Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 349 (2020) 81–102

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

# Improvements to the Robust Transform Based on the Weighted Median Operator Algorithm

Jesús E. Ramírez[1]

*Universidad Simón Bolívar*
*Caracas, Venezuela*

José Paredes[2]

*Universidad de Los Andes*
*Mérida, Venezuela*

Yudith Cardinale[3]

*Universidad Simón Bolívar*
*Caracas, Venezuela*

**Abstract**

The Robust Transform based on the Weighted Median operator algorithm calculates the transform of a signal when it has been exposed to impulsive noise. Since this algorithm demands very long execution time, it is not useful for real time signal processing systems. In this context, this work presents several strategies to improve its performance, such as the reduction of redundant calculations, optimization in the memory access, and a multithreads version of the algorithm. Besides, the original estimation method is modified to decrease even more the average execution time, keeping the quality level of the numeric results. The experimental results show a 30% performance improvement by reducing redundant calculations and optimizing the memory access, without making modifications to the estimation method and without using multi-threaded processing; 93% performance improvement by introducing modifications to the estimation method; and 97% performance improvement by incorporating the multi-threaded processing.

*Keywords:* Robust Transform, Weighted Median, Optimization, High Performance, Multhreading

## 1 Introduction

In the context of signal processing, a linear transformation consists in the mapping of a discrete signal from a domain to another, to expose characteristics of the signal that allow to process and analyze it in easier ways. When a signal is transmitted

[1] Email: 13-11170@usb.ve
[2] Email: jparedes@ula.edu.ve
[3] Email: ycardinale@usb.ve

https://doi.org/10.1016/j.entcs.2020.02.014

through a medium, it uses to be affected in a degenerative way. This effect is known as noise. Impulsive noise shows sharp increases in signals intensity, although the duration of these increases is short compared to the time between them. This kind of noises is usually modeled with distributions that have tails more weighted than those of the Gaussian Distribution, for example, the Laplacian Distribution.

In communications systems it is usually necessary to obtain the transformation of a signal that has been received through a channel with noise [1]. If this noise is impulsive in nature, it is necessary to use a robust method to calculate the transform, since impulsive noise is highly degenerative.

Several algorithms have been proposed in this context [2,3,4,5,6,7,8]. In particular, in a previous work [9], the Robust Transform based on the Weighted Medium operator (RTWM) algorithm was presented. Although these solutions for calculating robust transformations are effective, they demand high computational cost for the processing, due to their algorithmic complexities. Thus, their application in signal processing in real time is unsuitable. For this reason, it is necessary to develop less expensive algorithms in computational time that achieve competitive numerical precision.

In this sense, we propose several strategies to reduce the execution time of the RTWM algorithm, which consist on the reduction of redundant calculations, the use of the hierarchical memory system, and the use of multiple threads of execution. In addition, modifications are made to the original structure of the algorithm that allow improving the execution time while preserving the quality of the numerical results. The experimental results demonstrate an improvement in performance of up to 30% by reducing redundant calculations and optimizing the memory access, without making modifications to the estimation method and without using multithreaded processing; 93% performance improvement when introducing modifications to the estimation method; and 97% performance improvement by incorporating multi-threaded processing.

The reminder of this work is organized as follows. Section 2 describes the basis of the RTWM algorithm. In Section 3, we present the improvements related to redundant calculations, memory access, and the estimation method. Section 4 is dedicated to the multithreaded version. Experimental results are presented in Section 5. We conclude this work in Section 6.

## 2 Robust Transform based on the Weighted Median operator

The RTWM algorithm presented in [9], calculates the transformation of a discrete signal whose samples have been affected by impulse noise. RTWM is based on estimating the transform based on regression of minimum absolute deviation regularized by a $L1$ norm, which induces solutions that present dispersion in its components [10].

Discrete signals are represented as a vector $\mathbf{z} \in \mathbb{C}^n$. Calculating the transform of a signal consists in multiplying the signal by a transformation matrix $\Phi \in \mathbb{C}^{nxn}$ to obtain the transformed signal $\mathbf{x} \in \mathbb{C}^n$. This matrix is composed by columns

that represent orthogonal signals (i.e., its internal product is equal to 0), selected according to the characterization that is desired to be performed. The original signal can be recovered from its transformation by applying another transformation based on the inverse matrix $\Psi$ of the original transform; thus we have $\mathbf{z} = \Psi * \mathbf{x}$.

The information contained in the transformation of a signal calculated under the procedure described previously is seriously degraded when the original signal $\mathbf{z}$ has been subjected to noise, which can happen in its transmission medium, in the process of sampling, or at other stages of its manipulation. The sampled signal is characterized by $\mathbf{y} = \mathbf{z} + \mathbf{v}$. Where $\mathbf{z}$ is the original signal and $\mathbf{v} \in \mathbb{C}^n$, it is the noise vector. The components of the latter are assumed to be independent and identically distributed. When the noise vector $\mathbf{v}$ follows a Gaussian distribution with zero mean, the estimate of the transform is optimal under the principle of maximum probability, as the solution of the following optimization problem (see equation (1)):

$$\mathbf{x} = \arg\min_{\mathbf{x}} \|\mathbf{y} - \Psi * \mathbf{x}\|_2^2 \tag{1}$$

However, when noise follows a heavy-tailed distribution bigger than those of the Gaussian distribution, the performance of this method degrades significantly. The Laplace distribution is a distribution that maximizes the probability when the location parameter is adjusted to the median [11]. This distribution is usually used to model phenomena that have heavy-tailed distribution bigger than those of the Gaussian distribution, for example, impulsive noise.

The RTWM algorithm addresses the problem as a minimum absolute deviation regression regularized by a $L1$ norm, whose solution is found using the coordinate reduction method and the weighted median operator to calculate each coefficient of the transform. RTWM is based on the fact that under the Maximum A Posteriori (MAP) framework, the optimal transform is given by the solution to the problem of finding a $\mathbf{x}$ that minimizes $\|\mathbf{y} - \mathbf{Psi} * \mathbf{x}\|_1$; if the components of the noise vector are independent and identically distributed and follow a Laplacian distribution.

In certain practical applications, the transformation of a signal has a low percentage of non-zero coefficients. To favor solutions of this type, a term is added to the minimization problem that depends on the magnitude of the non-zero coefficients. This is the $L1$ norm of the solution vector scaled by a regularization parameter. Thus, the estimation of the robust transform is directed to solve the following optimization problem (see equation (2)):

$$\mathbf{x} = \arg\min_{\mathbf{x}} \{\|\mathbf{y} - \Psi * \mathbf{x}\|_1 + \lambda * \|\mathbf{x}\|_1\} \tag{2}$$

The selected method to estimate the solution vector in RTWM is based on a coordinated descent approach, which consists of reducing the N-dimensional problem to successive one-dimensional problems. The approach assumes that all entries of the estimated vector are known, with the exception of one of them. This allows the problem to be simplified to a widely known optimization problem, whose solution is

based on the weighted median operator (see equations (3) and (4)).

$$\mathbf{x}_k^{m+1} = \arg\min_{\mathbf{x}_k} \{\|\mathbf{y} - \Psi * \mathbf{x}^m + \Psi_k * \mathbf{x}_k^m - \Psi_k * x_k\|_1 + \lambda * \|\mathbf{x}^m\|_1\} \tag{3}$$

$$\mathbf{x}_k^{m+1} = \arg\min_{\mathbf{x}_k} \{\sum_{i=1}^{N} |\Psi_{k,i}| |\frac{(\mathbf{y} - \Psi * \mathbf{x}^m + \Psi_k * \mathbf{x}_k^m)_i}{\Psi_{k,i}} - x_k| + \lambda * |x_k|\} \tag{4}$$

In order to make explicit the characteristics of the function to be minimized in equation (4), a redefinition of its terms is proposed. The terms $\Psi_{k,i}$ are redefined as the components of a vector $\mathbf{w}$, while the terms $\frac{(\mathbf{y}-\Psi*\mathbf{x}^m+\Psi_k*\mathbf{x}_k^m)_i}{\Psi_{k,i}}$ are grouped as the components of a $\mathbf{z}$ vector. In addition, the term of penalty is attached in these vectors as it fits the same structure as the previous ones, with $\lambda$ being the last component of $\mathbf{w}$ and 0 the last one of $\mathbf{z}$. The definition of each of its inputs is defined in equations (5) and (6). This allows you to rewrite equation (4) as shown in equation (7).

$$\mathbf{z}_i = \begin{cases} \frac{(\mathbf{y} - \Psi * \mathbf{x}^m + \Psi_k * \mathbf{x}_k^m)_i}{\Psi_{k,i}} & for\ i = 1, ..., N \\ 0 & i = N + 1 \end{cases} \tag{5}$$

$$\mathbf{w}_i = \begin{cases} |\Psi_{k,i}| & for\ i = 1, ..., N \\ \lambda & i = N + 1 \end{cases} \tag{6}$$

$$\mathbf{x}_k^{m+1} = \arg\min_{\mathbf{x}_k} \sum_{i=1}^{N+1} |\mathbf{w}_i||\mathbf{z}_i - \mathbf{x}_k| \tag{7}$$

This is a piecewise linear function that has the particularity of being convex, therefore it has at least one minimum point. The minimum point of this type of functions can be found using the weighted median operator. The weighted median of a set of values with an associated weight with each of them is calculated as follows: the values $\mathbf{z}_i$ are sorted and then from lowest to highest, weights are added until the sum cumulative exceed or equal the value $W_o = \frac{1}{2}\sum_{i=1}^{N+1} \mathbf{w}_i$. Thus, the last $\mathbf{z}_i$ considered is the weighted median.

This operator, applied to the parameters of the convex piecewise linear functions, calculates the point at which they reach their minimum point. When reviewing the definition of this operator, it can be seen that in this context, it accumulates the slopes until the sum exceeds or equals half of the total sum of the slopes. It can be proved, that when the independent variable is equal to the weighted median of the $\mathbf{z}_i$, the function reaches its minimum point (see equation (8)).

$$\mathbf{x}_k^{m+1} = MEDIAN\ ((\mathbf{w}_i \diamond \mathbf{z}_i)|_{i=1}^{N+1}) \tag{8}$$

The approach described above is used in the construction of an iterative estimation method: starting from a null vector, each of its components is estimated in succession, setting all others to the value estimated in the previous iteration. This procedure is repeated starting from the last estimated vector until the error in the initial optimization problem converges to an appropriate value or once a certain

number of iterations has been made. The method used to determine the weighted median in each of the iterations has an important influence on the performance of the algorithm. In this work we use the method described in [12], whose algorithmic complexity is $O(N)$, with $N$ being the size of the vector, and consists of selecting the pivot so that it is an element close to the median and thus increase the amount of items discarded per iteration.

The optimal value of the parameter $\lambda$ in equation (2) is $\lambda = \frac{\theta_{\mathbf{x}}}{\theta_{\mathbf{x}_k}}$ according to the used framework. These involved parameters are unknown. According to experiments carried out in citebib:DXA, it is recommended that the $\theta_x$ parameter be set for each component. This suggests setting a different regularization parameter for each component to be estimated. Under the MAP framework, it is demonstrable that $\theta_{\mathbf{x}_k}$ can be estimated with $|\mathbf{x}_k|$. This leads to the fact that the optimum value of the regularization parameter depends on the value of the component to be estimated. The closest known value is that obtained in the previous iteration and is therefore used for estimation. On the other hand, since the value of the numerator is unknown, it is suggested that the regularization parameter be refined as the iterations happen and the solution vector converges. In this way, initially this parameter is given high values to favor dispersion, decreasing on each iteration; thus, it is possible to have non-null components with less impact on the constitution of the estimated vector.

The regularization parameter is decremented exponentially, which provides the necessary smoothness so that all non-null components of the vector can arise as iterations are performed. Therefore, we have equation (9) to model the regularization parameter.

$$\lambda_k^{m+1} = \frac{\tau_m}{|\mathbf{x}_k^m|} \tag{9}$$

Where $\tau_m = \tau_0 * \beta^m$, $\tau_0$ is a constant and $\beta$ is the decay parameter, which is usually a value close to 1 to ensure smoothness. However, since the components may be null, a $\epsilon$ is added to the denominator to avoid division by zero. The regularization parameter is calculated with the expression given in equation (10).

$$\lambda_k^{m+1} = \frac{\tau}{\epsilon + |\mathbf{x}_k^m|} \tag{10}$$

Now that all aspects of RTWM have been described, the Algorithm 1 is presented in pseudocode for clarity. Algorithm 1 is not exactly the same as the one presented in [9], whose algorithmic complexity is $O(itmax * N^3)$, being $itmax$ the number of iterations and $N$ the size of the vector. However, with a simple modification (i.e., rearranging the iterative cycles, a calculation of $O(N)$ was reduced to $O(1)$ remaining constant in all iterations), we achieved the Algorithm 1 of complexity $O(itmax * N^2)$.

**Algoritmo 1** RTWM. Original $O(itmax * N^2)$

```
1: function RTWM(y, Ψ, N, β, τ0, tolerance, iterationsNumber)
2:     x ← 0; y' ← 0; re ← 0; im ← 0; w' ← 0; threshold ← τ0
3:     for m = 0, m < iterationsNumber, m++ do
4:         for k = 0, k < N, k++ do
5:             λ ← threshold / (ϵ+||x[k]||)
```

6: $\mathbf{z}[0] \leftarrow 0$; $\mathbf{w}[0] \leftarrow \lambda$
7: **for** $i = 0, j = 1,\ i < N,\ i{+}{+}$ **do**
8: **if** $\|\Psi[i][k]\| \neq 0$ **then**
9: $\mathbf{z}[j] \leftarrow \frac{\mathbf{y}[i] - \mathbf{y}'[i] + \mathbf{x}[k] * \Psi[i][k]}{\Psi[i][k]}$
10: $\mathbf{w}[j] \leftarrow \|\Psi[i][k]\|$
11: $j{+}{+}$
12: $\mathbf{x}[k] \leftarrow getWeightedMedian(\mathbf{z}, \mathbf{w}, , \mathbf{w}', \mathbf{re}, \mathbf{im}, j)$
13: $\mathbf{y}' \leftarrow \Psi * \mathbf{x}$
14: $energy \leftarrow \frac{\|\mathbf{y} - \mathbf{y}'\|_2{}^2}{\|\mathbf{y}\|_2{}^2}$
15: **if** $energy < tolerance$ **then**
16: *break*
17: $threshold \leftarrow threshold * \beta$
18: **return x**

The algorithm receives as parameters the signal from which we want to obtain the transform $y$, the inverse matrix of the transform $\Psi$, the length of the signal to process $N$, the exponential decay parameter for regularization $\beta$, the initial value for the regularization parameter $\tau_0$, the minimum value required in the normalized error energy with respect to the original signal *tolerance*, and the number of iterations to be performed in the estimate. On line 2, the vectors that maintain the solution estimate for the $m$ and $m + 1$ iterations are initialized, and the initial boundary value for the calculation of the regularization parameter. The cycle that begins on line 3 estimates the solution the number of times indicated. The estimation of each of the components of the vector $\mathbf{x}$ is done within the cycle of line 4. There, the regularization parameter for the component to be estimated is calculated. From lines 7 to 11, the vectors $\mathbf{w}$ and $\mathbf{z}$ are calculated, placing the one linked to the regularization parameter as the first term and the others are calculated according to the equations described above. On line 12, estimate the current coordinate using the weighted median operator on the vectors $\mathbf{z}$ and $\mathbf{w}$. On lines 13 and 14 the vector $\mathbf{y}'$ is calculated with the last estimated solution and the normalized error energy is found between $\mathbf{y}'$ and $\mathbf{y}$. If this energy is below the minimum tolerance, the algorithm stops to return the last estimate of $\mathbf{x}$, else, the boundary value is decremented exponentially with the $\beta$ factor (lines 15 to 18).

## 3 Improvements to the RTWM's Performance

The performance improvement process of the RTWM calculation algorithm is divided into several stages. Each of them focuses on improving the proposed implementation in [9], and described in Section 2, so as to reduce the total execution time. This is justified because the practical applications of this algorithm aim to process in real time signals received by telecommunications devices. This implies that the algorithm must be suitable to run on devices with low computing power that can be integrated into the data processing systems. Each of the following sections presents incrementally, the improved versions until reaching the development of the final algorithm.

### *3.1* Version 1: Simplification of calculations

The calculation of the vector $\mathbf{z}$ in the current state of the algorithm is based on the equation (11). It can be seen that the previous value of the component being estimated is involved in $N$ multiplications and then in $N$ sums. Using algebra, the equation (11) can be simplified to replace these operations only with sums, as shown by the equation (12).

$$\mathbf{z} \leftarrow \frac{\mathbf{y}_i - \mathbf{y}'_i + \mathbf{x}_k * \Psi_{i,k}}{\Psi_{i,k}} \tag{11}$$

$$\mathbf{z}_j \leftarrow \frac{\mathbf{y}_i - \mathbf{y}'_i}{\Psi_{i,k}} + \mathbf{x}_k \tag{12}$$

This new expression reveals that all the components of the vector $\mathbf{z}$ are being summed by the previous value of the component being estimated. When considering that a $\mathbf{z}$ entry will be the new value of the component $\mathbf{x}_k$, the term $\frac{\mathbf{y}_i - \mathbf{y}'_i}{\Psi_{i,k}}$ is the difference between the previous and the new value. Therefore, it is right to remove the sum of the value of the previous component and subsequently, instead of replacing the value in $\mathbf{x}_k$, just add the difference necessary to perform the update. This is stated in the equations (13) and (14). It should be noted that this optimization assumes that once the component $\mathbf{x}_k$ has taken a non-zero value, it will not take this value again by selecting the element related to the regularization parameter in the weighted median. This is valid because the regularization parameter has a mainly decreasing behavior, although its value may certainly increase due to a reduction in the absolute value of $\mathbf{x}_k$ in the estimation process. However, if the trend in the estimation is to reduce the absolute value of $\mathbf{x}_k$, the algorithm will achieve it in subsequent iterations once the regularization parameter has been sufficiently decremented by the exponential decay and selected another element as a corrector through the weighted median operator.

$$\mathbf{z}_j \leftarrow \frac{\mathbf{y}_i - \mathbf{y}'_i}{\Psi_{i,k}} \tag{13}$$

$$\mathbf{x}_k += getWeightedMedian(\mathbf{z}, \mathbf{w}) \tag{14}$$

This optimization represents to save $N$ multiplications and $N$ sums, replacing them with just one sum. This is an important optimization because operations performed with complex floating point numbers and double precision are expensive at processing time. The Algorithm 2 shows the modifications made with respect to the original version presented in Section 2 on the lines marked in red.

**Algoritmo 2** RTWM. Version 1: Simplification of calculations

```
1: function RTWM(y, Ψ, N, β, τ0, tolerance, iterationsNumber)
2:     x ← 0; y' ← 0; re ← 0; im ← 0; w' ← 0; threshold ← τ0
3:     for m = 0, m < iterationsNumber, m++ do
4:         for k = 0, k < N, k++ do
5:             λ ← threshold / (ϵ + ||x[k]||)
6:             z[0] ← 0; w[0] ← λ
7:             for i = 0, j = 1, i < N, i++ do
8:                 if ||Ψ[i][k]|| ≠ 0 then
```

$\mathbf{z}[j] \leftarrow \frac{\mathbf{y}[i]-\mathbf{y}'[i]}{\Psi[i][k]}$
$\mathbf{w}[j] \leftarrow \|\Psi[i][k]\|$
$j{+}{+}$
$\mathbf{x}[k] \leftarrow \mathbf{x}[k] + getWeightedMedian(\mathbf{z}, \mathbf{w}, \mathbf{w}', \mathbf{re}, \mathbf{im}, j)$
$\mathbf{y}' \leftarrow \Psi * \mathbf{x}$
$energy \leftarrow \frac{\|\mathbf{y}-\mathbf{y}'\|_2{}^2}{\|\mathbf{y}\|_2{}^2}$
**if** $energy < tolerance$ **then**
  *break*
$threshold \leftarrow threshold * \beta$
**return x**

### *3.2* Version 2: Memory access improvements

In the algorithms that are characterized by constantly accessing memory to obtain the necessary data for their calculations, it is very important to carefully consider the access patterns. This is because the memory access times are much longer than the times in which the arithmetic operations are executed in the processor. To alleviate this bottleneck, most current processors have a hierarchical memory system, which is characterized by having various memory devices in which the access time is inversely proportional to its data storage capacity. One of the advantages of the hierarchical memory system is that the memory handler, when a data is accessed in the main memory, not only copies this data to one of the devices with the lowest latency (i.e., cache memory), but a set of data next to him, called page. The purpose of this operation is to take advantage of the high probability that next instructions need to access this data and have it available on low-latency devices, such as cache memory.

To make efficient use of the hierarchical memory system it is important that the algorithm implementation processes data that is located contiguously. The pattern of access to a matrix is a critical point that can generate a bottleneck by memory access. Matrices can be considered as an array of arrays. Therefore, if the data is required in such a way that a different array is accessed in each iteration, the hierarchical memory system will not be used efficiently since no contiguous data is accessed. Each programming language has a way to align the data in a matrix. In the case of C and C ++ languages, the alignment is done at the row level. That is, each row can be considered as an array of data stored contiguously.

RTWM requires to access a column vector of the matrix $\Psi$ to estimate each component of the solution vector and determine vectors $\mathbf{z}$ and $\mathbf{w}$. When accessing the matrix through its column vectors, the hierarchical memory system is not being used efficiently, since the data accessed is not stored in contiguous areas. To solve this problem, it is possible to use the transposed matrix $\Psi$, allowing to access the columns of $\Psi$ through the rows of the transposed. Therefore, it allows an access pattern that efficiently uses the memory system. However, the use of the transposed matrix forces to double the amount of memory used by the algorithm. As the original matrix is still necessary due to its use to calculate the vector $\Psi * \mathbf{x}$ in which the matrix is accessed through its rows.

Another improvement in regard to memory access is based on the alignment of memory arrangements with respect to the size of virtual memory pages. Since the memory handler copies the page on which the data required by the processor is located and stores it in a low-latency device, it is convenient that on that page there

be as much data as possible that will be accessed shortly. This can be ensured if the required memory is allocated by the memory handler in a position that is multiple of the page size. In this way the arrangement is adjusted so that it occupies as few pages as possible. A block of memory that is not aligned requires $k + 1$ access to main memory, one for each page that contains it. This block can be contained in only $k$ pages if it is aligned with a multiple of the number of pages. This prevents access to main memory, which is expensive at runtime. To implement alignment with page size, the Linux kernel, in conjunction with the POSIX library, offers a function called `posix_memalign` analogous to the classic `malloc`. It allocates blocks of memory at run time aligned with memory addresses of the indicated size, which in this case would be the page size. Algorithm 3 presents the improvements made through the transposed matrix and the location in the memory accesses.

**Algoritmo 3** RTWM. Version 2: Cache access improvements

```
1: function RTWM(y, Ψ, N, β, τ0, tolerance, iterationsNumber)
2:     x ← 0; y′ ← 0; re ← 0; im ← 0; w′ ← 0
3:     Ψ^T ← Transposed(Ψ)
4:     threshold ← τ0
5:     for m = 0, m < iterationsNumber, m++ do
6:         for k = 0, k < N, k++ do
7:             for i = 0, j = 0, i < N, i++ do
8:                 if ||Ψ^T[k][i]|| ≠ 0 then
9:                     z[j] ← (y[i] − y′[i]) / Ψ^T[k][i]
10:                    w[j] ← ||Ψ^T[k][i]||
11:                    j++
12:            z[j] ← 0
13:            w[j] ← threshold / (ϵ + ||x[k]||)
14:            j++
15:            x[k] ← x[k] + getWeightedMedian(z, w, w′, re, im, j)
16:        y′ ← Ψ ∗ x
17:        energy ← ||y − y′||_2^2 / ||y||_2^2
18:        if energy < tolerance then
19:            break
20:        threshold ← threshold ∗ β
21:    return x
```

### *3.3* Version 3: Estimate based on the latest available information

The original algorithm proposes to estimate each component using only the vector estimated in the previous iteration, as can be seen in the equations (5), (6), and (7). This approach does not use the latest information available to the algorithm. When estimating the component $k$, all components $\mathbf{x}_i^m$ in the range $0 \leq i < k$ have already been estimated. These values are better estimates of the solution vector than those belonging to the previous iteration. It is proposed to use the $\mathbf{y}'$ vector with the most recent information available for each component.

The proposed solution consists in subtracting the contribution of the previous value of the component from the vector $\mathbf{y}'$ and adding the contribution of the last estimated value. When grouping terms, this is equivalent to adding the difference between the last two values of the component by scaling to the corresponding column vector $\mathbf{y}'$. The difference that needs to be added is the result obtained from applying

the weighted median operator on the vector $\mathbf{z}$ with weights $\mathbf{w}$ (see equation (15)).

$$\mathbf{y'} \leftarrow \mathbf{y'} + (\mathbf{x}_k^{m+1} - \mathbf{x}_k^m) * \Psi_{\mathbf{k}} \tag{15}$$

This keeps the vector $\mathbf{y}$ updated with the latest information by simply adding a couple of vector addition operations. It also eliminates matrix-vector multiplication at the end of each outer iteration, since the calculation of matrix multiplication is distributed to a sum of vectors in the estimation of each component. Therefore, the need for the original $\Psi$ matrix is eliminated, being able to work only with the transposed. Thus, by transposing the matrix in the same memory space, the increase in the use of memory introduced in Version 2 (efficient data access) is eliminated. The Algorithm 4 shows the estimate using the obtained latest information.

**Algoritmo 4** RTWM. Version 3: Use of obtained latest information

**function** RTWM($\mathbf{y}, \Psi, N, \beta, \tau_0, tolerance, iterationsNumber$)
  $\mathbf{x} \leftarrow \mathbf{0}$; $\mathbf{y'} \leftarrow \mathbf{0}$; $\mathbf{re} \leftarrow \mathbf{0}$; $\mathbf{im} \leftarrow \mathbf{0}$; $\mathbf{w'} \leftarrow \mathbf{0}$
  $\Psi^T \leftarrow Transposed(\Psi)$
  $threshold \leftarrow \tau_0$
  **for** $m = 0,\ m < iterationsNumber,\ m{+}{+}$ **do**
    **for** $k = 0,\ k < N,\ k{+}{+}$ **do**
      **for** $i = 0, j = 0,\ i < N,\ i{+}{+}$ **do**
        **if** $\|\Psi^T[k][i]\| \neq 0$ **then**
          $\mathbf{z}[j] \leftarrow \frac{\mathbf{y}[i] - \mathbf{y'}[i]}{\Psi^T[k][i]}$
          $\mathbf{w}[j] \leftarrow \|\Psi^T[k][i]\|$
          $j{+}{+}$
      $\mathbf{z}[j] \leftarrow 0$
      $\mathbf{w}[j] \leftarrow \frac{threshold}{\epsilon + \|\mathbf{x}[k]\|}$
      $j{+}{+}$
      $diff_X \leftarrow getWeightedMedian(\mathbf{z}, \mathbf{w}, \mathbf{w'}, \mathbf{re}, \mathbf{im}, j)$
      $\mathbf{x}[k] \leftarrow \mathbf{x}[k] + diff_X$
      $\mathbf{y'} \leftarrow \mathbf{y'} + diff_X * \Psi^T[k]$
    $energy \leftarrow \frac{\|\mathbf{y} - \mathbf{y'}\|_2{}^2}{\|\mathbf{y}\|_2{}^2}$
    **if** $energy < tolerance$ **then**
      $break$
    $threshold \leftarrow threshold * \beta$
  **return** $\mathbf{x}$

### *3.4* Version 4: Acceleration in the decrease of the regularization parameter

The direct way to calculate the transformation of a signal that has been subjected to impulsive noise consists in applying matrix-vector multiplication, as would be done if the vector was not subjected to noise. Figure 1, presents the normalized energy of the error as a function of dispersion, applying this method to calculate the transform and the best results obtained using the RTWM algorithm.

It can be seen that for the matrix-vector multiplication, the error decreases along with the dispersion. This is naturally due to the ratio between noise and signal energy, which becomes lower as there is a greater number of non-zero components in the transform. In addition, the RTWM algorithm has an opposite behavior in the estimation error. It starts with a very high error compared to that obtained by the direct method and decreases as the signal dispersion increases. This shows that the algorithm does not have a good performance when executed in signals with little dispersion. According to Figure 1, the algorithm offers numerical gains compared to

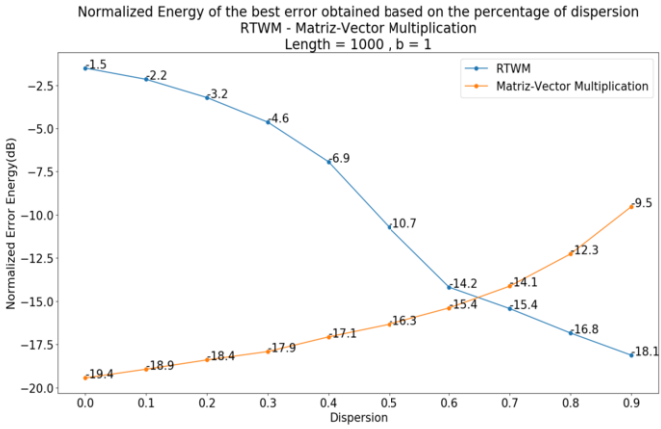


Fig. 1. Minimum error obtained by RTWM for a Laplacian noise with $b = 1$.

the direct method from 65% dispersion. From these limitations of the algorithm, it works properly only for signals whose transform has high levels of dispersion.

Figures 2 and 3 present the convergence based on the iterations of the current algorithm in the estimation of a vector with 1000 inputs, for two levels of dispersion in which the algorithm exceeds the performance of the matrix-vector multiplication method. A length of $N = 1000$, $\beta = 0.95$, and a Laplacian noise with scale parameter $b = 1$ have been used. It shows that for the first iterations, no progress is made towards convergence. This can be attributed to the fact that in this region the regularization parameter has not been reduced enough to allow the appearance of non-null components and thus reduce the error in the estimate.

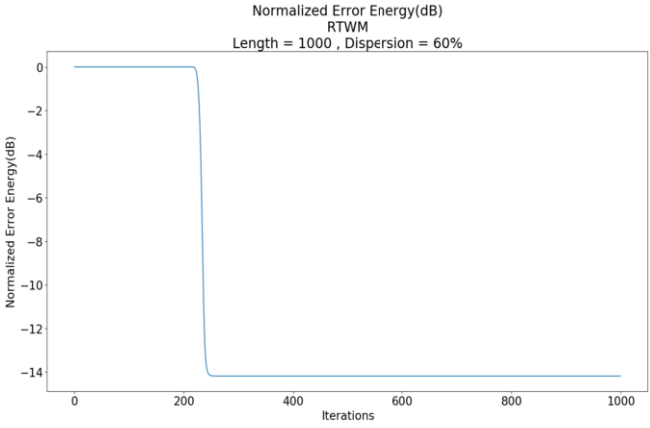


Fig. 2. Normalized error energy based on the iterations performed. Dispersion: 60%

In order to reduce the number of iterations in which convergence is not advanced, it is proposed to modify the algorithm in such a way that the normalized error of the previous iteration is stored. If the standardized error of the current iteration is not reduced by at least a percentage $p$ with respect to the previous one, the regularization parameter is automatically reduced in the equivalent of $S$ iterations, reducing by the

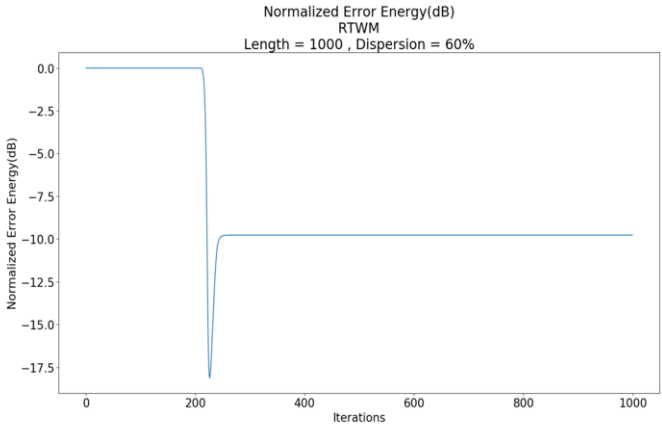


Fig. 3. Normalized error energy based on the iterations performed. Dispersion: 90%

same amount the number of iterations to perform. This generates a reduction in the execution time of the algorithm, since a smaller number of iterations is performed in which convergence does not improve. The values $p$ and $S$ are parameters of the algorithm and can be selected according to how much smoothness is required in the decay of the regularization parameter. In the Algorithm 5 the modifications with respect to the acceleration parameter and the jump in iterations are presented. This improvement, like Version 3, structurally affects the original algorithm.

**Algoritmo 5** RTWM. Version 4: Use of latest information obtained. Acceleration with jumps

**function** RTWM($\mathbf{y}, \Psi, N, , \tau_0, tolerance, iterationsNumber$)
  $\mathbf{x} \leftarrow \mathbf{0}$; $\mathbf{y'} \leftarrow \mathbf{0}$; $\mathbf{re} \leftarrow \mathbf{0}$; $\mathbf{im} \leftarrow \mathbf{0}$; $\mathbf{w'} \leftarrow \mathbf{0}$; $\Psi^T \leftarrow Transposed(\Psi)$
  $threshold \leftarrow \tau_0$
  $acceleration \leftarrow 1$
  **for** $i = 0,\ i < JUMP,\ i{+}{+}$ **do**
    $acceleration \leftarrow acceleration * \beta$
    $previousEnergy \leftarrow -1$
  **for** $m = 0,\ m < iterationsNumber,\ m{+}{+}$ **do**
    **for** $k = 0,\ k < N,\ k{+}{+}$ **do**
      **for** $i = 0, j = 0,\ i < N,\ i{+}{+}$ **do**
        **if** $\|\Psi^T[k][i]\| \neq 0$ **then**
          $\mathbf{z}[j] \leftarrow \frac{\mathbf{y}[i] - \mathbf{y'}[i]}{\Psi^T[k][i]}$
          $\mathbf{w}[j] \leftarrow \|\Psi^T[k][i]\|$
          $j{+}{+}$
      $\mathbf{z}[j] \leftarrow 0$
      $\mathbf{w}[j] \leftarrow \frac{threshold}{\epsilon + \|\mathbf{x}[k]\|}$
      $j{+}{+}$
      $diff_X \leftarrow getWeightedMedian(\mathbf{z}, \mathbf{w}, \mathbf{w'}, \mathbf{re}, \mathbf{im}, j)$
      $\mathbf{x}[k] \leftarrow \mathbf{x}[k] + diff_X$
      $\mathbf{y'} \leftarrow \mathbf{y'} + diff_X * \Psi[k]$
    $energy \leftarrow \frac{\|\mathbf{y} - \mathbf{y'}\|_2{}^2}{\|\mathbf{y}\|_2{}^2}$
    **if** $previousEnergy > 0 \&\& \frac{energy}{previousEnergy} > MINCHANGE$ **then**
      $threshold \leftarrow threshold * acceleration$
      $m \leftarrow m + JUMP$
    $previousEnergy \leftarrow energy$
    $threshold \leftarrow threshold * \beta$
  **return** $\mathbf{x}$

# 4 Parallelization using multiple execution threads (Version 5)

The parallelization scheme proposed for the RTWM algorithm consists on distributing the estimation of the components of the transformed vector among the execution threads. In this way, the inner loop executed on each iteration is parallelized. This implies that each thread must be able to calculate the vectors $\mathbf{z}$ and $\mathbf{w}$, calculate their weighted median, and update the vector $\mathbf{x}$. In addition, the results of these updates must be gathered to construct the vector $\mathbf{y}'$, with which the energy of the error reached in the current external iteration has to be checked.

Each thread must have private and shared memory resources, as well as synchronization mechanisms that allow avoiding race conditions when accessing shared data. This should be done trying to minimize the amount of private memory and the number of instructions made within critical sections (i.e., code sections where shared variables are accessed).

**Shared memory:** In the RTWM algorithm, the data structure that consumes a greatest amount of resources is the inverse transformation matrix. This consists of $N^2$ pairs of floating point numbers. This matrix and the $\mathbf{y}$ vector are not modified during execution, their entries are only used for reading. Because of this, these data structures are made available to the threads as shared memory between them.

The results of the algorithm are mainly the vector $\mathbf{x}$, which contains the estimated transform and the vector $\mathbf{y}'$ that maintains the estimate of the inverse transform. The vector $\mathbf{y}'$ is discarded at the end of the iterations, however, it maintains the information with which the estimation of the components of the vector $\mathbf{x}$ are made. Therefore, the vector $\mathbf{y}'$ is the main channel through which threads can expose their results when working together. For this reason, the $\mathbf{y}'$ and $\mathbf{x}$ vectors are stored in the shared memory. However, to avoid excessive synchronization instructions, each thread keeps a copy of the $\mathbf{y}'$ vector in its private memory and, at the end of each outer iteration, this copy vector is exchanged with $\mathbf{y}'$, which has the results of the last iteration performed.

Finally, synchronization instruments that allow access to critical sections of the code (semaphores and `mutex`) are stored in the shared memory.

bf Private memory: Each thread should be able to perform the operations to estimate the value of the components of the vector $\mathbf{x}$. This requires the calculation of the vectors $\mathbf{z}$ and $\mathbf{w}$ to which the weighted median operator is applied. Thus, this pair of vectors are necessary in each thread and stored in private memory.

In order to gather the results of the estimates of all the threads in the vector $\mathbf{y}'$, the same idea implemented in the optimization of use of the latest available information is used. This involves parallelizing the matrix-vector multiplication by adding $\mathbf{y}'$ to the vector columns scaled by the difference introduced in the iteration. Since access to the vector $\mathbf{y}'$ must be synchronized and in order to reduce the amount of access to it, the sum of each thread is accumulated in a private vector

**d**. In this way, at the end of the estimation of the components that correspond to them, the threads can add up their accumulated contribution within the vector $\mathbf{y}'$. The vector **d** must be reset to zero after each outer iteration.

**Distribution of the data to be processed:** Each thread has a set of components to estimate. The assignment of these consists of dividing the entire vector into contiguous sets of components and assigning a set to each thread.

**Structural improvements:** Incorporating the acceleration in the decay of the regularization parameter, requires that the threads be able to identify whether they should accelerate the decrease of the regularization parameter. To achieve this, the error energy obtained in the last iteration is placed in shared memory. When a thread performs the energy check, it exposes the value obtained through shared memory.

The multi-threaded algorithm consists of the initialization and control section, and the thread execution section. Algorithms 6 and 7 present these sections respectively.

**Algoritmo 6** RTWMC. Version 5: Multi-threads. Acceleration. Latest information

```
    function RTWMC(y, Ψ, N, β, tolerance, iterationsNumber, τ0,
    threadsNumber)
2:      x ← 0; y' ← 0; Transpose(Ψ)
        beginCounter ← 0; endCounter ← 0
4:      energy_y ← −1; energy_e ← 0
        width ← N / threadsNumber
6:      for k = 0, k < threadsNumber − 1, k++ do                ▷ Create Threads
            L ← k∗width; R ← (k+1)∗width
8:          RTWMCthread(y, y', x, Ψ, L, R, N, β, tolerance,
    iterationsNumber, τ0, &energy_y, &energy_e, &beginCounter,
    &endCounter, threadsNumber)
        L ← (threadsNumber − 1)∗width
10:     R ← N
        RTWMCthread(y, y', x, Ψ, L, R, N, β, tolerance,
    iterationsNumber, τ0, &energy_y, &energy_e, &beginCounter,
    &endCounter, threadsNumber)
12:     joinThreads()                          ▷ Wait for the threads to finish running
        return x                                          ▷ Return the estimated transform
```

**Algoritmo 7** Thread for RTWMC. Version 5: Acceleration. Latest information

```
    procedure RTWMCthread(y, y', x, Ψ, L, R, N, β, tolerance,
    iterationsNumber, τ0, ∗energy_y, ∗energy_e, ∗beginCounter,
    ∗ endCounter, threadsNumber)
2:      ; previousEnergy ← −1; threshold ← τ0; acceleration ← 1

4:      for i = 0, i < JUMP, i++ do
            acceleration ← acceleration ∗ β
6:          for m = 0, m < iterationsNumber, m++ do
                difference ← 0
8:              for k = L, k < R, k++ do
                    for i = 0, j = 1, i < N, i++ do
10:                     if ‖Ψ[k][i]‖ ≠ 0 then
                            z[j] ← (Y[i] − privateY'[i]) / Ψ[k][i]
12:                         w[j] ← ‖Ψ[k][i]‖
                            j++
14:                 z[j] ← 0
                    w[j] ← threshold / (ϵ + ‖x[k]‖)
16:                 j++
                    XDiff ← getWeightedMedian(z, w, w', re, im, j)
18:                 difference ← difference + XDiff ∗ Ψ[k]
                    privateY' ← privateY' + XDiff ∗ Ψ[k]
```

```
20:             x[k] ← x[k] + XDiff
                aquireMutex()
22:             while *beginCounter > 0 do
                    releaseMutex()
24:                 wait()
                    aquireMutex()
26:             y'← y'+difference
                *endCounter ← *endCounter + 1
28:             if *endCounter == threadsNumber then
                    if *energy_y == −1 then
30:                     *energy_y ← ||y||_2^2
                    *energy_e ← ||y−y'||_2^2 / *energy_y
32:                 if *energy_e < tolerance then
                        *endCounter ← −1
34:                 else
                        *endCounter ← 0
36:                 signalAll()
                while *endCounter > 0 do
38:                 releaseMutex()
                    wait()
40:                 aquireMutex()
                if *endCounter == −1 then
42:                 signalAll()
                    releaseMutex()
44:                 break
                else
46:                 if *beginCounter == threadsNumber then
                        *beginCounter ← 0
48:             if previousEnergy > 0&&*energy_e / previousEnergy > MINCHANGE then
                    threshold ← threshold * acceleration
50:                 m ← m + JUMP
                previousEnergy ← *energy_e
52:             privateY' ← y'
                signalAll()
54:             releaseMutex()
                threshold ← threshold * β
56:         return
```

# 5 Experimental Results

This section presents the results of the experiments performed to measure the total execution time as well as the quality of the results of the different versions of the developed algorithm. An incremental presentation is used that compares the execution times of each version with the original version.

## *5.1 Description of the experiments performed*

For the behavior's analysis of the original version of RTWM, the average execution time of executions on vectors of 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1500 and 2000 components was obtained. Each test with a specific vector size is executed 20 times and the median of all its execution times is taken. In this way, graphs describing the execution time of the algorithm can be obtained based on the quantity of components of the vector to be estimated. In addition, this is done in order to expose the algorithm to conditions similar to those that would be found in practical applications.

To bring the simulation closer to real conditions, the algorithm is used to calculate the discrete Fourier transform (DFT). In addition, it is known that in many practical applications, the transformed vector usually has considerable levels of dis-

persion and is symmetric. Therefore, to obtain vectors that, when applied the DFT, generates transformed with these characteristics, the transformed vector is first generated by introducing dispersion and symmetry. Then, using the inverse transform, IDFT, the vector representing the signal without noise is calculated. The last step is to add Laplacian noise in the components of the vector obtained following the previous procedure.

In the case of optimizations that involve changes in the original algorithm, graphs are presented that compare the energy of the standardized error obtained, to validate that the modifications made maintain the quality of the results. The numerical tests have been performed in the estimation of vectors of 1000 components and using a parameter $\beta = 0.95$ for the exponential decay of the regularization parameter, as used in [9]. The noise used for the contamination follows a Laplacian distribution with zero mean and scale parameter $b = 1$. The number of iterations dedicated to the estimation is 1000 and the algorithm stop parameter based on a minimum in error energy is not used. The latter allows to appreciate in greater detail the numerical behavior of the estimate. Finally, the tests were performed on fixed dispersion percentages ranging from 0% to 90%. The experiments are carried out on an Intel Core i7-7500U computer, with four cores and 16GB of RAM, with Ubuntu.

### *5.2 Performance evaluation of the original RTWM*

Figure 4 shows the execution time as a function of the length of the vector to be estimated based on the Algorithm 1. It can be seen that the execution time increases depending on the size of the problem following a behavior that corresponds to the theoretical quadratic complexity established as $O(itmax * N^2)$. The runtime in vectors of the order of thousands of components exceeds hundreds of seconds. 886 seconds are required for the estimation of a vector of 2000 elements. This reveals the impossibility of using this implementation in real-time signal processing.

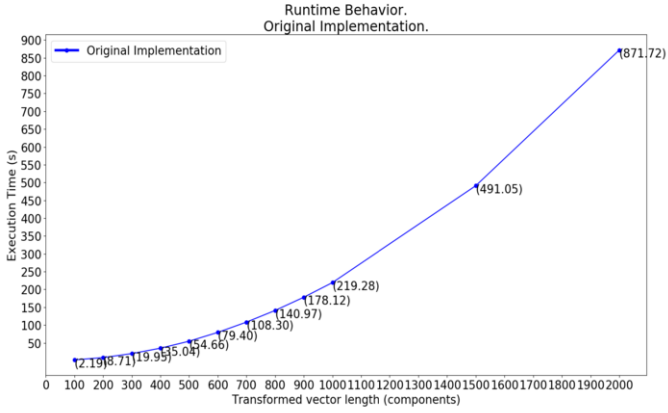


Fig. 4. Execution Time of the Original RTWM.

### 5.3 Version 1

Figure 5 presents the execution time obtained in the experiments for Versions 1 and the original implementation. It can be seen that in the worst case, where the length of the vector to be estimated is 2000 components, a reduction of about 100 seconds in the execution time has been achieved. The improvement in the average performance is 11% with respect to the original implementation. This reveals a significant improvement, as would be expected by reducing a substantial amount of floating point operations.

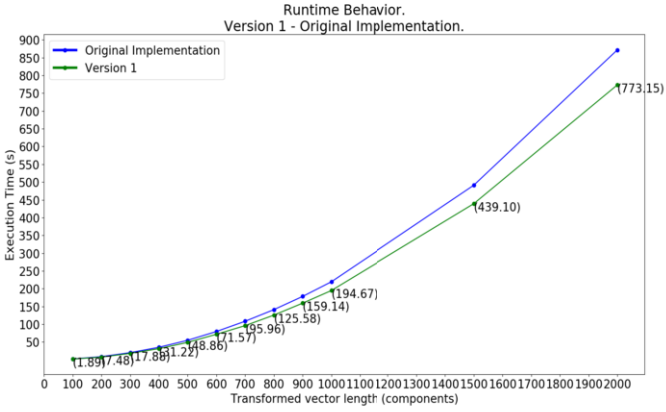


Fig. 5. Performance of Version 1 and Original.

### 5.4 Version 2

Figure 6 presents the execution time for Versions 1, 2, and the original implementation. As expected from an algorithm that makes intensive use of memory accesses, by making good use of the hierarchical memory structure an improvement in the considerable average performance of 19% has been obtained with respect to the Version 1 and a cumulative 28% with respect to the original implementation.

### 5.5 Version 3

This version modifies the mathematical procedure of the original algorithm, therefore it is necessary to verify that the quality of the results is maintained. Figure 7 presents a comparison between the best error obtained using the original implementation and Version 3. It can be seen that for dispersion levels greater than 60%, this version achieves better numerical results than the original algorithm, obtaining up to 1 dB of gain.

Figure 8 shows the execution time for Versions 2, 3, and the original implementation. It can be seen that they have been reduced more than 300 seconds in the worst case according to the length compared to the original implementation. In percentage, the improvement of the average performance with respect to Version 2 is 5%, while with respect to the original implementation it is 33%. For the lengths 100 and

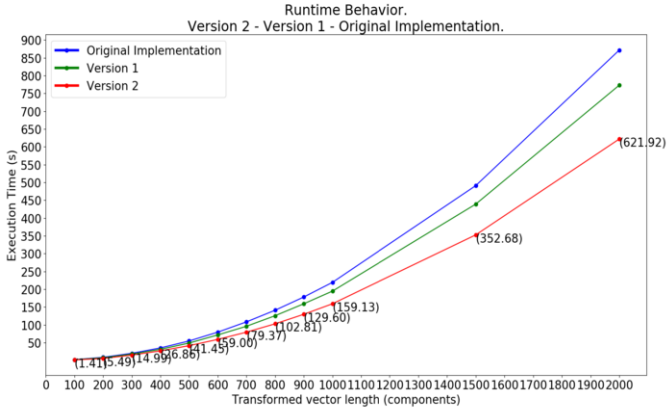


Fig. 6. Performance of Versions 1, 2, and Original.

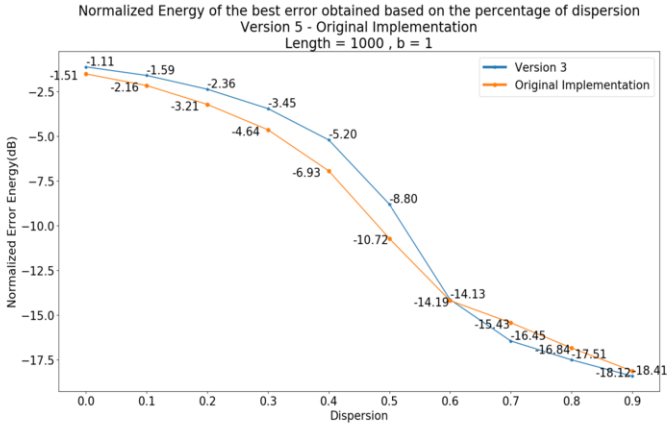


Fig. 7. Normalized energy of the best error (dB), Version 3 and Original. Laplacian noise with $\mu = 0$ and $b = 1$.

200 a reduction in the performance of 8% can be seen, however for the other lengths an average improvement of 7% is obtained. Noting that for the latest optimizations the shorter lengths have received a cumulative improvement greater than the rest, this reduction is opposed to this, but maintains a balance of performance growth. For this reason and for the improvement in the quality of the numerical results, the adoption of this optimization is justified despite the reduction in shorter lengths.

### *5.6* **Version 4**

Figure 9 presents the numerical results of Version 4 compared to the original algorithm. It can be seen that the performance achieved with Version 3 has been reduced, however, a performance greater than that of the original algorithm for dispersion greater than 70% is maintained with a Laplacian noise with a scale parameter $b = 1$.

The performance of this latest version is shown in Figure 10, which shows the execution time for Versions 3, 4, and the original implementation. On average, an improvement of 88% has been achieved with respect to Version 3 and 92% with

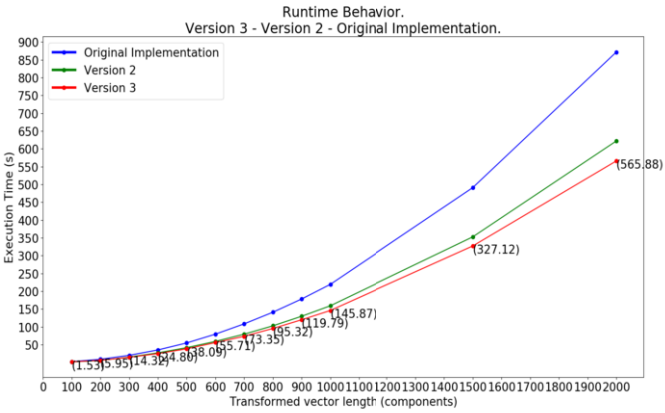


Fig. 8. Performance of Versions 2, 3, and Original.

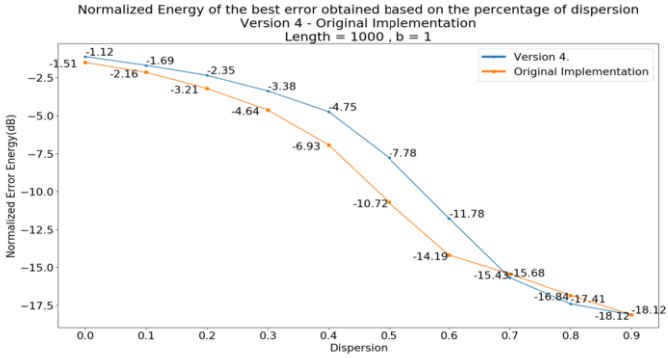


Fig. 9. Normalized energy of the best error (dB), Version 4 and Original. Laplacian noise with $mu = 0$ and $b = 1$.

respect to the original implementation.

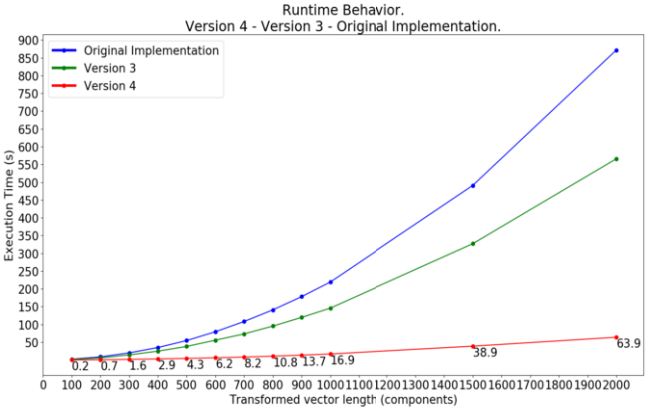


Fig. 10. Performance of Versions 3, 4, and Original.

### *5.7* Version 5

Parallelization involves an important parameter that is the number of threads. Increasing the number of threads does not necessarily generate a better performance in the program. This is due to the fact that the necessary mechanisms for the support and synchronization of the threads generate a considerable load in the execution of the program. For the experiments performed, a number of four threads was selected because the processor in which the algorithm is executed has four processing cores and is therefore optimized to support this number of threads.

Due to the design decisions in the parallel implementation of Version 4, modifications are made to the mathematical procedure that executes the algorithm. Therefore, a new validation of the numerical results of Version 5 is required. Figure 11 presents a comparison between the normalized energy of the best error obtained with Version 5 and with the original algorithm. It can be seen that for a dispersion greater than 70% the error obtained is very similar to that obtained with the original algorithm.

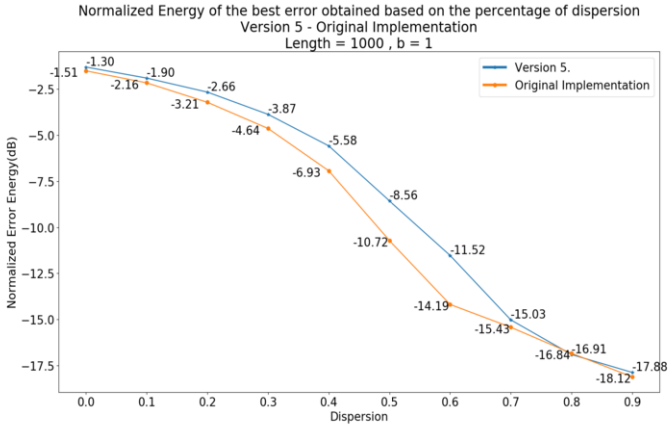


Fig. 11. Normalized energy of the best error (dB), Version 5 and Original. Laplacian noise with $\mu = 0$ and $b = 1$.

Figure 12 shows the execution time of Version 4, Version 5, and the original implementation. It shows that the execution time of the worst case has been reduced from 886 to almost 28 seconds. On average, a 55% has been improved with respect to Version 4 and a 97% with respect to the original implementation.

## 6 Conclusions

In this paper we present five improvements in the performance of the iterative algorithm of calculation of robust transforms based on the weighted median operator (RTWM), presented in [9]. Improvements of up to 30% were achieved without making modifications to its estimation method and without using multi-threaded processing. By introducing modifications to the estimation method an improvement in the average performance of 93% is achieved and by incorporating the multi-threaded processing an average improvement of 97% is achieved. The development scheme

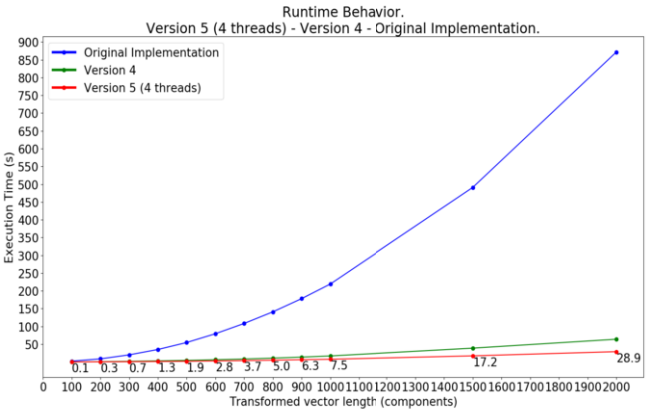


Fig. 12. Performance of Versions 4, 5, and Original.

and methods used in this investigation can be applied to similar algorithms that also depend on the calculation of medians such as those presented in [13] and [14].

Since arithmetic operations represent a significant fraction of the instructions executed in the algorithm, greater improvements could be achieved by using graphic processing units (i.e., GPU), which are optimized for the calculation of arithmetic operations.

# References

[1] R. Behera, S. Meignen, and T. Oberlin, "Theoretical analysis of the second-order synchrosqueezing transform," *Applied and Computational Harmonic Analysis*, vol. 45, no. 2, pp. 379–404, 2018.

[2] I. Djurovic, L. Stankovic, and J. F. Bohme, "Robust l-estimation based forms of signal transforms and time-frequency representations," *IEEE Transactions on Signal Processing*, vol. 51, no. 7, pp. 1753–1761, 2003.

[3] S. Lambert-Lacroix, L. Zwald *et al.*, "Robust regression through the huber's criterion and adaptive lasso penalty," *Electronic Journal of Statistics*, vol. 5, pp. 1015–1053, 2011.

[4] J. L. Paredes and G. R. Arce, "Compressive sensing signal reconstruction by weighted median regression estimates," *IEEE Transactions on Signal Processing*, vol. 59, no. 6, pp. 2585–2601, 2011.

[5] I. W. Selesnick and I. Bayram, "Sparse signal estimation by maximally sparse convex optimization," *IEEE Transactions on Signal Processing*, vol. 62, no. 5, pp. 1078–1092, 2014.

[6] D.-H. Pham and S. Meignen, "High-order synchrosqueezing transform for multicomponent signals analysis—with an application to gravitational-wave signal," *IEEE Transactions on Signal Processing*, vol. 65, no. 12, pp. 3168–3178, 2017.

[7] S. Meignen and D.-H. Pham, "Retrieval of the modes of multicomponent signals from downsampled short-time fourier transform," *IEEE Transactions on Signal Processing*, vol. 66, no. 23, pp. 6204–6215, 2018.

[8] H. Yang, "Statistical analysis of synchrosqueezed transforms," *Applied and Computational Harmonic Analysis*, vol. 45, no. 3, pp. 526–550, 2018.

[9] J. M. Ramirez and J. L. Paredes, "Robust transforms based on the weighted median operator," *IEEE Signal Processing Letters*, vol. 22, no. 1, pp. 120–124, 2014.

[10] L. Wang, "The l1 penalized lad estimator for high dimensional linear regression," *Journal of Multivariate Analysis*, vol. 120, pp. 135–151, 2013.

[11] S. Kotz, T. J. Kozubowski, and K. Podgörski, *The Laplace Distribution and Generalizations*, 1st ed. Springer Science+Business Media., 2001.

[12] A. Rauh and G. R. Arce, "Optimal pivot selection in fast weighted median search." *IEEE Trans. Signal Process.*, vol. 60, pp. 4108–4117, 2012.

[13] J. Lacruz, J. Paredes, and J. Ramírez, "Robust sparse channel estimation for ofdm system using an iterative algorithm based on complex median," *IEEE International Conference on Acoustic, Speech and Signal Processing*, pp. 6429–6433, 2014.

[14] J. Paredes and G. Arce, "Compressive sensing signal reconstruction by weighted median regression estimates," *IEEE TRANSACTIONS ON SIGNAL PROCESSING*, vol. 59, no. 6, pp. 2585–2601, 2011.