x = b_i \wedge y = b_i - b_{i-1}\big\} \qquad (2)$$

for $i = 2, \ldots, n$. See Figure 1 for an example. We view each point in P_i as a "state" in our DP algorithm, and our goal becomes to efficiently compute P_i from P_{i-1}. The key observation is that each P_i is indeed an $O(n)$-vertex convex polygon, and we only need an efficient way to compute the *vertices* of P_i from those of P_{i-1}. This needs a clever representation, though, since all vertices can change when going from P_{i-1} to P_i. A closer look reveals that we can represent the vertex transformations *implicitly*, without actually updating each vertex, and we can combine subsequent transformations into a single one. More specifically, if we consider the boundary of P_{i-1}, the transformation to P_i consists of two steps: (1) a linear transformation for the upper and lower hull of P_{i-1}, and (2) a truncation of the resulting polygon by vertical and horizontal lines (i.e., an intersection of the polygon and a half-plane).

The first step requires a more involved proof and uses that all line segments of P_i have weakly positive slope ("+SLOPED", formally defined below). Implicitly computing the first transformation as we move between P_i is straightforward, only requiring a composition of linear operations (a different one, though, for upper and lower hull). We can apply the cumulative transformation whenever we need to access a vertex.

The second step is conceptually simpler, but more difficult to implement efficiently, as we have to determine where a line cuts the polygon in amortized constant time. For this operation, we separately store the vertices of the upper and lower hull of P_i in two arrays, sorted by increasing x-coordinate; since P_i is +SLOPED, y-values are also increasing. A linear search for intersections has overall $O(n)$ cost since we can charge individual searches to deleted vertices.

Finally, if $P_n \neq \emptyset$, we compute a feasible vector $\boldsymbol{b}$ backwards, starting from any point in P_n. Since we do not explicitly store the P_i, this requires successively "undoing" all operations (going from P_i back to P_{i-1}); see Appendix C for details.

2.2. Transformation from state P_{i-1} to P_i

We first define the structural property "+SLOPED" that our method relies on.

Definition 2.1 (+sloped): *We say a polygon $P \subseteq \mathbb{R}^2$ with vertices $v_1, \ldots, v_k$ is* +SLOPED *if* $\operatorname{slope}(v_i, v_j) \geq 0$ *for all edges (v_i, v_j) of P. Here, the slope between two points $v_1 = (x_1, y_1)$, $v_2 = (x_2, y_2) \in \mathbb{R}^2$ is defined as* $\operatorname{slope}(v_1, v_2) = \frac{y_2 - y_1}{x_2 - x_1}$, *when $x_1 \neq x_2$, and* $\operatorname{slope}(v_1, v_2) = \infty$, *otherwise.*

We will now show that P_i can be computed by applying a simple geometric transformation to P_{i-1}. In passing, we will prove (by induction on i) that all P_i are +SLOPED. For the base case, note that $P_2 = \{(b_2, b_2 - b_1) \mid x_1^- \leq b_1 \leq x_1^+ \wedge x_2^- \leq b_2 \leq x_2^+ \wedge y_2^- \leq b_2 - b_1 \leq y_2^+\}$, which is an intersection of 6 half-planes. The slopes of the defining inequalities are all non-negative or infinite, so P_2 is +SLOPED.

Let us now assume that P_{i-1}, $i \geq 3$, is +SLOPED; we will consider the transformation from P_{i-1} to P_i and show that it preserves this property. We begin by separating the transformation from P_{i-1} to P_i into two main steps.

Step 1: Second-order constraint only. For the first step, we ignore the value and first-order constraints at index i. This will yield a convex polygon, $P_i^{(z)}$, that contains P_i; in Step 2, we will add the other constraints at i to obtain P_i itself.

Definition 2.2 ($\boldsymbol{P_i^{(z)}}$: 2nd-order-only polygons): *For a fixed i, consider the modified problem with $x_i^-, y_i^- = -\infty$ and $x_i^+, y_i^+ = \infty$. Define the second-order-only polygon, $P_i^{(z)}$, as the polygon P_i of this modified problem (considering only the z_i constraints at i).*

The statement of the following lemma is very simple observation, but allows us to compute $P_i^{(z)}$ from P_{i-1} with an explicit geometric construction, (whereas such seemed not obvious for the original feasibility polygons).

Lemma 2.3 ($\boldsymbol{P_i^{(z)}}$: scaled, sheared and shifted $\boldsymbol{P_{i-1}}$):
$P_i^{(z)} = \big\{(x + \alpha_i y + z, \alpha_i y + z) \mid (x, y) \in P_{i-1}, z \in [z_i^-, z_i^+]\big\}$.

Proof: The only constraint at i is $z_i^- \leq (b_i - b_{i-1}) - \alpha_i(b_{i-1} - b_{i-2}) \leq z_i^+$. We rewrite this as (a) a constraint for $b_i - b_{i-1}$, using that $b_{i-1} - b_{i-2}$ is the y-coordinate in P_{i-1}, and (b) a constraint for b_i, using that, additionally, b_{i-1} is the x-coordinate in P_{i-1}. □

Once we have computed this polygon $P_i^{(z)}$, computing P_i is easy: adding the constraints $x_i^- \leq x \leq x_i^+$ and $y_i^- \leq y \leq y_i^+$ requires only cutting $P_i^{(z)}$ with two horizontal and vertical lines. We give a visual representation of the mapping on an example in Figure 2. We break the above mapping into two simpler stages:

Corollary 2.4 ($\boldsymbol{P_i^{(z)}}$: sheared and shifted $\boldsymbol{P_{i-1}^{\alpha_i}}$):
Setting $P_{i-1}^{\alpha_i} = \{(x, \alpha_i y) \mid (x, y) \in P_{i-1}\}$, we have
$P_i^{(z)} = \big\{(x + y + z, y + z) \mid (x, y) \in P_{i-1}^{\alpha_i}, z \in [z_i^-, z_i^+]\big\}$.

We note that scaling the y-coordinate by α_i preserves the +SLOPED-property:

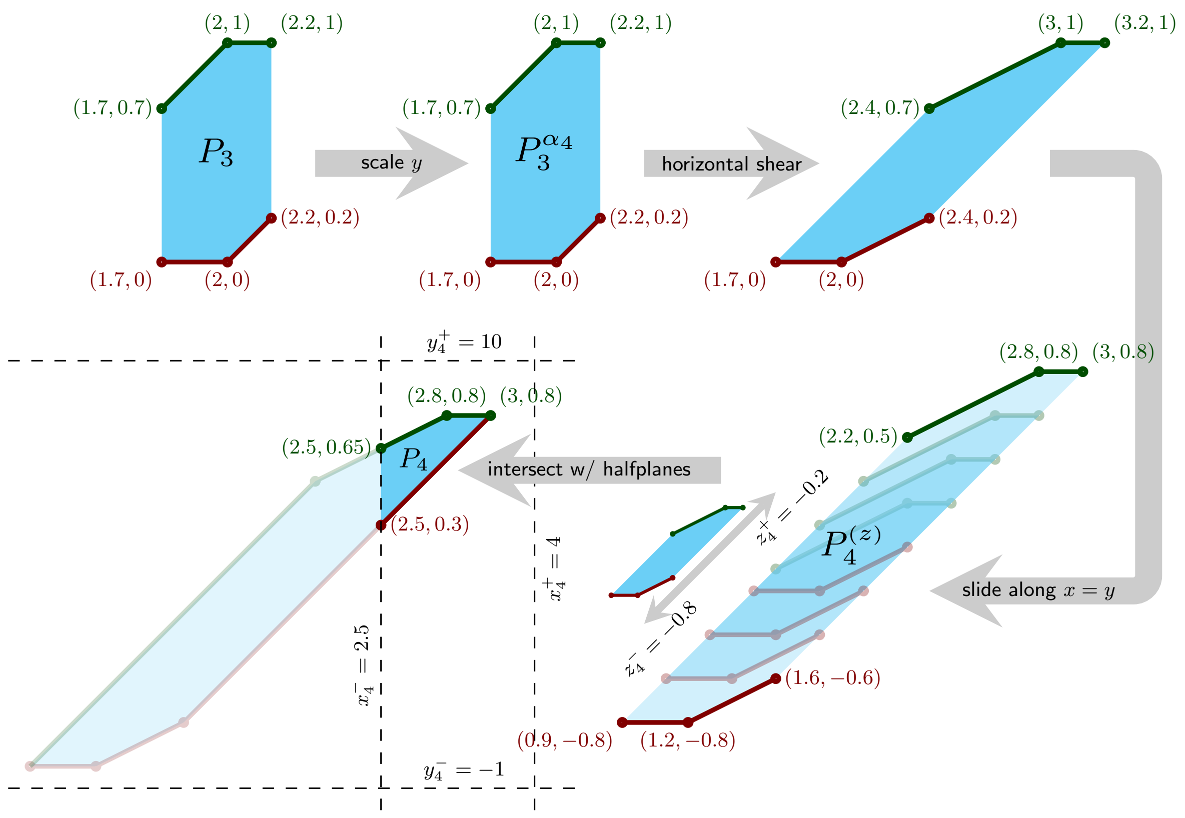

Figure 2: The transformation from P_3 to P_4 for the example instance of Figure 1. Upper and lower hull are shown separately in green resp. red.

Lemma 2.5: *Let $\alpha \geq 0$. If P is* +SLOPED, *so is $P^\alpha = \{(x, \alpha y) \mid (x, y \in P)\}$.*

Proof: Scaling the y-coordinates will preserve all of the vertices of P, and also scale the slope of each vertex pair by $\alpha \geq 0$. So, P^α is +SLOPED. $\square$

That leaves us with the core of the transformation, from $P_{i-1}^{\alpha_i}$ to $P_i^{(z)}$. Intuitively, it can be viewed as sliding $P_{i-1}^{\alpha_i}$ along the line $x = y$ by any amount $z \in [z_i^-, z_i^+]$ and taking the union thereof, (see Figure 2). To compute the result of this operation, we split the boundary into upper and lower hull.

Definition 2.6 (Upper/lower hull): *Let P be a convex polygon with vertex set V. We define the upper hull (vertices) resp. lower hull (vertices) of P as*

$$\begin{aligned}
\text{U-HULL}(P) &= \left\{u_i = (x_i, y_i) \in V \;\middle|\; \nexists (x_i, y) \in P : y > y_i\right\} \\
\text{L-HULL}(P) &= \left\{u_i = (x_i, y_i) \in V \;\middle|\; \nexists (x_i, y) \in P : y < y_i\right\}
\end{aligned}$$

Unless specified otherwise, hull vertices are ordered by increasing x-coordinate.

Note that a vertex can be in both hulls. Moreover, the leftmost vertices in U-HULL(P) and L-HULL(P) always have the same x-coordinate, similarly for the rightmost vertices. As proved in Lemma 2.3, each point in $P_{i-1}^{\alpha_i}$ is mapped to a line-segment with slope 1; we give this mapping a name.

Definition 2.7 (2nd-order P transform): *Let $f_i((x,y))$ be the line-segment $\{(x+y+z, y+z) \mid z \in [z_i^-, z_i^+]\}$ and denote by $f_i^-((x,y)) = (x+y+z_i^-, y+z_i^-)$ and $f_i^+((x,y)) = (x+y+z_i^+, y+z_i^+)$ the two endpoints of $f_i((x,y))$.*

We write $f(S) = \bigcup_{(x,y)\in S} f((x,y))$ for the element-wise application of f to a set S of points.

The vertices of $P_i^{(z)}$ result from transforming the upper hull of $P_{i-1}^{\alpha_i}$ by f_i^+ and the lower hull by f_i^-. The next lemma formally establishes that applying f_i^+ resp. f_i^- to the hulls of $P_{i-1}^{\alpha_i}$ correctly computes $P_i^{(z)}$, (again, compare Figure 2).

Lemma 2.8 (From $P_{i-1}^{\alpha_i}$ to $P_i^{(z)}$ via hulls): *If $P_{i-1}^{\alpha_i}$ is +SLOPED, then $P_i^{(z)}$ is +SLOPED and U-HULL$(P_i^{(z)}) = \{f_i^-(v_{ll})\} \cup f_i^+(\text{U-HULL}(P_{i-1}^{\alpha_i}))$ and L-HULL$(P_i^{(z)}) = f_i^-(\text{L-HULL}(P_{i-1})) \cup \{f_i^+(v_{ur})\}$, where v_{ll} (lower-left) and v_{ur} (upper-right) are the first vertex of L-HULL$(P_{i-1}^{\alpha_i})$ and the last vertex of U-HULL$(P_{i-1}^{\alpha_i})$, respectively.*

We defer the formal proof to Appendix B. Intuitively, since each point in $P_{i-1}^{\alpha_i}$ is mapped to a line-segment with slope 1 in $P_i^{(z)}$, $P_i^{(z)}$ is obtained by sliding $P_{i-1}^{\alpha_i}$ along the line $x = y$. Note here that we could allow $z_i^- = -\infty$ and/or $z_i^+ = \infty$, where the functions f_i^-, f_i^+ would instead map to the ray centered at $(x, x+y)$ and either pointed upwards or downwards with slope 1. The full transformation from P_{i-1} to $P_i^{(z)}$ can now be stated as:

Lemma 2.9 (P_{i-1} to $P_i^{(z)}$): *Let f_i^{*,α_i} be the function $f_i^{*,\alpha_i}(x,y) = (x + \alpha_i y + z_i^*, \alpha_i y + z_i^*)$ for $* \in \{-,+\}$. If P_{i-1} is +SLOPED, then $P_i^{(z)}$ is +SLOPED with*

$$\begin{aligned} \text{U-HULL}(P_i^{(z)}) &= \left\{f_i^{-,\alpha_i}(v_{ll})\right\} \cup f_i^{+,\alpha_i}(\text{U-HULL}(P_{i-1})) \\ \text{L-HULL}(P_i^{(z)}) &= f_i^{-,\alpha_i}(\text{L-HULL}(P_{i-1})) \cup \left\{f_i^{+,\alpha_i}(v_{ur})\right\} \end{aligned}$$

with v_{ll} and v_{ur} the lower-left resp. upper-right vertex of P_{i-1}.

Proof: This follows immediately from Corollary 2.4 and Lemmas 2.5 and 2.8. □

Step 2: Truncating by value and slope. To complete the transformation, we need to add the constraints $x_i^- \le b_i \le x_i^+$ and $y_i^- \le b_i - b_{i-1} \le y_i^+$ to $P_i^{(z)}$. This is equivalent to cutting our polygon with two vertical and horizontal planes. The following lemma shows that this preserves the +SLOPED-property.

Lemma 2.10 (# new vertices): *If P_{i-1} is +SLOPED with k vertices, then P_i is either empty or +SLOPED with at most $k+6$ vertices.*

It follows that over the course of the algorithm, only $O(n)$ vertices are added in total. This will be instrumental for analyzing the running time.

Proof: We know that $P_i^{(z)}$ is +SLOPED, and it follows easily from the definition that cutting by horizontal and vertical planes will preserve this property. Furthermore, note that cutting a convex polygon will increase the total number of vertices by at most one. We added at most 2 vertices to P_{i-1} to obtain $P_i^{(z)}$. We then cut $P_i^{(z)}$ by the inequalities $x \le x_i^+$, $x \ge x_i^-$, $y \le y_i^-$, and $y \ge y_i^+$, i.e., two horizontal and vertical planes. Each adds at most one vertex, giving the desired upper bound. □

2.3. Algorithm

A direct implementation of the transformation of Lemma 2.9 yields a "brute force" algorithm that maintains all vertices of P_i and checks if P_n is empty; (the running time would be quadratic). It works as follows:

1. *[Init]:* Compute the vertices of P_2.
2. *[Compute P_i]:* For $i = 3, \dots, n$, do the following:
   2.1. At step i, scale the y-coordinate of each vertex by α_i.
   2.2. Apply f_i^+ resp. f_i^- to each vertex, depending on which hull it is in.
   2.3. Add the new vertex to U-HULL and L-HULL, as per Lemma 2.9.
   2.4. Delete all the vertices outside $[x_i^-, x_i^+] \times [y_i^-, y_i^+]$ and add the vertices created by intersecting with $[x_i^-, x_i^+] \times [y_i^-, y_i^+]$.
3. *[Compute **b**]:* If $P_n \neq \emptyset$, compute $(b_1, \dots, b_n)$ by backtracing.

Observe that Lemma 2.9 applies the *same* linear function (multiplication of y-coordinate by α_i and f_i^+ or f_i^-) to *all* vertices in U-HULL resp. L-HULL. So, we do not need to modify every vertex each time; instead, we can store – separately for U-HULL and L-HULL – the *composition* of the linear transformations as a matrix. Whenever we access a vertex, we take the unmodified vertex and apply the cumulative transformation in $O(1)$ time.

At each step, after applying the linear transformations, by Lemma 2.9 we also need to copy the leftmost vertex of L-HULL, add it to the left of U-HULL and copy the rightmost vertex of U-HULL and add it to the right of L-HULL. To add these vertices, we simply apply the inverse of each respective cumulative transformation such that all stored vertices require the same transformation. This will also take $O(1)$ time.

Since all the slopes of $P_i^{(z)}$ are non-negative (+SLOPED) and we keep vertices sorted by x-coordinate, the truncation by a horizontal or vertical plane can only remove a prefix or suffix from U-HULL and L-HULL of $P_i^{(z)}$. Depending on the constraint we are adding, ($x \leq x_i^+$, $x \geq x_i^-$, $y \leq y_i^-$, or $y \geq y_i^+$), we start at the rightmost or leftmost vertex of the U-HULL and L-HULL, and continue until we find the intersection with the cutting plane. We remove all visited vertices.

This could take $O(n)$ time in any single iteration, but the total cost over all iterations is $O(n)$ since we start with $O(1)$ vertices and add $O(n)$ vertices throughout the entire procedure (by Lemma 2.10). This allows us to use two deques (double-ended queues), represented as arrays, to store the vertices of U-HULL and L-HULL. Putting this all together gives the linear time algorithm for the decision problem "$\mathcal{S} = \emptyset$?".

To compute an actual solution when $\mathcal{S} \neq \emptyset$, we compute $b_n, \dots, b_1$, in this order. From the last P_n, we can find a feasible b_n (the x-coordinate of any point in P_n). Then, we retrace the steps of our algorithm through specific points in each P_i. Since intermediate P_i were only implicitly represented, we have to recover P_i by "undoing" the algorithm's operations in reverse order; this is possible in overall time $O(n)$ by remembering the operations from the forward phase. The details on the backtracing step are deferred to Appendix C, where we also present the final algorithm.

3. Conclusion

In this article, we presented a linear-time dynamic-programming algorithm to decide whether there is a vector **b** that lies (componentwise) between given upper and lower bounds and additionally satisfies inequalities on its first- and second-order (successive) differences. This method can be used to approximate weighted-L_∞ shape-restricted function-fitting problems, where the shape restrictions are given as bounds on first- and/or second-order differences (local slope and curvature).

This is a first step towards much sought-after efficient methods for more general convex regression tasks. A main limitation of our approach is the restriction to one-dimensional problems. We show in Appendix D that a natural extension of the problem studied here to directed acyclic graphs is already as hard as linear programming, leaving little hope for an efficient generic solution. This is in sharp contrast to isotonic regression, where similar extensions to arbitrary partial orders do have efficient algorithms (for L_∞) [27]. This might also be bad news for multidimensional regression with second-order constraints, since higher dimensions entail, among other complications, a non-total order over the inputs.

A second limitation is the L_∞ error metric, which might not be adequate for all applications. We leave the question whether similarly efficient methods are also possible for other metrics for future work. A further extension to study is convex *unimodal* regression; here, finding the maximum is part of the fitting problem, and so not directly possible with our presented method.

Acknowledgments

We thank Richard Peng, Sushant Sachdeva, and Danny Sleator for insightful discussions, and our anonymous referees for further relevant references and insightful comments that significantly improved the presentation.

References

[1] Pankaj K. Agarwal, Jeff M. Phillips, and Bardia Sadri. Lipschitz unimodal and isotonic regression on paths and trees. In *LATIN 2010: Theoretical Informatics*, pages 384–396. Springer Berlin Heidelberg, 2010. `doi:10.1007/978-3-642-12200-2\_34`.

[2] Alok Aggarwal, Maria M. Klawe, Shlomo Moran, Peter Shor, and Robert Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(1-4):195–208, November 1987. `doi:10.1007/bf01840359`.

[3] Francis Bach. Efficient algorithms for non-convex isotonic regression through submodular optimization. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 1–10. Curran Associates, Inc., 2018.

[4] Gábor Balázs. *Convex Regression: Theory, Practice, and Applications.* PhD thesis, 2016. `doi:10.7939/R3T43J98B`.

[5] Bernard Chazelle. A theorem on polygon cutting with applications. In *Symposium on Foundations of Computer Science (SFCS)*, pages 339–349. IEEE, 1982. `doi:10.1109/SFCS.1982.58`.

[6] D. Eppstein, Z. Galil, and R. Giancarlo. Speeding up dynamic programming. In *Symposium on Foundations of Computer Science (SFCS)*. IEEE, 1988. `doi:10.1109/sfcs.1988.21965`.

[7] Jeff Erickson. Shortest homotopic paths, 2009. Lecture notes for computational topology. URL: `http://jeffe.cs.illinois.edu/teaching/comptop/2009/notes/shortest-homotopic-paths.pdf`.

[8] C. Fefferman. Smooth interpolation of data by efficient algorithms. In *Excursions in Harmonic Analysis, Volume 1*, pages 71–84. Birkhäuser Boston, November 2012. `doi:10.1007/978-0-8176-8376-4\_4`.

[9] Hervé Fournier and Antoine Vigneron. Fitting a step function to a point set. *Algorithmica*, 60(1):95–109, July 2009. `doi:10.1007/s00453-009-9342-z`.

[10] Hervé Fournier and Antoine Vigneron. A deterministic algorithm for fitting a step function to a weighted point-set. *Information Processing Letters*, 113(3):51–54, February 2013. `doi:10.1016/j.ipl.2012.11.003`.

[11] Zvi Galil and Raffaele Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, 64(1):107–118, apr 1989. `doi:10.1016/0304-3975(89)90101-1`.

[12] Ravi Sastry Ganti, Laura Balzano, and Rebecca Willett. Matrix completion under monotonic single index models. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 1873–1881. Curran Associates, Inc., 2015.

[13] Piet Groeneboom and Geurt Jongbloed. *Nonparametric estimation under shape constraints*, volume 38. Cambridge University Press, 2014.

[14] Adityanand Guntuboyina and Bodhisattva Sen. Nonparametric shape-restricted regression. *Statistical Science*, 33(4):568–594, November 2018. `doi:10.1214/18-sts665`.

[15] Alon Itai. Two-commodity flow. *Journal of the ACM (JACM)*, 25(4):596–611, 1978.

[16] Sham M Kakade, Varun Kanade, Ohad Shamir, and Adam Kalai. Efficient learning of generalized linear and single index models with isotonic regression. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 927–935. Curran Associates, Inc., 2011.

[17] Adam Tauman Kalai and Ravi Sastry. The isotron algorithm: High-dimensional isotonic regression. In *Annual Conference on Learning Theory (COLT)*, 2009.

[18] Philip N Klein. Multiple-source shortest paths in planar graphs. In *Symposium on Discrete Algorithms (SODA)*, pages 146–155. SIAM, 2005.

[19] Rasmus Kyng, Anup Rao, and Sushant Sachdeva. Fast, provable algorithms for isotonic regression in all l_p-norms. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2719–2727. Curran Associates, Inc., 2015.

[20] Rasmus Kyng and Peng Zhang. Hardness results for structured linear systems. In *Symposium on Foundations of Computer Science (FOCS)*, pages 684–695, 2017. Available at: https://arxiv.org/abs/1705.02944.

[21] Der-Tsai Lee and Franco P. Preparata. Euclidean shortest paths in the presence of rectilinear barriers. *Networks*, 14(3):393–410, 1984. `doi:10.1002/net.3230140304`.

[22] Cong Han Lim. An efficient pruning algorithm for robust isotonic regression. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 219–229. Curran Associates, Inc., 2018.

[23] Ronny Luss and Saharon Rosset. Generalized isotonic regression. *Journal of Computational and Graphical Statistics*, 23(1):192–210, January 2014. doi:10.1080/10618600.2012.741550.

[24] Rahul Mazumder, Arkopal Choudhury, Garud Iyengar, and Bodhisattva Sen. A computational framework for multivariate convex regression and its variants. *Journal of the American Statistical Association*, pages 1–14, January 2018. doi:10.1080/01621459.2017.1407771.

[25] Günter Rote. Isotonic regression by dynamic programming. In Jeremy T. Fineman and Michael Mitzenmacher, editors, *Symposium on Simplicity in Algorithms (SOSA 2019)*, volume 69 of *OASIcs*, pages 1:1–1:18. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/OASIcs.SOSA.2019.1.

[26] Quentin F. Stout. Unimodal regression via prefix isotonic regression. *Computational Statistics & Data Analysis*, 53(2):289–297, December 2008. doi:10.1016/j.csda.2008.08.005.

[27] Quentin F. Stout. Fastest isotonic regression algorithms, 2014. URL: http://web.eecs.umich.edu/~qstout/IsoRegAlg.pdf.

[28] Charalampos E. Tsourakakis, Richard Peng, Maria A. Tsiarli, Gary L. Miller, and Russell Schwartz. Approximation algorithms for speeding up dynamic programming and denoising aCGH data. *Journal of Experimental Algorithmics*, 16:1.1, May 2011. doi:10.1145/1963190.2063517.

[29] F. Frances Yao. Efficient dynamic programming using quadrangle inequalities. In *Symposium on Theory of Computing (STOC)*. ACM Press, 1980. doi:10.1145/800141.804691.

[30] F. Frances Yao. Speed-up in dynamic programming. *SIAM Journal on Algebraic Discrete Methods*, 3(4):532–540, December 1982. doi:10.1137/0603055.

Appendix

A. Simple greedy algorithm for convex regression

In this appendix, we give details on a simpler algorithm for the special case of unweighted convex function fitting.

Theorem A.1: *There exists an algorithm for the unweighted L_∞ convex regression that runs in $O(n)$ time.*

Proof: We consider the following problem. Given an n-dimensional vector $\boldsymbol{a}$, and parameter $\Delta \geq 0$, find a convex vector $\boldsymbol{b}$ such that $\|b - a\|_\infty \leq \Delta$, if such a vector exists.

This clearly fits under our parameters of Definition 1.1 by setting $\boldsymbol{x}^- = \boldsymbol{a} - \Delta, \boldsymbol{x}^+ = \boldsymbol{a} + \Delta$, both $\boldsymbol{y}^-$ and $\boldsymbol{y}^+$ to be unbounded, and $\boldsymbol{z}^- = 0, \boldsymbol{z}^+ = \infty$, along with $\boldsymbol{\alpha} = 1$. A binary search on Δ gives a $O(n \log \frac{U}{\varepsilon})$ algorithm.

However, this can also be solved by considering the set of points $(i, a_i + \Delta)$ for all i, and taking the lower hull,[2] $H(\Delta)$, such that for each point (i, h_i) in this lower hull we set $b_i = h_i$. We claim that the minimum possible Δ such that $b_i \geq a_i - \Delta$ is exactly the answer to this problem. If $(i, a_i + \Delta)$ is a vertex of the convex hull, $b_i = a_i + \Delta$ is always at least $a_i - \Delta$. Otherwise, let $(j, a_j + \Delta)$, $(k, a_k + \Delta)$ be two vertices of H such that $j < i < k$. We have

$$\begin{aligned} b_i &\geq a_i - \Delta \\ \iff a_j + \Delta + \frac{a_k - a_j}{k - j}(i - j) &\geq a_i - \Delta \\ \iff \Delta &\geq \frac{1}{2}\left(a_i - a_j + \frac{i - j}{k - j}(a_k - a_j)\right) \end{aligned}$$

If Δ violates this for some i, j, k, then it is impossible to fit a convex function through the intervals $[(j, a_j - \Delta), (j, a_j + \Delta)]$, $[(i, a_i - \Delta), (i, a_i + \Delta)]$, and $[(k, a_k - \Delta), (k, a_k + \Delta)]$.

Conversely, if Δ satisfies all of such constraints, $b_i \geq a_i - \Delta$ for all $1 \leq i \leq n$, then b_i cannot be greater than $a_i + \Delta$ as that would violate H being the convex lower hull of $(i, a_i + \Delta)$. Thus, $(b_1, \ldots, b_n)$ is a possible solution.

It takes $O(n)$ time to compute the lower convex hull and $O(n)$ time to calculate the minimum Δ. Thus, this algorithm solves L_∞ convex regression in $O(n)$ time. $\square$

The above method can also be adapted for inputs with x-values that are non-uniformly spaced. However, it does not directly generalize to weighted L_∞ regression: moving points up by $w_i \cdot \Delta$ can lead to different lower hulls for different values of Δ.

[2] The lower hull of a set of points is the subset of vertices (x_i, y_i) of the convex hull, where y_i is the minimal y-coordinate of all points with the x-coordinate x_i in the convex hull; see also Definition 2.6.

B. Proof of Lemma 2.8

The proof of Lemma 2.8 will be separated into two stages. First, we show that the polygon defined by $\{f_i^+(\text{U-HULL}(P_{i-1}^{\alpha_i}))\} \cup \{f_i^-(\text{L-HULL}(P_{i-1}^{\alpha_i}))\}$ has upper-hull $\{f_i^-(v_{ll}), f_i^+(\text{U-HULL}(P_{i-1}^{\alpha_i}))\}$ and lower-hull $\{f_i^-(\text{L-HULL}(P_{i-1}^{\alpha_i})), f_i^+(v_{ur})\}$, where v_{ll} is the first vertex of $\text{L-HULL}(P_{i-1}^{\alpha_i})$ and v_{ur} is the last vertex of $\text{U-HULL}(P_{i-1}^{\alpha_i})$. Furthermore, this polygon will have slopes between vertices in $[0,1]$. This property will then allow us to show that $P_i^{(z)}$ is equivalent to the convex hull of the vertices, which implies the claim.

In order to show that the $P_i^{(z)}$ has all slopes between 0 and 1, we consider how f_i^- and f_i^+ affect slopes.

Lemma B.1 (Bounded slopes): *If P is* +SLOPED*, then for any connected vertices $v_j, v_k \in V$, any i, and $* \in \{-,+\}$, we have*

$$0 \leq \mathit{slope}(f_i^*(v_j), f_i^*(v_k)) \leq 1$$

and for any connected vertices $v_j, v_k, v_l \in V$, if $\mathit{slope}(v_j, v_k) < \mathit{slope}(v_k, v_l)$, then

$$\mathit{slope}(f_i^*(v_j), f_i^*(v_k)) < \mathit{slope}(f_i^*(v_k), f_i^*(v_l))$$

Proof: We first write the slope function explicitly to obtain

$$\text{slope}(f_i^*(v_j), f_i^*(v_k)) \;=\; \frac{(y_j + z_i^*) - (y_k + z_i^*)}{(x_j + y_j + z_i^*) - (x_k + y_k + z_i^*)} \;=\; \frac{y_j - y_k}{(x_j - x_k) + (y_j - y_k)}.$$

This implies that if $\text{slope}(v_j, v_k) = \infty$ then $\text{slope}(f_i^*(v_j), f_i^*(v_k)) = 1$, and if $\text{slope}(v_j, v_k) = 0$ then $\text{slope}(f_i^*(v_j), f_i^*(v_k)) = 0$. Furthermore, this gives the identity

$$\text{slope}(f_i^*(v_j), f_i^*(v_k))^{-1} = \text{slope}(v_j, v_k)^{-1} + 1$$

when $\text{slope}(v_j, v_k) \in (0, \infty)$. Combined with the fact that all slopes are non-negative, this gives both of our desired inequalities. □

The first inequality of the lemma above will allow us to show that all of the slopes between vertices are bounded, and the second implies that each of the vertices remains a vertex, giving the following corollary.

Corollary B.2 (Hulls by elementwise transformation):
If $P_{i-1}^{\alpha_i}$ is +SLOPED*, then the convex hull P of $V = f_i^+(\text{U-HULL}(P_{i-1}^{\alpha_i})) \cup f_i^-(\text{L-HULL}(P_{i-1}^{\alpha_i}))$ has $\text{U-HULL}(P) = \{f_i^-(v_{ll}), f_i^+(\text{U-HULL}(P_{i-1}^{\alpha_i}))\}$ and $\text{L-HULL}(P) = \{f_i^-(\text{L-HULL}(P_{i-1}^{\alpha_i})), f_i^+(v_{ur})\}$, where v_{ll} is the first (lower-left) vertex of $\text{L-HULL}(P_{i-1}^{\alpha_i})$ and v_{ur} is the last (upper-right) vertex of $\text{U-HULL}(P_{i-1}^{\alpha_i})$. Furthermore, for any connected vertices v_j, v_k in P, we have $0 \leq \mathit{slope}(v_j, v_k) \leq 1$.*

Proof: By construction, the first and last vertices of $\text{U-HULL}(P)$ and $\text{L-HULL}(P)$ are the same. Let v_{u1} be the first vertex of $\text{U-HULL}(P_{i-1}^{\alpha_i})$, which gives two possibilities, either (1): $v_{u1} = v_{ll}$, or (2) $\text{slope}(v_{u1}, v_{ll}) = \infty$. For case (1) it is easy to see that $\text{slope}(f_i^+(v_{u1}), f_i^-(v_{ll})) = 1$, and for case (2), we showed in the proof of Lemma B.1 that $\text{slope}(v_{u1}, v_{ll}) = \infty$ implies $\text{slope}(f_i^+(v_{u1}), f_i^+(v_{ll})) = 1$, which combined with $\text{slope}(f_i^+(v_{ll}), f_i^-(v_{ll})) = 1$ gives $\text{slope}(f_i^+(v_{u1}), f_i^-(v_{ll})) = 1$. Furthermore the slopes between all vertices in $\text{U-HULL}(P_{i-1}^{\alpha_i})$ are less than ∞ by Definition 2.6, and therefore less than 1 under the transformation by Lemma B.1. Along with the second inequality of Lemma B.1, this implies that $\text{U-HULL}(P)$ makes up a concave function from $f_i^-(v_{ll})$ to $f_i^+(v_{ur})$.

By symmetric reasoning we see that L-HULL(P) makes up a convex function from $f_i^-(v_{ll})$ to $f_i^+(v_{ur})$. Additionally, the second inequality states that every element in $\{f_i^-(\text{L-HULL}(P_{i-1}^{\alpha_i})), f_i^+(v_{ur})\}$ and $\{f_i^-(v_{ll}), f_i^+(\text{U-HULL}(P_{i-1}^{\alpha_i}))\}$ must be a vertex. Accordingly, P must be a convex polygon with all slopes between 0 and 1. $\square$

We now have fixed upper and lower hulls of a polygon, and we use the representation as the convex hull its vertices, along with the bounded-slope property, to show that this polygon is in fact equal to $P_i^{(z)}$. In particular, all the slopes being bounded by 1 will be critical here because each point $(x, y) \in P_{i-1}^{\alpha_i}$ maps to a line segment from $(x+y+z_i^-, y+z_i^-)$ to $(x+y+z_i^+, y+z_i^+)$, which has slope 1. If we then consider (x, y) to be in the upper hull, if the slopes of our new upper-hull for $P_i^{(z)}$ were greater than 1, the point $(x+y+z_i^-, y+z_i^-)$ would lie outside of this hull. Our bounded slopes prevent this, though, and lead to the following lemma.

Lemma B.3: *Let $P_{i-1}^{\alpha_i}$ be* +SLOPED *and let P be the convex hull of*

$$V = \left\{f_i^+(\text{U-HULL}(P_{i-1}^{\alpha_i}))\right\} \cup \left\{f_i^-(\text{L-HULL}(P_{i-1}^{\alpha_i}))\right\}.$$

Then $P = P_i^{(z)}$.

Proof: We show both inclusions.

- $P \subseteq P_i^{(z)}$.
  By definition of P, any point $u \in P$, can be written as a convex combination
  $$\sum_{(x_j,y_j)\in V(P_{i-1}^{\alpha_i})} p_j((x_j+y_j, y_j) + (z_i^*, z_i^*)),$$
  where the sum is over the vertices (x_j, y_j) of $P_{i-1}^{\alpha_i}$, $* \in \{-, +\}$, and $\sum p_j = 1$. We set $z = \sum p_j z_i^*$; clearly, $z \in [z_i^-, z_i^+]$. Furthermore set $x = \sum p_j x_i$ and $y = \sum p_j y_j$. We know each (x_j, y_j) is a vertex in $P_{i-1}^{\alpha_i}$, so by convexity (x, y) must be in $P_{i-1}^{\alpha_i}$, implying $(x+y+x, y+z) \in P_i^{(z)}$ by Corollary 2.4.

- $P_i^{(z)} \subseteq P$.
  Assume towards a contradiction there were $(x+y+z, y+z) \in P_i^{(z)}$ with $(x, y) \in P_{i-1}^{\alpha_i}$ and $z \in [z_i^-, z_i^+]$, but $(x+y+z, y+z) \notin P$. By definition and assumption, both P and $P_{i-1}^{\alpha_i}$ are convex, so there must be a *vertex* (x_v, y_v) of $P_{i-1}^{\alpha_i}$ such that $(x_v+y_v+z, y_v+z) \notin P$. Furthermore, by convexity of P, there must also exist $z \in \{z_i^-, z_i^+\}$ such that $(x_v+y_v+z, y_v+z) \notin P$. Assume w.l.o.g. that $(x_v, y_v) \in \text{U-HULL}(P_{i-1}^{\alpha_i})$. By definition of P, we have $(x_v+y_v+z_i^+, y_v+z_i^+) \in P$, so we must have $z = z_i^-$.

  Since $P_{i-1}^{\alpha_i}$ is +SLOPED and f_i is monotone, $f_i^-(v_{ll})$ is dominated[3] by $(x_v+y_v+z_i^-, y_v+z_i^-)$, and similarly, $f_i^+(v_{ur})$ dominates $(x_v+y_v+z_i^-, y_v+z_i^-)$. Furthermore, by Corollary B.2 the upper hull lies above the line segment from from $f_i^-(v_{ll})$ to $(x_v+y_v+z_i^+, y_v+z_i^+)$ and has slope at most 1. But the slope between $(x_v+y_v+z_i^-, y_v+z_i^-)$ and $(x_v+y_v+z_i^+, y_v+z_i^+)$ is exactly 1, so $(x_v+y_v+z_i^-, y_v+z_i^-)$ cannot lie above the upper hull.

  Finally, $(x_v+y_v+z_i^-, y_v+z_i^-)$ also cannot lie below L-HULL(P) because otherwise there would exist $(x_v, y) \in P_{i-1}^{\alpha_i}$ that lies above (x_v, y_v), contradicting (x_v, y_v) being in U-HULL$(P_{i-1}^{\alpha_i})$. Because the upper hull and lower hull combine to the convex polygon P and because the x-coordinate of $(x_v+y_v+z_i^-, y_v+z_i^-)$ is within the range of x-coordinate of P, we have $(x_v+y_v+z_i^-, y_v+z_i^-) \in P$, a contradiction. $\square$

With this, we finish the proof of our lemma.

Proof of Lemma 2.8: Follows directly from Corollary B.2 and Lemma B.3. $\square$

[3](x_1, y_1) is said to dominate (x_2, y_2) if $x_1 \geq x_2$ and $y_1 \geq y_2$.

C. Complete algorithm

In this appendix, we give detailed pseudocode for our entire algorithm. We also discuss the details on the backtracing step, i.e., computing an actual solution $\boldsymbol{b} \in \mathcal{S}$ from the (implicitly represented) feasibility polygons $P_2, \ldots, P_n$. The final procedure is shown in Algorithm 1.

C.1. Implicitly computing the P_i

The main ideas have been described in Section 2.3. We represent points in homogeneous coordinates, i.e., (x, y) becomes the column vector $(x, y, 1)^T$. That allows our transformation to be represented as a single matrix, and we can compose them by multiplying the matrices. We store the current matrix in Algorithm 1 in S_u (for the upper hull) and S_v for the lower hull. u and v denote the deques storing the (untransformed) points of U-HULL and L-HULL in homogeneous coordinates and in sorted order.

To compute P_i from P_{i-1} (Step 2), we update the transformation matrices and add the new points to the hull (following Lemma 2.9). After that (line 9), u and v represent $P_i^{(z)}$. To implement the intersection with the half planes corresponding to the value and first-order constraints at i, we separately cut upper and lower hull with all four boundaries. Since we store upper and lower hull separately, vertical line segments are not explicitly represented in either hull, which requires some care in cutting with horizontal lines. We therefore use the following strategy –it is illustrated on an example in Figure 3: We first cut with the left and right boundaries (the value constraints), then transform our representation temporarily to left and right hulls (lines 17–18), which can easily handle cutting by horizontal line segments. Cutting is always implemented as a linear scan of u resp. v, during which all vertices outside the constraint halfplane are removed. Then we add a new vertex at the intersection of the last segment with the constraint. (We remember the last removed vertex r for doing so.)

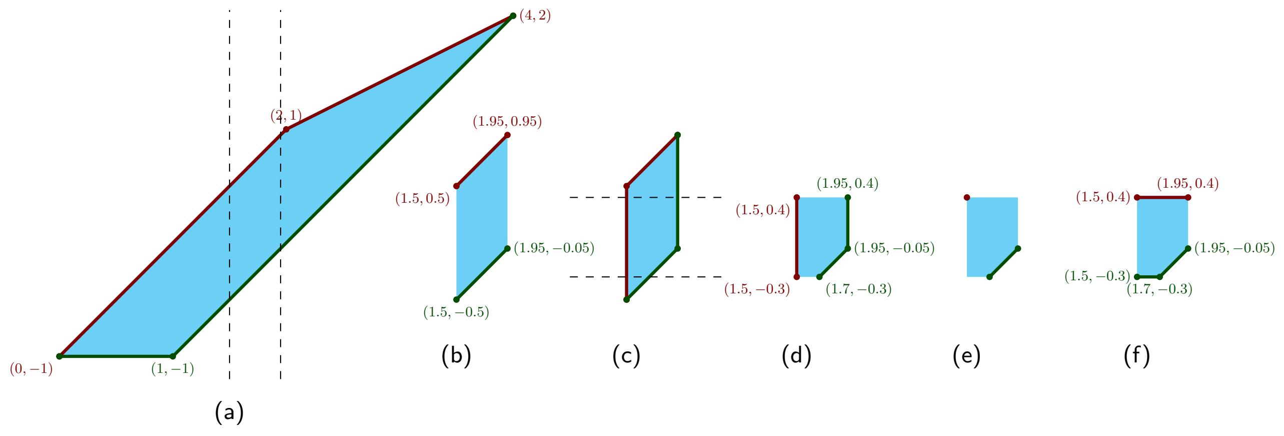

Figure 3: Example for lines 10–30 of Algorithm 1. (a) The polygon $P_i^{(z)}$ (after line 9). (b) After vertical cuts at $x_i^- = 1.5$ and $x_i^+ = 1.95$. (c) After adding the vertical line segments (line 18). (d) After horizontal cuts at $y_i^- = -0.3$ and $y_i^+ = -0.4$ (line 26); u and v represent left and right hull of the correct polygon now, but we have to transform them back to upper and lower hull. For that, we store the LL and UR vertices. (e) After deletion of vertices with same x-coordinate (line 28); neither vertical, nor horizontal line segments are represented. (f) After adding the stored LL and UR vertices (line 30), we obtain the final upper and lower hulls.

Algorithm 1: 1st/2nd-Diff-Constrained Decision Algorithm

```
Input: Vectors $\boldsymbol{x}^- \leq \boldsymbol{x}^+$, $\boldsymbol{y}^- \leq \boldsymbol{y}^+$, $\boldsymbol{z}^- \leq \boldsymbol{z}^+$, $\boldsymbol{\alpha} \geq 0$
Output: Some $\boldsymbol{b} \in \mathcal{S}$, or infeasible if $\mathcal{S} = \emptyset$.
Note: We represent vertex $(x, y)$ by real vector $(x, y, 1)^T$,   (homogeneous coordinates).
[Step 1: Init]
$u \leftarrow$ deque with vertices of upper hull of $P_2$ (sorted by $x$-coordinates);
$v \leftarrow$ deque with vertices of lower hull of $P_2$ (sorted by $x$-coordinates);
$S_u \leftarrow I_3$;  $S_v \leftarrow I_3$ ;        /* init maps to the identity matrix $I_3$ in $\mathbb{R}^{3\times 3}$ */
[Step 2: Compute $P_i$]
for $i \leftarrow 3$ to $n$ do
    $S_u \leftarrow \begin{pmatrix} 1 & \alpha_i & z_i^+ \\ 0 & \alpha_i & z_i^+ \\ 0 & 0 & 1 \end{pmatrix} \cdot S_u$;  $S_v \leftarrow \begin{pmatrix} 1 & \alpha_i & z_i^- \\ 0 & \alpha_i & z_i^- \\ 0 & 0 & 1 \end{pmatrix} \cdot S_v$;        /* Update maps */
    /* Add new LL / UR vertex to hulls after transformation */
    $u.push\_front\big((S_u)^{-1} \cdot S_v \cdot v.front()\big)$;  $v.push\_back\big((S_v)^{-1} \cdot S_u \cdot u.back()\big)$;
    for $c \in \{u, v\}$ do /* Cut left and right boundary */
        $r \leftarrow null$; while $c.size() \geq 1 \wedge (S_c \cdot c.front())_x < x_i^-$ do $r \leftarrow S_c \cdot c.pop\_front()$;
        if $c.empty()$ then return infeasible;
        if $r \neq null$ then $\{q \leftarrow S_c \cdot c.front();\ c.push\_front\big((S_c)^{-1} \cdot \big(q + \frac{q_x - x_i^-}{q_x - r_x} \cdot (r - q)\big)\big)\}$;
        $r \leftarrow null$; while $c.size() \geq 1 \wedge (S_c \cdot c.back())_x > x_i^+$ do $r \leftarrow S_c \cdot c.pop\_back()$;
        if $c.empty()$ then return infeasible;
        if $r \neq null$ then $\{q \leftarrow S_c \cdot c.front();\ c.push\_front\big((S_c)^{-1} \cdot \big(q + \frac{x_i^+ - q_x}{r_x - q_x} \cdot (r - q)\big)\big)\}$;
    /* Temporarily add vertices for vertical line segments (simplifies cuts) */
    if $(S_u \cdot u.front())_y > (S_v \cdot v.front())_y$ then $u.push\_front\big((S_u)^{-1} \cdot S_v \cdot v.front()\big)$;
    if $(S_u \cdot u.back())_y > (S_v \cdot v.back())_y$ then $v.push\_back\big((S_v)^{-1} \cdot S_u \cdot u.back()\big)$;
    for $c \in \{u, v\}$ do /* Cut upper and lower boundary */
        $r \leftarrow null$; while $c.size() \geq 1 \wedge (S_c \cdot c.front())_y < y_i^-$ do $r \leftarrow S_c \cdot c.pop\_front()$;
        if $c.empty()$ then return infeasible;
        if $r \neq null$ then $\{q \leftarrow S_c \cdot c.front();\ c.push\_front\big((S_c)^{-1} \cdot \big(q + \frac{q_y - y_i^-}{q_y - r_y} \cdot (r - q)\big)\big)\}$;
        $r \leftarrow null$; while $c.size() \geq 1 \wedge (S_c \cdot c.back())_y > y_i^+$ do $r \leftarrow S_c \cdot c.pop\_back()$;
        if $c.empty()$ then return infeasible;
        if $r \neq null$ then $\{q \leftarrow S_c \cdot c.front();\ c.push\_front\big((S_c)^{-1} \cdot \big(q + \frac{y_i^+ - q_y}{r_y - q_y} \cdot (r - q)\big)\big)\}$;
        $ll_c \leftarrow S_c \cdot c.front()$;  $ur_c \leftarrow S_c \cdot c.back()$;      /* Store current LL/UR for later */
        /* Remove generated duplicate nodes and vertical segments */
        /* (sndFront/sndBack denote the second / second-to-last elements) */
        while $c.size() \geq 2 \wedge (S_c \cdot c.front())_x = (S_c \cdot c.sndFront())_x$ do $c.pop\_front()$;
        while $c.size() \geq 2 \wedge (S_c \cdot c.back())_x = (S_c \cdot c.sndBack())_x$ do $c.pop\_back()$;
    /* Add stored LL/UR vertices if horizontal segments missing */
    if $(S_v \cdot v.front())_x > (S_u \cdot u.front())_x$ then $v.push\_front\big((S_v)^{-1} \cdot ll_u\big)$;
    if $(S_u \cdot u.back())_x < (S_v \cdot v.back())_x$ then $u.push\_back\big((S_u)^{-1} \cdot ur_v\big)$;
[Step 3: Compute b]
$(x, y) \leftarrow S_u \cdot u.back()$;  $b_n \leftarrow x$;  $p \leftarrow$ index of the last element of $u$;
for $i \leftarrow n$ to 3 do
    Revert $u$, $v$, $S_u$, $S_v$ to the previous stage;
    $x' \leftarrow x - y$;
    while $p_x < x'$ do $p$++;
    Use $u_p$ and $u_{p-1}$ (if exists) to compute $y_m \leftarrow \max\{y' \mid (x', y') \in P_{i-1}\}$;
    if $y \geq \alpha_i y_m + z_i^-$ then $(x, y) \leftarrow (x', y_m)$ else $(x, y) \leftarrow (x', (y - z_i^-)/\alpha_i)$;
    $b_{i-1} \leftarrow x$;
$b_1 \leftarrow x - y$;
return $(b_1, \ldots, b_n)$;
```

C.2. Backtracing

Suppose we have computed P_n as described above, and then partially backtraced through a sequence of feasible points. We are now at $(b_{i+1}, b_{i+1} - b_i)$ in P_{i+1}. Since $(b_{i+1}, b_{i+1} - b_i) = (x + y + z, y + z)$, $z \in [z^-_{i+1}, z^+_{i+1}]$ for some (unknown) $(x, y) = (b_i, \alpha_{i+1}(b_i - b_{i-1})) \in P_i$, we can recover $x = b_i$ from $(b_{i+1}, b_{i+1} - b_i)$ by subtracting the two coordinates of $(b_{i+1}, b_{i+1} - b_i)$. To recover y, suppose we can find $y_{\max} = \max\{y \mid (b_i, y) \in P_i^{\alpha_{i+1}}\}$ efficiently. Since $\{y \mid (b_i, y) \in P_i^{\alpha_{i+1}}\}$ is an interval, the following lemma allow us to find $b_i - b_{i-1}$.

Lemma C.1 (back 1 step): *Let* $f_{i+1}(x, y) = \{(x + y + z, y + z) \mid z \in [z^-_{i+1}, z^+_{i+1}]\}$. *Either* $(b_{i+1}, b_{i+1} - b_i) \in f_{i+1}((b_i, y_{\max}))$ *or* $(b_{i+1}, b_{i+1} - b_i) = (b_i + y + z^-_{i+1}, y + z^-_{i+1})$ *for some* $y < y_{\max}$.

Intuitively, a vertical line segment L inside P_i is mapped to a line-segment with slope 1 in P_{i+1}, because the line segments the points in L are mapped to lie all on the same line (overlapping with each other).

Proof: If $(b_{i+1}, b_{i+1} - b_i) \notin f_{i+1}(b_i, y_{\max})$, by the maximality of $y_{\max}$, $b_{i+1} - b_i < y_{\max} + z^-_{i+1}$. Since there exists (b_i, y') such that $(b_{i+1}, b_{i+1} - b_i) \in f_{i+1}(b_i, y')$, $(b_i + y' + z, y' + z) = (b_{i+1}, b_{i+1} - b_i)$ for some $z \in [z^-_{i+1}, z^+_{i+1}]$. Consider $f_{i+1}(b_i, y + z - z^-_{i+1})$. Then $(b_{i+1}, b_{i+1} - b_i) = (b_i + (y' + z - z^-_{i+1}) + z^-_{i+1}, (y' + z - z^-_{i+1}) + z^-_{i+1})$. Since $b_{i+1} - b_i < y_{\max} + z^-_{i+1}$, $y' + z - z^-_{i+1} < y_{\max}$. The lemma is proven by letting y be $y' + z - z^-_{i+1}$. □

In the former case of Lemma C.1, we can take $(x, y_{\max})$ as $(b_i, \alpha_{i+1}(b_i - b_{i-1}))$. In the latter case, we can take $(b_i, (b_{i+1} - b_i) - z^-_{i+1})$ as $(b_i, \alpha_{i+1}(b_i - b_{i-1}))$.

Since $y_{\max}$ is the y-coordinate of the intersection of U-HULL(P_i) and the vertical line $(b_i, \cdot)$, to compute $y_{\max}$, we want to find two vertices in U-HULL(P_i), (x_l, y_l) and (x_r, y_r), such that $x_l \le b_i \le x_r$. $(b_i, y_{\max})$ is just the intersection of the line segment between (x_l, y_l) and (x_r, y_r) and the vertical line $(b_i, \cdot)$. The following lemma shows how to find (x_l, y_l) and (x_r, y_r) efficiently using an amortized constant-time algorithm.

Lemma C.2 (Computing $y_{\max}$): *Suppose* (x_l, y_l) *and* (x_r, y_r) *are two vertices in* U-HULL(P_i), *and some point* $(b_i, y) \in P_i$ *satisfies* $x_l \le b_i \le x_r$. *Let* (x', y') *be some point in* P_{i+1} *with* $(x', y') \in f_{i+1}(b_i, \alpha_{i+1} y)$. *Then* $x' \le (f^+_{i+1}(x_r, \alpha_{i+1} y_r))_x$, *where* $(\cdot)_x$ *means taking the* x*-coordinate of a point and* $(\cdot)_y$ *takes the* y*-coordinate.*

Proof: Assume towards a contradiction that $x' > (f^+_{i+1}(x_r, \alpha_{i+1} y_r))_x$. Since $x' - y' = b_i \le x_r = (f^+_{i+1}(x_r, \alpha_{i+1} y_r))_x - (f^+_{i+1}(x_r, \alpha_{i+1} y_r))_y$, we have $y' > (f^+_{i+1}(x_r, \alpha_{i+1} y_r))_y$. But $y' = \alpha_{i+1} y + k \le \alpha_{i+1} y + z^+_{i+1} \le \alpha_{i+1} y_r + z^+_{i+1} = (f^+_{i+1}(x_r, \alpha_{i+1} y_r))_y$. Contradiction. □

The amortized constant-time algorithm to retrieve $(b_n, \ldots, b_1)$ depends on the implementation of the deques. Since we will add n vertices to the deques during the whole algorithm, the (textbook) fixed-size array-based implementation suffices; we recall it to fix notation. A deque d is represented by array A and two indices p_l, p_r. p_l is the index of the first element of d and p_r is the index of the last element. If we want to add an element e to the left of the deque, the two operations $p_l \leftarrow p_l - 1$, $A[p_l] = e$ suffice. Similarly, we can add/pop elements from left/right. During our algorithm, p_l (resp. p_r) can move to the left (resp. right) by at most n positions, so A can be an array of length $2n + O(1)$. If we store the vertices of P_2 in the middle of A initially, we never exceed the boundaries of A when running the algorithm.

Definition C.3 (Position): *We define* $pos_i(x')$ *as the smallest index (in the array representing deque* u*) of a vertex of* U-HULL$(P_i(\cdot))$ *with* x*-coordinate at least* x'.

Note that adding or removing elements does not change the vertex at a given index (unless that vertex itself is removed).

Lemma C.4 (Monotonicity of positions):
$pos_i(b_i) \geq pos_{i+1}(x')$ for some $(x', y') \in f_{i+1}(b_i, \alpha_{i+1} y)$.

Proof: By Lemma C.2, $x' \leq (f^+_{i+1}(x_r, \alpha_{i+1} y_r))_x$. So $f^+_{i+1}(x_r, \alpha_{i+1} y_r)$ is stored after $pos_{i+1}(x')$. And since our algorithm stores $f^+_{i+1}(x_r, \alpha_{i+1} y_r)$ at the same place as (x_r, y_r), $pos_{i+1}(x') \leq pos_i(b_i)$. □

Lemma C.4 allows us to find $pos_i(z)$ by moving a pointer monotonically to the right. Thus, we can retrieve $b_n, \ldots, b_1$ in order by unrolling our linear algorithm for the decision problem and moving the pointer $pos_i(z)$. This process takes $O(n)$ time overall.

C.3. Analysis

We conclude with the proof of our main theorem.

Proof of Theorem 1.2: The correctness of Algorithm 1 follows from the preceding discussions: By Lemma 2.9, the iterative transformations compute the P_i as defined in (2), and $\mathcal{S} \neq \emptyset$ iff $P_n \neq \emptyset$. Moreover, Lemma C.1 shows that, when $\mathcal{S} \neq \emptyset$, Step 3 computes a valid $\boldsymbol{b} \in \mathcal{S}$. It remains to analyze the running time.

- Step 1 takes $O(1)$ time since the vertices of P_2 are a subset of the (at most) 12 intersection points of the defining lines. (P_2 is the trapezoid spanned by $(x_2^-, x_2^- - x_1^+), (x_2^-, x_2^- - x_1^-), (x_2^+, x_2^+ - x_1^+), (x_2^+, x_2^+ - x_1^-)$, intersected with the halfspaces $y \geq y_2^-$ and $y \leq y_2^+$.)
- Step 2. The operations inside the loops are all constant-time and the outer loop runs $O(n)$ times. Moreover, the inner while-loops all remove a node from a deque, so their total cost over all iterations of the for-loop is $O(n)$, too: We start with $O(1)$ vertices and adding at most $O(n)$ vertices throughout the entire procedure (Lemma 2.10), so we cannot remove more than $O(n)$ vertices.
- Step 3. All operations except for the first line inside the for-loop take constant time. The inner while-loop runs for overall $O(n)$ iterations, since p only moves right and we add $O(n)$ vertices in total.

  It remains to implement the first line of the loop body in $O(n)$ overall time. To be able to undo the changes to u, v, S_u, S_v, we keep a *log* for each instruction executed in Step 2, so that we can undo their changes here (in the opposite order). Since Step 2 runs in $O(n)$ total time, the rollback also runs in $O(n)$ time.

Since all three steps run in linear time, so does the whole algorithm. □

D. Generalization to DAGs is hard

In this appendix, we will give a natural generalization of Definition 1.1 to arbitrary DAGs and investigate its complexity. Our original setting with differences of adjacent indices only corresponds to a directed-path graph.

In light of rather general results for isotonic regression, the path setting might appear quite restrictive; we will argue here why these conditions probably cannot be relaxed much further if we want an $O(n)$ time algorithm.

Definition D.1: *Suppose we are given a directed acyclic graph $G = (V, E)$ with $m = |E|$ edges and m_p number of length two directed paths in G , n-dimensional vectors $x^- \leq x^+$, m*

dimensional vector $y^- \leq y^+$, and m_p dimensional vectors $z^- \leq z^+$ and $\alpha \geq 0$. We define $\mathcal{S}_G$ to be the set of all n-dimensional vectors b such that $x_i^- \leq b_i \leq x_i^+$ for all i, $y_{ij}^- \leq b_j - b_i \leq y_{ij}^+$ for all edges $(i, j) \in E$, and $z_{ijk}^- \leq (b_k - b_j) - \alpha_{ijk}(b_j - b_i) \leq z_{ijk}^+$ for all pairs of edges $(i, j), (j, k) \in E$.

In contrast to Theorem 1.2, we show that determining if $\mathcal{S}_G$ if empty or not is as hard as solving linear programs.

Theorem D.2: *With notation as in Definition D.1, if we can determine $\mathcal{S}_G$ is empty or not in time $f(n + m + m_p)$, then we can determine feasibility of any set of linear constraints defined by s bounded integer coefficients in $c_1 f(c_2 s \log M))$ time, where c_1 and c_2 are two constants and the absolute value of each coefficient in the linear constraints is no more than M.*

Our reduction to prove Theorem D.2 is closely motivated by the hardness of isotropic total variation from [20], as well as subsequent works on extending such hardness results to positive linear programs. Compared to these results though, it sidesteps linear systems, and is a more direct invocation of the completeness of 2-commodity flow linear programs from [15].

We first consider a more restricted class of problems than Definition D.1 allows (where all the α's in Definition D.1 are set to be 1). Formally we define the problem as:

Definition D.3: *A generalized second-order constrained feasibility problem is defined by variables $b_1 \dots b_n$, combined with a set of m constraints parameterized by*

1. *Upper and lower bounds on the variables x_i^- and x_i^+.*
2. *Upper and lower bounds on the first order differences y_i^- and y_i^+ and corresponding indices $p_i < q_i$.*
3. *Upper and lower bounds on the second order differences z_i^- and z_i^+ and corresponding indices $r_i < s_i < t_i$*

and constraints

Value Constraints: $x_i^- \leq b_i \leq x_i^+$

First Order Constraints: $y_i^- \leq b_{q_i} - b_{p_i} \leq y_i^+$

Second Order Constraints: $z_i^- \leq (b_{t_i} - b_{s_i}) - (b_{s_i} - b_{r_i}) \leq z_i^+$.

The goal is to decide whether there exists $b_1, \dots, b_n$ that satisfy all these constraints simultaneously.

Proof of Theorem D.2: It is easy to see that the problem defined in Definition D.3 is a special case of the problem in Definition D.1. This is obtained by forming a DAG with edges (p_i, q_i), (r_i, s_i), (s_i, t_i) for all p_i, q_i, r_i, s_i, t_i. We will prove that a general linear programming feasibility problem with s polynomially-bounded integer coefficients can be expressed as a second-order-constrained feasibility problem (Definition D.3). In particular, we will show that a feasibility of a set of linear constraints containing at most s non-zero coefficients whose absolute values are integers no more than M can be reduced to $O(s \log M)$ value, first order and second order constraints as in Definition D.3.

Note that the second constraint in Definition D.3 is the same as

$$z_i^- \leq 2b_{q_i} - b_{r_i} - b_{p_i} \leq z_i^+.$$

In particular, it allows us to create constraints of the form

$$2b_{q_i} = b_{p_i} + b_{r_i}.$$

We will now show how we can restate a feasibility of a set of general linear constraints can be expressed as a second order constrained feasibility problem as in Definition D.1. The main idea will be clear when we consider a linear constraint of the form

$$b_{i_1} + b_{i_2} + \dots b_{i_k} \leq c_i,$$

with k a power of 2, and $i_1 < i_2 < \dots < i_k$ in increasing order. To express this in terms of second order constraints, we can introduce new variables

$$\begin{aligned} i_1 < i_{12} < i_2 \\ i_3 < i_{34} < i_4 \\ \dots \end{aligned}$$

and use $b_{i_{12}}$ to represent the sum of b_{i_1} and b_{i_2} and so on. Repeating this halves the value of k, but aggregates the whole sum into a single variable. Therefore, we can express the above linear constraint as one value constraint

$$b_{i_{12\dots k}} \leq x^+_{12\dots k} := c_i$$

and $k-1$ second order constraints

$$b_{i_{12}} = b_{i_1} + b_{i_2}, \dots, b_{i_{(k-1)k}} = b_{i_{k-1}} + b_{i_k}, \dots, b_{i_{1\dots k}} = b_{i_{1\dots k/2}} + b_{i_{k/2+1\dots k}}.$$

In case k is not a power of 2, we can add dummy variables whose values we restrict to zero using the value constraints. This process uses at most k value constraints. So we have shown that we can express any linear constraint of the form $b_{i_1} + b_{i_2} + \dots b_{i_k} \leq c_i$ in terms of $O(k)$ second order constraints and $O(k)$ value constraints.

Now consider the case with both positive and negative values in the linear constraint

$$b_{i_1} \pm \dots \pm b_{i_k} \leq c_i.$$

We can aggregate the sums of the variables with positive coefficients and negative coefficients separately, and let us denote the resulting variables by $b_{\text{pos}}, b_{\text{neg}}$. We can now bound the difference using a first order constraint of the form

$$b_{\text{pos}} - b_{\text{neg}} \leq c_i.$$

This results in additional $O(1)$ first order constraints for each linear constraint.

Finally, when the coefficients are arbitrary integers, we can do pairing based on the binary representation. The second order constraint and value constraint allows us to create constrains of the form

$$\begin{aligned} 0 \leq 2b_i - b_0 - b_j \leq 0 \\ 0 \leq b_0 \leq 0 \end{aligned}$$

which are equivalent to

$$b_j = 2b_i.$$

So we can introduce new variables d_{kj} representing $2^k b_j$ for any $1 \leq k \leq c$ where c is a constant. Thus, given any linear constraint in k variables with integer coefficients that are bounded by

M, we first represent each coefficient by its binary representation, increasing the number of non-zero coefficients by $O(\log M)$ times and creating $O(k \log M)$ second order constraints and value constraints. Then all the coefficients in the linear constraints are $+1$ or -1 and we can use the reduction above. In summary, we can solve any linear programming feasibility problem with $O(s)$ non-zero coefficients which are integers bounded by M by a generalized second-order constrained feasibility problem of $O(s \log M)$ constraints. This together with our assumption of an algorithm solving generalized second-order constrained feasibility problem in $f(\cdot)$ time prove the theorem. $\square$