Repeated sorting on sliding window for OS-CFAR

Joseph Haim Didi [1], Nadav Levanon [1*]

[1] Electrical Engineering - Systems, Tel-Aviv University, Tel-Aviv, Israel
[*] nadav@eng.tau.ac.il

Abstract: An algorithm is suggested for repeated sorting of a sliding window of *n* consecutive reference cells, required in OS-CFAR radar detection. The sliding window surrounds a central Cell Under Test (CUT) with guard cells on its sides, which are excluded from the sorted reference cells. The algorithm's computational complexity is smaller by a factor of *n*/4 compared to performing a new independent sorting after each slide. That factor does not depend on the number of guard cells. The advantage of the new algorithm was confirmed by theoretical and numerical comparison with respect to two other recently reported approaches.

1. Introduction

Radar detection of targets in the presence of noise and clutter background requires a detection threshold. The changing level of the background suggests an adjustable threshold that can be derived, for each range (or Doppler) cell, from its neighbouring cells. The purpose of the adjustable threshold is to maintain a predetermined constant false alarm rate (CFAR). Order Statistics (OS) [1] is a popular CFAR approach because it tolerates singularities in the background, e.g., neighbouring targets. What hampers the use of OS CFAR is the need to numerically sort the values in the reference cells in order to pick a representative reference cell (usually near the 75% level) that will be used to determine the threshold. Censored Cell Averaging (CCA) [2] is a similar CFAR technique that also requires sorting.

Sorting is a rather computationally complex operation. There are many well-known sorting algorithms [3,4,5]. Their computational complexity changes for different initial state of the window to be sorted. For a random initial state (typical in detected noise), the average complexity of sorting an n element window changes between $O(n^2)$ (e.g., Bubble sort [3]) to $O(n\log_2 n)$ (e.g., Merge sort [4] or Quick sort [5]).

If the radar range span contains N range cells (e.g., N = 1000) and the window of reference cells around the CUT contains n range cells (e.g., n = 50), then the brute-force CFAR approach will require approximately $N-n\approx N$ repeats of sorting of a sliding window of size n, namely approximately $N\,n\log_2 n\approx 280{,}000$ operations.

Our approach makes use of the advantageous fact that in each move of the sliding window there is only a minor change in the content of the n reference cells. When using that advantage the number of required operations will drop by a factor of $n/4$. In the above example it will drop to $N\,4\log_2 n\approx 22{,}600$ operations. We will also show that the improvement by a factor of $n/4\,(=50/4=12.5)$ is not affected by the number of guard cells around the CUT (that are not used as reference cells).

In our approach, sorting of the reference window is performed only once, hence any one of the efficient $O(n\log_2 n)$ sorting algorithms will do. On the other hand, updating of the sorted reference window is required after each slide of the window. That update uses an efficient algorithm to place a new element in its appropriate location in the sorted reference array. The algorithm employs short representative arrays and a search algorithm based on the principle of the "Lion in the desert" search algorithm. That algorithm is described in the Appendix. The following sections of the paper will include: A brief description of OS-CFAR; The process of updating the sorted reference array after each slide, without and with guard cells; Computation complexity analysis; Simulation examples. Finally we will describe two previously published approaches of efficient sorting for OS-CFAR, estimate their complexity and compare their performances with ours.

2. OS-CFAR

Order Statistics CFAR [1], as described in Fig. 1, follows a non-coherent envelope detector. If the background noise is Rayleigh distributed, the output of a square-law detector will be exponentially distributed. The upper shift register contains the intensities measured in consecutive range cells. Hence, Z_i and Z_{i+1} exhibit the intensities measured at two neighbouring range cells. The range cell presently examined against threshold, is the CUT, usually located at the centre of the window. In some cases there may be one or more guard cells (G) on each side of the CUT to prevent self-masking of a target whose width is wider than the width of a range cell. Excluding the CUT and G cells the window contains n reference cells. Those n reference cells are sorted to produce a value ordered reference window (VORW), where $z_{(1)}\le z_{(2)}\le z_{(3)}\cdots\le z_{(k)}\cdots\le z_{(n)}$. The k'th ordered cell is picked as the representative cell. Its value $z_{(k)}$ is then multiplied by a coefficient α to produce the threshold T, against which the value in the CUT is compared, to decide if there is a target at that range cell. If the probability density function (PDF) of the values in the cells z_i, including the CUT, is identical and exponential, the probability of false alarm will be given by (1). Note that the scale parameter of the exponential PDF does not appear in (1), which is what makes this a CFAR detector.

$$P_{FA}=\prod_{i=1}^{k}\left(1+\frac{\alpha}{n+1-i}\right)^{-1} \qquad (1)$$

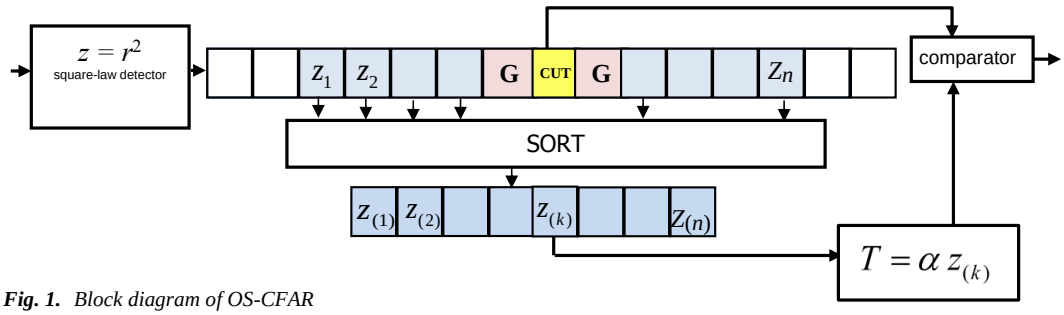


Fig. 1. *Block diagram of OS-CFAR*

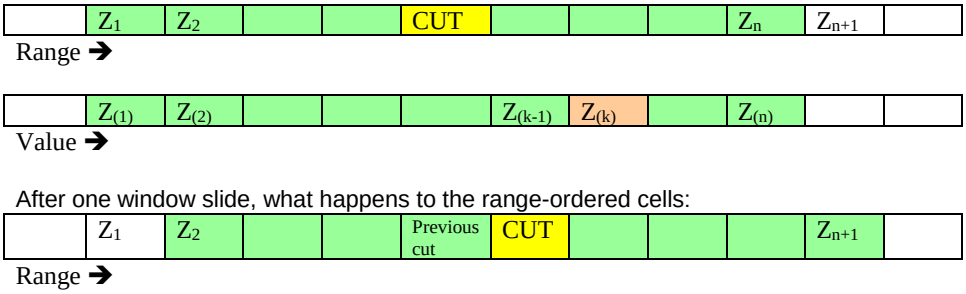


Fig. 2. *The concept of updating the reference window*

What determines the P_{FA} are the size n of the reference cells window, the selection of the representative cell k, and the coefficient α.

The main advantage of OS-CFAR is its inherent robustness against the presence of other targets within the range span of the reference window. When using averaging instead of sorting, the presence of target(s) among the reference cells will raise the threshold dramatically, causing miss detection. Choosing a relatively low value of k will improve the protection but will increase the Signal to Noise Ratio (SNR) loss of the CFAR detector, relative to detection with predetermined threshold. A common compromise is to choose $k \approx 0.75n$.

The main disadvantage of OS-CFAR is the need to obtain a new VORW of reference cells, as the CUT and its surrounding cells slide along the radar range span. The entire range span includes N range cells, and typically $N >> n$. Performing N repeated independent sorts creates a large computing load, which may be prohibitive. The next section suggests a mitigation of this difficulty - instead of performing a new complete sort operation after each window slide, update the previous VORW (Fig. 2).

3. Updating an ordered window after one slide (no guard cells)

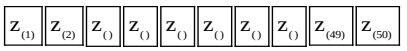
Only for the first CUT we sort the range-ordered reference window and create a value-ordered reference window (VORW). Fig. 2 shows what happens after a slide of the reference window:
(1) The new CUT will be the cell to the right of the previous CUT.
(2) The previous CUT will join the reference cells.
(3) The first reference cell will exit.
(4) A new cell from the right will join the reference cells.

In order to update the previous VORW we need to perform four tasks:
(1) Take the value of Z_1 out from the VORW.
(2) Take the value of the new CUT out from the VORW.
(3) Add the value in the previous CUT to the VORW.
(4) Add the value in the new cell Z_{n+1}, which entered from the right, into the VORW.

To execute tasks (1) and (2) we use the "Lion in the desert" algorithm (see Appendix) to find where the values of those two cells are located in the VORW and take them out. The complexity of each search in a window of length n is $\log_2 n$. Extracting both will therefore require $2\log_2 n$ operations.

The first step in all the iterations involved in adding values to a VORW is to check if the value of the new element is smaller or larger than all the values in the VORW, if it is, placing it in the VORW is straight forward and no search is needed. If search is needed we propose an efficient approach, which is a modification to "Lion in the desert". The difference is that here we have to insert a new element exactly between two values, one that is smaller (or equal) to the new element and one that is larger (or equal) to the new element. If the length of the VORW was a power of 2, then a "Lion in the desert", as is, would perform that task. The modified algorithm allows other lengths.

We will demonstrate the concept on a VORW of length n = 50 (see below). Into that VORW we want to insert a new value Z_m at the correct location.



We will split the original 50 element VORW (above) into 5 sections, each 10 element long, and pick the first element of each section to create a new 5 element value ordered window (below).

$z_{(1)}$	$z_{(11)}$	$z_{(21)}$	$z_{(31)}$	$z_{(41)}$

This 5 element window is called the representative window. Each element in it will represent itself and the 9 elements following it. We now define:

$$left = 1$$
$$right = 5$$
$$mid = ceil\left(\frac{left + right}{2}\right) \quad (2)$$
$$if\ (mid \geq right) \rightarrow mid = mid - 1$$

If the last line of (2) is not true, the value of *mid* does not change. Initially $Z(mid)$ is the value in the centre of the representative window, namely $Z(21)$. Let Z_m be the new value entering the reference window, then

$$if\ \left(\left(Z_m \geq Z(mid)\right) and \left(Z_m < Z(mid+1)\right)\right) \Rightarrow \textbf{Point A} \quad (3)$$

If (3) is not true we check if

$$\left(Z_m > Z(mid)\right) and \left(Z_m > Z(mid+1)\right) \quad (4)$$

or if

$$\left(Z_m < Z(mid)\right) and \left(Z_m < Z(mid+1)\right) \quad (5)$$

If the first option (4) holds then we perform:

$$left = mid + 1$$
$$right,\ \text{no change}$$
$$mid = ceil\left(\frac{left + right}{2}\right) \quad (6)$$
$$if\ (mid \geq right) \rightarrow mid = mid - 1$$
$$if\ \left(\left(Z_m \geq Z(mid)\right) and \left(Z_m < Z(mid+1)\right)\right) \Rightarrow \textbf{Point A}$$

If the second option (5) holds then we perform

$$left,\ \text{no change}$$
$$right = mid - 1$$
$$mid = ceil\left(\frac{left + right}{2}\right) \quad (7)$$
$$if\ \left(\left(Z_m \geq Z(mid)\right) and \left(Z_m < Z(mid+1)\right)\right) \Rightarrow \textbf{Point A}$$

Point A. Reaching **Point A** implies that either (3) or the last statement in (6) or the last statement in (7) is true. That means that we found the representative element that the new value is larger than it, and the representative element that the new value is smaller than it.

The process above is repeated by creating a new representative window, and so on, until only two elements are left. The new element will be smaller than (or equal to) the larger of the two elements and larger than (or equal to) the smaller of the two elements. Hence will be placed between them.

We will introduce a numerical example that will help follow the concept. Let the initial VORW be all the 50 odd numbers from 1 to 99 (see below). Into that window we want to add the value 64, in its proper location.

1	3	5	7	9			.....		99

The representative window will be:

1	21	41	61	81

Applying (2) to the example will produce a false statement:

$$left = 1$$
$$right = 5$$
$$mid = ceil\left(\frac{left + right}{2}\right) = ceil\left(\frac{1+5}{2}\right) = 3$$
$$if\ (mid \geq right) \rightarrow mid = mid - 1$$
$$if\ (3 \geq 5) \Rightarrow \textbf{not true} \Rightarrow \textbf{no change in } mid$$

We now check (3) , repeated below as (8)

$$if\ \left(\left(Z_m \geq Z(mid)\right) and \left(Z_m < Z(mid+1)\right)\right) \quad (8)$$

And express it using the numerical example:

$$\text{If } \left(64 \geq 41\right) and \left(64 < 61\right) \Rightarrow \textbf{not true}$$

Because it was not true we check (4)

$$\left(Z_m > Z(mid)\right) and \left(Z_m > Z(mid+1)\right) \Rightarrow$$
$$\left(64 > 41\right) and \left(64 > 61\right) \Rightarrow \textbf{\textit{true}}$$

Hence we will proceed and use (6)

$$left = mid + 1 = 3 + 1 = 4$$
$$right = 5$$
$$mid = ceil\left(\frac{left + right}{2}\right) = ceil\left(\frac{4+5}{2}\right) = 5$$
$$if\ (mid \geq right) \rightarrow mid = mid - 1 \Rightarrow mid = 4 \Rightarrow Z(mid) = 61$$
$$if\ \left(\left(Z_m \geq Z(mid)\right) and \left(Z_m < Z(mid+1)\right)\right) \Rightarrow$$
$$if\ \left(\left(64 > 61\right) and \left(64 < 81\right)\right) \Rightarrow \textbf{\textit{true}}$$

Once the statement is true, it means that we reached **Point A**, namely we found the representative element that the new value is larger than it and the representative element that the new value is smaller than it. Using the result from the example, the two elements were 61 and 81. We now proceed by taking all the elements located between those two elements and create a new representative window:

1	21	41	61	81

$$\Downarrow$$

63	65	67	69	71	73	75	77	79	81

63	67	71	75	79

The last row above is the new representative window. We will repeat the proceeding again until we reach a window containing only two elements. The new value of 64 will enter between those remaining two elements.

1	21	41	61	81						
63	65	67	69	71	73	75	77	79	81	
63	67	71	75	79						
63	65									

Summary: The initial window of length n = 50 was split into n_1=5 windows each of length 10. Using an intermediate representative window, out of the five windows the correct one was found (into which the new element will be inserted). That correct 10 element window was split to n_2 =5 windows each of length n_3=2 elements, and the new element was inserted between those two. Note that $n = n_1 \cdot n_2 \cdot n_3$. The same three steps will be used for other initial size windows. Examples from some lengths used in Section 9 are: $80 = 8 \cdot 5 \cdot 2$, $150 = 15 \cdot 5 \cdot 2$, $500 = 50 \cdot 5 \cdot 2$.

4. Calculating the Computational Complexity of Adding a New Value

(1) The search within the 5 element representative window $Z_{(1)}, Z_{(11)}, Z_{(21)}, Z_{(31)}, Z_{(41)}$ required $\log_2 5$ operations.

(2) Similarly the search within the 5 element array $Z_{(32)}, Z_{(34)}, Z_{(36)}, Z_{(38)}, Z_{(40)}$ also required $\log_2 5$ operations.

(3) The search within the last 2 element array required $\log_2 2$ operations.

The total was $\log_2 5 + \log_2 5 + \log_2 2 = \log_2 (5 \cdot 5 \cdot 2) = \log_2 50$.

Selecting representative arrays of sizes 5, 5 and 2, whose product of lengths, 50, is the length n of the VORW, leads the total complexity of adding one new value to the VORW to be $\log_2 n$. However, we had to perform **two** such insertions: for the CUT that entered the VORW (task 3) and for the new element that joined in from the right (task 4). Hence the complexity of entering the two new values was $2\log_2 n$. Recall that extracting two elements (tasks 1 and 2) also required $2\log_2 n$ operations, Hence the complexity of one repeated sorting is $4\log_2 n$.

That brings the total complexity of N repeats of the new approach to $N 4\log_2 n$. Recall that the prevailing approach – performing an independent sort after each window slide, entailed complexity of $N n \log_2 n$. The conclusion is therefore that **the new approach reduces the complexity by a factor of *n*/4** . That conclusion says that the improvement in complexity increases with the length n of the reference window.

5. Updating an Ordered Window After one slide (with Guard Cells)

The original window with guard cells (G_1, G_2) is seen in Fig. 3. Recall that the initial sort did not include (G_1, CUT, G_2). The updating steps:

(1) The old G_1 joins the reference cells.
(2) The old CUT becomes G_1.
(3) The old G_2 becomes the CUT.
(4) The reference cell on the right of the old G_2 becomes G_2.
(5) The first reference cell Z_1 exits the reference window.
(6) The cell on the right of the reference window, joins the reference window.

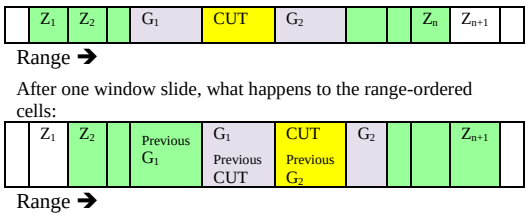


Fig. 3. *Updating when guard cells are present*

Those changes require the following modifications in the VORW:
(1) Take out the value of Z_1 from the VORW.
(2) Add the value of the old G_1 to the VORW.
(3) Extract from the VORW the value of the reference cell that was located to the right of G_2.
(4) Add to the VORW the value of the new cell that joined the reference window from the right.

We see that when guard cells are used we still have to extract 2 values from the VORW and add 2 values to the VORW. Exactly what we had to do when guard cells were not used. The conclusion is that our proposed algorithm is not affected by the number of guard cells. The computational complexity of an update, with guards, remains $4\log_2 n$. Hence the complexity improvement factor remains $n/4$.

6. Demonstration of the Algorithm Operation

In a typical simulation run a vector of $N + n$ (≈1000) random values from a Rayleigh PDF was generated. From which the first n (=50) values were used as the initial range-ordered background samples (Fig. 4).

Three central cells (filled by blue, red and green stars) represent the l.h.s. guard, the CUT and the r.h.s. guard, respectively. Their values appear in the figure's title. Those cells do not appear in the initial value-ordered sorted window (Fig. 5). Creating the first VORW was performed by the first and only sorting operation.

After one window slide the range-ordered window appears in Fig. 6. From now on only sorting updates are required, which will result in the new VORW (Fig. 7). The similarity between Figs. 5 and 7 provides intuitive justification for performing a sorting update instead of a full new sort. The two VORWs are very similar and differ in only 4 places, out of 50. The window's slides, each followed by a sorting update, will be repeated 1000 times, until the CUT reaches its furthest range.

In three such simulation runs, with exponentially distributed background, the computation improvement ratios were: 12.75, 12.98 and 13.1. In three similar simulation runs, with Rayleigh distributed background, the computation improvement ratios were: 12.95, 13.25 and 13.55. These results agree with the expected worst-case improvement ratio, which is $n/4 = 50/4 = 12.5$.

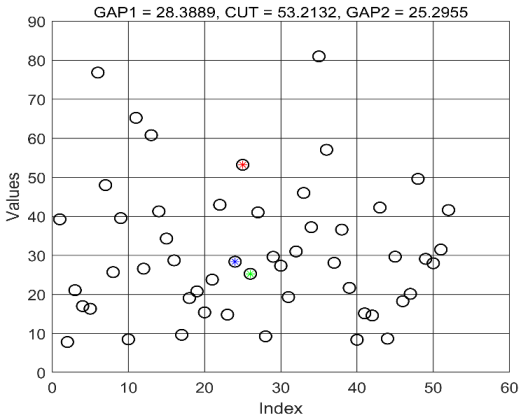


***Fig. 4**. Initial range-ordered window*

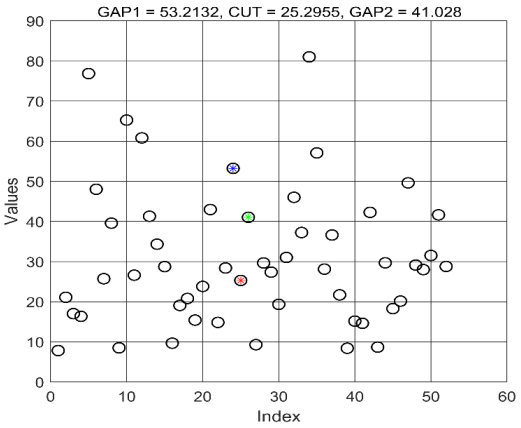


***Fig. 6**. The range-ordered window after one slide*

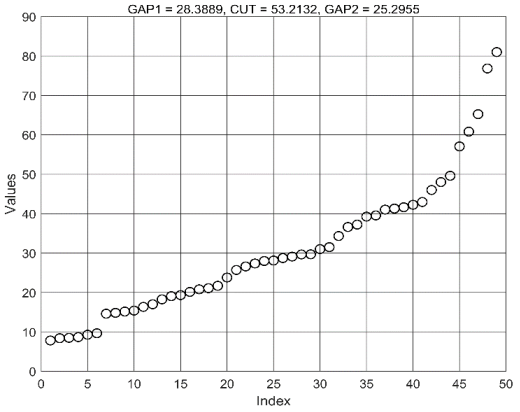


***Fig. 5**. Initial value ordered reference window (VORW)*

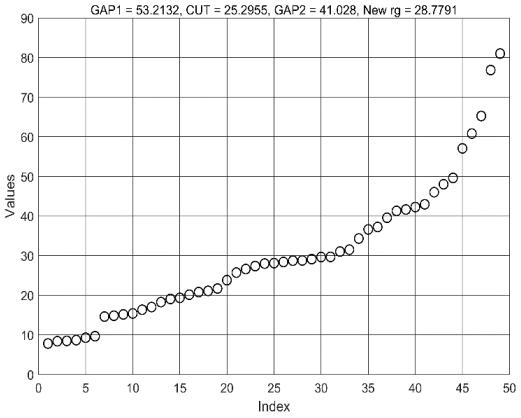


***Fig. 7**. VORW after one slide, obtained by sorting update*

7. Other Sorting Algorithms

7.1. Bubble sort [3]

In this comparative sorting, each iteration causes the largest element in a shrinking array to move toward its final location. The sorting starts by comparing the value of every two neighbouring elements in an *n* size array. If $a_i > a_{i+1}$ the two elements will switch positions. At the end of the iteration the largest element will move to the last place. Next the process is repeated on an *n*-1 long array that does not include the last element. This is repeated until there are no consecutive elements that are not in the correct order.

Example: Consider the array [6 5 3 1 8]. 1st iteration – 6 and 5 will be compared and will switch positions. Then 6 and 3 will be compared and switch positions, and 6 and 1 will switch positions, while 6 and 8 remain in that order, resulting the array [5 3 1 6 8]. 2nd iteration – 5 and 3 will switch, 5 and 1 will switch and 5 and 6 will not, resulting [3 1 5 6 8]. 3rd iteration – 3 and 1 will switch positions, while 3 and 5 will not, resulting the sorted array [1 3 5 6 8].

Computational complexity – The 1st iteration on an *n*–size array requires *n*-1 comparisons, the 2nd requires *n*-2 comparisons, and so on. The *n*-1 iteration will require only one comparison. The total number of comparisons is therefore

$$1+2+3+\ldots+(n-1)=n(n-1)/2=O(n^2) \qquad (9)$$

Namely, the average computational complexity of Bubble sort is n^2 .

7.2 Merge sort [4]

This is a recursive sorting algorithm utilizing the simplicity of merging sorted arrays. An unsorted array of size *n* is divided into two smaller sub-arrays each of approximate *n*/2 size. This is repeated until the sub-arrays are of length one, which is obviously a sorted sub-array. Next we start merging pairs of sub-arrays. We look at the first element of each one of the two sub-array to be merged. Since the sub-arrays are sorted the first element is the smallest element in that sub-array. The smaller of the two sub-arrays (*a* and *b*) will be the smallest in the merged array.

Assuming that it came from sub-array *a*, we check the 2nd element in sub-array *a* and compare it to the first (hence smallest) of sub-array *b*. The smaller of the two becomes the next element in the merged array, and so on. This will continue until we are done with all the elements of one of the sub-arrays. Then all the remaining elements of the un-finished array are entered together, since they are already sorted. An example of sorting the array [38 27 43 3 9 82 10] is given in Fig. 8.

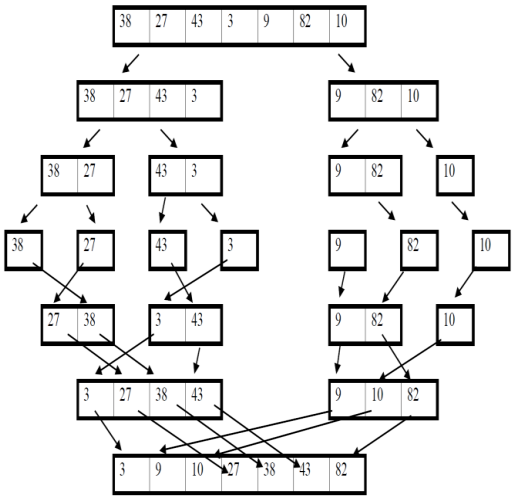


Fig. 8*. Example of Merge Sort*

Computational complexity – An *n*–size array is split into two sub-arrays of sizes i and $n-i-1$. The average computational time $T(n)$ of sorting the *n*–size array is given by

$$T(n)=\frac{1}{n}\sum_{i=0}^{n-1}\left(T(i)+T(n-1-i)+cn\right)$$
$$=\frac{2}{n}\left(T(0)+T(1)+....+T(n-2)+T(n-1)\right)+cn \quad (10)$$

where *c* is the constant time needed to perform a comparison between the values of two elements. Re-writing (10) we get

$$nT(n)=2\left(T(0)+T(1)+....+T(n-2)+T(n-1)\right)+cn^2 \quad (11)$$

Similarly, we can write

$$(n-1)T(n-1)=2\left(T(0)+T(1)+....+T(n-2)\right)+c(n-1)^2 \quad (12)$$

Subtracting (12) from (11) yields

$$nT(n)-(n-1)T(n-1)=2T(n-1)+2cn-c$$
$$\approx 2T(n-1)+2cn \quad (13)$$

$$\frac{T(n)}{n+1}=\frac{T(n-1)}{n}+\frac{2c}{n+1} \quad (14)$$

(14) can be developed to a telescoping series,

$$\frac{T(n)}{n+1}+\frac{T(n-1)}{n}+\frac{T(n-2)}{n-1}+....+\frac{T(2)}{3}+\frac{T(1)}{2}$$
$$-\frac{T(n-1)}{n}-\frac{T(n-2)}{n-1}-....-\frac{T(2)}{3}-\frac{T(1)}{2}-\frac{T(0)}{1} \quad (15)$$
$$=\frac{2c}{n+1}+\frac{2c}{n}+....+\frac{2c}{3}+\frac{2c}{2}$$

Cancelling identical elements yields,

$$\frac{T(n)}{n+1}=\frac{T(0)}{1}+2c\left(\frac{1}{2}+\frac{1}{3}+....+\frac{1}{n}+\frac{1}{n+1}\right) \quad (16)$$

$$T(n)=(n+1)T(0)+(n+1)2c\log_2 n$$
$$\approx n\log_2 n \Rightarrow O(n\log_2 n) \quad (17)$$

8. Other Repeated Sorting Algorithms for OS-CFAR with Reduced Computational Complexity

Reducing the complexity of repeated sorting was the subject of previous works. We will limit the discussion to independent algorithms rather than approaches in which the hardware and algorithm are embedded together. We will discuss two approaches suggested for OS-CFAR.

8.1. K-Finder algorithm [6]

The K-Finder algorithm does not sort the array but only finds the k'th ordered element in an array of n elements. The algorithm steps are as follows:

1. Define the desired order *k*.
2. From the array's elements choose one as a pivot. Also define two variables:
Tu – The number of elements in the initial reference window with value larger than the present pivot.
Cu – The number of elements in the current window that are larger than the present pivot.
3. Find the number *Tu* of elements in the initial array that are larger than the present pivot.
4. Find the number *Cu* of elements in the present array that are larger than the present pivot. At the initial step *Cu* = *Tu*.
5. If $Tu > n-k$ leave in the array only the elements that are large than the pivot. Also set $Tu = Tu - Cu$. If $Tu \le n-k$ leave in the array only the elements that are smaller than the pivot.
6. Find the average of the elements remaining in the array and make that average the new pivot.
7. For the present array calculate *Cu* (the number of elements in the present array) that are larger than the pivot. Also set $Tu = Tu + Cu$.
8. Repeat steps 3 – 7 until there are no elements whose values are larger than the value of the pivot, hence $Cu = 0$ and the algorithm stops. The value of the pivot is the value of the *k*'th ordered element.

A numerical example is given in Table 1.

Table 1. Example of a K-Finder algorithm for n =10, k = 7 $\Rightarrow n-k=3$

Reference window	**Pivot**	*Cu*	*Tu*	comment
5 ↓13 2 ↓11 ↓19 ↓28 6 3 1 ↓9	6	5	5	$Tu > n\text{-}k$ Hence choose values larger than pivot.
13 11 19 28 9			0	Set $Tu = Tu - Cu$
↓13 ↓11 19 28 ↓9	16	2	2	Pivot = mean. Calculate *Cu*. *Tu* =2 < n-k elements in the initial array are larger than the pivot, hence the smaller than pivot values will remain in the array
13 ↓11 ↓9	11	1	3	3 elements in the initial array are larger than the new pivot ➔ $Tu = 3 \le n-k$ ➔ the smaller elements stay. Only 1 element of the present array is larger than the pivot ➔ *Cu*=1.
↓11 9	10	1	4	$Tu = 4 > n-k$ ➔ the larger elements stay.
11	11			This is the *k*'th ordered element

Computational complexity – In the first iteration n comparisons are conducted to find which element in the array has a larger value than the defined pivot. In the second iteration $n-k$ comparisons are required, and so on until the k'th ordered element is reached. The total complexity $T(n)$ is therefore

$$T(n)=n+(n-k)+(n-k-k_1)+(n-k-k_1-k_2)+... \quad (18)$$

Because the various k values in each iteration are significantly smaller than n, (18) implies $T(n)\Rightarrow O(n)$.

8.2 ABIS algorithm [7]

The Anchor Based Insertion Sorting (ABIS), does make use of the fact that the reference window was initially sorted. In addition to the original array containing the input samples in their order of arrival, the algorithm creates two additional arrays: Cell Unit List (CUL) and Ordered Sequence. The initial Ordered Sequence is created by initial sorting of the reference window, which includes the first n input cells, excluding the CUT).

Each of the cells in the CUL contains 4 fields: Its location index in the CUL, its numerical value, the location index of the cell that preceded it in the Ordered Sequence and the index of the cell that followed it in the Ordered Sequence.

After a shift, two elements have to exit the reference window (the earliest cell and the cell to the right of the CUT). The various indexes are used to achieve that. Then two new elements will enter the reference window (the CUT from the previous shift and the new cell entering from the right). Each one of the two new elements will be compared to the value of the recent k'th ordered element. If the tested entering new element has a higher value it will be compared to each of the elements whose values are higher than the k'th ordered element, until it reaches a cell with a higher value. A similar operation is performed if the new entering cell has a smaller value then the k'th ordered cell. This sorting approach is based on the "Insertion sorting algorithm". In addition, after each insertion the indexes of the cells with values higher than the inserted cell need to be updated.

Computational complexity – For the more difficult case in which the two new inserted cells are near the two edges of the Ordered sequence the number of required operations is approximately $(n-k)+k+n_{\text{additions}} \Rightarrow O(n)$.

9. Comparing Algorithms

Here we will numerically compare the computational complexities of four approaches: (a) Repeated **Quick Sorting** without using the previous sort, (b) the **ABIS** approach, (c) the **K–Finder** approach, and (d) the approach proposed in this paper, labelled **New Alg**. Note that the estimated complexity of the four approaches are, respectively (up to a constant): (a) $n\log_2(n)$, (b) n, (c) n and (d) $4\log_2(n)$.

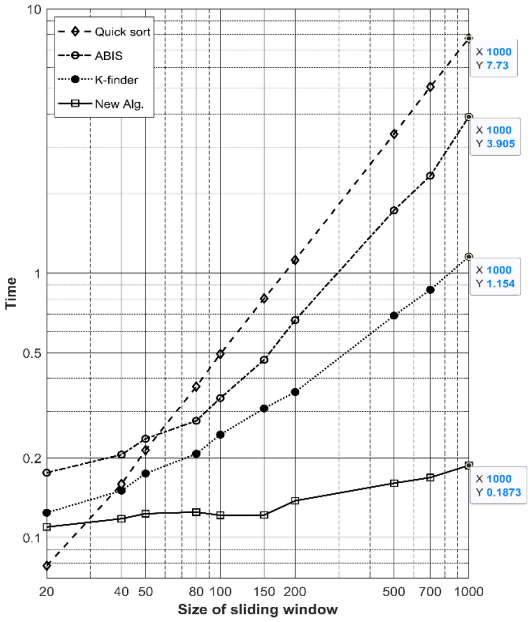


***Fig. 9**. Computational duration of 1000 shifts and repeated calculations of the k'th ordered cell*

All four approaches were programmed in MATLAB using only basic instructions. The reference window sizes changed from n = 20 to 1000. For all window sizes the number of window slides was 1000. The window shift occurred after the k'th ordered cell was calculated. The k'th order was chosen near 0.75 n.

Fig. 9 displays a typical result of the time required to complete 1000 window shifts. It shows that the different computational complexities become apparent for large window size n. The 'New Alg.' graph in Fig. 9 shows very slow dependence on the window length n, pointing its advantage when large sliding window sizes are used.

10. Conclusions

Order Statistics CFAR gained popularity because of its relative immunity to the presence of neighbouring targets among the reference cells. However, the need to repeatedly sort the window of n reference cells, as it slides along the range axis, is a severe penalty.

The algorithm proposed in this paper performs sorting update, which is computationally simpler by a factor of $n/4$, compared to a new sort after each slide of the window. The algorithm was described in details and its computational complexity was analysed and confirmed by simulations. Theoretical and numerical comparisons were also performed for two previously reported approaches. The comparison confirms the advantage of the presently reported algorithm.

11. References

[1] Rohling, H.: 'Radar CFAR thresholding in clutter and multiple target situations' IEEE Trans. Aerospace and Electronic Systems, 1983, 19, (4), pp. 608-621

[2] Ritcey, J. A.: 'Performance analysis of the censored mean-level detector' IEEE Trans. Aerospace and Electronic Systems, 1986, 22, (4), pp. 443-454

[3] Biggar, P. and Gregg, D. 'Sorting in the presence of branch prediction and caches' Technical Report TCD-CS-2005-57, Dept. Comp. Sci., Univ. of Dublin, Trinity College, August 2005. (https://www.scss.tcd.ie/publications/tech-reports/reports.05/TCD-CS-2005-57.pdf)

[4] Knuth, D. E.: 'The Art of Computer Programing: Vol. 3: Sorting and Searching' (Addison-Wesley, 2nd edn. 1998)

[5] Aho, A. V., Hopcroft, J. E. and Ullman, J. D.: 'The Design and Analysis of Computer Algorithms' (Addison-Wesley, 1974)

[6] Ali, Z., Arshad, A., Razzaq, U., Sana, S., Ahmed, A. H., Harris, A. M.: 'Design and implementation of an OS-CFAR processor based on a new rank order filtering algorithm' 2010 Int'l Symp. On Systems On Chip, Tampere, Finland, September 2010, pp. 158 – 162.

[7] Shin, D., Kim, J., Kim J., Bang. J., Kwon, K. K.: 'Anchor based insertion sorting algorithm for OS-CFAR' 2014 IEEE Radar Conference, Cincinnati, OH, USA, September 2014, pp. 391-395.

12. Appendix – "Lion in the desert" search algorithm

"Lion in the desert" or "Binary search" is a well-known search algorithm for finding a numerical value element in an already sorted window, where the left-most element contains the lowest numerical value and the right-most element exhibits the highest value. If the element in the middle of the window contains the value we look for, the search ends successfully right there. (If the number of elements is even we can arbitrarily define the mid-element as the one on the right of the middle.) If the mid-element is not the one we look for, we compare the sought-for value to the value of the mid-element. If the sought-for value is smaller than the mid-element value, we limit the search to the l.h.s. of the mid-element, where all the elements exhibit smaller values, and vice-versa. That process continues repeatedly as long as the sought-for value is not equal to the value of the mid-element. In the worst case the array size shrinks to 1 or 2 elements.

Example:

Consider the sorted window in the top row of Fig. 10. We are looking for the number 82. The first step is to choose the middle cell. Its value is 52. Since 82>52, the search will continue in the sub-window to the right of the mid element, where all the values are higher than (or equal to) 52.

The middle cell of the sub-window (middle row) is 94 > 82, so the search will now be limited to its left sub-window, repeated in the last row, where the mid-element (as defined for even length arrays) exhibits the value we look for.

10	18	27	35	43	**52**	60	82	94	97	100

60	82	**94**	97	100

60	**82**

***Fig. 10**. Example of a "Lion in the desert" search*

Computation complexity: In each step there is one comparison and the array size is halved. The worst case is when the array reaches a size of one element. In that case the number of comparisons m satisfies the equation $2^m = n$. In the worst case $m = \log_2 n$. The complexity is therefore $O(\log_2 n)$.