Journal of Statistical Software

May 2022, Volume 102, Code Snippet 1. *doi: 10.18637/jss.v102.c01*

Monotone Regression: A Simple and Fast $O(n)$ PAVA Implementation

Frank M. T. A. Busing
Leiden University

Abstract

Efficient coding and improvements in the execution order of the up-and-down-blocks algorithm for monotone or isotonic regression leads to a significant increase in speed as well as a short and simple $O(n)$ implementation. Algorithms that use monotone regression as a subroutine, e.g., unimodal or bivariate monotone regression, also benefit from the acceleration. A substantive comparison with and characterization of currently available implementations provides an extensive overview of up-and-down-blocks implementations for the pool-adjacent-violators algorithm for simple linear ordered monotone regression.

Keywords: monotone regression, isotonic regression, L_2 metric, pool-adjacent-violators algorithm, PAVA, up-and-down-blocks algorithm, unimodal, umbrella orderings, bivariate.

1. Introduction

Monotone or isotonic regression minimizes the weighted least squares loss function

$$f(\mathbf{x}) = \sum_{i=1}^{n} w_i \left(y_i - x_i\right)^2 , \qquad (1)$$

under the restriction that $x_1 \leq x_2 \leq \ldots \leq x_{n-1} \leq x_n$, where $y_i, i = 1, \ldots, n$, are given data values (in proper order) and $w_i, i = 1, \ldots, n$, are corresponding non-negative weights.

There is a substantial amount of literature on monotone or isotonic regression, starting with Ayer, Brunk, Ewing, Reid, and Silverman (1955) and Van Eeden (1958) and followed by, among others, Barlow, Bartholomew, Bremner, and Brunk (1972), Robertson, Wright, and Dykstra (1988), and De Leeuw, Hornik, and Mair (2009). And there is also, as Best and Chakravarti (1990) put it, "a bewildering variety of algorithms" available for solving the problem. For the special case of simple linear orders in the L_2 metric, addressed in the current paper and given in (1), the pool-adjacent-violators algorithm (PAVA), due to Ayer *et al.*

(1955), Miles (1959), and Kruskal (1964b), is commonly known as the fastest algorithm available (cf. Stout 2019). But although PAVA is known for its $O(n)$ algorithmic complexity (see Bentley 1976; Best and Chakravarti 1990), implementations do not necessarily inherit this algorithmic characteristic without a struggle: An $O(n)$ algorithmic complexity does neither guarantee an $O(n)$ nor a fast implementation. See, for an early reference on PAVA implementations, for example, Van Waning (1976), Grotzinger and Witzgall (1984), and for more recent references Wikipedia (2022), Tulloch (2014a), and De Leeuw (2017).

Isotonic regression has applications in fields like operations research, signal processing, and statistics, (see, e.g., Kyng, Rao, and Sachdeva 2015; Chatterjee, Guntuboyina, and Sen 2015). More specifically, concerning statistics, the pool-adjacent-violators algorithm and its implementations have applications in, for example, test statistics (Bartholomew 1961), maximum likelihood estimation of ordered means (Robertson 1978), unimodal function estimation (Turner and Wollan 1997; Eggermont and LaRiccia 2000; Reboul 2005), calculation of adjusted response probabilities (Pace, Stylianou, and Warltier 2007), assessing the quality of micro-array titration data (Klinglmueller 2010), and efficient first-order generalized gradient algorithms (Tibshirani and Suo 2016). In our case, monotone regression is used as part of a larger procedure, like for one of its originators (Kruskal 1964a,b): the transformation step in a nonmetric multidimensional scaling algorithm.

Due to the computational slowness of some monotone regression implementations (cf. Stout 2015), algorithmic workarounds have been used to reduce these computational burdens in multidimensional scaling, such as regularly skipping the transformation step in favor of the configuration fitting step (Busing, Commandeur, and Heiser 1997) or using another, less appropriate but faster method, such as rank images (Guttman 1968), for at least the first few steps of the iterative process (Stoop and De Leeuw 1982). In any case, multidimensional scaling would certainly benefit from a reliable and fast monotone regression procedure, especially when n becomes large.

This code snippet suggests yet another implementation of PAVA, that is both $O(n)$ and fast, and the snippet will proceed as follows. In Section 2, we give a general description of the pool-adjacent-violators algorithm, provide examples of some algorithmic choices, and describe our implementation of the algorithm. Extensions of the algorithm, procedures in which the algorithm is used as a subroutine, are described in Section 3. In Section 4, we provide an extensive comparison with several known and currently available implementations in terms of speed. The extensions are additionally compared with available and comparable procedures. The snippet ends with a conclusion.

2. Pool-adjacent-violators algorithm

The pool-adjacent-violators algorithm was first described by Ayer *et al.* (1955), but became well-known by the up-and-down-blocks implementation of the algorithm described in Kruskal (1964b). To clarify the link: in the latter, blocks are the collection of pooled values.

The general idea of the pool-adjacent-violators algorithm is as follows: Find an optimal solution such that if $y_i \geq y_{i+1}$, $x_i = x_{i+1} = (w_i x_i + w_{i+1} x_{i+1})/(w_i + w_{i+1})$, for all i, i.e., in short: if there is a violation, pool. The result is a composition of elements with non-decreasing values.

Although the up-and-down-blocks algorithm seems quite straightforward, works very efficiently (Dykstra and Robertson 1982), and is trivial to implement in linear time (Stout 2019), Best and Chakravarti (1990) state that only a skillful implementation is of computational complexity $O(n)$. Grotzinger and Witzgall (1984, page 262) discuss the algorithm and the implementation decisions for the unweighted case. The differences between weighted and unweighted algorithms vary widely across metrics, but for the L_2 metric there is essentially no difference (Stout 2019, Remark 4). This equivalence is not true for the L_1 and L_∞ metrics that are much more difficult to solve, with or without weights. Important implementation decisions for the current L_2 case concern 1. the processing of pooled elements or block values and 2. the direction and scope of violation checking. We will now discuss these two issues in detail.

1. In essence, there are two ways to administrate pooled elements. The first (direct) approach updates the complete result vector after coalescing violating elements: the block value is directly inserted in the results vector, at the designated positions. According to Grotzinger and Witzgall (1984), this will result in an $O(n^2)$ procedure. Implementations using this approach are, for example, from Van Waning (1976), Venables and Ripley (2013), Dumelle, Kincaid, Olsen, and Weber (2022), and Turner (2020). The other (postponed) approach collects all block values in a separate vector, which is expanded later to obtain the final results vector. Although the direct updates approach does not require additional memory to store the block values and lacks the need for an index vector, keeping the results vector up-to-date might lead to a very slow procedure at worst, while expanding the block values, in the postponed approach, can be performed rather quickly.

2. Concerning the second issue on direction and scope of violation checking, first note that all up-and-down-block implementations of the pool-adjacent-violators algorithm access $\mathbf{x}$ sequentially, going from x_1 to x_n, repairing violations along the way. These repairs may concern (a) only neighboring elements, (b) extend the scope to multiple neighboring elements, either forward (up) or backward (down), or (c) extend the scope to multiple neighboring elements in both directions.

   (a) Implementations that only repair a single violation at the time might need to restart the whole sequence, probably more than once, in order to eliminate all violations. This approach is illustrated in the first part of Table 1 (1-up-0-down). The block that arises in eliminating the violation in Step 4, $8 = x_3 > x_4 = 2$, does not resolve the re-created violation with the previous block ($x_1 = x_2 = 6$), so Steps 8–10 are needed to properly solve the complete problem. This procedure corresponds to one given by Miles (1959) and implementations are, for example, given by Bril, Dykstra, Pillers, and Robertson (1984) and Turner (2020).

   (b) Most up-and-down-blocks implementations check for a violation with the next element, 1-up, followed by the elimination of all down-block violations (k-down) when there was indeed a violation. When there was not, the procedure simply progresses to the next element. Since the forward violations are resolved sequentially, and the backward violations are immediately resolved too, this procedure uses exactly n steps. An example of this procedure is given in the middle part of Table 1 (1-up-k-down). Implementations of this kind are given by, for example, Kruskal (1964b), Van Waning (1976), Cran (1980), Danisch (2016), and De Leeuw (2017).

		1-up-0-down										1-up-k-down							k-up-k-down				
i	**x**	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	1	2	3	4	5
1	8	8	6	6	6	6	6	6	6	4	4	8	6	6	5.5	4.8	4	4	8	6	6	4	4
2	4		6	6	6	6	6	6	6	4	4		6	6	5.5	4.8	4	4		6	6	4	4
3	8			8	5	4	3	3		4	4			8	5.5	4.8	4	4			8	4	4
4	2				5	4	3	3		4	4				5.5	4.8	4	4				4	4
5	2					4	3	3		4	4					4.8	4	4				4	4
6	0						3	3		4	4						4	4				4	4
7	8							8			8							8					8

Table 1: Pool-adjacent-violators algorithm examples using different violation checking strategies.

(c) The implementation suggested in this snippet adds yet another step. Observe that the violation $8 = x_3 > x_4 = 2$ is solved by combining two values, 8 and 2, resulting in a (new) block value of 5, i.e., $(8 + 2)/2 = 5$. Instead of immediately turning around and start solving down block violation, we may first look ahead for the next value in the sequence, k-up, for if this element is smaller than or equal to 5, the next value can immediately be pooled into the current block, i.e., $(8+2+2)/3 = 4$. Looking ahead can be continued until the next element is larger than the current block value or if we reach the end of the sequence. For the current example $x_6 = 0$ is pooled into the block too, since $4 > 0$, and the current block value becomes $(8 + 2 + 2 + 0)/4 = 3$. The next element $x_7 = 8$ is not a violator, since the current block value equals 3 which is smaller than 8. The lookahead procedure is displayed in Step 4 of the last part of Table 1 (k-up-k-down). Superfluous lookaheads, e.g., Step 2 under k-up-k-down, more than offset the advantages of being able to complete the sequence faster. Every additional upwards violation captured by the lookahead procedure (here additionally combining elements x_5 and x_6 with x_3 and x_4) avoids one or more up-and-down block violation eliminations that costs more than double.

The pseudo-code for the implementation is given in Algorithm 1. We can clearly distinguish the single upwards violation check in Line 11, the k-up procedure in Lines 16–21, and the k-down violation checks in Lines 22–27. The lookahead procedure moves ahead one element at the time with minimal cost: the new block value x (Line 15) is computed anyway and subsequent elements are handled with maximum time efficiency. The down-block violations (Lines 22–27) are handled with the same efficiency. In Lines 33–41 the block values are expanded to the results vector in reversed order to avoid overwriting the data block values.

Although the current implementation of the pool-adjacent-violators algorithm is written in C, a conversion to any other language should pose no serious problem given the pseudo-code in Algorithm 1. The function `monotone()` is provided in the package **monotone**[1] (Busing and Claramunt Gonzalez 2022) which is available from the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/package=monotone`.

[1] Compiled with `gcc -DNDEBUG -O2 -Wall -std=gnu99 -mfpmath=sse -msse2 -mstackrealign`

Algorithm 1 Monotone regression minimizing $f(\mathbf{x}) = \sum_{i=1}^{n} w_i \left(y_i - x_i\right)^2$ using the up-and-down-blocks implementation of the pool-adjacent-violators algorithm.

```
procedure MONOTONE($n, \mathbf{x}, \mathbf{w}$)            ▷ $\mathbf{x}$ in expected order and $\mathbf{w}$ nonnegative
    $r_0 \leftarrow 0$                                        ▷ initialize index 0
    $r_1 \leftarrow 1$                                        ▷ initialize index 1
    $b \leftarrow 1$                                          ▷ initialize block counter
    $\widehat{x} \leftarrow x_b$                              ▷ set previous block value
    $\widehat{w} \leftarrow w_b$                              ▷ set previous block weight
    for $i \leftarrow 2, n$ do                                ▷ loop over elements
        $b \leftarrow b + 1$                                  ▷ increase number of blocks
        $x \leftarrow x_b$                                    ▷ set current block value
        $W \leftarrow w_b$                                    ▷ set current block weight
        if $\widehat{x} > x$ then                             ▷ check for down violation of $x$
            $b \leftarrow b - 1$                              ▷ decrease number of blocks
            $S \leftarrow \widehat{w} \times \widehat{x} + W \times x$    ▷ set current weighted block sum
            $W \leftarrow W + \widehat{w}$                    ▷ set new current block weight
            $x \leftarrow S/W$                                ▷ set new current block value
            while $i < n$ and $x \geq x_{i+1}$ do             ▷ repair up violations
                $i \leftarrow i + 1$                          ▷ increase element counter
                $S \leftarrow S + w_i \times x_i$             ▷ set new current weighted block sum
                $W \leftarrow W + w_i$                        ▷ set new current block weight
                $x \leftarrow S/W$                            ▷ set new current block value
            end while
            while $b > 1$ and $x_{b-1} > x$ do                ▷ repair down violations
                $b \leftarrow b - 1$                          ▷ decrease number of blocks
                $S \leftarrow S + w_b \times x_b$             ▷ set new current weighted block sum
                $W \leftarrow W + w_b$                        ▷ set new current block weight
                $x \leftarrow S/W$                            ▷ set new current block value
            end while
        end if
        $x_b \leftarrow \widehat{x} \leftarrow x$             ▷ save block value
        $w_b \leftarrow \widehat{w} \leftarrow W$             ▷ save block weight
        $r_b \leftarrow i$                                    ▷ save block index
    end for
    $f \leftarrow n$                                          ▷ initialize “from” index
    for $k \leftarrow b, 1$ do                                ▷ loop over blocks
        $t \leftarrow r_{k-1} + 1$                            ▷ set “to” index
        $x \leftarrow x_k$                                    ▷ set block value
        for $i \leftarrow f, t$ do                            ▷ loop “from” downto “to”
            $x_i \leftarrow x$                                ▷ set all elements equal to block value
        end for
        $f \leftarrow t - 1$                                  ▷ set new “from” equal to old “to” minus one
    end for
end procedure
```

3. Extensions

In several cases the pool-adjacent-violators algorithm is used as part of a larger algorithm. It is expected that a faster sub-algorithm also improves the speed of the entire algorithm. In this context we will discuss two algorithms: unimodal monotone regression and bivariate monotone regression.

3.1. Unimodal monotone regression

Unimodal monotone regression minimizes the same weighted least squares loss function already given in (1), but with the following, slightly different restriction:

$$x_1 \leq \ldots \leq x_{m-1} \leq x_m \geq x_{m+1} \geq \ldots \geq x_n,$$

indicating that the sequence increases first and then decreases. When the position of mode m, where increase changes in decrease, is known, we are left with an isotonic regression on $i = 1, \ldots, m-1$ and an antitonic regression on $i = m+1, \ldots, n$. For an unknown mode, we choose the one with the lowest error sum-of-squares, which by the way is not necessarily unique. The unimodal restriction is also referred to as an umbrella ordering and further discussions can be found in Barlow *et al.* (1972); Robertson *et al.* (1988); Geng and Shi (1990), and Stout (2008). The following example is from the **Iso** package (Turner 2020), which contains the function `ufit()`, and it clearly shows the increase, the decrease, and the mode (element number 8 with mode value 293.8).

```
R> y <- c(0.0, 61.9, 183.3, 173.7, 250.6, 238.1, 292.6, 293.8, 268.0, 285.9,
+    258.8, 297.4, 217.3, 226.4, 170.1, 74.2, 59.8, 4.1, 6.1)
R> library("monotone")
R> unimonotone(y)

[1]     0.000  61.900 178.500 178.500 244.350 244.350 292.600 293.800 277.525
[10] 277.525 277.525 277.525 221.850 221.850 170.100  74.200  59.800   5.100
[19]   5.100
```

Turner and Wollan state that "fitting an isotonic model with such a constraint is essentially trivial; one simply fits two separate models with linear order constraints." (Turner and Wollan 1997, page 308). Turner (2020) uses `pava()` for the separate models, whereas Geng and Shi (1990) use `amalgm()` (Cran 1980). However, minimizing (1) for all modes $m = 1, \ldots, n$, including corresponding antitonic regressions, takes $O(n^2)$ time (Stout 2000, Section 2), although concerning time Geng and Shi (1990) state: "Not much time is consumed, but it is worth noting that the execution time increases linearly with K increasing" (here, K represents the length of the vector, i.e., n).

The prefix approach proposed by Stout (2008) and implemented in the package **UniIsoRegression** (Xu, Sun, Karunakaran, and Stout 2017) reduces the calculations for unimodal isotonic regression to $O(2n)$, using `reg_1d_l2()` to fit the separate, isotonic and antitonic, models. Instead of fitting an isotonic and an antitonic regression for each $m = 1, \ldots, n$, only one of each regressions is needed. The information contained in the vectors with block values and block sizes (see Stout 2008, Figure 5 and Figure 7) suffices to rapidly rebuild the monotone regressions for one specific mode m.

The current k-up-k-down approach, as described in Section 2, uses indices to identify blocks and collects all block values at the origin of the data vector (see Algorithm 1). Although such an approach makes it unnecessary to allocate an extra vector for the blocks, at the same time it becomes impossible to rebuild the regressions for one single mode m afterwards. This can easily be solved by using block sizes instead of indices and leaving the block values in place. This recover-afterwards-possibility approach corresponds not only to the prefix approach of Stout (2008), but also to the `_inplace_contiguous_isotonic_regression` procedure of Varoquaux, Tulloch, and Lee (2016). The general scheme of the unimodal algorithm follows the procedure described by Stout (2008).

The procedure is implemented in the function `unimonotone()` in the **monotone** package. Note that since the mode value x_m corresponds to an actual data vector value y_m (see, for example, Stout 2008), the mode cannot be a pooled element or combined block value. This means that candidate mode values emerge when the condition $\widehat{x} > x$ (Algorithm 1, Line 11) is actually false, in which case the error sum-of-squares needs to be saved for future use. This also means that the block size for the candidate mode will be equal to 1. Since the error sum-of-squares is only needed in the prescribed case, it suffices to keep track of the total sum-of-squares and the regression sum-of-squares and update the error sum-of-squares based on these two sums progressively.

3.2. Bivariate monotone regression

In some monotone regression problems we have more than one independent variable. When there are only two variables this is referred to as bivariate monotone regression. In that case, the data consists of a matrix and a solution is sought where both rows and columns are monotonically increasing. The following example is from Dykstra (1981) and the solution is given by `bimonotone()` (**monotone** package).

```
R> G <- matrix(c( 1, 5.2, 0.1, 0.1, 5, 0, 6, 2, 3, 5.2, 5, 7, 4, 5.5,
+    6, 6), 4, 4)
R> print(G)

     [,1] [,2] [,3] [,4]
[1,]  1.0    5  3.0  4.0
[2,]  5.2    0  5.2  5.5
[3,]  0.1    6  5.0  6.0
[4,]  0.1    2  7.0  6.0

R> library("monotone")
R> bimonotone(G)

     [,1] [,2] [,3] [,4]
[1,]  1.0  2.5  3.0  4.0
[2,]  1.8  2.5  5.1  5.5
[3,]  1.8  4.0  5.1  6.0
[4,]  1.8  4.0  6.5  6.5
```

In an iterative algorithm to solve the bivariate monotone regression problem, simple univariate monotone regression is used as a subroutine. The algorithm, due to Dykstra and

Implementation	Year	Original language	Allows weights	Auxiliary floats	Auxiliary integers	Direct updates	Forward checks	Backward checks
`fitm()`	1964	Fortran	Yes	n	n	No	1	k
`wmrmnh()`	1978	Fortran	Yes	-	-	Yes	1	k
`amalgm()`	1980	Fortran	Yes	$2n$	-	No	1	k
`pav()`	1984	Fortran	Yes	$4n$	n	No	1	0
`isoreg()`	1995	C	No	n	-	Yes	k	0
`isopava()`	1997	Fortran	Yes	-	n	Yes	1	0
`isotonic()`	2001	C	No	-	-	Yes	k	0
`isomean()`	2010	C	Yes	$2n$	n	No	0	k
`pooledpava()`	2013	Python	Yes	$2n$	n	No	0	k
`linearpava()`	2014	Julia	Yes	-	-	Yes	k	0
`inplacepava()`	2016	Python	Yes	-	n	No	1	k
`mdpava()`	2016	C++	No	n	n	No	1	k
`reg1dl2()`	2017	C++	Yes	$3n$	$2n$	No	0	k
`jbkpava()`	2017	C	Yes	$2n$	$3n$	No	1	k
`monotone()`	2019	C	Yes	-	n	No	k	k

Table 2: Overview of implementations (year = first year of publication; original language = original implementation language; allows weights = weighted or unweighted implementation; auxiliary floats = additional memory allocation for floating point numbers; auxiliary integers = additional memory allocation for integers, used for indices or block sizes; direct updates = continuous fully updated result vector; forward checks = none (0-up), single (1-up), or multiple (k-up); backward checks = none (0-down) or multiple (k-down)).

Robertson (1982) and, independently, to Sasabuchi, Inutsuka, and Kulatunga (1983), improves significantly on earlier algorithms proposed by Gebhardt (1970) and Dykstra (1981) (cf. Dykstra and Robertson 1982) and only needs to solve univariate monotone regression problems repeatedly along rows and columns until convergence. The algorithm extends naturally to more than two variables, in which case each iteration consists of more than two sub-cycles, something we are not going to pursuit here.

An implementation, published as Algorithm AS 206 (Bril *et al.* 1984), with corrections, is available as `biviso()` from **Iso**. The Fortran code uses `pav()` as the univariate monotone regression subroutine. The simple order case in the algorithm of Sasabuchi *et al.* (1983) is solved using the procedure `amalgm()` of Cran (1980). Both `pav()` and `amalgm()` are introduced in the next section.

An entirely different approach, but not pursuit here due to several reasons, is taken by Stout (2015). It is based on an order preserving embedding into a directed acyclic graph (DAG) and compared to previously mentioned algorithms it is claimed to improve running time by a significant factor. The procedure is implemented in **UniIsoRegression** as function `reg_2d()`.

4. Comparisons

In this section, we will compare the current implementation with other PAVA implementations. Many implementations of the pool-adjacent-violators algorithm can be found and most are some variant of the up-and-down-blocks algorithm, mainly written in C, Fortran, Java, Julia, Python, or R.

Table 2 provides a listing of all participating implementations[2]. For a proper comparison, all implementations have been translated into plain C (when needed) to avoid differences due to compiler characteristics. The implementations are provided in the `legacy()` function of the package **monotone**, accompanying this snippet.

Algorithm suggestions without a clear implementation (e.g., Gebhardt 1970), or implementations too deeply nested in other software (e.g., **Weka** by Witten, Frank, Hall, and Pal 2016), or implementations that could not be translated into C without a certain amount of force (e.g., `gpava()` by De Leeuw *et al.* 2009, `pava()` by Raubertas 1994, contained in **SAGx** Broberg 2020, or `pava()` contained in lecture notes on isotonic regression Geyer 2020), were not taken into account.

An historical overview and characterization of each implementation listed in Table 2, as well as a comparison with the original function in either R, Fortran, or C, if available, is given in Appendix A.

Now, the simulation study consists of two stages[3]. In the first stage, all implementations are included in the comparison in order to distinguish the $O(n)$ implementations from the $O(n^2)$ implementations. In the second stage, only the $O(n)$ implementations are compared on speed and ranked using different random data vectors.

4.1. Stage 1: Distinguishing $O(n)$ and $O(n^2)$ implementations

From the implementation decisions in Section 2, the overview of implementations in Table 2, and the characterization given in Appendix A, it is clear that some implementation might be fast when the vector is already (almost) in the right order, for example `wmrmnh()` or `isotonic()`, but not in the opposite direction (almost complete disorder). Other implementations might just be fast in this opposite direction, e.g., `pooledpava()` or `isoreg()`, and not in the ordered situation. One situation that is not especially favorable for any implementation given in Table 2 is the following:

```
set_data_vector <- function(n) {
  half <- n / 2
  return(c(1:half, half:1))
}
```

To distinguish $O(n)$ and $O(n^2)$ implementations, it suffices to use this (nonrandom) data vector with four different vector sizes: $n = 100, 1000, 10000, 100000$.

The package **microbenchmark** (Mersmann 2021) is used to time the implementations. Each implementation run is evaluated 100 times by `microbenchmark()` in random order with 2 warm-up evaluations. We will consider average timing performance only[4].

Table 3 shows the results of the simulation study. As expected, the timings increase with either n or n^2. The last column clearly shows which implementations are slowest. Comparing the last to the second to last column for these implementations shows a close to 100 times increase in timings while the data vector size is only tenfold. For example, `isoreg()` increases

[2] The weights w_i in (1) are set to one, in order to allow `isoreg()`, `isotonic()`, and `mdpava()` to compete.

[3] All simulations have been run on an Apple Macbook Pro with an Intel(R) Core(TM) i9-9880H CPU @ 2.30GHz, running the 64-bit Windows 10 Operating System.

[4] Full results, including best and worst timings, are provided as supplementary materials.

Implementation	$n = 100$	$n = 1000$	$n = 10000$	$n = 100000$
`fitm()`	4.690	15.932	159.580	4170.953
`wmrmnh()`	5.202	134.934	12971.086	1963115.876
`amalgm()`	5.458	103.358	9564.534	1081874.874
`pav()`	8.327	299.754	29117.754	3050869.166
`isoreg()`	7.525	296.066	28054.968	2782377.579
`isopava()`	18.237	1047.448	108726.536	11003871.990
`isotonic()`	8.616	380.128	36798.018	3688134.227
`isomean()`	4.692	14.420	112.229	1970.113
`pooledpava()`	4.786	15.522	123.404	1952.878
`linearpava()`	7.780	294.595	29512.901	2994164.059
`inplacepava()`	4.547	13.169	135.845	1443.377
`mdpava()`	4.629	14.958	185.757	1515.005
`reg1dl2()`	4.857	15.184	120.942	2323.007
`jbkpava()`	5.230	56.540	217.683	2291.755
`monotone()`	3.734	9.859	73.557	1141.088

Table 3: Mean timing performance in microseconds for the 15 PAVA implementations (see Table 2) for the nonrandom data vector and four different data vector sizes.

in time from 28 milliseconds for $n = 10000$ to 2782 milliseconds for $n = 100000$, while, for example, `jbkpava()` only increases from 0.217 milliseconds to 2.291 milliseconds for the same vector sizes, respectively. It is more difficult to deduce this conclusion based on the first two columns, probably due to a lack of precision or some code timings independent of n or n^2. Hereafter these slowest implementations, i.e., `wmrmnh()`, `amalgm()`, `pav()`, `isoreg()`, `isopava()`, `isotonic()`, and `linearpava()`, will be referred to as the $O(n^2)$ implementations.

When we compare the $O(n^2)$ implementations to the design decisions shown in Table 2, we notice that all implementations that use direct updates are among the $O(n^2)$ implementations, i.e., `wmrmnh()`, `isoreg()`, `isotonic()`, `isopava()`, and `linearpava()`. This observation confirms Grotzinger and Witzgall's claim that direct updates lead to an $O(n^2)$ implementation. Further, the implementations that only have one single upwards violation check (1-up-0-down), i.e., `pav()` and `isopava()`, also belong to the $O(n^2)$ implementations. Besides, `isopava()` falls in both categories, direct updates and 1-up-0-down, which is clearly illustrated by the highest mean timings for all vector sizes. Although `amalgm()` does not belong to the just mentioned categories, this implementations has one serious delaying factor: the update scheme shifts the whole vector backwards on every single pool operation, quite a time consuming process.

4.2. Stage 2: Comparing $O(n)$ implementations

For the $O(n)$ implementations timing experiment, we use five *random* data vectors (see Table 4), scaled between 0 and 10. An impression for $n = 100$ is given in Figure 1. We use the same vector sizes as before, which brings the total number of cells to $5 \times 4 = 20$. Each cell is replicated 100 times, with different random error for each replication. We will consider the average performances for vector size and data vectors, respectively, of all $O(n)$ implementations[5], and let `microbenchmark()` use the **multicomp** package (Hothorn, Bretz, and

[5] Full results are provided as supplementary materials.

Description	Equation	R code
Order	$x_i = i$	`x <- 1:n`
Sinus order	$x_i = 5i/n + \sin(10i/n)$	`x <- 5 * (1:n) / n + sin(10 * (1:n) / n)`
No order	$x_i = 5$	`x <- rep(5, n)`
Sinus disorder	$x_i = n - 5i/n + \sin(10i/n)$	`x <- n - 5 * (1:n) / n + sin(10 * (1:n) / n)`
Disorder	$x_i = n - i + 1$	`x <- n:1`
scaling between 0 and 10		`x <- x - min(x); x <- x / max(x); x <- 10 * x`
standard normal error		`x <- x + rnorm(n)`

Table 4: Overview of random data vector variants, scaling of data, and adding random error.

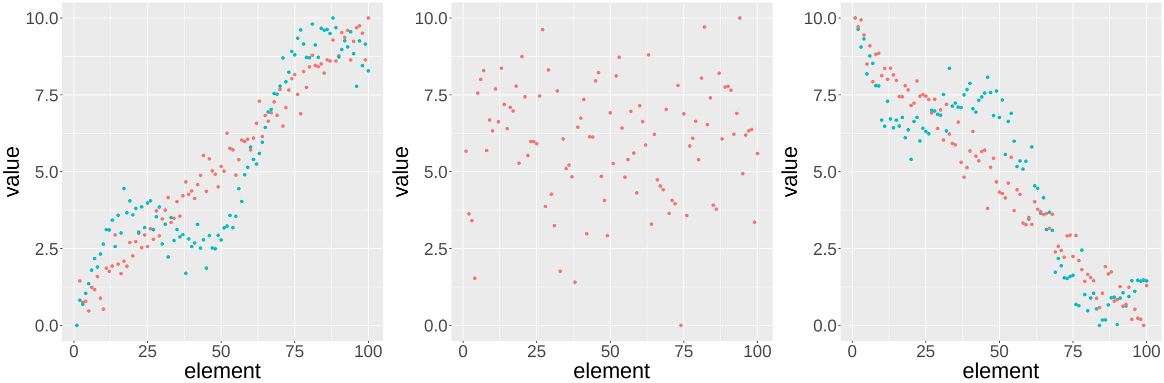

Figure 1: Overview of data vector variants: order (left panel, red dots), sinus order (left panel, green dots), no order (middle panel), sinus disorder (right panel, green dots), and disorder (right panel, red dots).

	$n = 100$		$n = 1000$		$n = 10000$		$n = 100000$	
Implementation	Mean	cld	Mean	cld	Mean	cld	Mean	cld
`fitm()`	4.324	c	20.071	g	178.634	c	1909.611	e
`isomean()`	4.363	e	20.158	h	184.779	cd	2003.587	f
`pooledpava()`	4.425	f	17.859	d	163.636	b	1806.446	d
`inplacepava()`	4.132	b	15.733	c	153.224	b	1654.275	b
`mdpava()`	4.346	d	18.690	e	162.874	b	1737.621	c
`reg1dl2()`	4.584	g	15.056	b	126.953	a	1923.846	e
`jbkpava()`	4.686	h	19.659	f	193.563	d	2432.005	g
`monotone()`	3.425	a	12.530	a	118.900	a	1322.806	a

Table 5: Mean timing performance in microseconds for the $O(n)$ implementations on four different vector sizes, augmented with a statistical ranking in compact letter display (cld).

Westfall 2021) for multiple comparisons and a statistical ranking, displayed in compact letter display (cld, Piepho 2004).

Table 5 shows the results for all vector sizes, averaged over data vector variants. Seemingly there is only little difference in the timings. Multiple comparisons indicate nevertheless that `monotone()` is significantly faster in all cases.

Table 6 shows the results for the five data vector variants from Table 4, as well as an overall mean and a ranking. The timings are averaged over all data vector sizes, taking into account

	Order		Sinus order		No order		Sinus disorder		Disorder			
Implementation	Mean	cld	Mean	cld	Mean	cld	Mean	cld	Mean	cld	Overall	Rank
`fitm()`	28.059	d	26.707	d	28.233	d	21.930	c	20.409	b	25.068	6
`isomean()`	28.238	d	26.792	d	27.880	d	23.084	d	21.881	cd	25.575	7
`pooledpava()`	25.389	c	24.606	c	25.087	c	23.004	d	22.582	de	24.134	5
`inplacepava()`	23.783	b	23.240	b	23.554	b	20.876	b	19.694	b	22.229	2
`mdpava()`	25.247	c	24.780	c	25.618	c	22.430	cd	21.689	c	23.953	4
`reg1dl2()`	23.366	b	23.352	b	23.432	b	23.125	d	22.756	e	23.206	3
`jbkpava()`	28.448	d	28.349	e	28.294	d	26.695	e	25.963	f	27.550	8
`monotone()`	20.008	a	19.250	a	19.847	a	15.852	a	14.914	a	17.974	1

Table 6: Mean timing performance in nanoseconds for the $O(n)$ implementations on the five data vector variants from Table 4, augmented with a statistical ranking in compact letter display (cld) and a final ranking.

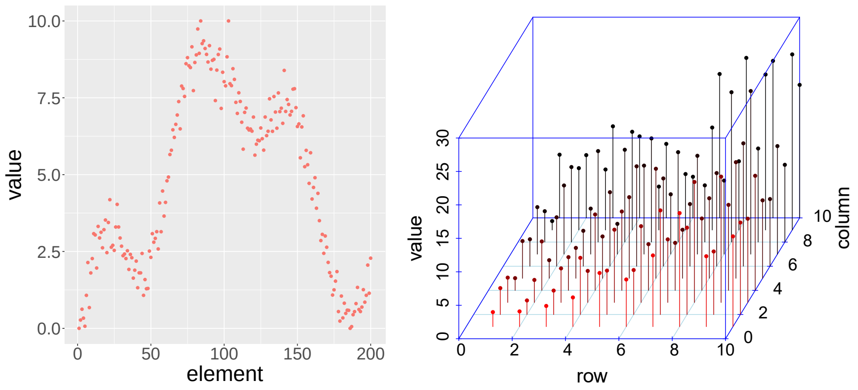

Figure 2: Data used for unimodal (left hand side) and bivariate (right hand side) monotone regression simulations.

the vector sizes, that is, the timing is divided by the vector size before averaging. The results thus indicate the number of nanoseconds it takes to process one single data vector element.

It is striking how little difference there seemingly is between the different data vector variants. Apparently, for these $O(n)$ implementations, it takes between 15 and 29 nanoseconds to process one single element, almost irrespective of the constitution of the data vector. This is not something to be said from the $O(n^2)$ implementations, that can differ markedly for the different data vector variants. We can, nevertheless, observe that (sinus) disordered vectors take less time for all implementations, improving on their ordered variant by 13% on average.

Considering the weights, the results confirm the statement of Stout (2019) that weights do not play an important role for L_2 implementations. Although `isoreg()` and `isotonic()` belong to the $O(n^2)$ implementations, on different grounds than lacking weights, `mdpava()` does not. The latter resides in the middle of the $O(n)$ implementations using weights.

A special remark is in order for `fitm()` (Kruskal 1964b): it is the oldest implementation, apparently slightly messy coded, but still extremely fast. In the end, `monotone()` is more than 19% faster than the next implementation (`inplacepava()`) and more than 34% faster than the slowest $O(n)$ implementation (`jbkpava()`), not to mention the speed differences with the $O(n^2)$ implementations.

4.3. Extensions

In a small simulation study, we compare the three different implementations of unimodal monotone regression, i.e., `Iso::ufit()` (Turner 2020), `UniIsoRegression::d1_l2()` (Xu *et al.* 2017), based on Stout (2008), and the current implementation, `unimonotone()`. We use a concatenation of the random data vector variants sinus order and sinus disorder given in Table 4. An example data vector is displayed in Figure 2 (left hand side). Each $n = 1000$ vector is replicated 100 times, with different random error for each replication, and `microbenchmark()` `(times = 100, check = "equivalent", control = list(warmup = 2))` is used for the timings.

```
Unit: microseconds
                 expr      min       lq         mean   median       uq
               ufit() 779090.1 806475.3 810007.01363 810962.6 814629.0
 uniisoregression()     37.0     46.7     61.00116     61.4     72.4
        unimonotone()     14.0     20.8     27.98501     27.7     34.1
      max neval cld
 991932.2 10000   b
    196.1 10000  a
    113.1 10000  a
```

The timing results show that `ufit()` is, by far, the slowest procedure and for two reasons: it solves $2n$ regression problems each time and uses an $O(n^2)$ procedure for each problem. Since Geng and Shi (1990) use a similar procedure, it falsifies their statement on time, as using an $O(n^2)$ procedure does consume time and execution time increases quadratically with n. The difference between `UniIsoRegression::d1_l2()` and `unimonotone()` is not significant, although that is due to the high timings of `Iso::ufit()`. Practically speaking, `unimonotone()` is more than twice as fast as `UniIsoRegression::d1_l2()`. The difference can be explained by the faster monotone regression procedure (cf. Sections 4.1–4.2) for both isotonic and antitonic regressions and by the selective and progressive updating of the error sum-of-squares.

For the bivariate monotone regression case, we compare the `biviso()` and `bimonotone()` functions from the **Iso** and **monotone** packages, respectively[6]. Both implementations use the simple univariate implementation as a subroutine, i.e., `pav()` and `monotone()`, respectively. The simulated 32×32 data matrix $\mathbf{G}$ with $n \approx 1000$ is generated as $g_{ij} = i + j + r$, where error r is an uniform random number between $-i$ and j. An example 10×10 data matrix is displayed in Figure 2. Each matrix is replicated 100 times, with different random error for each replication, and `microbenchmark()` `(times = 100, control = list(warmup = 2))` is used for the timings.

```
Unit: milliseconds
                     expr   min    lq  mean median    uq   max neval cld
                 biviso() 7.249 7.311 7.465  7.438 7.589 8.708   100   c
     bimonotone(default) 3.052 3.091 3.164  3.124 3.188 3.437   100  b
 bimonotone(eps=1.0e-5) 1.989 2.010 2.058  2.022 2.057 2.422   100 a
```

[6] Although indeed seriously faster, `reg_2d()` from **UniIsoRegression** based on Stout (2015) is omitted from the comparison due to a different algorithmic approach and due to different, probably invalid, results.

It is clear from the **microbenchmark** results that `bimonotone()` is significantly faster, although both procedure are quite slow, considering the timings are in milliseconds, not in microseconds. Compared to the original implementation almost forty years ago, the precision has increased considerably, namely from four significant digits (cf. Dykstra and Robertson 1982) to eight significant digits (cf. Turner 2020)[7]. Decreasing precision to the original status reduces the time from 3.164 to 2.058 milliseconds. It is expected that the implementation of Sasabuchi *et al.* (1983) is faster than `biviso()` because `amalgm()` is faster than `pav()`, just as that `bimonotone()` is faster than the other implementations, because `monotone()` is faster.

5. Conclusion

An $O(n)$ algorithm does not imply an $O(n)$ implementation, nor does it imply a fast implementation. Many skillfully implemented versions of the pool-adjacent-violators algorithm have preceded the current implementation, some of them with computational complexity $O(n)$ or close, some of them fast, some of them both.

A close inspection of the up-and-down-blocks algorithm revealed that there was still something to be gained: an important acceleration occurs during the sequential handling of the up-blocks checks. In case of a violation, continued up-block checks or lookaheads are performed first, accelerating the overall sequence by cutting down on some operations, before initiating the common series of down-block amalgamations.

For the comparison of implementations in terms of speed, we have used **microbenchmark**. We could also have looked at the number of operations, like multiplication, division, addition, assignment, and comparison, an implementation has to perform to get to the answer. Just counting the number of different operations confirms the working ingredient of the lookahead procedure and reveals why `monotone()` is fast: it simply does not duplicate any operation. For example, `isopava()` needs 56928 operations (in total) for an $n = 100$ non-random data vector variant (see Section 4.1), while `monotone()` only needs 1950, which is only about 3.5%.

In everyday use, such as part of an iterative algorithm, e.g., an alternating non-metric multidimensional scaling algorithm, there will be little difference between `monotone()` and the next $O(n)$ implementation. However, for heavily repeated use, like in unimodal or bivariate monotone regression procedures, or for large problem sizes, it is unwise to consider slower procedures as an option when speed is an issue.

Low-level optimizations, or cleverly constructed assembly code, might even improve on the current version, but that might reduce the readability, simplicity, elegance, and most certainly the transferability of the current implementation.

Acknowledgments

This paper could not have existed lacking the inspiration and feedback of Jan de Leeuw and his thesis student Ernst van Waning. The author also wishes to thank the two anonymous reviewers for their extensive and supportive reports, and Willem Heiser, Maximilien Danisch,

[7] `microbenchmark()` considered the results *not equivalent* using decreased precision, so that option was turned off for the simulation.

Nelle Varoquaux, Tom Kincaid, Rolf Turner, Korbinian Strimmer, and Jacqueline Meulman for their constructive feedback on an earlier (partly) draft of the snippet, Juan Claramunt Gonzalez for the creation of and help with the **monotone** package, and finally both Boris and Lexy for their considerable support during the testing of the implementations and the writing of the snippet.

References

Ayer M, Brunk HD, Ewing GM, Reid WT, Silverman E (1955). "An Empirical Distribution Function for Sampling with Incomplete Information." *The Annals of Mathematical Statistics*, pp. 641–647. doi:10.1214/aoms/1177728423.

Balabdaoui F, Rufibach K, Santambrogio F (2011). ***OrdMonReg**: Compute Least Squares Estimates of One Bounded or Two Ordered Isotonic Regression Curves.* R package version 1.0.3, URL https://CRAN.R-project.org/package=OrdMonReg.

Barlow RE, Bartholomew DJ, Bremner JM, Brunk HD (1972). *Statistical Inference under Order Restrictions: The Theory and Application of Isotonic Regression.* 1st edition. John Wiley & Sons.

Bartholomew DJ (1961). "A Test of Homogeneity of Means under Restricted Alternatives." *Journal of the Royal Statistical Society B*, **23**(2), 239–272. doi:10.1111/j.2517-6161.1961.tb00410.x.

Bentley JL (1976). *Divide and Conquer Algorithms for Closest Point Problems in Multidimensional Space.* Ph.D. thesis, University of North Carolina at Chapel Hill. URL https://www.cs.unc.edu/techreports/76-103.pdf.

Best MJ, Chakravarti N (1990). "Active Set Algorithms for Isotonic Regression; A Unifying Framework." *Mathematical Programming*, **47**(1-3), 425–439. doi:10.1007/bf01580873.

Bril G, Dykstra R, Pillers C, Robertson T (1984). "Algorithm AS 206: Isotonic Regression in Two Independent Variables." *Journal of the Royal Statistical Society C*, **33**(3), 352–357. doi:10.2307/2347723.

Broberg P (2020). ***SAGx**: Statistical Analysis of the GeneChip.* R package version 1.64.0, URL https://www.bioconductor.org/packages/3.12/bioc/html/SAGx.html.

Busing FMTA, Claramunt Gonzalez J (2022). ***monotone**: Performs Monotone Regression.* R package version 0.1.2, URL https://CRAN.R-project.org/package=monotone.

Busing FMTA, Commandeur JJF, Heiser WJ (1997). "PROXSCAL: A Multidimensional Scaling Program for Individual Differences Scaling with Constraints." In W Bandilla, F Faulbaum (eds.), *Softstat '97 Advances in Statistical Software*, pp. 237–258. Lucius, Stuttgart, Germany.

Chatterjee S, Guntuboyina A, Sen B (2015). "On Risk Bounds in Isotonic and Other Shape Restricted Regression Problems." *The Annals of Statistics*, **43**(4), 1774–1800. doi:10.1214/15-aos1324.

Cran GW (1980). "Algorithm AS 149: Amalgamation of Means in the Case of Simple Ordering." *Journal of the Royal Statistical Society C*, **29**(2), 209–211. doi:10.2307/2986312.

Danisch M (2016). "C Implementation of the Pool Adjacent Violators Algorithm for Isotonic Regression." GitHub, URL https://github.com/maxdan94/pava.

Danisch M, Chan THH, Sozio M (2017). "Large Scale Density-Friendly Graph Decomposition via Convex Programming." In *Proceedings of the 26th International Conference on World Wide Web*, pp. 233–242. International World Wide Web Conferences Steering Committee. URL https://i.cs.hku.hk/~hubert/www17.pdf.

De Leeuw J (1977). "Correctness of Kruskal's Algorithms for Monotone Regression with Ties." *Psychometrika*, **42**(1), 141–144. doi:10.1007/bf02293750.

De Leeuw J (2016). "Exceedingly Simple Isotone Regression with Ties." URL https://www.researchgate.net/profile/Jan_De_Leeuw/publication/291165786_Exceedingly_Simple_Isotone_Regression_with_Ties/links/569ea8c208aee4d26ad043d2.pdf.

De Leeuw J (2017). "Exceedingly Simple Monotone Regression." URL https://www.researchgate.net/publication/315744239_Exceedingly_Simple_Monotone_Regression.

De Leeuw J, Hornik K, Mair P (2009). "Isotone Optimization in R: Pool-Adjacent-Violators Algorithm (PAVA) and Active Set Methods." *Journal of Statistical Software*, **32**(5), 1–24. doi:10.18637/jss.v032.i05.

De Leeuw J, Mair P (2009). "Gifi Methods for Optimal Scaling in R: The Package **homals**." *Journal of Statistical Software*, **31**(4), 1–20. doi:10.18637/jss.v031.i04.

Dumelle M, Kincaid T, Olsen T, Weber M (2022). ***spsurvey**: Spatial Sampling Design and Analysis.* R package version 5.3.0, URL https://CRAN.R-project.org/package=spsurvey.

Dykstra RL (1981). "An Isotonic Regression Algorithm." *Journal of Statistical Planning and Inference*, **5**(4), 355–363. doi:10.1016/0378-3758(81)90036-7.

Dykstra RL, Robertson T (1982). "An Algorithm for Isotonic Regression for Two or More Independent Variables." *The Annals of Statistics*, **10**(3), 708–716. doi:10.1214/aos/1176345866.

Eggermont PPB, LaRiccia VN (2000). "Maximum Likelihood Estimation of Smooth Monotone and Unimodal Densities." *The Annals of Statistics*, **28**(3), 922–947. doi:10.1214/aos/1015952005.

Gebhardt F (1970). "An Algorithm for Monotone Regression with One or More Independent Variables." *Biometrika*, pp. 263–271. doi:10.1093/biomet/57.2.263.

Geng Z, Shi NZ (1990). "Algorithm AS 257: Isotonic Regression for Umbrella Orderings." *Journal of the Royal Statistical Society C*, **39**(3), 397–402. doi:10.2307/2347399.

Geyer CJ (2020). "Stat 8045 Lecture Notes: Isotonic Regression." URL http://www.stat.umn.edu/geyer/8054/notes/isotonic.html.

Gifi A (1990). *Nonlinear Multivariate Analysis*. 1st edition. John Wiley & Sons.

Grotzinger SJ, Witzgall C (1984). "Projections onto Order Simplexes." *Applied Mathematics and Optimization*, **12**(1), 247–270. doi:10.1007/bf01449044.

Guttman L (1968). "A General Nonmetric Technique for Finding the Smallest Coordinate Space for a Configuration of Points." *Psychometrika*, **33**(4), 469–506. doi:10.1007/bf02290164. URL https://link.springer.com/article/10.1007/BF02290164.

Hothorn T, Bretz F, Westfall P (2021). ***multcomp**: Simultaneous Inference in General Parametric Models*. R package version 1.4-16, URL https://CRAN.R-project.org/package=multcomp.

Klaus B, Strimmer K (2021). "**fdrtool**: Estimation of (Local) False Discovery Rates and Higher Criticism." R package version 1.2.17, URL https://CRAN.R-project.org/package=fdrtool.

Klinglmueller F (2010). ***orQA**: Order Restricted Assessment Of Microarray Titration Experiments*. R package version 0.2.1, URL https://CRAN.R-project.org/package=orQA.

Kruskal JB (1964a). "Multidimensional Scaling by Optimizing Goodness of Fit to a Nonmetric Hypothesis." *Psychometrika*, **29**(1), 1–27. doi:10.1007/bf02289565.

Kruskal JB (1964b). "Nonmetric Multidimensional Scaling: A Numerical Method." *Psychometrika*, **29**(2), 115–129. doi:10.1007/bf02289694.

Kyng R, Rao A, Sachdeva S (2015). "Fast, Provable Algorithms for Isotonic Regression in All L_p-Norms." In *Advances in Neural Information Processing Systems*, pp. 2719–2727. URL http://papers.nips.cc/paper/5824-fast-provable-algorithms-for-isotonic-regression-in-all-l_p-norms.pdf.

Mair P, De Leeuw J, Groenen PJF (2019). ***Gifi**: Multivariate Analysis with Optimal Scaling*. R package version 0.3-9, URL https://CRAN.R-project.org/package=Gifi.

Mair P, Groenen PJF, De Leeuw J (2022). *More on Multidimensional Scaling in R: **smacof** Version 2*. doi:10.18637/jss.v102.i10.

Mersmann O (2021). ***microbenchmark**: Accurate Timing Functions*. R package version 1.4-9, URL https://CRAN.R-project.org/package=microbenchmark.

Miles RE (1959). "The Complete Amalgamation into Blocks, by Weighted Means, of a Finite Set of Real Numbers." *Biometrika*, **46**, 317–327. doi:10.1093/biomet/46.3-4.317.

Pace NL, Stylianou MP, Warltier DC (2007). "Advances in and Limitations of Up-and-down Methodology: A Précis of Clinical Use, Study Design, and Dose Estimation in Anesthesia Research." *Anesthesiology*, **107**(1), 144–152. doi:10.1097/01.anes.0000267514.42592.2a.

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011). "**scikit-learn**: Machine Learning in Python." *Journal of Machine*

Learning Research, **12**, 2825–2830. URL https://jmlr.org/papers/v12/pedregosa11a.html.

Piepho HP (2004). "An Algorithm for a Letter-Based Representation of All-Pairwise Comparisons." *Journal of Computational and Graphical Statistics*, **13**(2), 456–466. doi:10.1198/1061860043515.

Raubertas RF (1994). "Pooled Adjacent Violators Algorithm (PAVA) Implemented Entirely in S."

R Core Team (2022). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Reboul L (2005). "Estimation of a Function under Shape Restrictions. Applications to Reliability." *The Annals of Statistics*, **33**(3), 1330–1356. doi:10.1214/009053605000000138.

Ripley BD (2003). isoreg.c*: Isotonic/Monotone Regression*. R package **stats**, URL http://svn.R-project.org/R/trunk/src/library/stats/src/isoreg.c.

Robertson T (1978). "Testing for and against an Order Restriction on Multinomial Parameters." *Journal of the American Statistical Association*, **73**(361), 197–202. doi:10.1080/01621459.1978.10480028.

Robertson T, Wright FT, Dykstra RL (1988). *Order Restricted Statistical Inference*. Wiley Series in Probability and Mathematical Statistics, 1st edition. John Wiley & Sons, New York.

Sasabuchi S, Inutsuka M, Kulatunga DDS (1983). "A Multivariate Version of Isotonic Regression." *Biometrika*, **70**(2), 465–472. doi:10.1093/biomet/70.2.465.

Stoop I, De Leeuw J (1982). "How to Use SMACOF-IB (A Complete User's Guide)." *Technical report*, Leiden University, Leiden.

Stout QF (2000). "Optimal Algorithms for Unimodal Regression." *Computing Science and Statistics*, **32**, 1–8.

Stout QF (2008). "Unimodal Regression via Prefix Isotonic Regression." *Computational Statistics & Data Analysis*, **53**, 289–297. doi:10.1016/j.csda.2008.08.005.

Stout QF (2015). "Isotonic Regression for Multiple Independent Variables." *Algorithmica*, **71**(2), 450–470. doi:10.1007/s00453-013-9814-z.

Stout QF (2019). "Fastest Known Isotonic Regression Algorithms." URL https://web.eecs.umich.edu/~qstout/IsoRegAlg.pdf.

Strimmer K (2008). "**fdrtool**: A Versatile R Package for Estimating Local and Tail Area-Based False Discovery Rates." *Bioinformatics*, **24**(12), 1461–1462. doi:10.1093/bioinformatics/btn209.

Tatti N, Gionis A (2015). "Density-Friendly Graph Decomposition." In *Proceedings of the 24th International Conference on World Wide Web*, pp. 1089–1099.

Tibshirani R, Suo X (2016). "An Ordered Lasso and Sparse Time-Lagged Regression." *Technometrics*, **58**(4), 415–423. doi:10.1080/00401706.2015.1079245.

Tulloch A (2014a). *Isotonic Regression in Julia.* URL https://github.com/ajtulloch/Isotonic.jl.

Tulloch A (2014b). *Speeding up Isotonic Regression in **scikit-learn** by* 5000×. URL http://tullo.ch/articles/speeding-up-isotonic-regression/.

Turner R (2020). ***Iso**: Functions to Perform Isotonic Regression.* R package version 0.0-18.1, URL https://CRAN.R-project.org/package=Iso.

Turner TR, Wollan PC (1997). "Locating a Maximum Using Isotonic Regression." *Computational Statistics & Data Analysis*, **25**(3), 305–320. doi:10.1016/s0167-9473(97)00009-1.

Van Eeden C (1958). *Testing and Estimating Ordered Parameters of Probability Distribution.* Ph.D. thesis, University of Amsterdam. URL https://ir.cwi.nl/pub/23611/23611A.pdf.

Van Waning WE (1976). *A Set of Programs to Perform a Kruskal-Type Monotone Regression.* Master's thesis, Leiden University, Leiden.

Varoquaux N, Tulloch A, Lee A (2016). "Inplace Contiguous Isotonic Regression." **scikit-learn** implementation of isotonic regression, URL https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/_isotonic.pyx.

Venables WN, Ripley BD (2013). *Modern Applied Statistics with S-PLUS.* 3rd edition. Springer-Verlag.

Wikipedia (2022). "Isotonic Regression – Wikipedia, The Free Encyclopedia." URL https://en.wikipedia.org/wiki/Isotonic_regression, accessed 2022-03-22.

Witten IH, Frank E, Hall MA, Pal CJ (2016). *Data Mining: Practical Machine Learning Tools and Techniques.* 4th edition. Morgan Kaufmann Publishers, San Francisco.

Xu Z, Sun C, Karunakaran A, Stout Q (2017). ***UniIsoRegression**: Unimodal and Isotonic L_1, L_2 and L_∞ Regression.* R package version 0.0-0, URL https://CRAN.R-project.org/package=UniIsoRegression.

A. Legacy functions

The package **monotone** contains a function called `legacy()` that holds all PAVA implementations used in the comparison of Section 4. An overview of the implementations is provided in Table 2. In this appendix, we provide a description of the origin of the algorithm, provide basic references, and indicate algorithmic particularities. Given the original function, if available, we use the **microbenchmark** package to compare the original function to the legacy function (**microbenchmark** parameters: `times = 100, check = "equivalent", control = list(warmup = 2))`. These comparisons use all five data vector variants (see Table 4) with a vector length of $n = 1000$ and 100 replications, i.e., 100 different data vectors. The following list of implementations is in historical order as much as possible.

A.1. `fitm() : legacy(x, w, type = 1)`

The up-and-down-blocks implementation of the pool-adjacent-violators algorithm, written in Fortran by Kruskal, can be found in `mdscal()` (Version 5MS, October 1971, Unchanged from Version 4, January 1968) and later in `kyst()` (Modified for KYST by J. Kruskal and J. Seery, 1973). These versions deviate slightly from the original unweighted description in Kruskal (1964b). Both versions are quite similar and the actual PAVA algorithm is only a part of the subroutine `fitm()`, which computes a nonmetric or ordinal transformation using either primary or secondary approach to ties, that is, either untying tied values or keeping tied values tied, respectively. The implementation uses a vector with elements $w_i x_i$ preset to check for violations. Two large coding blocks handle either a down-block or an up-block violation. Only when both violations are dealt with, which is a maximum of one up-block coalescence and possibly more down-block fusions, the process is continued to the next, active, element. In the end, the result vector is created from the non-violating blocks. The original Fortran code is full of clever `goto`'s and inimitable memory management. Unfortunately, the implementation does not exist as a single function.

A.2. `wmrmnh() : legacy(x, w, type = 2)`

In Van Waning (1976), several PAVA implementations are compared with respect to memory requirements and speed. The subroutine `wmrmnh()` comes out as a winner (fast and without additional memory requirements, the latter being quite important in the Seventies) and has been implemented in `smacof-1b()` by van Waning and Stoop (Stoop and De Leeuw 1982). In 2014, the original Fortran code has been translated into C, after translations into languages like MATLAB and R in intermediate years, by Groenen and Van den Burg for the use in **smacof** (Mair, Groenen, and De Leeuw 2022). The clear and short implementation uses a simple check to start the recovering of violations. If the vector is already ordered and a few violations remain, this implementation is really fast. Unfortunately, if this is not the case, only solving down-block violations and, moreover, using direct updates, results in a somewhat slow implementation. The original Fortran code is still available (`subroutine wmrmnh`) and so is the derived C code implementation supplied with the **smacof** package in `wmonreg.c`. The original functions are significantly slower than the legacy function.

```
Unit: microseconds
               expr  min   lq     mean median    uq    max neval cld
 original (Fortran) 13.3 44.4 198.5744   72.3 399.9 2485.6 50000   b
       original (C) 14.4 45.7 199.7564   73.3 400.9 2761.1 50000   b
             legacy 16.5 47.0 161.2173   66.4 319.6  655.0 50000  a
```

A.3. `amalgm() : legacy(x, w, type = 3)`

The up-and-down-blocks algorithm was first described by Kruskal (1964b) and subsequently drawn up in a flow chart by Barlow *et al.* (1972). The implementation was published more than a decade later as Algorithm AS 149 by Cran (1980). The algorithm implements `amalgm()`, short for "the amalgamation of means", is written in ISO Fortran, and contains just a little less `goto`'s than Kruskal's original version. Like `fitm()`, this implementation uses two large coding blocks to take care of the violations. However, there is a lot more overhead in the calculation of block values in case of violations compared to `fitm()`. The continuous updating of the result vector in both coding blocks does not help to create a fast implementation. There is a small statistical difference between the original Fortran function and the legacy function in C in favor of the legacy function.

```
Unit: microseconds
     expr   min    lq     mean median    uq    max neval cld
 original 226.8 249.0 264.2632  256.7 280.8 5289.7 50000   b
   legacy 220.0 241.5 256.5742  249.6 272.4  479.7 50000  a
```

A.4. `pav() : legacy(x, w, type = 4)`

Bril *et al.* publish Algorithm AS 206: Isotonic Regression in Two Independent Variables, which implements the isotonic regression algorithm of Dykstra and Robertson (1982). Part of the algorithm, more specifically part 206.1, contains the subroutine `pav()`, an implementation of the pool-adjacent-violators algorithm, discussed by Barlow *et al.* (1972) (Bril *et al.* 1984), but nevertheless quite different from `fitm()` and `amalgm()`. The implementation allocates an abundance of memory for all possible vectors. For each violation, the complete vector is shifted to the left, merging the violator with the preceding block into one new block, before continuing with the next element. Only for completely ordered data or for data with only a few violations this implementation is fast. Before using the legacy function and comparing it with the original, a bug was removed to obtain proper results (for the enthusiast: the number of block elements, contained in `NW`, was lacking from the shift operation in the loop with label `40`). Both implementations are not very fast and the original function clearly outperforms the legacy function.

```
Unit: microseconds
     expr   min    lq     mean median    uq   max neval cld
 original 348.8 372.9 397.8627  386.7 411.8 861.4 50000  a
   legacy 344.0 417.9 451.5271  441.7 477.0 913.6 50000   b
```

A.5. `isoreg() : legacy(x, w, type = 5)`

In their book on Modern Applied Statistics with S, Venables and Ripley (2013) describe the function `isoMDS()`, which implements Kruskal's multidimensional scaling algorithm (Kruskal 1964b). The monotone transformation in `isoMDS()` is contained in the function `VR_mds_fn()`, written by Ripley in 1995, and later added as `isoreg()` (Ripley 2003) to the **stats** package in R (R Core Team 2022). It is one of the shortest implementations and uses the cumulative sum of the data elements to handle violation checks. After determining the smallest slope (violation), the violation is dealt with. For vectors in (almost) the right order, there is quite some redundancy due to unnecessary calculations, and the implementation is not very fast. Unfortunately, this implementation does not allow for weights, nor is the strategy with cumulative sums claimed to be suitable for a weighted implementation. Comparison of the original function with the legacy function reveals that there is indeed quite some overhead in `isoreg()` when it comes to simple isotonic regression.

```
Unit: microseconds
     expr  min   lq     mean median    uq   max neval cld
 original 32.0 35.4 87.39017   45.6 149.1 321.7 50000   b
   legacy  9.6 12.0 32.96522   16.5  57.6 147.7 50000  a
```

A.6. `isopava() : legacy(x, w, type = 6)`

In 1997, Turner and Wollan developed a technique for estimating the location of the maximum of a set of data by applying isotonic regression to unimodal orderings. Code to effect the necessary calculations was initially written in S-PLUS and later translated into R. In 2008, Turner incorporated the code into the **Iso** package, including the functions `pava()` (Fortran-based) and `pava.sa()` (R-based). We use the name `isopava()` to avoid name conflicts. The implementation resembles `wmrmnh()` by Van Waning (1976), but lacks the speedy updates as for each violation, the whole vector range is assessed, but only one up-block violations is solved. Combined with direct updates, this implementation is by far the slowest implementation for simple linear orderings. A comparison between the legacy function and the original function, the R function `pava()` containing the call to the compiled Fortran code, shows that the latter is significantly faster, although both functions are quite slow (note the measuring unit milliseconds instead of microseconds).

```
Unit: milliseconds
     expr    min     lq     mean median     uq    max neval cld
 original 1.1273 1.1895 1.241744 1.2432 1.2642 2.7154 50000  a
   legacy 1.2074 1.3053 1.361961 1.3608 1.3943 1.9876 50000   b
```

A.7. `isotonic() : legacy(x, w, type = 7)`

The R package **spsurvey** (Dumelle *et al.* 2022) contains a test for the parallel regression assumption, which uses `isotonic()`, an R function for isotonic regression. The function was written by Kincaid in 2001 and one-to-one translated into the legacy C function. The shortest implementation of all only checks for up-block violations and is required to do so multiple times. Each time, it is determined whether the vector is monotone or not. Although there is some overhead in the violation checking, the short and clear approach might perform reasonably fast, also because there are no weights involved. The comparison between the legacy function and the original function is not completely fair, because `isotonic()` is completely written in R, including three nested while loops.

```
Unit: microseconds
     expr    min     lq       mean median     uq      max neval cld
 original 3743.8 5248.7 6450.36712 5916.3 7659.0 139887.7 50000   b
   legacy   17.8   29.0   37.16069   34.5   43.1    174.7 50000  a
```

A.8. `isomean() : legacy(x, w, type = 8)`

In package **fdrtool** (Strimmer 2008; Klaus and Strimmer 2021), part of the function `monoreg()` uses C code to execute the PAVA algorithm. The referring function `isomean()` is ported from R by Strimmer (`https://github.com/cran/fdrtool/blob/master/src/isomean.c`), originally written by Balabdaoui, Rufibach, and Santambrogio (2011), and also used in, for example, **misoreg** (Klinglmueller 2010). The function is described as similar to `isoreg()`, with the addition that `monoreg()` accepts weights. This seems to be an understatement considering the substantial speed difference between the two implementations. The `isomean()` implementation uses additional memory for blocks, weights, and indices. There is only a down-block violation check and an original formula for the block value update. Block values are expanded in the final stage of the function. Comparing the legacy function with the original R and C implementations shows that the R function `monoreg()` is clearly the slowest implementation, mainly due to quite some overhead not needed for simple monotone regression.

```
Unit: microseconds
         expr  min    lq      mean median    uq    max neval cld
 original (R) 97.6 113.7 120.26143  119.0 124.7  234.4 50000   c
 original (C) 13.3  17.0  20.61951   21.3  23.2 3130.7 50000  b
       legacy 12.9  16.8  20.02457   19.3  22.9   55.5 50000 a
```

A.9. `pooledpava() : legacy(x, w, type = 9)`

In the years 2012–2014 an accessible page on isotonic regression appears on Stat Wiki (Wikipedia 2022). On that page, the pool-adjacent-violators algorithm is described in pseudocode and references are provided to implementations in R, Java, and Python. As for Python, isotonic regression is implemented in **scikit-learn** (Pedregosa *et al.* 2011) in 2013 for which Tulloch (2014b) creates a considerably faster Cython implementation, still using the initially used active set method (cf. De Leeuw *et al.* 2009). Switching to a true PAVA implementation, a whole team of contributors writes `pooledpava()`, which closely corresponds to the pseudocode of Wikipedia. `pooledpava()` uses additional memory, for indices, data, and weights, does not use direct updates but expands the block values afterwards, and only checks for down-block violations. The coalescence, however, handles one element at the time, slowing the process down considerably. The legacy function closely follows the Cython implementation.

A.10. `linearpava() : legacy(x, w, type = 10)`

The Wikipedia (2022) team also works on `linearpava()`, apparently the fastest of the implementations. All versions, `pooledpava()`, `linearpava()`, and `inplacepava()` (to be discussed hereafter), later appear in Julia (Tulloch 2014a). The `linearpava()` implementation is almost identical to `isotonic()`, but for the weights: it only checks for up-block violations, but does so beyond the next element and no memory allocation is needed due to the direct update approach. Despite the correspondence with `isotonic()` and the additional use of weight, `linearpava()` is way faster than `isotonic()`. It clearly dismisses the Grotzinger and Witzgall (1984) claim that direct updates lead to $O(n^2)$ algorithmic behavior. There is not an original function available for R.

A.11. `inplacepava() : legacy(x, w, type = 11)`

The scikit-learn branch undergoes another major change in 2016, when Varoquaux *et al.* (2016) implement `inplacepava()` (originally called `_inplace_contiguous_isotonic_regression`), an implementation matching `linearpava()` in terms of speed. The implementation of `inplacepava()` is probably the most sophisticated implementation. The procedure checks for an up-block violations first, and once found, it switches to checks for down-block violations. The block values are kept at the first element location of a block, hence the term "inplace". It resembles the PAVA procedure described by Stout (2008) used for unimodal regression. In the end, only the first elements of the blocks need to be expanded to the whole block. There is no original implementation available for the use in R.

A.12. `mdpava() : legacy(x, w, type = 12)`

An implementation in C++, `pava()`, is from Danisch (2016), and part of a program (`https://github.com/maxdan94/Density-Friendly`) that is used for computing the density-friendly decomposition (Tatti and Gionis 2015) and the densest subgraph in large sparse graphs (see Danisch, Chan, and Sozio 2017). `mdpava()` (renamed to avoid name conflicts again) is the shortest implementation, although it does not allow for weights. It only checks for down-block violations (after one up-wards check) and expands the block values afterwards. Due to its simplicity and the absence of weights, it makes a fast implementation. The original function is faster than the legacy function, a difference we cannot explain, since both codes are remarkably similar, and it makes `mdpava()` nearly as fast as `monotone()`.

```
Unit: microseconds
     expr  min     lq     mean median     uq    max neval cld
 original  8.2  9.699 10.49971 10.202 10.701 63.599 50000  a
   legacy 12.9 14.700 15.54204 15.100 15.600 53.900 50000   b
```

A.13. `reg1dl2() : legacy(x, w, type = 13)`

In Stout (2000) an algorithm is introduced for unimodal regression, as noted earlier. A good part of the algorithm consists of an isotonic regression procedure, called prefix isotonic regression, that saves the block values for subsequent use to form the isotonic (and antitonic) regressions for the unimodal regression problem. The actual paper is published years later as Stout (2008) due to "a series of absurd delays" as Stout puts it on his website (`https://web.eecs.umich.edu/~qstout/abs/UniReg.html`). In 2017, the first version of the R package **UniIsoRegression** is released that, among other things, performs the simple isotonic regression. The code for the isotonic regression can be found on GitHub `https://github.com/xzp1995/UniIsoRegression`. The function `reg_1d_l2()`, written in C++, has some overhead for the simple isotonic regression, preserving memory for weights, weighted values, block values, weighted squared values, level errors, and left and right block indices. The function only checks for down violations, collects the pooled elements in a separate vector, and thus expands the block values at the end into the result vector. The legacy function differs from the original function in three aspects: The original function is written in C++, a bug has been removed that invalidated the results, and overhead (needed for unimodal monotone regression) was removed to increase speed. It seems that this latter adjustment paid off.

```
Unit: microseconds
     expr  min   lq     mean median   uq  max neval cld
 original 22.1 25.8 26.99130   26.3 26.9 93.1 50000   b
   legacy 13.0 15.4 16.08711   15.9 16.4 46.1 50000  a
```

A.14. `jbkpava() : legacy(x, w, type = 14)`

The last contribution comes from one of the early adapters, as De Leeuw supervised the work of Van Waning (1976), proved the correctness of Kruskal's algorithms in case of ties (De Leeuw 1977), published on techniques involving isotonic regression, like `SMACOF()` (Stoop and De Leeuw 1982) and `GIFI()` (Gifi 1990), and contributed to software for these techniques (Mair *et al.* 2022; De Leeuw and Mair 2009; Mair, De Leeuw, and Groenen 2019). In De Leeuw (2016, 2017) (work in progress), a PAVA implementation in C, `jbkpava()`, is described that is *easily modified* for other purposes and *performs relatively uniformly* under difference problem instances (De Leeuw 2017). The implementation uses a C structure to keep block information: values, weights, sizes, and previous and next block indices. Violations are solved in both directions by constantly ensuring the blocks are either up-satisfied or down-satisfied. `jbkpava()` is clearly the more readable and modern version of `fitm()`. The legacy function is identical to the original function.

Affiliation:

Frank M. T. A. Busing
Leiden University
Methodology and Statistics Section
Psychological Institute
Faculty of Social and Behavioral Sciences
Wassenaarseweg 52
2300 RB Leiden, The Netherlands
E-mail: busing@fsw.leidenuniv.nl
URL: https://www.universiteitleiden.nl/en/staffmembers/frank-busing

Journal of Statistical Software https://www.jstatsoft.org/
published by the Foundation for Open Access Statistics https://www.foastat.org/
May 2022, Volume 102, Code Snippet 1 *Submitted:* 2020-03-16
doi:10.18637/jss.v102.c01 *Accepted:* 2021-06-30