# An $O(n)$ algorithm to construct the greatest convex minorant and least concave majorant

Andrew J. Radcliffe

September 4, 2023

**Abstract**

TODO

## 1 Introduction

Consider a set of points, $(x_i, f(x_i)),\ i = 0, \ldots, n-1$. Suppose that one conjectures that $f$ is convex, but for arbitrary reasons, the observation of $f$ is such that the set is not convex. However, we would like to construct a convex set $(x_i, \bar{f}(x_i),\ i = 0, \ldots, n-1$ which contains the set of points, and which satisfies the property of $\bar{f}(x_i) \leq f(x_i)\ \forall i$, i.e. $\bar{f}$ is the greatest convex minorant (GCM), if we were to impose the restriction that $x \geq 0$. We assume that the set $\Omega = \{x_i :\ i = 0, \ldots, n-1\}$ is convex. Here, we admit a slight departure from the conventional definition of the greatest convex minorant by permitting $x < 0$.

Let us consider the points to be ordered such that $x_{i-1} < x_i$. We have already assumed that $f$ is convex. If we assume that that $f$ is once continuously differentiable then we have [1]

$$Df(x_{i-1})(x_i - x_{i-1}) \leq f(x_i) - f(x_{i-1}) \tag{1}$$

which implies that

$$f(x_i) \geq f(x_{i-1}) + Df(x_{i-1})(x_i - x_{i-1}) \tag{2}$$

That is, the $f(x_i)$ always lies below the linear approximation of $f$ at $x_i$. We can generalize this to the case where $f$ is not differentiable by substituting for the Jacobian, $Df(x)$, the transpose of the subgradient, $g$, such that

$$f(x_i) \geq f(x_{i-1}) + g^{\mathrm{T}} f(x_{i-1})(x_i - x_{i-1}) \tag{3}$$

Thus, if one had a method to obtain the Jacobian (or subgradient), one could compute the nearest convex approximation to $f(x_i)$ as

$$\begin{aligned} f(x_i) &= f(x_{i-1}) + Df(x_{i-1})(x_{i-x_{i-1}}) \\ &= f(x_0) + \sum_{i^*=1}^{i} Df(x_{i^*-1})(x_{i^*} - x_{i^*-1}) \end{aligned} \tag{4}$$

If we restrict ourselves to $x \in \mathbb{R}$, then one is justified in re-writing (1) as

$$Df(x_{i-1}) \leq \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}} \tag{5}$$

It is self-evident that the values of $Df(x)$ which one could compute from the data would simply recapitulate the observed values of $f(x)$ when plugged in to (4). This appears to merely re-state the problem, but provides an insight: we must construct an approximation to $Df(x)$ such that $f$ is convex.

Consider that if $f$ is twice continuously differentiable, then $f$ is convex on $\Omega$ if and only if for each $x \in \Omega$ the Hessian of $f$ at $x$ is a positive semi-definite matrix. Thus, to construct a $Df(x)$ such that $f$ is convex, we must have $D^2 f(x) \succeq 0, \; \forall x \in \Omega$. When $\Omega \subset R$, i.e. the one-dimensional case, this simplifies to $\frac{d^2 f(x)}{dx^2} \geq 0, \forall x \in \Omega$. In other words, the derivative $Df(x)$ must be $\geq 0$. One such way to construct such a $Df(x)$ is to perform isotonic regression on the values of $Df(x_{i-1})$ computed using (5). By definition, the resultant function has derivative $\geq 0$, hence, $f(x_i)$ computed by this approximation will be convex.

Assume that the framework above constructs the GCM. We note that if $f$ is convex, then $-f$ is concave, hence, we can apply the same methodology to construct the least concave majorant from a set of points $(x_i, f(x_i)),\; i = 0, \ldots, n-1$, where $x \in \Omega$ is a convex set, by constructing the GCM of the set of points $(x_i, -f(x_i)),\; i = 0, \ldots, n-1$, then negating the resultant $\bar{f}$. That is, starting from a concave $f$, we construct a convex approximation to $-f$, denoted $\bar{f}$, then negate said approximation to recover a concave function as $-\bar{f}$. It is a simple result, but demonstrates that the algorithm presented in the next section solves two problems (which are in fact the same problem, masquerading under different names).

## 2 Algorithm

The mention of isotonic regression above hints at what the $O(n)$ algorithm will entail. Specifically, the constrained optimization problem to be solved is:

$$\begin{gathered} \min \frac{1}{2} \sum_{i=0}^{n-2} w_i (Df(x_i) - \mu_i)^2 \\ \text{subject to } \mu_0 \leq \mu_1 \cdots \leq \mu_{n-3} \leq \mu_{n-2} \end{gathered} \tag{6}$$

in which the order of the points, $(x_i, f(x_i))$, is such that $x_0 < x_1 \cdots < x_{n-2} < x_{n-1}$. This problem can be treated using Langrange multipliers. Let us define the Lagrangian,

$$L(\mu, \lambda) = \frac{1}{2} \sum_{i=0}^{n-2} w_i (Df(x_i) - \mu_i)^2 + \sum_{j=0}^{k-3} \lambda_j (\mu_j - \mu_{j+1}) \tag{7}$$

The partial derivative of which is

$$\frac{\partial L(\mu, \lambda)}{\partial \mu_m} = \lambda_m - \lambda_{m-1} - w_i (Df(x_m) - \mu_m) \tag{8}$$

We now write the Karush-Kuhn-Tucker (KKT) conditions:

$$\frac{\partial L(\mu, \lambda)}{\partial \mu_m} = 0, \qquad m = 0, \ldots, k-2 \qquad \text{derivative of Lagrangian equal to zero} \tag{9}$$

$$\mu_j \leq \mu_{j+1}, \qquad j = 0, \ldots, k-3 \qquad \text{primal feasibility} \tag{10}$$

$$\lambda_j \geq 0, \qquad j = 0, \ldots, k-3 \qquad \text{dual feasibility} \tag{11}$$

$$\lambda_j (\mu_j - \mu_{j+1}) = 0, \qquad j = 0, \ldots, k-3 \qquad \text{complementary slackness} \tag{12}$$

Where we define $\lambda_{-1} = \lambda_{k-2} = 0$ in order to make (**??**) valid for $m = 0, \ldots, k-2$.

The classic approach to solving this problem is the pool adjacent violators (PAV) algorithm. An excellent exposition of this algorithm, in terms of order simplexes, with geometric interpretation, is provided by 3. Grotzinger and Witzgall demonstrate not only that a PAV algorithm, carefully implemented, has $O(n)$ time and space complexity, but also that the solution obtained satisfies the KKT conditions. Their work forms the basis for the present algorithm, which differs only in a subtle aspect.

Recall that our objective is construct an approximation to $Df(x)$, from values computed from the data, i.e.

$$Df(x_{i-1}) = \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}} \tag{13}$$

If we were to naively follow intuition, we might simply attempt isotonic regression on the values of $Df(x_{i-1})$ directly. This would work if the points in $\Omega$ were uniformly-spaced, i.e. $x_i - x_{i-1} = x_{i+1} - x_i$ for $i = 1, \ldots, n-2$. Unfortunately, this will not in general be the case. However, it is possible to construct a generalized pooling update which, irrespective of the spacing of the $x$ points preserves the geometric properties of the uniform pooling update; this is where we make our contribution.

## 2.1 Generalization of pooling mechanism

The PAV algorithm is initialized with $\boldsymbol{\mu} = Df(\boldsymbol{x})$, $\boldsymbol{w} = \mathbf{1}$, and $\boldsymbol{\lambda} = \mathbf{0}$. The algorithm proceeds by finding adjacent violators, defined by any $j$ for which $\mu_{j-1} > \mu_j$, then "pooling" them by constructing a weighted average of the two values

$$\mu' = \frac{w_{j-1}\mu_{j-1} + w_j\mu_j}{w_{j-1} + w_j} \tag{14}$$

$$w' = w_{j-1} + w_j \tag{15}$$

then replacing the $(j-1)^{\text{th}}$ block and weight with $\mu'$ and $w'$, respectively; $j^{\text{th}}$ block is then removed. The transpose is possible – replace $j^{\text{th}}$, remove $(j-1)^{\text{th}}$ – but, as we shall see below, this leads to $O(n^2)$ in the worst case as we cannot represent the blocks as a stack (unless we use a linked list, which we prefer to avoid).

Note that the above pooling mechanism is equivalent pooling via

$$\mu' = \frac{\sum_{i \in I_{j-1}} w_i\mu_i + \sum_{i \in I_j} w_i\mu_i}{\sum_{i \in I_{j-1}} w_i + \sum_{i \in I_j} w_i} \tag{16}$$

setting the blocks at $i \in I_{j-1} \cup I_j$ equal to $\mu'$, then updating the record of the contiguous indices which belong to each "pooled" block, which necessitates that one of the elements in said record be removed (either $(j-1)^{\text{th}}$ or $j^{\text{th}}$, it does not matter which).

The only difference between the two approaches is whether one represents the weight of pooled blocks via maintenance of (possibly many) block instantiations (the latter option), or by construction of a single block which retains the same information (the former option). The former option is much preferable from an implementation standpoint, as it leads to $O(n)$ time complexity, whereas the latter option is inevitably $O(n^2)$ due to the sums which must be computed from (possibly many) weights each time the pooling mechanism is invoked. The latter option has $O(n^2)$ time complexity even when $w_i = 1\ \forall i$ due to the requirement that values be copied into just-pooled block; this results from the literal representation of the basic blocks throughout the PAV algorithm.

To account for the possibility of non-uniformity of the values $x_i - x_{i-1}$, we re-write the definition

$$\mu_j = \frac{\nu_j}{\xi_j} \tag{17}$$

and change the pooling mechanism to

$$\nu' = \frac{w_{j-1}\nu_{j-1} + w_j\nu_j}{w_{j-1} + w_j} \tag{18}$$

$$\xi' = \frac{w_{j-1}\xi_{j-1} + w_j\xi_j}{w_{j-1} + w_j} \tag{19}$$

$$w' = w_{j-1} + w_j \tag{20}$$

in which $\nu_i = df_i = f_i - f_{i-1}$, $\xi_i = dx_i = x_i - x_{i-1}$. Noting that we have $w_i = 1\ \forall i$,

$$\nu_j = \frac{\sum_{i\in I_j} df_i}{w_j} \qquad \Longrightarrow \qquad w_j\nu_j = \sum_{i\in I_j} df_i \tag{21}$$

$$\xi_j = \frac{\sum_{i\in I_j} dx_i}{w_j} \qquad \Longrightarrow \qquad w_j\xi_j = \sum_{i\in I_j} dx_i \tag{22}$$

Thus,

$$\nu' = \frac{\sum_{i\in I_{j-1}} df_i + \sum_{i\in I_j} df_i}{w_{j-1} + w_j} \tag{23}$$

$$\xi' = \frac{\sum_{i\in I_{j-1}} dx_i + \sum_{i\in I_j} dx_i}{w_{j-1} + w_j} \tag{24}$$

The pooled value is therefore

$$\begin{aligned}\mu' &= \frac{\nu'}{\xi'} \\ &= \frac{\sum_{i\in I_{j-1}} df_i + \sum_{i\in I_j} df_i}{\sum_{i\in I_{j-1}} dx_i + \sum_{i\in I_j} dx_i}\end{aligned} \tag{25}$$

It is useful to elucidate the properties of the generalized pooling mechanism in (25). First, we demonstrate that it is equivalent to the canonical weighted-average pooling of $Df(x_i)$ values given uniform $dx_{i-1} = x_i - x_{i-1}$ values. If $dx_i = c\ \forall i$, then by (21) and (22), we have

$$\begin{aligned}\mu_j &= \frac{\sum_{i\in I_j} df_i}{c\sum_{i\in I_j} 1} \\ &= \frac{\sum_{i\in I_j} df_i}{c w_j}\end{aligned} \tag{26}$$

and by (25) we have

$$\mu' = \frac{\sum_{i\in I_{j-1}} df_i + \sum_{i\in I_j} df_i}{c\sum_{i\in I_{j-1}} 1 + c\sum_{i\in I_j} 1} \tag{27}$$

$$= \frac{\sum_{i\in I_{j-1}} df_i + \sum_{i\in I_j} df_i}{c(w_{j-1} + w_j)} \tag{28}$$

If we substitute the definition $\mu_i = \frac{df_i}{dx_i} = \frac{df_i}{c}$ into (14), then we have

$$\mu' = \frac{\sum_{i\in I_{j-1}} \frac{df_i}{dx_i} + \sum_{i\in I_j} \frac{df_i}{dx_i}}{w_{j-1} + w_j} \tag{29}$$

$$= \frac{\sum_{i\in I_{j-1}} df_i + \sum_{i\in I_j} df_i}{c(w_{j-1} + w_j)} \tag{30}$$

One observes that the generalized mechanism with constant $dx_i$, (28), is equal to the original mechanism with constant $dx_i$, (30). This demonstrates that the generalized mechanism preserves the original weighted average mechanism.

### 2.1.1 Inductive proof of mechanism equivalence given constant $dx_i$

A formal proof is possible using induction. Consider a sequence of violator blocks, indexed to $j-1, j, j+1, j+2$, which maintain violations after pooling such that after pooling $j-1$ and $j$, the resultant block must be pooled with $j+1$, and this second resultant block must be pooled with $j+2$. Let $'$ denote the number of pooling operations required to reach a given step, with pooling occurring left-to-right. We have for the generalized mechanism,

$$\mu' = \frac{w_{j-1}df_{j-1} + w_j df_j}{w_{j-1}dx_{j-1} + w_j dx_j} \tag{31}$$

$$\mu'' = \frac{w_{j-1}df_{j-1} + w_j df_j + w_{j+1}df_{j+1}}{w_{j-1}dx_{j-1} + w_j dx_j + w_{j+1}dx_{j+1}} \tag{32}$$

$$\mu''' = \frac{w_{j-1}df_{j-1} + w_j df_j + w_{j+1}df_{j+1} + w_{j+2}df_{j+2}}{w_{j-1}dx_{j-1} + w_j dx_j + w_{j+1}dx_{j+1} + w_{j+2}dx_{j+2}} \tag{33}$$

We see that if $dx_j = c\ \forall j$, then the denominator simplifies to $c(w_{j-1} + w_j + w_{j+1} + w_{j+2})$. For the original mechanism,

$$\mu' = \frac{w_{j-1}\frac{df_{j-1}}{dx_{j-1}} + w_j\frac{df_j}{dx_j}}{w_{j-1} + w_j} \tag{34}$$

$$\mu'' = \frac{w_{j-1}\frac{df_{j-1}}{dx_{j-1}} + w_j\frac{df_j}{dx_j} + w_{j+1}\frac{df_{j+1}}{dx_{j+1}}}{w_{j-1} + w_j + w_{j+1}} \tag{35}$$

$$\mu''' = \frac{w_{j-1}\frac{df_{j-1}}{dx_{j-1}} + w_j\frac{df_j}{dx_j} + w_{j+1}\frac{df_{j+1}}{dx_{j+1}} + w_{j+2}\frac{df_{j+2}}{dx_{j+2}}}{w_{j-1} + w_j + w_{j+1} + w_{j+2}} \tag{36}$$

If $dx_j = c\ \forall j$, then $dx_j$ terms can be brought to the denominator, and the expressions become equal.

From this exposition, we observe that if the weight of a $\frac{df_j}{dx_j}$ point is $dx_j$, then the original mechanism becomes

$$\frac{df_{j-1} + df_j}{dx_{j-1} + dx_j} \tag{37}$$

which demonstrates that the effect of the generalized mechanism is to make the weight of each point equal to $dx_j$.

## 2.2 $O(n)$ algorithm using generalized pooling mechanism

Prior to writing down the algorithm, let us re-examine (17), (21) and (22). Substituting (21) and (22) into (17) yields

$$\mu_j = \frac{\sum_{i \in I_j} df_i}{\sum_{i \in I_j} dx_i} \tag{38}$$

for which the update mechanism is given by (23), (24) and (25). We notice that the $w_{j-1} + w_j$ terms cancel from both $\mu_j$ and $\mu'$. This enables a simplification of the pooling of $\nu_{j-1}, \nu_j$ and $\xi_{j-1}, \xi_j$, as we need only retain the numerators of (21) and (22) in order to compute $\mu_j$ without loss of information. To illustrate this, we can re-define

$$\nu_j = \sum_{i \in I_j} df_i \tag{39}$$

$$\xi_j = \sum_{i \in I_j} dx_i \tag{40}$$

and therefore re-write the pooling as

$$\nu' = \nu_{j-1} + \nu_j = \sum_{i \in I_{j-1}} df_i + \sum_{i \in I_j} df_i \tag{41}$$

$$\xi' = \xi_{j-1} + \xi_j = \sum_{i \in I_{j-1}} dx_i + \sum_{i \in I_j} dx_i \tag{42}$$

Clearly, the ratios $\frac{\nu_j}{\xi_j}$ and $\frac{\nu'}{\xi'}$ remain unchanged. This saves 4 `mul` and 2 `div` (and 3 type casts) per pooling operation, which, while it does not change the time complexity of the algorithm, is beneficial nonetheless.

We now present the algorithm.

```
   Data: $(x_i, f_i) \in \mathbb{R}^2$ such that $x_i < x_{i+1},\ i = 0, \ldots, n-1,\ n \geq 2$
   Result: $\bar{f}$ such that $\bar{f}(x_i) \leq f_i\ \forall i$
1  // compute forward differences
2  for $i \leftarrow 0$ to $n-2$ do
3  |   $df_i \leftarrow f_{i+1} - f_i$;
4  |   $dx_i \leftarrow x_{i+1} - x_i$;
5  end
6  // initialization of PAV
7  $\nu_0 \leftarrow df_0$;
8  $\xi_0 \leftarrow dx_0$;
9  $w_0 \leftarrow 1$;
10 $j \leftarrow 0$;
11 $i \leftarrow 1$;
12 $n \leftarrow n-1$;
13 while $i < n$ do
14 |   // propose new block
15 |   $j \leftarrow j+1$;
16 |   $\nu_j \leftarrow df_i$;
17 |   $\xi_j \leftarrow dx_i$;
18 |   $w_j \leftarrow 1$;
19 |   $i \leftarrow i+1$;
20 |   // pool adjacent violating blocks:  stop when none remain, or only
   |      single block remains
21 |   while $j > 0$ and $\frac{\nu_{j-1}}{\xi_{j-1}} > \frac{\nu_j}{\xi_j}$ do
22 |   |   $\nu_{j-1} \leftarrow \nu_{j-1} + \nu_j$;
23 |   |   $\xi_{j-1} \leftarrow \xi_{j-1} + \xi_j$;
24 |   |   $w_{j-1} \leftarrow w_{j-1} + w_j$;
25 |   |   $j \leftarrow j-1$;
26 |   end
27 end
28 // construct the GCM
29 $\bar{f}_0 \leftarrow y_0$;
30 $m \leftarrow j+1$;
31 $j \leftarrow 0$;
32 $i \leftarrow 1$;
33 while $j < m$ do
34 |   $D\bar{f}_j \leftarrow \frac{\nu_j}{\xi_j}$;
35 |   for $k \leftarrow i$ to $i + w_j - 1$ do
36 |   |   $\bar{f}_k \leftarrow \bar{f}_{k-1} + D\bar{f}_j dx_{k-1}$;
37 |   end
38 |   $i \leftarrow i + w_j$;
39 |   $j \leftarrow j+1$;
40 end
```

**Algorithm 1:** Left-to-right GCM

With the algorithm clearly stated, we can proceed with the analysis of its time and space complexity. By inspection, it is apparent that the forward differences on lines 2 through 5 require $2(n-1)$ subtraction operations and storage space equal to $2(n-1)$. The core of algorithm, lines 7-27, requires a bit more care to elucidate; it is helpful to provide an informal description. The idea behind such an implementation of PAV, which 3 also describe, is to construct the blocks one at a time, starting from a single block. Given the initial block, initialized as shown on lines 7-11, one proceeds to propose a new block, then checks whether it violates the KKT conditions, starting with the preceding block. If it violates primal feasibility, then it is pooled with the preceding ($(j-1)^{th}$) block; we then proceed to check the just-pooled block against its preceding block, repeating this process until the adjacent blocks no longer violate primal feasibility, or only a single block remains. Once the cycle of violation checks and pooling has terminated, we return to the top, propose a new block and enter the cycle again.

More formally, we observe that given the value of $j$ upon entry into the outer while loop, at most $j+1$ violation checks can occur; there are $n-2$ increments of $j$. The best case behavior corresponds to $\frac{\nu_{j-1}}{\xi_{j-1}} \leq \frac{\nu_j}{\xi_j}$ without any pooling, which incurs $n-2$ violation checks. There are two equivalent worst case behaviors. The first corresponds to $\frac{\nu_{j-1}}{\xi_{j-1}} \geq \frac{\nu_j}{\xi_j}$ upon proposal of each new block, followed by pooling of $\frac{\nu_{j-1}}{\xi_{j-1}}$ and $\frac{\nu_j}{\xi_j}$ such that $\frac{\nu_{j-2}}{\xi_{j-2}} \leq \nu'$. This incurs 2 violation checks and 1 pooling operation per proposal (i.e. per outer loop iteration), hence, $2(n-2) = 2n-4$ violation checks. The second type of worst case behavior can be realized when $\frac{\nu_{j-1}}{\xi_{j-1}} \leq \frac{\nu_j}{\xi_j}$ up to but not including $j = n-2$, followed by $\frac{\nu_{n-2}}{\xi_{n-2}}$ such that all elements must be pooled. This incurs $n-3$ violation checks up to $j = n-2$, then $n-1$ violation checks, hence, $2n-4$ violation checks.

Hence, in both the best case and worst case, the time complexity of the algorithm is $O(n)$. This is an asymptotically tight bound, irrespective of the inputs, as we must check at least $n-2$ pairs in the best case, and $2n-4$ in the worst case. From the equivalence of the worst case behavior, we observe that any reduction in the number of violators will reduce the total number of violation checks and pooling operations; depending on where the violators occur, the behavior will be some admixture of the first and second type of worst case behavior, but the total checks can be no more than $2n-4$.

## 2.3 Verification of KKT conditions

The loop invariant which is maintained by the algorithm consists of the KKT conditions, 9 10, 11 12. The KKT conditions provide a method to ascertain whether an implementation of the algorithm is correct, subject to finite precision of computer arithmetic. By combining the expression for the derivative of the Lagrangian, 9, and combining this with complementary slackness, 12, one observes that within a pooled block, defined by the index set $I_j$, it must be the case that

$$\sum_{i \in I_j} dx_i (Df_i - D\bar{f}_i) = 0 \tag{43}$$

This follows directly from complementary slackness, which between blocks, $j$ and $j+1$, necessitates that $\lambda_j (D\bar{f}_i - D\bar{f}_{i+1}) = 0$, which implies that $\lambda_j$ must be zero. When dual feasibility is applied, we observe that each partial sum computed as part of (43) must be $\geq 0$.

The Lagrange multipliers can be computed in several ways, but the most direct method is to proceed by re-writing 9 as

$$\lambda_m = \lambda_{m-1} + dx_i (Df_i - D\bar{f}_i) \tag{44}$$

which provides a recurrence relation for $m = 0, \ldots, k-2$. This provides a direct method of computation, but there exists another method which is worthwhile to pursue for its geometric interpretation. Begin by noting that

$$\begin{aligned} f_i &= f_{i-1} + Df_{i-1}dx_{i-1} \\ &= f_{i-1} + \frac{df_{i-1}}{dx_{i-1}}dx_{i-1} \\ &= f_{i-1} + df_i \\ &= f_0 + \sum_{i^*=0}^{i-1} df_{i^*} \end{aligned} \tag{45}$$

$$\begin{aligned} \bar{f}_i &= \bar{f}_{i-1} + D\bar{f}_{i-1}dx_{i-1} \\ &= \bar{f}_{i-1} + \frac{d\bar{f}_{i-1}}{d\bar{x}_{i-1}}dx_{i-1} \\ &= f_0 + \sum_{i^*=0}^{i-1} \frac{d\bar{f}_{i-1}}{d\bar{x}_{i-1}}dx_{i-1} \end{aligned} \tag{46}$$

This enables one to substitute into the recurrence, (44), obtaining

$$\begin{aligned} \lambda_m &= \lambda_{m-1} + df_m - \frac{f_m}{d\bar{x}_m}dx_m \\ &= \lambda_{-1} + \sum_{i^*}^{m} df_{i^*} - \frac{f_{i^*}}{d\bar{x}_{i^*}}dx_{i^*} \\ &= f_{m+1} - \bar{f}_m \end{aligned} \tag{47}$$

We now have a clear interpretation for the Lagrange multipliers: they are the distance between the input ($f$) and the greatest convex minorant ($\bar{f}$). Hence, given only a result obtained by an implementation of the algorithm and the original data, a computer program can be used to verify that the KKT conditions are satisfied (subject to error introduced due to finite precision).

It may seem trivial to verify something which is known from the construction of an algorithm, but the disconnect between theory and implementation is an ever-present problem. At the very least, this enables us check our implementation (which could contain errors!), hence assure users of the library that the code implements what it purports to.

## 3 Implementation

For the algorithm described above, we provide a Rust implementation (`https://github.com/andrewjradcliffe/gcm-lcm`). The test suite utilizes the KKT conditions, computed as described above and also via less direct method, to enforce correctness of the implementation.

## 4 References

1. Chong, Edwin KP, and Stanislaw H. Żak. An introduction to optimization. Vol. 75. John Wiley & Sons, 2013. (Vol. 75). John Wiley & Sons.

2. Geyer, Charles J. "Stat 8054 Lecture Notes: Isotonic Regression". (`https://www.stat.umn.edu/geyer/8054/notes/isotonic.pdf`) (2023).

3. Grotzinger, Stephen J., and Christoph Witzgall. "Projections onto order simplexes." Applied mathematics and Optimization 12 (1984): 247-270.