Appeared in *Computational Statistics, Volume 1* (1992), edited by Y. Dodge and J. Whittaker, Heidelberg: Physica-Verlag, pp. 411-428.

Time-efficient algorithms for two highly robust estimators of scale

Christophe Croux and Peter J. Rousseeuw

Department of Mathematics and Computer Science, Universitaire Instelling Antwerpen (U.I.A.), Universiteitsplein 1, B-2610 Wilrijk, Belgium.

Abstract
In this paper we present deterministic algorithms of time $O(n \log n)$ and space $O(n)$ for two robust scale estimators with maximal breakdown point. The actual source codes are included, and execution times are compared.

1. INTRODUCTION

In robust estimation one frequently needs an initial or auxiliary estimator of scale. For this one usually takes the median absolute deviation $\mathrm{MAD}_n = 1.4826\ \mathrm{med}_i\{|x_i - \mathrm{med}_j\, x_j|\}$ because it has a simple explicit formula, needs little computation time, and is very robust as witnessed by its bounded influence function and its 50% breakdown point. However, there is still room for improvement in two areas: the fact that MAD_n is aimed at symmetric distributions, and its low (37%) gaussian efficiency. Rousseeuw and Croux (1991) proposed two explicit and 50% breakdown scale estimators that are more efficient. These are the estimator $S_n = 1.1926\ \mathrm{med}_i\{\mathrm{med}_j\, |x_i - x_j|\}$ and the estimator Q_n which is given by the 0.25-quantile of the distances $\{|x_i - x_j|; i < j\}$. Note that S_n and Q_n do not need any location estimate. The gaussian asymptotic efficiency of S_n is 58%, whereas Q_n attains 82%. Other statistical properties such as finite-sample efficiencies and influence functions turned out to be satisfactory as well, even at very asymmetric distributions.

At first sight these estimators appear to need $O(n^2)$ computation time, which would be a disadvantage. However, both S_n and Q_n can actually be computed using no more than $O(n \log n)$ time and $O(n)$ storage, by means of the fast algorithms described below. We also give correction factors that make our estimators (and the MAD) nearly unbiased at small samples. The Fortran source code of both algorithms is listed in this paper, and can be obtained in machine-readable form from the authors (email: croux@wins.uia.ac.be and rousse@wins.uia.ac.be). A Pascal version is also available.

2. THE ESTIMATOR S_n

We define the estimator S_n as

$$S_n = 1.1926 \underset{i=1,\ldots,n}{\mathrm{lomed}} \underset{j=1,\ldots,n}{\mathrm{himed}} |x_i - x_j| \tag{1}$$

where the outer operation is a low median (that is, the $[(n+1)/2]$-th order statistic out of n numbers) and the inner is a high median (the $([n/2]+1)$-th order statistic out of n). This should be read as follows: for each i we compute the high median of $\{|x_i - x_j|;\ j = 1, .., n\}$. This yields n numbers, the low median of which gives our final estimate S_n. (The factor 1.1926 is for consistency at normal distributions). The idea to apply medians repeatedly was introduced by Tukey (1977) for estimation in two-way tables, and by Siegel (1982) for estimating regression coefficients. Rousseeuw and Bassett (1990) used recursive medians for estimating location in large data sets.

Note that (1) is an explicit formula, hence S_n is always uniquely defined. We see immediately that S_n does behave like a scale estimator, in the sense that transforming the observations x_i to $ax_i + b$ will multiply S_n by $|a|$. We will refer to this property as *affine equivariance.*

Definition and breakdown point: It seems a bit superfluous to include the zero values $|x_i - x_i|$ in definition (1), but this is resolved by noting that

$$S_n = 1.1926 \operatorname*{lomed}_{i=1,\dots,n} \operatorname*{lomed}_{j \neq i} |x_i - x_j|. \tag{2}$$

This equivalent definition will be used in the actual computation of S_n.

In order to measure the robustness of S_n we want to know how many data points need to be replaced to make it explode (tend to infinity) or implode (tend to zero). For any sample $X = \{x_1, \dots, x_n\}$ the explosion breakdown point is defined by

$$\varepsilon_n^+(S_n, X) = \min\{\frac{m}{n};\ \sup_{X'} S_n(X') = \infty\}$$

and the implosion breakdown point by

$$\varepsilon_n^-(S_n, X) = \min\{\frac{m}{n};\ \inf_{X'} S_n(X') = 0\},$$

where X' is obtained by replacing any m observations by arbitrary values. The overall breakdown point of S_n is then defined as

$$\varepsilon_n^*(S_n, X) = \min\{\varepsilon_n^-(S_n, X), \varepsilon_n^+(S_n, X)\}.$$

For this particular estimator we obtain the following proposition:

Proposition 1. *For any sample $X = \{x_1, \dots, x_n\}$ the explosion breakdown point of S_n is given by*

$$\varepsilon_n^+(S_n, X) = [(n+1)/2]/n,$$

and for any sample X in which no 2 points coincide, the implosion breakdown point is

$$\varepsilon_n^-(S_n, X) = [n/2]/n.$$

Therefore its overall breakdown point is $[n/2]/n$, which is the best possible value. In fact, if we replace the outer lomed in (2) by an order statistic of rank at most $[n/2] + 1$ we still keep the same breakdown points. (Even more combinations, like himed(himed) or

Table 1: Average value of S_n, MAD_n and Q_n before the incorporation of a finite-sample correction factor.

	S_n		MAD_n		Q_n	
n	ave	SE	ave	SE	ave	SE
3	0.5381	0.0045	0.6689	0.0056	1.0025	0.0084
4	1.0479	0.0057	0.7336	0.0042	1.9523	0.0106
5	0.7485	0.0041	0.8291	0.0048	1.1973	0.0064
6	0.9996	0.0045	0.8331	0.0040	1.6276	0.0068
7	0.8335	0.0036	0.8774	0.0042	1.1638	0.0048
8	0.9951	0.0038	0.8855	0.0036	1.4942	0.0051
9	0.8812	0.0031	0.9032	0.0037	1.1411	0.0039
10	0.9941	0.0033	0.9128	0.0033	1.3925	0.0041
11	0.9113	0.0029	0.9236	0.0035	1.1240	0.0034
20	0.9983	0.0022	0.9582	0.0025	1.1899	0.0023
21	0.9643	0.0020	0.9611	0.0025	1.0716	0.0021
30	1.0024	0.0017	0.9756	0.0021	1.1270	0.0017
31	0.9748	0.0017	0.9743	0.0021	1.0510	0.0016
50	1.0012	0.0013	0.9848	0.0016	1.0763	0.0013
51	0.9874	0.0013	0.9840	0.0016	1.0295	0.0012
80	1.0017	0.0010	0.9892	0.0013	1.0487	0.0009
81	0.9949	0.0010	0.9920	0.0013	1.0176	0.0009
100	0.9998	0.0009	0.9907	0.0012	1.0393	0.0008

lomed(himed), are allowed if we are willing to accept an explosion breakdown point of $[n/2]/n$ combined with an implosion breakdown point of $[(n+1)/2]/n$.) Here we don't adopt the usual definition of the sample median as an average of the low and high median since this is only necessary in the location case (to obtain affine equivariance), and doubles the computation time.

In order to check whether the factor 1.1926 (which was obtained by means of an asymptotic argument) succeeds in making S_n approximately unbiased for finite samples, we performed a simulation study. For each n we generated 10,000 samples of n gaussian observations and then computed the average value of (2) and the standard error (SE) on that value. To generate these observations we used a random generator provided by the compiler on our Unix system and the Box-Muller transformation. The random generator of Cheney and Kincaid (1985, page 335) and the generator AS 183 (Wichmann and Hill 1982, McLeod 1985) yielded results that were within standard error bounds of the reported values. In Table 1 we can see that for n even there is practically no bias. However, for n odd a small bias appears. Therefore, from now on we redefine S_n as

$$S_n = c_n\, 1.1926 \operatorname*{lomed}_{i=1,\ldots,n} \operatorname*{lomed}_{j\neq i} |x_i - x_j| \tag{3}$$

where the correction factor c_n is given by

n	2	3	4	5	6	7	8	9
c_n	0.743	1.851	0.954	1.351	0.993	1.198	1.005	1.131

for $n \leq 9$, and for $n > 9$ is defined as

$$c_n = \frac{n}{n - 0.9} \quad \text{for } n \text{ odd}$$

$$c_n = 1 \text{ for } n \text{ even} .$$

In order to be able to give c_n with three decimals, we repeated the simulation for small n with 200,000 replications. Moreover, the factor c_2 can actually be obtained analytically. Indeed, for $n = 2$ it holds that

$$\operatorname*{lomed}_{i=1,2} \operatorname*{lomed}_{j \neq i} |x_i - x_j| = |x_1 - x_2|$$

and we know that

$$E|X_1 - X_2| = \sqrt{2}E|X_1| = \frac{2}{\sqrt{\pi}} = 1.1284$$

hence $c_2 = 1/(1.1284 * 1.1926) = 0.743$.

Note that the factor c_n is generally close to 1. By way of comparison, let us consider the factor b_n that is needed to make

$$\mathrm{MAD}_n = b_n \; 1.4826 \operatorname*{med}_i |x_i - \operatorname*{med}_j x_j| \tag{4}$$

approximately unbiased, where "med" now stands for the usual (lomed+himed)/2 version of the median. By repeating the above procedure we obtain

n	2	3	4	5	6	7	8	9
b_n	1.196	1.495	1.363	1.206	1.200	1.140	1.129	1.107

and for $n > 9$ we find

$$b_n = \frac{n}{n - 0.8}.$$

Therefore, the finite-sample correction factor for S_n is of roughly the same size as the one needed for MAD_n.

A naive algorithm for $\mathbf{S_n}$: A primitive method for computing S_n is given by the following Fortran code:

```
      do 10 i=1,n
         do 20 j=1,n
20       a1(j)=abs(x(i)-x(j))
10    a2(i)=himed(a1,n)
      s=cn*1.1926*lomed(a2,n)
```

Apart from the array x of observations, this only uses two additional arrays of length n, so the total storage we need is $O(n)$. Selecting an order statistic among n elements can be done in $O(n)$ time (Knuth 1973, page 216) so this simple algorithm needs a computation time of $O(n^2)$. This algorithm computes S_n as the remedian with base n (see Rousseeuw and Bassett 1990) of the collection of interpoint distances $D = \{|x_i - x_j|; 1 \leq i, j \leq n\}$, assuming that the elements of D are considered row by row.

If we use a matrix language, then the naive algorithm above becomes a "one-liner". For example, in ISP we can write s=cn*1.1926*median(trn(median(abs(x-trn(x)))),

where "median" computes the column-wise median of a matrix and "trn" denotes the transpose of a matrix. However, on our PC this ran into space problems for $n \geq 150$ because a matrix of $n^2 \geq 22500$ elements has to be stored in memory.

The naive algorithm really needs a large parallel computer, on which the n inner medians could all be computed simultaneously.

An efficient algorithm: The basic idea of our $O(n \log n)$-time and $O(n)$-space algorithm is the following. If we first sort the observations (this takes $O(n \log n)$ time), then each a2(i) is in fact the overall (low) median of two sorted arrays, namely

$$\{x_i - x_{i-1}, \ldots, x_i - x_1\} \text{ and } \{x_{i+1} - x_i, \ldots, x_n - x_i\}. \tag{5}$$

Finding the overall median of two sorted arrays, of which the largest contains n elements, can be done in $O(\log n)$ time. Shamos (1976) has described how to do this for arrays of equal length. We extended this idea to unequal lengths, and adapted it to our situation. (Note that it would *not* do to actually compute the $n-1$ differences in (5) and then to apply some subroutine to them, since that would increase the computation time to $O(n^2)$ again.) In the following paragraph we give an outline of the algorithm for finding the median of two sorted arrays A and B, of which B is the largest. We use this for computing a2(i) with

$$A = \{x_i - x_{i-j};\, j = 1, \ldots, i-1\} \text{ and } B = \{x_{i+j} - x_i;\, j = 1, \ldots, n-i\},$$

when $1 < i \leq [(n+1)/2]$. For $[(n+1)/2] < i < n$ we interchange A and B. The numbers a2(1) and a2(n) are trivially found.

Finding the overall median of 2 sorted arrays: Suppose that we have two sorted arrays A and B. Denote $\mathbf{diff}{=}n_B - n_A \geq 0$ (where n_B denotes the number of elements of the array B). In our imagination we extend the array A to an array A_0 with n_B elements, by filling up the remainder with $[\mathbf{diff}/2]$ times $-\infty$ and $[(\mathbf{diff}+1)/2]$ times $+\infty$. Then we have the property that $\text{lomed}(A \cup B) = \text{lomed}(A_0 \cup B)$. (If we were searching for the himed, we would have to plug in $[(\mathbf{diff}+1)/2]$ times $-\infty$ and $[\mathbf{diff}/2]$ times $+\infty$.) We initialize four variables $\mathbf{leftA} = 1 = \mathbf{leftB}$ and $\mathbf{rightA} = n_B = \mathbf{rightB}$. Our potential outcomes correspond to

$$\textit{Candidates} = \{a_j;\, \mathbf{leftA} \leq j \leq \mathbf{rightA}\} \cup \{b_j;\, \mathbf{leftB} \leq j \leq \mathbf{rightB}\}.$$

The purpose is to refine *Candidates* until it has only 2 elements. Denote by A_m and B_m the elements that are still in the running. We adapt the limits of *Candidates* until $\mathbf{leftA} = \mathbf{rightA}$ (and $\mathbf{leftB} = \mathbf{rightB}$) by steps satisfying:

(a) $n_{A_m} = n_{B_m} = length_m$
(b) $\text{lomed}(A_m \cup B_m) = \text{lomed}(A_{m-1} \cup B_{m-1})$
(c) $length_m = [(length_{m-1}+1)/2]$

Property (c) ensures that we only need $O(\log n)$ steps. In the final step $n_{A_m} = n_{B_m} = 1$, so that from (b) it follows that the smallest of the 2 elements in $A_m \cup B_m$ is our outcome. How do we adapt our bounds? In step m, we compute **medA** and **medB** as the lomeds of A_m and B_m. If for example **medA** $\geq$ **medB** then we know that all $a_j >$ **medA** are too large to be the overall median, and all $b_j <$ **medB** are too small to be the overall median. (The algorithm needs to be quite carefully designed to end up with the correct ranks for even and odd array lengths.) We also use two numbers **Amin** and **Amax**, to mark the position of A in A_0 (note that **Amin** and **Amax** remain unchanged during the refinement steps). If an index j is smaller than **Amin** or greater than **Amax**, we know what the outcome of comparing a_j with any element of B will be. Note that it is impossible that all elements of A_m are $-\infty$, because then we would have $\text{lomed}(A_m \cup B_m) = -\infty$. It could happen that all elements of A_m are $+\infty$, and then the overall himed equals $\max(B_m) = b_{\mathbf{rightB}}$. One could test in every loop whether **leftA** >**Amax** in order to use the latter property, but it turns out that this does not improve the speed.

Source code: The function **Sn** was written in Fortran77 and has been compiled with version 3.31 of the Microsoft Fortran compiler and the UTX/32 Fortran 77 compiler running under UNIX. It has been thoroughly tested against the result of the naive algorithm.

```
cc
cc  Efficient algorithm for the scale estimator:
cc
cc        Sn = cn * 1.1926 * LOMED_{i} HIMED_{j} |x_i-x_j|
cc
cc  Parameters of the function Sn :
cc        x  : real array containing the observations
cc        n  : number of observations (n>=2)
cc
cc  The function Sn uses the procedures:
cc        sort(x,n,y) : sorts an array x of length n, and stores the
cc                      result in an array y (of size at least n)
cc        pull(a,n,k) : finds the k-th order statistic of an
cc                      array a of length n
cc
cc  The function Sn also creates an auxiliary array a2
cc        (of size at least n) in which it stores the values
cc        LOMED_{j<>i} |x_i-x_j|    for i=1,...,n
cc
      function Sn(x,n)
      dimension x(n),y(1000),a2(1000)
      integer rightA,rightB,tryA,tryB,diff,Amin,Amax,even,half
      real medA, medB
      call sort(x,n,y)
      a2(1)=y(n/2+1)-y(1)
      do 10 i=2,(n+1)/2
```

```
      nA=i-1
      nB=n-i
      diff=nB-nA
      leftA=1
      leftB=1
      rightA=nB
      rightB=nB
      Amin=diff/2+1
      Amax=diff/2+nA
15    continue
      if (leftA.lt.rightA) then
          length=rightA-leftA+1
          even=1-mod(length,2)
          half=(length-1)/2
          tryA=leftA+half
          tryB=leftB+half
          if (tryA.lt.Amin) then
              rightB=tryB
              leftA=tryA+even
          else
              if (tryA.gt.Amax) then
                  rightA=tryA
                  leftB=tryB+even
              else
                  medA=y(i)-y(i-tryA+Amin-1)
                  medB=y(tryB+i)-y(i)
                  if (medA.ge.medB) then
                      rightA=tryA
                      leftB=tryB+even
                  else
                      rightB=tryB
                      leftA=tryA+even
                  endif
              endif
          endif
      go to 15
      endif
      if (leftA.gt.Amax) then
          a2(i)=y(leftB+i)-y(i)
      else
          medA=y(i)-y(i-leftA+Amin-1)
          medB=y(leftB+i)-y(i)
          a2(i)=min(medA,medB)
      endif
10  continue
```

```
      do 20 i=(n+1)/2+1,n-1
          nA=n-i
          nB=i-1
          diff=nB-nA
          leftA=1
          leftB=1
          rightA=nB
          rightB=nB
          Amin=diff/2+1
          Amax=diff/2+nA
25        continue
          if (leftA.lt.rightA) then
              length=rightA-leftA+1
              even=1-mod(length,2)
              half=(length-1)/2
              tryA=leftA+half
              tryB=leftB+half
              if (tryA.lt.Amin) then
                  rightB=tryB
                  leftA=tryA+even
              else
                  if (tryA.gt.Amax) then
                      rightA=tryA
                      leftB=tryB+even
                  else
                      medA=y(i+tryA-Amin+1)-y(i)
                      medB=y(i)-y(i-tryB)
                      if (medA.ge.medB) then
                          rightA=tryA
                          leftB=tryB+even
                      else
                          rightB=tryB
                          leftA=tryA+even
                      endif
                  endif
              endif
          go to 25
          endif
          if (leftA.gt.Amax) then
              a2(i)=y(i)-y(i-leftB)
          else
              medA=y(i+leftA-Amin+1)-y(i)
              medB=y(i)-y(i-leftB)
              a2(i)=min(medA,medB)
          endif
```

```
20    continue
      a2(n)=y(n)-y((n+1)/2)
      cn=1
      if (n.le.9) then
          if (n.eq.2) cn=0.743
          if (n.eq.3) cn=1.851
          if (n.eq.4) cn=0.954
          if (n.eq.5) cn=1.351
          if (n.eq.6) cn=0.993
          if (n.eq.7) cn=1.198
          if (n.eq.8) cn=1.005
          if (n.eq.9) cn=1.131
      else
          if (mod(n,2).eq.1) cn=n/(n-0.9)
      endif
      Sn=cn*1.1926*pull(a2,n,(n+1)/2)
      return
      end
```

The first do-loop (**do 10**) computes the values of a2(i)=$\text{lomed}_{j\neq i}\,|x_i - x_j|$ for $i = 2,\ldots,[(n+1)/2]$. In **do 20** the same is done for $i = [(n+1)/2]+1,\ldots,n-1$. The similarity between these loops is striking, but there are some crucial distinctions due to the interchange of A and B. It is possible to shorten the source code by combining both cases in a single loop, making use of a few additional if-tests and some multiplications by a dummy integer taking the values -1 and 1. However, we found that this resulted in running times that were roughly 35% higher.

Remark: Note that this algorithm provides the complete array a2 without additional computational effort. This is important because the a2(i) can be used to construct confidence intervals around S_n. They also allow us to compute different scale estimators. For instance, we could compute an L-statistic on the $[n/2]+1$ smallest values of a2(i).

3. THE ESTIMATOR Q_n

Our second estimator is defined as

$$Q_n = 2.2219\,\{|x_i - x_j|;\, i < j\}_{(k)} \tag{6}$$

where the factor 2.2219 is for consistency and $k \approx \binom{n}{2}/4$. That is, we take the k-th order statistic of the $\binom{n}{2}$ interpoint distances. Like S_n, the estimator Q_n has a simple and explicit formula and is affine equivariant.

If we would replace the k-th order statistic by a median we would recover the median interpoint distance mentioned by Bickel and Lehmann (1979), the breakdown point of which is lower (about 29%). The latter is similar to the location estimator of Hodges and Lehmann (1963) which uses the averages $(x_i + x_j)/2$ instead of the distances $|x_i - x_j|$.

Let us investigate which values of k in (6) yield the maximal breakdown point:

Proposition 2. *At any sample X in which no 2 points coincide we have*

$$\varepsilon_n^+(Q_n, X) = [(n+1)/2]/n \quad and \quad \varepsilon_n^-(Q_n, X) = [n/2]/n$$

if we take

$$\binom{h-1}{2} + 1 \leq k \leq \binom{h}{2}$$

where $h = [n/2] + 1$.

This means that there are $h-1$ possible choices for k, which all yield the same breakdown behavior. Extensive simulations have confirmed that the standardized variances

$$n \operatorname{var}(Q_n)/(\operatorname{ave}(Q_n))^2$$

are strictly decreasing in k, so the larger k yield the more efficient estimators. Therefore, we decided to use the largest value $k = \binom{h}{2}$. For this choice of k, we have to determine the appropriate correction factor. As in the case of S_n and MAD_n, we computed the average of (6) over 10,000 samples. The results are given in Table 1, and again we see that this behavior is slightly different for n odd or even. We therefore redefine Q_n as

$$Q_n = d_n\, 2.2219\{|x_i - x_j|;\, i < j\}_{\binom{h}{2}} \tag{7}$$

where the correction factor d_n is given by

n	2	3	4	5	6	7	8	9
d_n	0.399	0.994	0.512	0.844	0.611	0.857	0.669	0.872

for $n \leq 9$, and for $n > 9$ is defined as

$$d_n = \frac{n}{n+1.4} \quad \text{for } n \text{ odd}$$

$$d_n = \frac{n}{n+3.8} \quad \text{for } n \text{ even.}$$

Note that d_2 has been obtained analytically, as were c_2 and b_2. In fact, for $n = 2$ all affine equivariant scale estimators (including S_n, MAD_n, and Q_n) reduce to a multiple of $|x_1 - x_2|$.

Computation of $\mathbf{Q_n}$: At first sight the estimator Q_n requires a large computational complexity, because the naive algorithm (which begins by computing and storing all $\binom{n}{2}$ pairwise distances) needs $O(n^2)$ space and $O(n^2)$ time. Fortunately, Johnson and Mizoguchi (1978) provide a fast method for finding the k-th order statistic in a table of the type $X + Y = \{x_i + y_j; 1 \leq i, j \leq n\}$. They first sort X and Y (using $O(n \log n)$ time for each) and then construct the $n \times n$ matrix $U = X + Y$ in their imagination. They define two arrays **left** and **right** for keeping track of the points on the i-th row that are still in the running for being the k-th order statistic. In other words,

$$Candidates = \{U_{ij};\ \mathbf{left(i)} \leq j \leq \mathbf{right(i)}\}.$$

$$\begin{pmatrix} & & & & & & & & & 0 \\ & & & & & & & & 0 & x_{(2)} - x_{(1)} \\ & & & & & & & & & \vdots \\ & & & & & & & & & \vdots \\ & & & 0 & x_{(i)} - x_{(n-i+2)} & \cdots & x_{(i)} - x_{(n-j+1)} & \cdots & & x_{(i)} - x_{(1)} \\ & & & & & & & & & \vdots \\ & & & & & & & & & \vdots \\ 0 & x_{(n)} - x_{(n-1)} & \cdots & \cdots\ \cdots & x_{(n)} - x_{(n-i+2)} & \cdots & x_{(n)} - x_{(n-j+1)} & \cdots & & x_{(n)} - x_{(1)} \end{pmatrix}$$

Figure 1: The matrix $U = (x_{(i)} - x_{(n-j+1)})_{1 \leq i,j \leq n}$

By comparison with a certain trial value, one can make **left(i)** greater and **right(i)** smaller. Each refinement step takes $O(n)$ time, and there are $O(\log n)$ such steps. For the trial value, Johnson and Mizoguchi chose the weighted median of the medians of the rows in *Candidates* (with weight equal to their length). Since the computation of that weighted median takes a substantial fraction of the total time, Monahan (1984) proposed to use other trial values. This yields algorithms that are faster on average, but whose worst case time becomes larger than $O(n \log n)$.

For the computation of Q_n we note that

$$\{|x_i - x_j|; i < j\}_{(k)} = \{x_{(i)} - x_{(n-j+1)};\ 1 \leq i, j \leq n\}_{(k+n+\binom{n}{2})}$$

where $x_{(1)} \leq \ldots \leq x_{(n)}$ are the ordered observations. Thus if we take $X = \{x_{(1)}, \ldots, x_{(n)}\}$ and $Y = \{-x_{(n)}, \ldots, -x_{(1)}\}$ then we can apply the Johnson-Mizoguchi approach to the table $U = X + Y = (x_{(i)} - x_{(n-j+1)})_{1 \leq i,j \leq n}$ which is shown in Figure 1. We see that the elements in the upper triangle are negative or zero, so we initialize the arrays **left** and **right** by

$$\textbf{left(i)} = n - i + 2 \text{ and } \textbf{right(i)} \ = n$$

for all $i \geq 2$.

Remark. By definition, at most $k = \binom{h}{2}$ elements in the lower triangle can be smaller than the value we are looking for. Hence certain elements in the bottom right corner of U can be excluded beforehand, because each of them dominates more than k elements. Therefore, we could initialize **right** by

$$\textbf{right(i)} \ = n - (i - h)$$

for all $i > h$. But although this modification reduced the initial size of *Candidates* by roughly 25% it did not shorten the total computation time in a substantial way, so it is not included in the present version of the algorithm.

Source code: The function **Qn** implements the above algorithm. It needs a function called **whimed** for computing a weighted high median in $O(n)$ time. Both functions are provided here. Note that **Qn** could also be used to compute an order statistic for a different k, by changing one line in the source code. As in the function **Sn** we did not need double precision, because the only arithmetic operation on reals is the subtraction $x_i - x_j$.

```
cc
cc  Time-efficient algorithm for the scale estimator:
cc
cc       Qn = dn * 2.2219 * {|x_i-x_j|; i<j}_(k)
cc
cc  Parameters of the function Qn :
cc       x  : real array containing the observations
cc       n  : number of observations (n >=2)
cc
cc  The function Qn uses the procedures:
cc     whimed(a,iw,n): finds the weighted high median of an array
cc                     a of length n, using the array iw (also of
cc                     length n) with positive integer weights.
cc     sort(x,n,y) : sorts an array x of length n, and stores the
cc                   result in an array y (of size at least n)
cc     pull(a,n,k) : finds the k-th order statistic of an
cc                   array a of length n
cc
      function Qn(x,n)
      dimension x(n)
      dimension y(500),work(500)
      integer left(500),right(500),weight(500),Q(500),P(500)
      integer h,k,knew,jhelp,nL,nR,sumQ,sumP
      logical found
      h=n/2+1
      k=h*(h-1)/2
      call sort(x,n,y)
      do 20 i=1,n
          left(i)=n-i+2
          right(i)=n
20    continue
      jhelp=n*(n+1)/2
      knew=k+jhelp
      nL=jhelp
      nR=n*n
      found=.false.
200   continue
      if ( (nR-nL.gt.n).and.(.not.found) ) then
          j=1
          do 30 i=2,n
          if (left(i).le.right(i)) then
```

```
          weight(j)=right(i)-left(i)+1
          jhelp=left(i)+weight(j)/2
          work(j)=y(i)-y(n+1-jhelp)
          j=j+1
      endif
30    continue
      trial=whimed(work,weight,j-1)
      j=0
      do 40 i=n,1,-1
45        continue
          if ((j.lt.n).and.((y(i)-y(n-j)).lt.trial)) then
              j=j+1
              goto 45
          endif
          P(i)=j
40    continue
      j=n+1
      do 50 i=1,n
55        continue
          if ((y(i)-y(n-j+2)).gt.trial) then
              j=j-1
              goto 55
          endif
          Q(i)=j
50    continue
      sumP=0
      sumQ=0
      do 60 i=1,n
          sumP=sumP+P(i)
          sumQ=sumQ+Q(i)-1
60    continue
      if (knew.le.sumP) then
          do 70 i=1,n
              right(i)=P(i)
70        continue
          nR=sumP
      else
          if (knew.gt.sumQ) then
              do 80 i=1,n
                  left(i)=Q(i)
80            continue
              nL=sumQ
          else
              Qn=trial
              found=.true.
```

```
               endif
           endif
       goto 200
       endif
       if (.not.found) then
           j=1
           do 90 i=2,n
           if (left(i).le.right(i)) then
               do 100 jj=left(i),right(i)
                   work(j)=y(i)-y(n-jj+1)
                   j=j+1
100            continue
           endif
90         continue
           Qn=pull(work,j-1,knew-nL)
       endif
       if (n.le.9) then
           if (n.eq.2) dn=0.399
           if (n.eq.3) dn=0.994
           if (n.eq.4) dn=0.512
           if (n.eq.5) dn=0.844
           if (n.eq.6) dn=0.611
           if (n.eq.7) dn=0.857
           if (n.eq.8) dn=0.669
           if (n.eq.9) dn=0.872
       else
           if (mod(n,2).eq.1) dn=n/(n+1.4)
           if (mod(n,2).eq.0) dn=n/(n+3.8)
       endif
       Qn=dn*2.2219*Qn
       return
       end


cc
cc  Algorithm to compute the weighted high median in O(n) time.
cc
cc  The whimed is defined as the smallest a(j) such that the sum
cc  of the weights of all a(i) <= a(j) is strictly greater than
cc  half of the total weight.
cc
cc  Parameters of this function:
cc        a: real array containing the observations
cc        n: number of observations
cc       iw: array of integer weights of the observations.
```

```
cc
cc  This function uses the function pull.
cc
cc  The size of acand, iwcand must be at least n.
cc
      function whimed(a,iw,n)
      dimension a(n),iw(n)
      dimension acand(500),iwcand(500)
      integer wtotal,wrest,wleft,wmid,wright
      nn=n
      wtotal=0
      do 20 i=1,nn
          wtotal=wtotal+iw(i)
20    continue
      wrest=0
100   continue
      trial=pull(a,nn,nn/2+1)
      wleft=0
      wmid=0
      wright=0
      do 30 i=1,nn
          if (a(i).lt.trial) then
              wleft=wleft+iw(i)
          else
              if (a(i).gt.trial) then
                  wright=wright+iw(i)
              else
                  wmid=wmid+iw(i)
              endif
          endif
30    continue
      if ((2*wrest+2*wleft).gt.wtotal) then
          kcand=0
          do 40 i=1,nn
              if (a(i).lt.trial) then
                  kcand=kcand+1
                  acand(kcand)=a(i)
                  iwcand(kcand)=iw(i)
              endif
40        continue
          nn=kcand
      else
          if ((2*wrest+2*wleft+2*wmid).gt.wtotal) then
              whimed=trial
              return
```

```
        else
            kcand=0
            do 50 i=1,nn
                if(a(i).gt.trial) then
                    kcand=kcand+1
                    acand(kcand)=a(i)
                    iwcand(kcand)=iw(i)
                endif
50          continue
            nn=kcand
            wrest=wrest+wleft+wmid
        endif
    endif
    do 60 i=1,nn
        a(i)=acand(i)
        iw(i)=iwcand(i)
60  continue
    go to 100
    end
```

4. COMPUTATION TIMES

We have seen that the naive algorithms for S_n and Q_n require $O(n^2)$ time, whereas the more efficient algorithms constructed in this paper have a time complexity of $O(n \log n)$. However, this does not yet give us the proportionality constants hidden by the $O(.)$ notation, or tell us how large n has to be before these expressions provide a good approximation to actual computation times.

Table 2 lists the CPU times (in seconds) for each of the four algorithms and for several values of n. Each entry was computed as the average time over 100 or more runs. The

Table 2: Average computation time (in seconds) of the naive and efficient algorithms for S_n and Q_n

	S_n		Q_n	
n	naive	efficient	naive	efficient
10	0.011	0.009	0.008	0.013
20	0.024	0.017	0.016	0.026
40	0.077	0.034	0.042	0.056
60	0.153	0.050	0.085	0.088
80	0.260	0.068	0.137	0.122
100	0.394	0.086	0.210	0.158
200	1.502	0.185	0.795	0.342
300	3.340	0.284	1.900	0.559
400	5.930	0.390	3.240	0.900
500	9.130	0.500	4.760	1.000

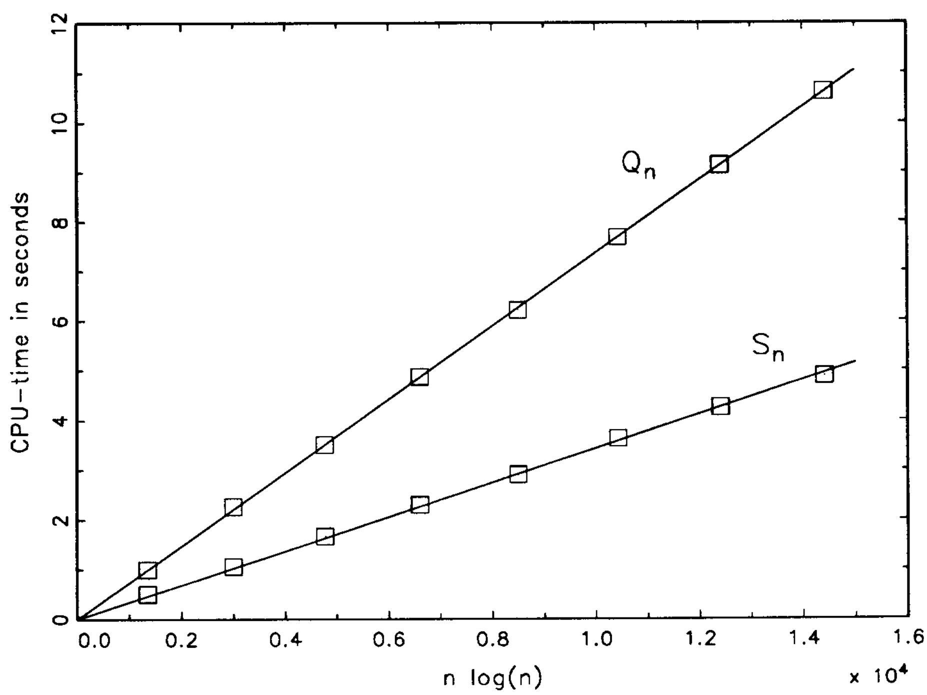

Figure 2: Computation time (in seconds) of the efficient algorithms versus $n \log n$

timings were carried out on a 386 PC with 33 MHz clock, running under DOS.

We see that the efficient algorithm for S_n is uniformly faster than the naive one. For Q_n this is only true when $n > 60$. For $n = 500$, the efficient Q_n is already 5 times faster than the naive Q_n, and the efficient S_n is 20 times faster than its naive counterpart.

Comparing both naive algorithms with each other, we find that time(S_n) $\sim n^2 \sim$ time(Q_n) with time(Q_n)/time(S_n) $\to 1/2$. This ratio can be explained by noting that for Q_n we select one order statistic among $n(n-1)/2$ elements. For S_n we select n order statistics, each among $n-1$ elements, which needs roughly the same time as one order statistic among $n(n-1)$ elements.

Figure 2 plots the computation time of the efficient algorithms versus $n \log n$ for values of n beyond those in Table 2, ranging from 500 to 4000. We see that the relations remain linear. This confirms that time(S_n) $\sim n \log n \sim$ time(Q_n). In this case we obtain time(Q_n)/time(S_n) ≈ 2, which is the opposite from what we found for the naive algorithms. When comparing the efficient algorithms of S_n and Q_n, it should also be noted that S_n needs far less storage than Q_n (although it is $O(n)$ in both cases), which is also to the advantage of S_n.

Acknowledgment: We wish to thank Carine Segers for helping with the Pascal versions of these algorithms.

REFERENCES

Bickel, P.J., and Lehmann, E.L. (1979), "Descriptive Statistics for Nonparametric Models IV: Spread," in *Contributions to Statistics, Hájek Memorial Volume,* ed. J. Jurečková, Prague: Academia, pp. 33–40.

Cheney, W., and Kincaid, D. (1985). *Numerical Methods and Computing,* Pacific Grove: California.

Hodges, J.L., Jr., and Lehmann, E.L. (1963), "Estimates of Location Based on Rank Tests," *Annals of Mathematical Statistics,* 34, 598–611.

Johnson, D.B., and Mizoguchi, T. (1978), "Selecting the Kth Element in $X + Y$ and $X_1 + X_2 + \ldots + X_m$," *SIAM Journal of Computing,* 7, 147–153.

Knuth, D.E. (1973). *The Art of Computer Programming, Vol. III, Sorting and Searching.* Reading, Massachusetts: Addison-Wesley.

McLeod, A.I. (1985), "Remark AS R58. A remark on algorithm AS 183", *Applied Statistics,* 34, 198–200.

Monahan, J.F. (1984), "Algorithm 616 – Fast computation of the Hodges-Lehmann estimator", *A.C.M. Trans. Math. Software,* 10, 265–270.

Rousseeuw, P.J., and Bassett, G.W., Jr. (1990), " The Remedian: A Robust Averaging Method for Large Data Sets," *Journal of the American Statistical Association,* 85, 97–104.

Rousseeuw, P.J., and Croux, C. (1993), "Alternatives to the Median Absolute Deviation," *Journal of the American Statistical Association,* 88, 1273–1283.

Shamos, M.I. (1976), "Geometry and Statistics: Problems at the Interface," in *New Directions and Recent Results in Algorithms and Complexity,* ed. J.F. Traub, New York: Academic Press, pp. 251–280.

Siegel, A.F. (1982), "Robust Regression using Repeated Medians," *Biometrika,* 69, 242–244.

Tukey, J.W. (1977), *Exploratory Data Analysis,* Reading, Massachusetts: Addison-Wesley.

Wichmann, B.A. and Hill, I.D. (1982), "Algorithm AS 183. An efficient and portable pseudo-random number generator," *Applied Statistics,* 31, 188–190.