Appl. Statist. (1995)
44, *No.* 2, *pp.* 258–268

Algorithm AS 296

Optimal Median Smoothing

By W. Härdle†

Humboldt-Universität Berlin, Germany

and W. Steiger

Rutgers University, New Brunswick, USA

[Received May 1988. Final revision September 1994]

Keywords: Heaps; Running medians; Smoothing

Language

Pascal

Description and Purpose

High variability in a given series $X_1, \ldots, X_N$ may obscure important structural features which would become evident if the data were replaced by a smoother, less variable version $X_1^*, \ldots, X_N^*$. One smoothing strategy consists of replacing X_i by

$$X_i^* = \text{median}(X_{i-k}, \ldots, X_{i+k}), \qquad i = k+1, \ldots, N-k \tag{1}$$

which, owing to the robustness of the median, will often give a superior result to

$$\text{average}(X_{i-k}, \ldots, X_{i+k}).$$

It is common terminology to call the series X_i^* the *running median* of order $K = 2k + 1$ and we refer to $X_{i-k}, \ldots, X_{i+k}$ as the *window of size K around* X_i. In this paper we describe an efficient running median algorithm using the *heap* data structure and we mention an interesting recent lower bound which shows that the algorithm has, up to constants, optimal running time.

An obvious approach (method 1) to the computation of X_i^* could use the fast median algorithm of Schönhage *et al.* (1976) with which each X_i^* could be obtained in at most $3K$ steps. As is usual, we count each pairwise comparison as a *step* and argue that the running time of any 'reasonable' implementation would be proportional to this complexity measure. The smoothed series would then be obtained in at most $3K(N - K)$ steps. By this method we obtained $N - K$ running medians at an average cost of $3K$.

Another possibility (method 2) would be to maintain the window

$$X_{i-k}, \ldots, X_{i+k}$$

in sorted order as

$$Z_1 \leqslant \ldots \leqslant Z_K.$$

†*Address for correspondence*: Institut für Statistik und Ökonometrie, Wirtschaftswissenschaftliche Fakultät, Humboldt-Universität Berlin, Spandauer Strasse 1, D-10178 Berlin, Germany.

 0035-9254/95/44258

In this case $X_i^* = Z_{k+1}$ and the Zs may be prepared for the determination of X_{i+1}^* by locating, then removing, X_{i-k}, and then correctly inserting X_{i+k+1} so that the Zs are once again sorted. Each operation may be done via a binary search requiring at most $\log K$ steps (all logarithms are to the base 2). Although it is possible to implement this strategy so that it actually runs in time $O(\log K)$ (this means $b \log K$ or less for all K, where b is an absolute constant), it is quite difficult to do so and, in any case, the constant b is quite large. Easily implemented algorithms to maintain a sorted window use pointers into the Zs. We expect $K/2$ pointer values to be changed per update, on average, and this would certainly contribute to the running time. Although an update really takes only $2 \log K$ *comparison* steps, the *hidden cost* of pointer updates would force the actual running time of this algorithm to be

$$(2 \log K + cK/2)(N - K),$$

where c is a constant reflecting the time for a pointer manipulation in relation to that of a comparison.

Our proposal (method 3) maintains $X_{i-k}, \ldots, X_{i+k}$ in a priority queue that supports easy and quick updates. We use a data structure based on two heaps which we call *partitioning heaps*. It may be updated in time proportional to $\log K$. The array $Z_1, \ldots, Z_m$ is called a maxheap if it is partially ordered so that it satisfies

$$Z_i \geqslant \max(Z_{2i}, Z_{2i+1}); \tag{2}$$

a minheap has the reverse inequality. A convenient reference for heaps and other data structures is Baase (1988) or Mehlhorn (1984). There it is shown that a heap of size m may be constructed in at most $4m$ steps and that a new item may be inserted to preserve the heap property in at most $\log m$ steps. It is clear that after an item has been deleted the heap property can be restored in at most $2 \log m$ steps.

The complexity of the algorithm that we describe in the next section is no more than

$$4K + \{3 \log(K/2)\}(N - K) \tag{3}$$

steps. More importantly, the implementation uses at most $4 \log(K/2)$ pointer manipulations per update, each using one multiplication or division. This gives a total running time bounded by

$$4K + (3 + 4d) \log(K/2)(N - K), \tag{4}$$

where d is a constant reflecting the time for a division relative to that of a comparison. If, as is reasonable, the window size increases with N, this method becomes much more efficient than the two methods that were mentioned previously.

It is interesting to consider the behaviour of an optimal algorithm for running medians. This would give a standard against which all algorithms should be compared and has practical as well as theoretical significance. Though it seems likely that the average cost of determining X_i^* would grow with K, the number of Xs determining each median, the only obvious statement is the trivial lower bound of $2(N - K)$ steps for smoothing by running medians which is established as follows. If $X_i, \ldots, X_N$ is partitioned into N/K non-overlapping segments of length K each, it is necessary to perform $2K$ steps to obtain each of $X_{K+1}^*, X_{K+k+1}^*, \ldots, X_{(N/K-1)K+k+1}^*$, via a result of Bent and John (1985). This gives a total cost of

$2KN/K$, or an average update cost of two steps per median. By expression (3), our method has an average cost of

$$3\log(K/2) + (K-1)/(N-K)$$

steps per update. The lower bound of Gill *et al.* (1988) shows that at least $(a \log K)(N-K)$ steps must be performed by any algorithm that correctly computes $X^*_{k+1}, \ldots, X^*_{N-K}$, where $a > 0$ is an absolute constant. This means that our algorithm is optimal at least up to the constant of proportionality in expression (4). It is difficult to imagine a realizable algorithm that does less work than $2\log(K/2)$ steps per update.

Finally, it is worth mentioning some recent active interest in data structures aimed at implementing priority queues for specialized types of insertions and deletions (see Atkinson *et al.* (1986) or Carlsson (1987)). Neither of these techniques seems as well suited to compute the running median as the partitioning heaps that we describe in the next section.

Numerical Method

We use two heaps of size k to represent the current window

$$X_{i-k}, \ldots, X_{i+k}, \qquad i = k+1, \ldots, N-k.$$

The window is stored in the array

$$H_{-k}, \ldots, H_{-1}, H_0, H_1, \ldots, H_k$$

that preserves the following structure:

(a) H_0 is X^*_i;
(b) $\max(H_{-2i}, H_{-2i-1}) \leqslant H_{-i} \leqslant X^*_i$;
(c) $\min(H_{2i+1}, H_{2i}) \geqslant H_i \geqslant X^*_i$.

These conditions mean that X^*_i partitions H into a maxheap of those window elements $H_{-1}, \ldots, H_{-k}$ not larger than X^*_i and a minheap

$$H_1, \ldots, H_k$$

of those window elements that are not smaller than X^*_i. The data structure

$$H_{-k}, \ldots, H_{-1}, H_0, H_1, \ldots, H_k$$

may be visualized as shown in Fig. 1.

The ordering between levels obeys conditions (b) and (c). No order is imposed on values within the same level. The structure has a depth at most $2(1 + \log k)$.

To update this structure so that it will represent the next window, we need to find X_{i-k} and to remove it, to insert X_{i+k+1} correctly and to restore the structure so that condition (b) still holds. Our method involves pointers $i_{-k}, \ldots, i_k$ and $j_{-k}, \ldots, j_k$, both permutations of $-k, \ldots, -1, 0, 1, \ldots, k$. They have the property that

$$\begin{aligned} H_{i_m} &= X_{i+m}, \qquad m = -k, \ldots, k, \\ X_{i+j_m} &= H_m, \qquad m = -k, \ldots, k. \end{aligned} \tag{5}$$

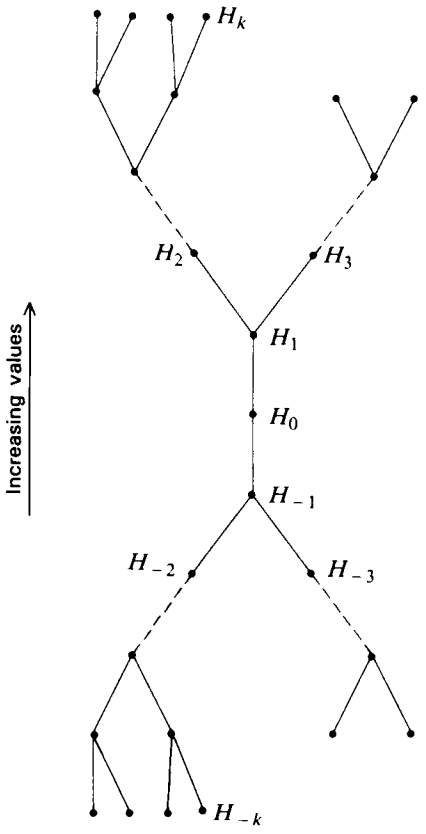

Fig. 1

The pointers i_m locate a particular window element in the heap whereas the j_m locate a particular heap element in the window. Clearly

$$\begin{aligned} X_{i+j_{i_m}} &= X_{i+m}, \\ H_{i_{j_m}} &= H_m. \end{aligned} \tag{6}$$

When X_{i-k} is removed from the window the data structure has an empty place at i_{-k}. To update we propagate the 'hole' to the apex of the relevant heap. For example if $i_{-k} < -1$ the steps

$$\begin{aligned} j_{i_{-k}} &\leftarrow j_{i_{-k}}/2, \\ i_{-k} &\leftarrow i_{-k}/2 \end{aligned} \tag{7}$$

swap the hole at i_{-k} with the value of the parent node $H_{i_{-k}}/2$ and correctly adjust the relevant pointers. The case $i_{-k} > 1$ is analogous. At most $\log k$ steps are performed, each involving a division of the pointer value by 2.

Inserting the new value X_{i+k+1} into the data structure is analogous: a comparison of X_{i+k+1} with X_i^* determines whether the new value goes in the top or bottom heap. In the former case, for example, assuming that X_{i-k} came from the

bottom heap X_{i+k+1} it is initially placed in the hole just created at position -1; $i_k \leftarrow -1$ and $j_{-1} \leftarrow k$ reflect this operation. H_{-1} is compared with H_{-2} and H_{-3} and if condition (b) holds, the insertion is finished. Otherwise the larger of H_{-2} and H_{-3} is swapped with H_{-1}, etc., until the new item trickles down to its correct place and condition (b) is once again satisfied. At most $2 \log k$ comparisons are required along with $2 \log k$ pointer updates. Thus, the total cost of maintaining the structure in H while moving from the current window to the next is at most $2 \log k$ comparisons and $4 \log k$ pointer updates.

The running times reported in the next section show that the average cost to compute for each X_i^* is proportional to $6 \log k$. Maintaining pointer values as permutations of $-k, \ldots, k$ facilitates quick updates because it is very well suited to the data structure that we use: the parent node of node j is $j/2$ and the children are $2j$ and $2j + 1$. Finally we observe that round-off errors and stability are not important because the algorithm does not perform floating point arithmetic.

Structure

Required global declarations

Constant		
maxk		the maximum window size (e.g. 60)
maxn		the maximum number of observations (e.g. 1023)
Type		
keytype	real	the type of value
elem	RECORD key: keytype END	
nelemarray	ARRAY [1..maxn] of elem	an array of all observations
kelemarray	ARRAY [−maxk..maxk] of elem	an array of observations in a window
outlistype	ARRAY [1..maxn] of integer	an array of pointer indices
nrlistype	ARRAY [−maxk..maxk] of integer	an array of pointer indices
medianarray	ARRAY [1..maxn] of elem	an array of medians

PROCEDURE runmed(n, k: integer; VAR unsorted: nelemarray; VAR median: medianarray);

Formal parameters			
n	integer	Value:	the real size of the array, less than or equal to maxn
k	integerthe	Value:	the real size of the window less than or equal to maxk
unsorted	nelemarray	Value:	only var because of storage
median	medianarray	Value:	returns the running median

TABLE 1
Timings for optimal median smoothing

k	*Times (s) for the following series:*		
	Descending	*Ascending*	*Random*
7	11.2	10.8	6.8
15	15.6	15.3	9.2
31	20.3	19.8	11.4
63	25.1	24.2	13.8
127	29.1	28.6	16.2
255	33.3	32.8	18.4
511	37.1	36.6	20.4
1023	41.2	39.7	22.0
2047	41.3	40.9	22.4
4095	38.6	38.2	21.2

Time

To check the approximations of expression (4) we smoothed series of $n = 16000$ by using windows of $K = 7, 15, 31, 63, 127, 255, 511, 1023, 2047, 4095$. The X_i were generated to be uniformly distributed (0, 1) random numbers by using the internal random mechanism of an Atari home computer. For each value of K the smoothing experiment was repeated 10 times with independently generated series $X_1, \ldots, X_N$. The running times were then averaged over the 10 replications. As a basis of comparison with the proposed optimal median smoothing algorithm we also smoothed by using a simplification of method 2 where $X_i^* = W_{k+1}$ is simply extracted from the sorted window $W_1, \ldots, W_k$ and then X_{i+k+1} is correctly inserted by a sequential search costing $O(K)$ steps on average.

Table 1 shows the timings for the optimal running median algorithm in three situations. In the second column of Table 1 we report the timings for smoothing a series of values that were initially in descending order. The third column gives the timing for a series that was initially increasing. The last column of Table 1 shows the timings averaged over 10 repetitions of smoothing purely random sequences.

It is interesting that the optimal running median algorithm is worst on an initially sorted series, whether ascending or descending. The reason is that the new element

TABLE 2
Timings for the straight insertion method

k	*Times (min:s) for the following series:*		
	Descending	*Ascending*	*Random*
7	0 : 11.2	0 : 08.4	0 : 08.3
15	0 : 22.7	0 : 16.3	0 : 11.8
31	0 : 45.6	0 : 33.2	0 : 16.7
63	1 : 31.4	0 : 06.1	0 : 36.9
127	3 : 02.4	2 : 11.6	1 : 14.9
255	6 : 02.1	4 : 21.0	2 : 21.5

entering the smoothing window will trickle all the way downwards (or upwards) through a heap.

Table 2 presents the timings for the straight insertion method. The columns refer to the same experiments as in Table 1. Since the algorithm was extremely slow for large K we present results only for K up to 255. A comparison of Tables 1 and 2 reveals that for the experiment the optimal median smoothing algorithm is about 10 times faster than the insertion method. Inserting the new value X_{i+k+1} into the window would not improve the insertion algorithm appreciably owing to the linear cost of moving data on each update. The timings for smoothing random series by the new method (Table 1) grow proportionally to $\log K$; those for the insertion method increase proportionally to K.

Acknowledgements

We would like to thank a referee for his careful reading of the code. This research was supported by the Deutschen Forschungsgemeinschaft, Sonderforschungsbereich 303 and 373, and by CORE and the Institut de Statistique, Université Catholique de Louvain, Belgium.

References

Atkinson, M., Sack, J. R., Santoro, N. and Strothotte, T. (1986) Min-max heaps and generalized priority queues. *Communs ACM*, **29**, 996–1000.

Baase, S. (1988) *Computer Algorithms: Introduction to Design and Analysis*, 2nd edn. Reading: Addison-Wesley.

Bent, S. W. and John, J. (1985) Finding the median requires $2n$ comparisons. In *Proc. 17th Association of Computing Machines Symp. Theory of Computing*, pp. 213–216.

Carlsson, S. (1987) The deap—a double ended heap to implement double ended priority queues. *Inf. Proc. Lett.*, **26**, 33–36.

Gill, J., Steiger, W. and Wigderson, A. (1988) The complexity of weighted median smoothing is $O(\log K)$ steps per median. *Technical Report*. Department of Computer Science, Rutgers University, New Brunswick.

Mehlhorn, K. (1984) *Data Structures and Algorithms*, vol. 1, *Sorting and Searching*. Berlin: Springer.

Schönhage, A., Patterson, M. and Pippenger, N. (1976) Finding the median. *J. Comput. Syst. Sci.*, **13**, 184–199.

Algorithm AS 297

Orthogonal Polynomial and Hybrid Estimators for Nonparametric Regression

By A. S. Azari and H. G. Müller†

University of California, Davis, USA

[Received June 1989. Final revision September 1994]

Keywords: Curve estimation; Estimation of derivatives; Ultraspherical polynomials

†*Address for correspondence*: Division of Statistics, University of California at Davis, Davis, CA 95616-8705, USA.
E-mail: asazari@ucdavis.edu