# Clustering evaluation for deinterleaving

Eirik Jensen Opland

Norwegian Defence Research Establishment (FFI)

23 February 2013

FFI-rapport 2013/00567

1219

P: ISBN 978-82-464-2230-5
E: ISBN 978-82-464-2231-2

## Keywords

Algoritmer

Deinterleaving

Klynging

Radar

ESM

EST

Testing

Optimalisering

## Approved by

| | |
|---|---|
| Berit Jahnsen | Project Manager |
| Anders Eggen | Director |

## English summary

Deinterleaving is a fundamental step in a lot of processing in ESM and radar systems, enabling users and/or client programs to focus on data from a single emitter at a time, rather than a mixture of data from several emitters.

When developing deinterleaving algorithms, it is sometimes useful to compare deinterleaving results with a model solution. Such a solution may be available because one has some alternative way of deinterleaving existing interleaved data, because one generates the data oneself, for example with a PDW simulator, or because one combines (interleaves) several existing non-interleaved data sets.

This report discusses some methods of assessing clustering results in the context of deinterleaving, and focuses on evaluation criteria that consider the problem of clustering as a binary classification problem, where the objects to be classified are all the possible pairs of distinct input points. A pair is classified as positive or negative, respectively, if the two points in the pair are in the same or different clusters. An efficient evaluation algorithm is developed, which avoids visiting every pair of points, but instead calculates the necessary information based on the sizes of clusters.

Finally, the evaluation criteria are used along with some already deinterleaved data in order to optimize a key parameter to the LINE deinterleaving algorithm. This leads to a new choice of evaluation criterion that is more suitable to the current data set. The GEOIDE deinterleaving algorithm is tested on the same data and the results are discussed. The same data set is used for generating new data sets, by combining deinterleaved emissions in new ways. The new data sets are used for testing LINEs deinterleaver further.

## Sammendrag

Deinterleaving er et grunnleggende steg i ESM- og radarsystemer, og gjør det mulig for brukere og/eller programmer å fokusere på data fra en enkelt emitter, istedenfor en blanding av data fra flere emittere.

Ved utvikling av deinterleavingsalgoritmer, er det noen ganger nyttig å sammenligne deinterleavingsresultater med en fasit. En fasit kan være tilgjengelig fordi man har en alternativ måte å deinterleave eksisterende interleavede data, fordi en genererer data selv, for eksempel ved hjelp av en PDW simulator, eller fordi man kombinerer (interleaver) flere eksisterende sett av ikke-interleavede pulsdata.

Denne rapporten tar for seg noen metoder for å evaluere klyngingsresultater i forbindelse med deinterleaving, og fokuserer på evalueringskriterier som betrakter klynging som et binært klassifiseringsproblem, hvor objektene som skal klassifiseres er alle mulige *par* av distinkte inputpunkter (pulser). Et par er klassifisert som henholdsvis positivt eller negativt, dersom de to punktene i paret er i samme eller forskjellige grupper. En effektiv evalueringsalgoritme er utviklet, som unngår å besøke hvert mulig par av punkter, men i stedet beregner den nødvendige informasjon basert på størrelsene av punktklynger.

Evalueringskriteriene brukes sammen med noen ferdig deinterleavede data for å optimalisere en viktig parameter i LINEs deinterleavingsalgoritme. Evalueringen av disse dataene leder til valg av et nytt evalueringskriterium, som er mer egnet til det gjeldende datasettet. GEOIDEs deinterleavingsalgoritme er testet på samme datasett, og resultatene diskuteres. Det samme datasettet brukes til å generere nye datasett, ved å sette emisjonene sammen på nye måter, og LINEs deinterleaver prøves på nye måter.

# Contents


| | | |
|---|---|---|
| **1** | **Introduction** | **7** |
| **2** | **Measuring deinterleaving results against a model solution** | **8** |
| 2.1 | Clustering | 8 |
| 2.1.1 | Terminology and conventions | 9 |
| 2.2 | Measuring clustering results against a target solution | 10 |
| 2.2.1 | Matching solution clusters with result clusters | 10 |
| 2.2.2 | Considering all pairs of points | 11 |
| **3** | **Efficiently calculating TP, FP, FN and TN** | **12** |
| 3.1 | Efficiently calculating TP | 12 |
| 3.2 | Efficiently calculating FN | 13 |
| 3.3 | Efficiently calculating FP | 14 |
| 3.4 | Efficiently calculating TN | 15 |
| 3.5 | Summary of the algorithmic complexity | 15 |
| **4** | **Applying the evaluation criteria to real data** | **15** |
| 4.1 | Using navigation radar data that have already been deinterleaved | 16 |
| 4.2 | Testing the GEOIDE deinterleaver on the same data as for LINE_Deint | 24 |
| 4.3 | Using the same data to generate new scenarios | 25 |
| 4.3.1 | Randomly moving every emission | 26 |
| 4.3.2 | Randomly moved emissions on a resized time axis | 26 |
| 4.4 | Further work | 28 |
| **5** | **Summary** | **29** |
| | **Appendix A LINE Deinterleaver Algorithm, LINE_Deint** | **30** |
| | **Appendix B Complexity analysis and the big O notation** | **31** |
| | **Appendix C Algorithm for calculating TP, FN, FP and TN** | **32** |
| | **References** | **32** |

# 1 Introduction

Deinterleaving is a fundamental step in a lot of processing in ESM and radar systems, enabling users and/or client programs to focus on data from a single emitter at a time, rather than a mixture of data from several emitters. Deinterleaving of pulsed signals is the problem of building groups of pulses, so that pulses originating from the same emitter are put in the same group, but pulses originating from different emitters are put in different groups. Deinterleaving can be quite easy or very difficult, depending on the signal environment, which parameters are available and to what extent different emitters are sufficiently different, with respect to these parameters.

Having deinterleaved some pulses, it is very useful to be able to evaluate the result. This is valuable both during the development of deinterleaving algorithms and when applying them in practice. The ways to do this can be split into two main categories of approaches:

- Approach 1: Check the result for agreement with an ideal solution.
- Approach 2: Look at new data, or different attributes in the data to validate the result.

Approach 2 is applicable to all stages of deinterleaving. For example, if one deinterleaved the data based on the assumption that each emitter transmitted pulses at a different radar frequency, one may use continuity in the pulse repetition interval and amplitude to strengthen the confidence in the result considerably. The LINE[1] deinterleaving algorithm (Appendix A) takes an approach similar to this, using the amplitude to generate candidate pulse trains, and then stability in the PRI to confirm the result. Approach 2 can also be used for testing a deinterleaving algorithm, particularly if the validation is strong enough that one can assume the result to be sufficiently correct. The greatest strengths of this approach, compared with approach 1, are its wide applicability and the fact that one does not need to know the solution in advance.

The greatest weakness of approach 2 is the (sometimes great) uncertainty of its correctness. This is avoided in approach 1, which uses a definitive answer, with which to compare the result. Of course, such an answer may be difficult or impossible to obtain, but that is another matter, which will be addressed below. However, given a perfect solution, approach 1 is guaranteed to give a definitive assessment, and given a good, but imperfect solution, approach 1 may still give a highly relevant answer. How to measure the agreement between a result and a solution is another matter that requires some consideration. This will be discussed in chapter 2.

[1] "LINE (**LI**ten **N**avigasjonsradar **E**SM) was an activity at FFI that culminated in the spring of 2012 at Unified Vision 2012 (UV12), in which two experiment sensors were used to track ships by their navigation radars at Ørlandet. This was accomplished by combining the difference in the rotation phase of the radar (based on the strength of the signal received at each ESM-sensor) with the difference in the time of arrival (TDOA) of pulses at the two ESM-sensors. The two methods gave several geographical curves (arcs and half hyperbolas), such that the radar would be located at an intersection between the curves." (Quoted directly from (1)).

There are several situations in which a sufficiently good ideal solution may be available. Here are some examples:

1. One has some alternative way of deinterleaving existing interleaved data. Perhaps one is developing a deinterleaver that is meant to work in real-time with very limited hardware resources. In a lab environment, one may have access to abundant physical resources and be able to use the most sophisticated algorithm. One may even have a human available to do the job manually or help the algorithm with the most difficult tasks, since humans are frequently better at pattern discovery. This may enable the creation of excellent solutions to realistic problems, and then other deinterleavers can be tested against this solution.
2. One generates the data oneself, for example with a PDW[2] simulator. One may know enough about the process generating the pulses to make pretty realistic simulations, which can provide the solutions along with the input data.
3. One combines existing non-interleaved data into an interleaved data set. Then each component that is being combined represents all the data from one single emitter, which can be used as a model solution.

# 2 Measuring deinterleaving results against a model solution

Given a model solution, a deinterleaving output can be automatically measured in a more reliable way than otherwise, and the quality of different deinterleaving results can be compared more easily. This can be used in order to optimize free parameters in the algorithm or improve the algorithm in other ways. It can also be used to assess the algorithm by exposing it to realistic scenarios for which real recordings are not yet available, to expose weaknesses and to identify the limits of the algorithm, both in terms of what kinds of inputs it can handle and the quality of the results for such inputs. These are key criteria, when determining whether the algorithm can be expected to meet the needs of a particular client program.

## 2.1 Clustering

Deinterleaving is a type of clustering problem. Clustering is the more general problem of grouping together data points, so that points within the same group are related to each other in some way, but less related to points within the other groups. For example, given time of arrival, radar frequency, pulse length, bearing and pulse repetition frequency of some pulses, a clustering algorithm may group together pulses with similar values for one or several of those measures. Exactly which pulses are grouped together, and on the bases of which criteria all depends which clustering algorithm is used and how it is configured.

[2] A pulse description word (PDW) is a description of the attributes of a particular pulse. Typically, it is a vector of numbers, where each number represents one attribute.

### 2.1.1 Terminology and conventions

The following terminology and conventions are used when discussing clustering problems, results and solutions in the following sections. This list is relatively long, but it should make subsequent sections easier to read, and provides a single place of reference for the sections that follow.

| Concept | Description |
|---|---|
| **Clustering Input** | A set **D** of $\mathbf{N_D}$ data points, $\mathbf{d_i}$ **(i ∈ {0, ..., $\mathbf{N_D}$})**[3]. The index "i" will be used exclusively to refer to *input data points*, so that one can refer to the data point **i** or $\mathbf{d_i}$ interchangeably and without any ambiguity. Unless stated otherwise, i and $d_i$ will refer to an *arbitrary data point*. |
| **Pairs of input data points** | It will be useful to discuss arbitrary pairs of input data points. **($\mathbf{i_1}$, $\mathbf{i_2}$)** is used as a short-hand notation for any pair of data points, $i_1$ and $i_2$. |
| **Result Clusters** | A given clustering algorithm will group the input, D, into some number, $\mathbf{N_R}$, of clusters, known as result clusters.<br><br>Let **R** be the set of result clusters, $\mathbf{r_j}$ **(j ∈ {0, ..., $\mathbf{N_R}$})**, that is, outcomes of applying some clustering algorithm to D. Note that each $r_j$ is a subset of D. The index "j" will be used exclusively to refer to result clusters, so that one can refer to the result cluster j or $d_j$ interchangeably and without ambiguity. Unless stated otherwise, j and $r_j$ will refer to an *arbitrary result cluster*.<br><br>Let $\mathbf{r_0}$ be reserved for unclustered data points, i.e. points that do not belong to any cluster. This is not actually a cluster, which would be self-contradictory, but, for convenience, is treated as part of the cluster set. |
| **Cluster Memebership Array** | Let the cluster membership of all the data points be represented by an array, $\mathbf{A_R}$, of length $\mathbf{N_D}$. The value of $\mathbf{A_R(i)}$ gives the **index, j**, of the result cluster to which the data point i belongs. |
| **Clustering Solution** | A clustering solution is expressed just like a clustering result, except that the letters **R**, **r** and **j** are replaced, respectively, by the letters **S**, **s** and **k**. In particular, the terms **S**, $\mathbf{s_k}$, **k**, $\mathbf{A_S}$, $\mathbf{N_S}$ and $\mathbf{s_0}$ mean the same about solution clusters as **R**, $\mathbf{r_j}$, **j**, $\mathbf{A_R}$, $\mathbf{N_R}$ and $\mathbf{r_0}$ mean about result clusters. |
| **Big O notation, O(·)** | When discussing algorithms, concepts like **time complexity**, **memory complexity** and the **big O notation, O(·)**, are very useful. These concepts are explained in detail in Appendix B. |

[3] A set, unlike a sequence, does not impose an order on the elements within it, but the elements here are still enumerated with natural number subscripts. This is only for ease of reference, and implies no meaningful ordering of the elements.

## 2.2 Measuring clustering results against a target solution

Now consider the situation in which one knows in advance which points should, or should not, belong together in the same clusters, so that the quality of a result can be measured by comparing it to the solution.

Measuring a result against a solution is complicated by the fact that there may be no definitive matching between each solution cluster and a corresponding result cluster. If there was, one way would be to simply consider each solution cluster in turn and check how many of the data points within that cluster have been correctly/incorrectly put in the corresponding result cluster. These two figures could then be measured against the total number of data points, and would provide a good basis for measuring the quality of a result, relative to the size of the problem.

### 2.2.1 Matching solution clusters with result clusters

As discussed above, there may be no definitive matching between solution clusters and result clusters. However, there may still be a very strong correspondence. For example, one result cluster, j, may contain 95% of the data points of some solution cluster, k. If so, then those 95% of the points have been correctly grouped together with each other (but not with the remaining 5% of $s_j$), and so it may be reasonable to match j with k, and this matching can be discovered by checking which k has the most common elements with j. However, this may lead to conflict when trying to match up the other clusters.

In order to find the optimal matching of result clusters with solution clusters, one may consider all possible ways of associating any j with any k, and see which set of matchings leads to the best performance. One great disadvantage of this approach is that, as the number of clusters grows, it leads to a combinatorial explosion in the number of ways one can match the result j with the solution k. In particular, based on the sizes of S and R, the number of such combinations is given by formula (2.1):

$$\begin{aligned} combinations(N_R, N_S) &= \binom{\max(N_R, N_S)}{\min(N_R, N_S)} = \frac{\max(N_R, N_S)\,!}{(\max(N_R, N_S) - \min(N_R, N_S))!} \\ &\geq O(\min(N_R, N_S)\,!) = \min(O(N_R)!, O(N_S)!) \end{aligned} \tag{2.1}$$

Hence, the time complexity is effectively factorial (accounting for the worst cases) in the smallest of the inputs, and so unless either $N_R$ or $N_S$ is fairly small, this number is huge. For example, if $N_R$=30 and $N_S$=30, then the number of combinations is more than $10^{32}$, which would take tens of thousands of times the age of the universe to run on a modern computer, if it could run that long. In other words, even for fairly small problems, it will take the computer much too long to check all the different combinations.

One can make some assumptions that reduce the time complexity to be completely manageable. For example, one can start with the largest result cluster, and then identify the best matching solution cluster. Then one could stick with that matching and progress in the same way with the remaining clusters. In general, one disadvantage of this approach is that one may misrepresent the

quality of the clustering in some cases. However, when this is the case, the clustering result is typically pretty poor anyway. Therefore, if ones goal is to distinguish pretty good clustering results from mediocre and poor clustering results, and one would also like to distinguish the very best clustering results from the slightly poorer (but potentially still excellent) clustering results, then this way of measuring results can be suitable. On the other hand, if one wants to compare mediocre or poor results with other results of approximately the same (mediocre or poor), but slightly different quality, then this measure may be totally misleading.

Another disadvantage of matching result and solution clusters is that, even if one finds the optimal matching, it may in some cases not be a very fair measure. Suppose a solution cluster is split into two result clusters of approximately equal size. Then only the contents of the first result cluster will contribute to the positive performance. However, the data points in the second cluster were successfully grouped together – with each other, but not with the data points in the first cluster. This is just one example of how this method is frequently not the best expression of the quality of a clustering result.

One positive property of this approach is that it can be easily visualized. Every data point is either in the right cluster or in the wrong cluster, which can be easily visualized, for example by a color in a plot of the input points.

### 2.2.2 Considering all pairs of points

Alternatively, one can consider the set, P, of all possible pairs of points, $(i_1, i_2)$ in D. There are $N_D \times (N_D-1)/2$ such data points. Then a clustering result or solution associates every such pair with truth values about whether the pair is in the same cluster or not, according to that clustering result or solution. Then, by inspecting $A_R(i_1)$, $A_R(i_2)$, $A_S(i_1)$ and $A_S(i_2)$ for every pair to determine whether both points are in the same result cluster and solution cluster, every pair can be assigned one of the following four categories:

1. Correctly grouped together (True Positives, TP)
   - They are in the same result cluster and solution cluster
2. Incorrectly grouped together (False Positives, FP)
   - They are in the same result cluster, but different solution clusters
3. Incorrectly not grouped together (False Negatives, FN)
   - They are in different result clusters, but the same solution clusters
4. Correctly not grouped together (True Negatives, TN)
   - They are in different result clusters and solutions clusters.

Counting the number of pairs in each of the above categories, and associating these counts with the category labels, TP, FP, FN and TN, the following additional measures can be derived:

$$TP_{Max} = TP + FN \quad (2.2)$$

$$TN_{Max} = TN + FP \quad (2.3)$$

$$\text{Sensitivity} = TP / TP_{Max} \quad (2.4)$$

$$\text{Specificity} = TN / TN_{Max} \quad (2.5)$$

$$\text{Accuracy} = (TP + TN) / (TP_{Max} + TN_{Max}) \quad (2.6)$$

$TP_{Max}$ and $TN_{Max}$ are the largest possible respective values of TP and TN, for a given problem. $TP=TP_{Max}$ and $TN=TN_{Max}$ are both true whenever a result is equal to the model solution with which one compares it.

When counting the categories TP, FP, FN and TN, one must make special considerations for $A_R(i)=0$ and $A_S(i)=0$, since the value zero was reserved for unclustered data points. If one of the two points satisfies $A_R(i)=0$, then they are not clustered together in the result, and if one of the two points satisfies $A_S(i)=0$, then it is not clustered together in the solution. This is a trivial addition to the counting process, but important to include.

It is worth noting that, for any clustering problem, there is a trivial solution, grouping all inputs together, giving a sensitivity of 1 (optimal) and another trivial solution, assigning a separate cluster to each data point (or putting them all in the unclustered group), giving a specificity of 1. Thus one cannot measure clustering well with only one of these criteria. One needs to either consider both or combine the criteria in some way. Accuracy is one such combination, which makes the assumption that false positives/negatives are equally bad. For now, specificity and sensitivity will be observed together. It will be seen lager (section 4.1) that specificity is not always a suitable, and an alternative criterion, precision, will be introduced.

Note that categorizing each pair in P explicitly and hence counting the number of pairs in each category, TP, FP, TN, FN, has a time complexity of $O(N_D^2)$, which is not remotely as bad as the factorial time complexity of combining result and solution clusters (2.2.1). Nevertheless, it can make quality measuring very time consuming as N grows large. Fortunately, one does not need to count all these pairs explicitly. This can be calculated much faster, with time complexity $O(N_D+N_R \times N_S)$, as described in chapter 3. This is always better than $O(N_D^2)$, since both $N_R$ and $N_S$ must always be smaller than $N_D$. It makes no sense for there to be more clusters than data points, since some clusters would then have to be empty.

# 3 Efficiently calculating TP, FP, FN and TN

It is not necessary to read this rather technical section in order to follow the rest of this document, but it is included for completeness and for the interested reader, because evaluating clustering performance is a critical component of the work in this report, without which cluster quality evaluation would be much less efficient. Algorithm for calculating TP, FN, FP and TNAppendix C gives pseudo-code that implements the steps described in the rest of this chapter.

## 3.1 Efficiently calculating TP

Any pair of points, $(i_1, i_2)$ is a true positive (TP) if and only if both points are within the same result and solution clusters, $r_j$ and $s_k$. This is the case if and only if both points are within the

intersection[4] $r_j \cap s_k$. Given a 2-dimensional array, Z, of cardinalities[5], $Z_{j,k}=|r_j \cap s_k|$, of each intersection, the total number of pairs of points within each intersection is $Z_{j,k} \times (Z_{j,k}-1)/2$. Finding $Z_{j,k}$, for a particular intersection, can be done in $O(N_D)$ time by processing every data point, $d_i$, once, and incrementing a counter whenever $A_R(i)$ equals j and $A_S(i)$ equals k.

In order to find the total number of TPs, one could repeat the process of the previous section for each intersection in turn, before adding together the values $Z_{j,k} \times (Z_{j,k}-1)/2$. This means that the process would be repeated $N_R \times N_S$ times, making the whole calculation of TP require $O(N_D \times N_R \times N_S)$ processing time. Fortunately, there is a much more efficient alternative, based on the fact that each data point, $d_i$, is a member of exactly one of the intersections $r_j \cap s_k$.

1. Initialize all elements in $Z_{j,k}$ to zero.
2. For each data point, $d_i$:
   2.1. $j=A_R(i)$
   2.2. $k=A_S(i)$
   2.3. Increment $Z_{j,k}$ by 1.
3. $TP = \sum_{j=1}^{N_R} \sum_{k=1}^{N_S} Z_{j,k} \times (Z_{j,k} - 1)/2$

Note that the unclustered elements at j=0 and k=0 are accounted for here by simply ignoring any j=0 and k=0 in the sums of step 3. This is correct because pairs that are unclustered in the result are (by definition) not positive and pairs that are unclustered in the solution can only be true if they are not positive.

Steps 1 and 3 in the above algorithm have a time complexity of $O(N_R \times N_S)$, whilst step 2 has time complexity $N_D$. So, the whole algorithm has time complexity $O(N_D+N_R \times N_S)$. The memory complexity is $O(N_R \times N_S)$, for storing the array Z.

### 3.2 Efficiently calculating FN

Recall that $TP_{Max}=TP+FN$ is the total number of pairs of points that should have been grouped together in an ideal case. $TP_{Max}$ is the maximum number of TPs that a clustering result could ever accomplish for a given problem. TP equals $TP_{Max}$ whenever the result clustering is equal to the solution clustering, so one can simply find the number of TPs for each solution cluster.

The solution, by definition, cannot contain any FP, so all positives must be TPs. Thus all points within $s_k$ will be paired with all points that are not in $s_k$. Given an array, C, of cardinalities, $C_k=|s_k|$, of each solution cluster, the total number of such pairs for a given solution cluster,

[4] The intersection of two sets, A and B, is the set containing all elements that are both in A and B.
[5] The cardinality, |S|, of any set, S, is the number of elements within S.

$s_k$, is $C_k(C_k - 1)/2$. C can be found by processing all points in D once, counting how many times each different value of $A_S(d_i)$ occurs. Then $TP_{Max}$ is given by formula (3.1):

$$TP_{Max} = \sum_{k=1}^{N_S} \frac{C_k \times (C_k - 1)}{2} = \frac{\sum_{k=1}^{N_S} C_k \times (C_k - 1)}{2} \quad (3.1)$$

Notice that the summation ignores k=0, since the group of unclustered points does not contribute toward $TP_{Max}$. One can now calculate FN by $FN=TP_{Max}-TP$. Calculating $TP_{Max}$ does not depend on the clustering result, only the solution. Therefore, if many different clustering results are to be compared, for example in order to optimize cluster parameters, then $TP_{Max}$ can be calculated only once and reused for all cluster evaluations. This does not change the time complexity of the whole algorithm, but will reduce the execution time somewhat. The time and memory complexities for calculating FN are respectively $O(N_D+S_D)$ and $O(S_D)$. These complexities are of lower order than the ones for TP, so they do not influence the overall complexity analysis.

### 3.3 Efficiently calculating FP

Every pair of clusters gives a certain contribution of FPs, similar to the calculation of TPs. For a given pair of clusters $r_j$ and $s_k$, every point within the intersection, $r_j \cap s_k$ will be part of an FP with every point that is in $r_j$ but not $s_k$. So, if one knew the size, $Z_{j,k}$, of the intersection $r_j \cap s_k$, and the size, $Q_{j,k}$ of the set difference[6] $r_j \backslash s_k$, then the contribution of FPs from the combination of $r_j$ and $s_k$ would be $Z_{j,k} \times Q_{j,k}$. $Z_{j,k}$ has already been counted in section 3.1. Any given $Q_{j,k}$ can be calculated by formula (3.2):

$$Q_{j,k} = |r_j \backslash s_k| \quad (3.2)$$

The set difference operation removes from $r_j$ all elements contained in $r_j$ that are also in $s_k$. These elements are the intersection $r_j \cap s_k$, and so the formula is equivalent to:

$$Q_{j,k} = |r_j \backslash (r_j \cap s_k)| \quad (3.3)$$

The elements in $r_j \cap s_k$ are all part of $r_j$, and so the number of elements removed from $r_j$ by the set difference operation is the size, $|r_j \cap s_k|$. Hence the formula can be simplified as follows:

$$Q_{j,k} = |r_j| - |r_j \cap s_k| = |r_j| - Z_{j,k} \quad (3.4)$$

Simply adding the products $Z_{j,k} \times Q_{j,k}$ together would count every FP twice. The reason for this is that every point, $d_i$, that is within an intersection with some result cluster $r_j$ is also outside another intersection with $r_j$. So the sum must be divided by two. The sum must also ignore any points in $r_0$, since these are explicitly unclustered, and therefore not false positives, or positives at all.

[6] Given two sets, S1 and S2, the set difference, S1\S2 is the set of elements that are in S1 but not in S2.

Finally, all points in some intersection, $r_j \cap s_0$, do not only form FPs with points in the set differences $r_j \setminus s_0$. They also form false positives with all other points within that intersection. Thus the number of pairs of points within those intersections must be added to the total count of FPs. These considerations lead to the following formula for calculating FP:

$$FP = \left( \sum_{j=1}^{N_R} \left( \sum_{k=0}^{N_S} Z_{j,k} \times Q_{j,k} \right) \right) \Big/ 2 + \sum_{j=1}^{N_R} Z_{j,0} \times \left( Z_{j,0} - 1 \right) \tag{3.5}$$

Having already calculated all $Z_{j,k}$, this calculation has both time and memory complexity $O(N_R \times N_S)$, which causes no increase in the complexity of the whole algorithm.

### 3.4 Efficiently calculating TN

Having already found FP, one only needs to find $TN_{Max}$ in order to calculate $TN=TN_{Max}-FP$. Consider each solution cluster, $s_k$. The points within $s_k$ should not be clustered with any points outside $s_k$. Having already counted the number of points, $|s_k|$, within each solution cluster, the number of points that are not within each solution cluster is easily calculated as $N_D-|s_k|$. Then $TN_{Max}$ is given by formula (3.6):

$$TN_{Max} = \sum_{k=1}^{N_S} |s_k| \times (N_D - |s_k|) \tag{3.6}$$

Hence, TN can be easily calculated, and this part of the algorithm has time and memory complexities $O(N_S)$ and $O(1)$ respectively. Again, this causes no increase in the complexity of the overall algorithm.

### 3.5 Summary of the algorithmic complexity

This chapter showed how TP, FN, FP and TN can be calculated, based on the *sizes* of result clusters, solution clusters, intersections, differences and complements of these sets. The algorithm is summarized with pseudo-code in Appendix C. The overall time complexity of the algorithm is $O(N_D+N_R \times N_S)$. The memory complexity is $O(N_R \times N_S)$, which is pretty small except when there are a very large number of clusters. This is a great improvement compared with the time complexity $O(N_D^2)$, which would inspect the cluster memberships of every pair of data points explicitly.

## 4 Applying the evaluation criteria to real data

The introduction mentioned three scenarios in which one may have access to an ideal solution, with which to compare a deinterleaving result. One situation was when the pulses were generated by a PDW simulator. This was the approach taken in (1), where the LINE deinterleaver was tested using data from the PDW simulator described in (1). The sensitivity and specificity criteria were optimized with respect to the relative jitter tolerance (J in the algorithm of Appendix A), and

then after the optimization, the performance was measured on other parts of the same data set. The optimization gave great improvements of the deinterleaving, which was verified by visual inspection of some of the cases. (1) also suggests some further applications of the PDW simulator data, in the context of deinterleaver development.

In the following sections, the LINE deinterleaver is instead applied to real data, for which a solution has been developed manually in advance. Inputs and parameters are manipulated in order to optimize the deinterleaver and test it on new challenges.

## 4.1 Using navigation radar data that have already been deinterleaved

One interesting application, mentioned in the introduction, is to measure deinterleaving performance on a real data set. Figure 4.1 shows an example data set that will be used here. The data set shown in Figure 4.1 was first deinterleaved by the LINE deinterleaver, hereafter known as LINE_Deint, before the result was improved further using a manual deinterleaver, developed by the author. Figure 4.2 shows the results of this deinterleaving, and will be used as the solution, against which deinterleaving results will be compared.

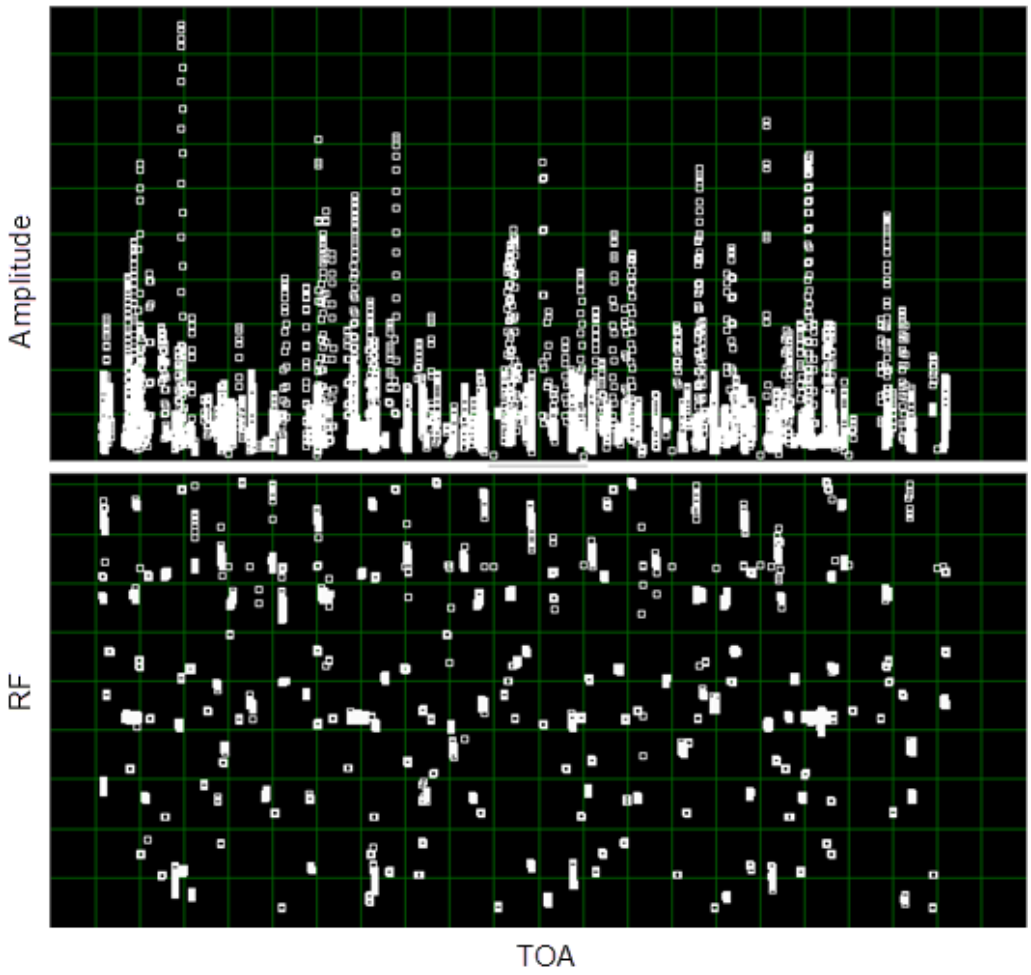


*Figure 4.1 Amplitude (top) and radar frequency (bottom) plotted against time for an ESM recording of navigation radars. Numbers and further information about the data is left out in order to keep this report unclassified.*

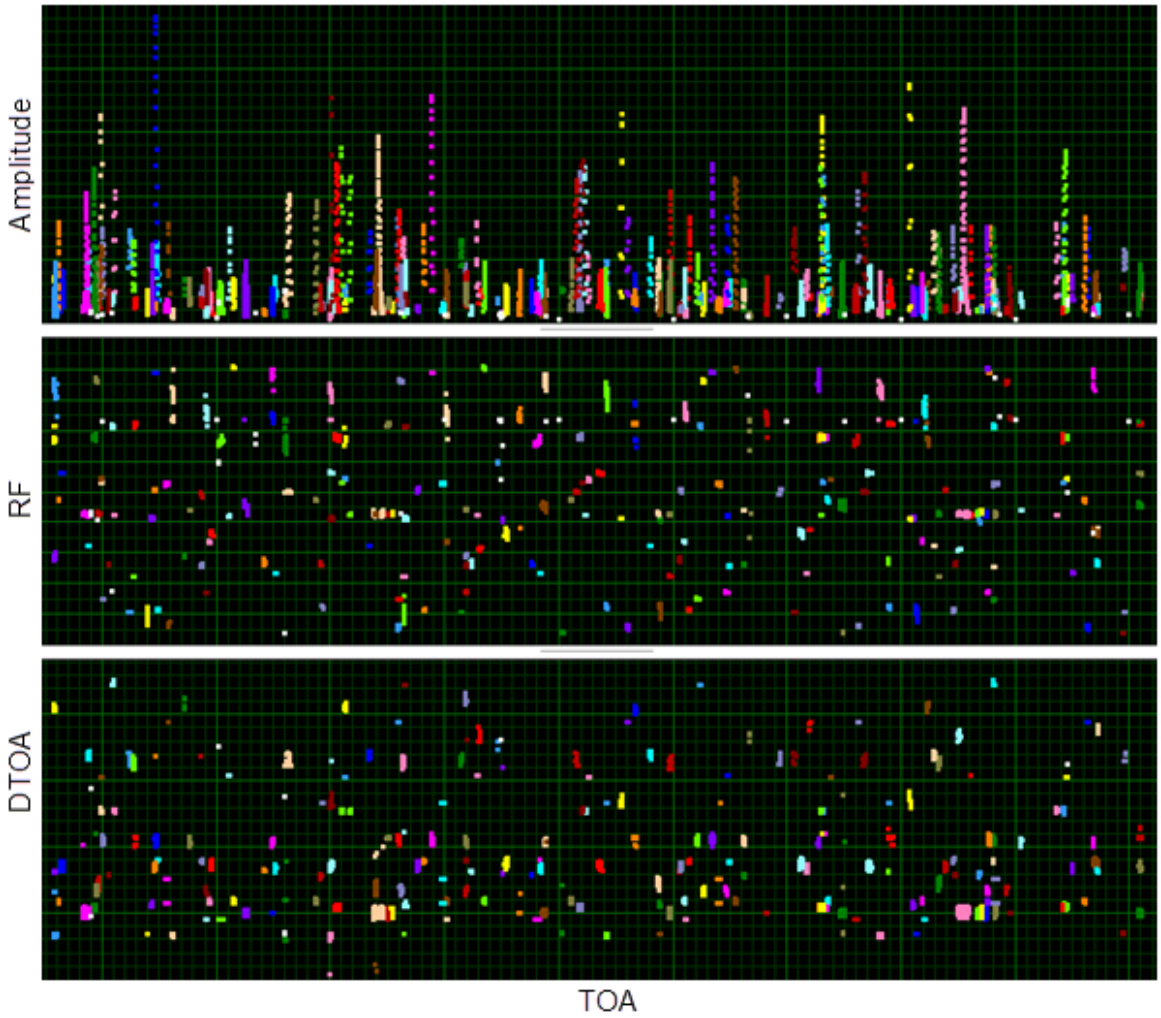


*Figure 4.2* *This figure shows the data from Figure 4.1 after careful deinterleaving. The result is presented with a different color for each emitter, but with only 18 different colors, since similar colors are hard to discriminate for a human reader. Therefore, colors are repeated for different emitters, which will be the case for all subsequent deinterleaving plots. The bottom plot shows the difference in TOA between consecutive pulses (i.e. the PRI). The deinterleaving result is not guaranteed to be correct throughout the plot, but it is more than good enough as a solution against which to compare the results of automated deinterleaving.*

This solution does not merge (put together) sequences of pulses when there are big gaps of time between the sequences. Hence, each emission is considered to be terminated when there is a big gap in time. Figure 3 shows an example of two such sequences, which have practically identical distributions, both of PRI (bottom plot) and frequency (middle plot). With merging, one would expect these to be put together.

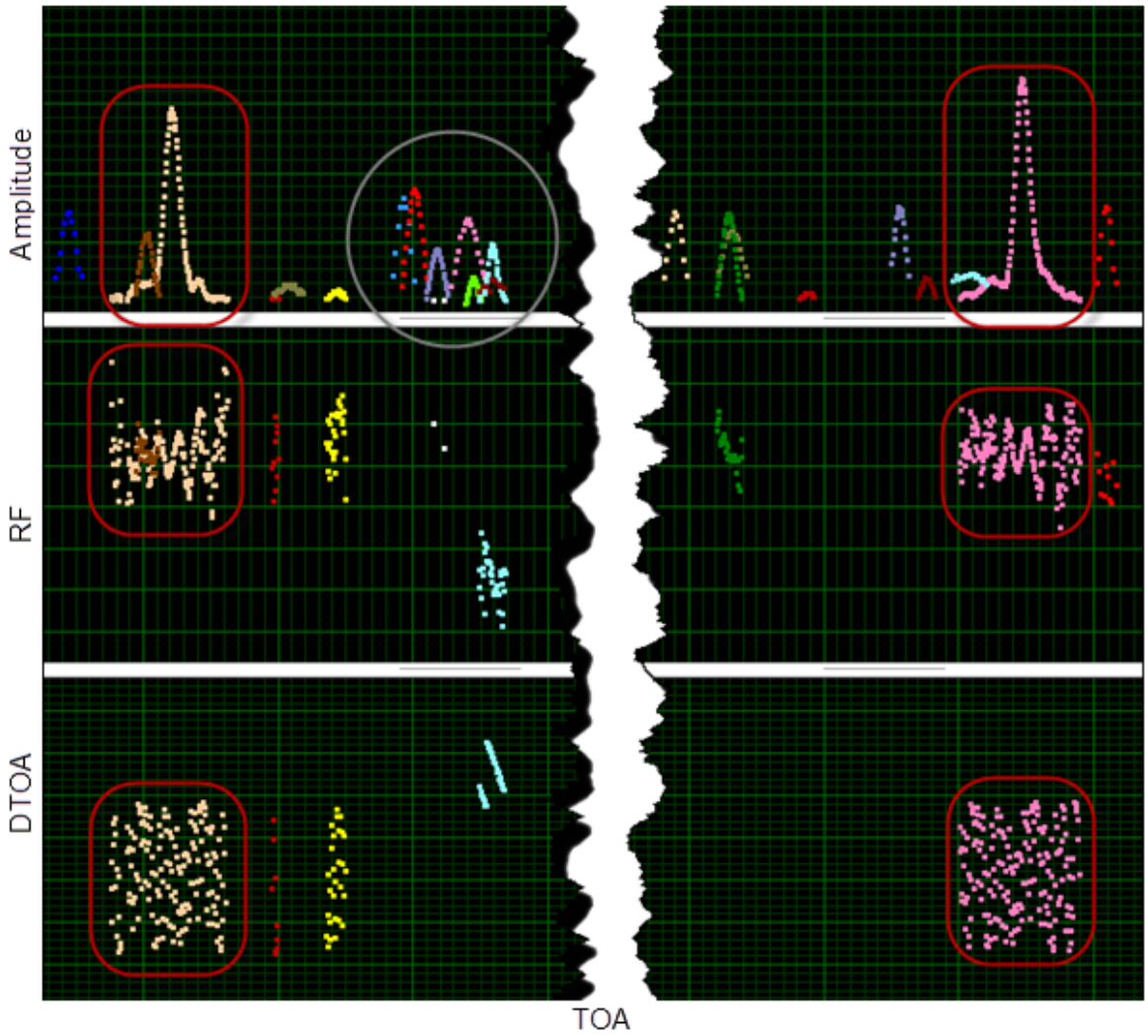


*Figure 4.3 Zoomed in version of two parts of Figure 4.2. The majority of the pulses in the rounded rectangles have practically the same PRI- and frequency-distributions and similar lobe shapes, suggesting that they may originate from the same emitter.*

Having created a solution, it is time to evaluate some results against the solution. Appendix A describes the LINE deinterleaving algorithm, which was also used in (1). One important parameter in LINE_Deint is the relative jitter tolerance; J. *Figure 4.4* shows the performance criteria specificity and sensitivity, discussed earlier.

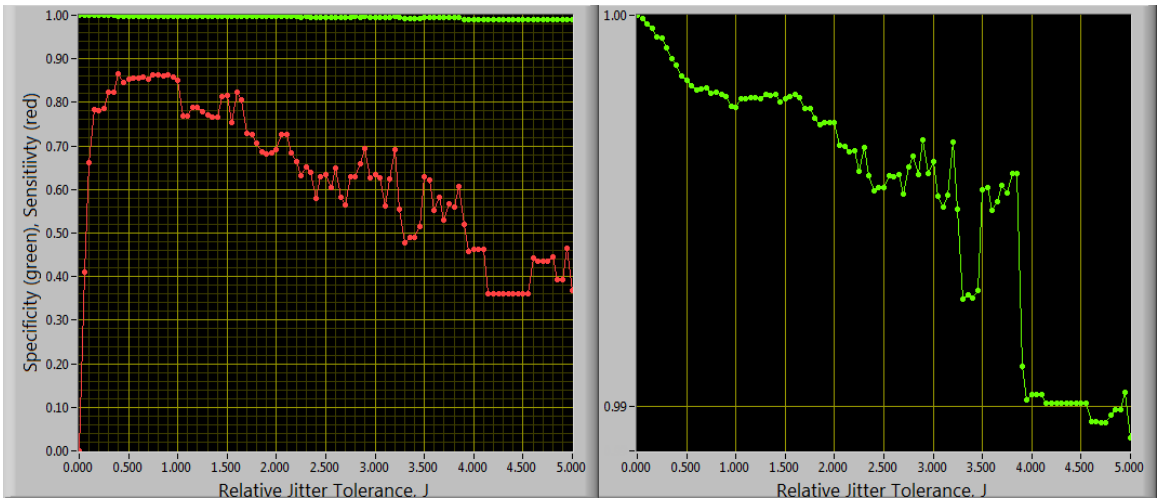


*Figure 4.4 Specificity (green) and sensitivity (red) after using LINE_Deint, varying J from 0 to 5.*

There are two remarkable things to observe about *Figure 4.4*. Firstly, the greatest value of J is 5. This means that, in the search for sequences of pulses with a fixed PRI, one tolerates a leeway that is 5 times greater than the PRI itself. This is an enormous value, since the PRI normally does not vary by more than 20%, and that is already a lot. However, the remarkable observation is that the sensitivity remains as high as around 0.4, i.e. 40% of the pairs of pulses that should be grouped together are grouped together. The reason for this is probably that the data are relatively uninterleaved, in the first place. Many emissions overlap no other emission, and so, since the algorithm searches for sequences of *consecutive uninterleaved* pulses, it successfully finds these non-overlapping emissions. On the other hand, if the data had been highly interleaved, the large value of J would have likely caused more arbitrary deinterleaving results, and hence a lower performance.

The second remarkable thing, which is even more remarkable, is that the specificity is approximately 0.99-1, independent of the value of J. In other words, the pairs of pulses that should not be together are not together, in most cases.

The reason for the great specificities in this case, is that the data consist of lots of small emissions of data separated by relatively big time gaps (see e.g. Figure 4.3). Even if one (largely erroneously) grouped together many pulses that were relatively near each other in time, there would still be a much larger number of pulses, before and after a time gap, with which these pulses were successfully not grouped together. Hence the numerator in the right hand side of formula (2.5) is practically as big as the denominator. Unfortunately, this means the deinterleaver gets away with a great mark (specificity) for a terrible deinterleaving job, which is illustrated in Figure 4.5. Notice the rainbow patterns, which suggest that a new emitter is generated for nearly every pulse.

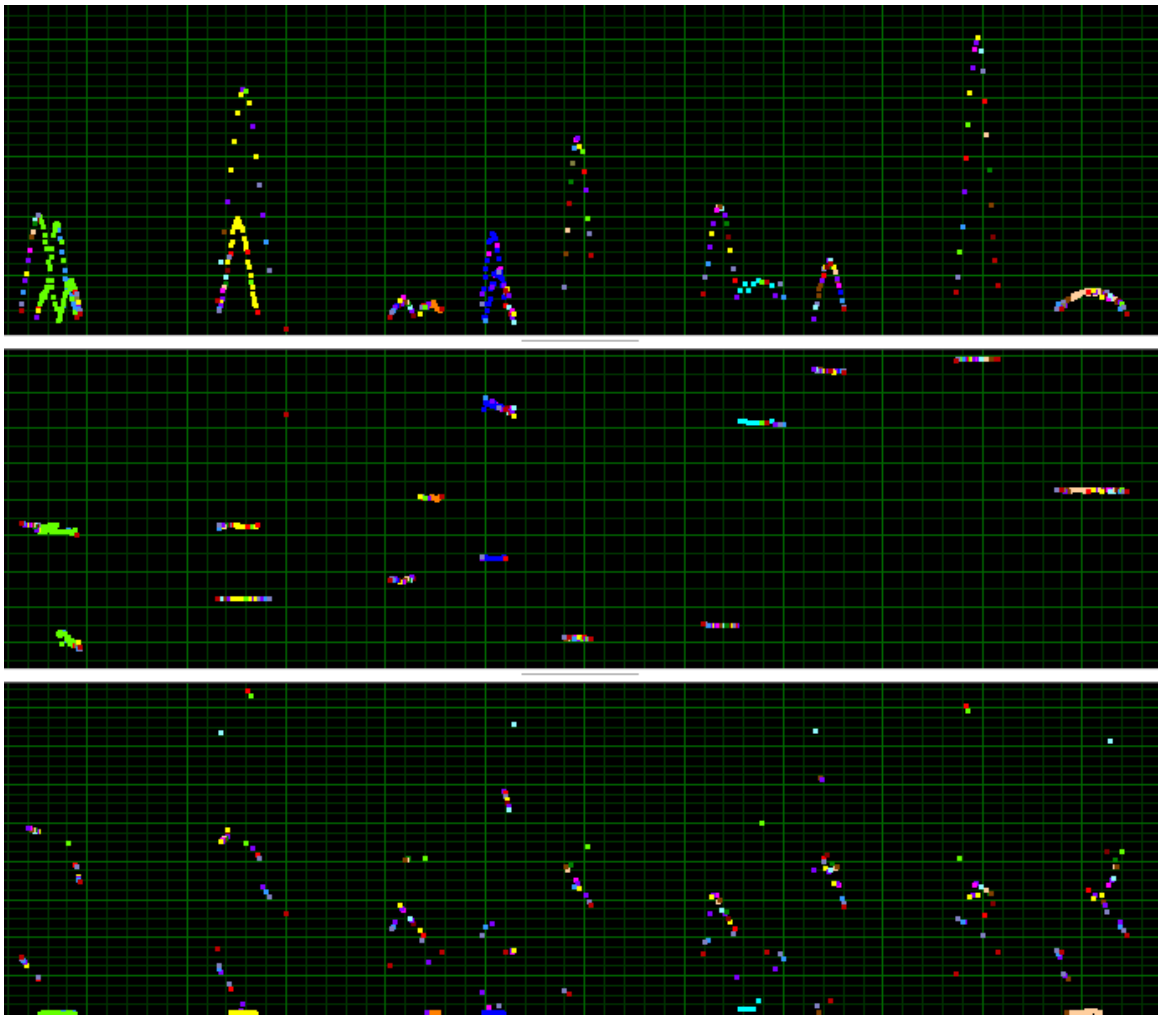

*Figure 4.5 Zooming in on the result of applying LINE_Deint with J=5 to the data shows how terrible the result is, despite the respective specificity and sensitivity values of 0.99 and 0.37.*

This shows that specificity does not necessarily say a lot, in absolute terms, about how well a result avoids forming the pairs that it is not supposed to form. Specificity may still be quite applicable as an optimization criterion, but then one could just as well use TN (section 2.2.2), from which specificity was derived, since the numerator, $TN_{Max}$ is independent of the result. An alternative criterion is precision, given as:

$$Precision = \frac{TP}{(TP + FP)} \tag{4.1}$$

This is the proportion of the pairs that were rightfully formed, relative to the total number of pairs that were formed. This gives a much more reasonable performance curve, shown (with sensitivity) in Figure 4.6. The curves follow each other pretty closely for extreme values of J, but diverge for the more interesting values of J, below 0.5.

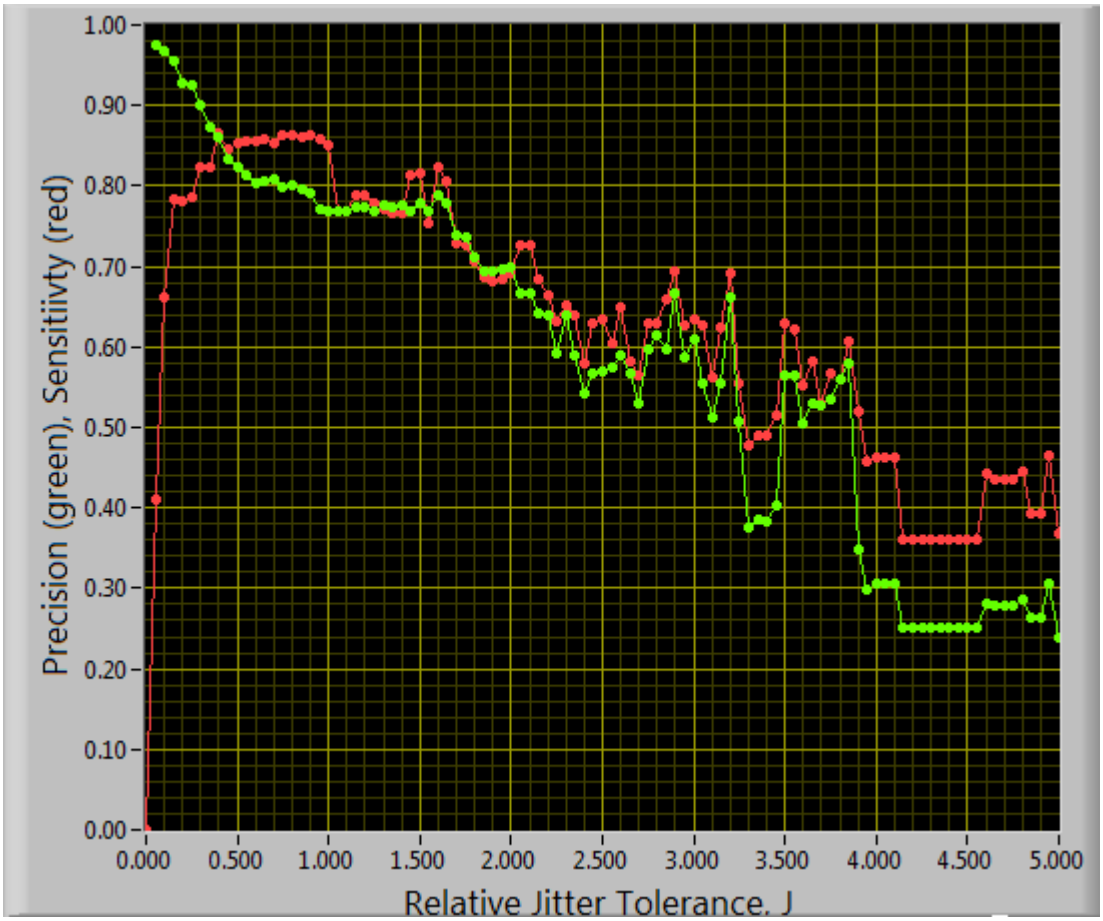


*Figure 4.6 Precision (green) and sensitivity (red) for the same values of J as in* Figure 4.4.

Now, the performance can be analyzed further, by looking at the sensitivity and precision in the range J∈[0, 0.5], which is shown in Figure 4.7. J=0.15 gives a pretty good precision of nearly 0.98, and a reasonably good sensitivity of 0.78. As one would expect, increasing J trades some of the precision for an improved sensitivity. Beware that the crossover point of the green and red curve is irrelevant. Instead, one should establish how important the precision is, relative to the sensitivity, which depends on the usage of the deinterleaver.

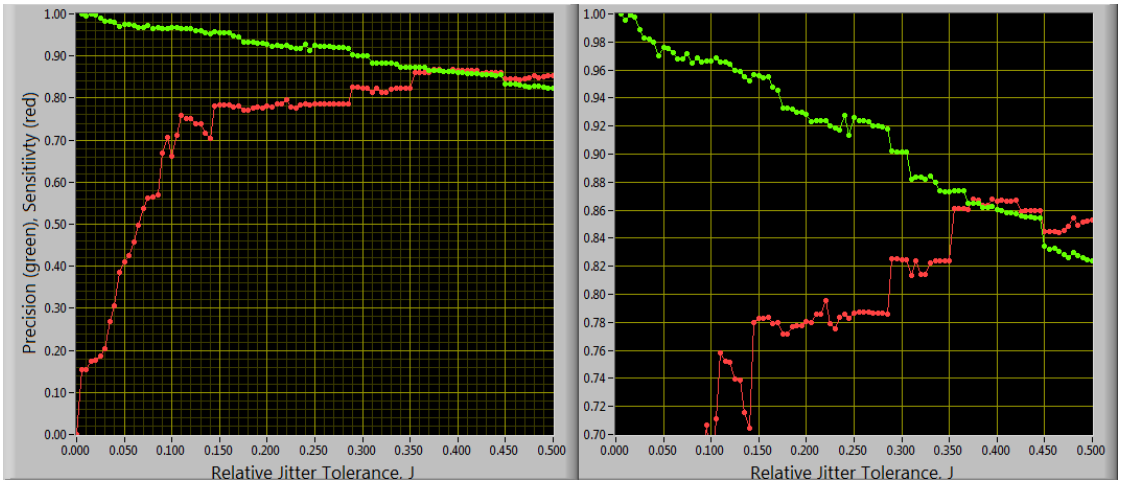


*Figure 4.7 Precision (green) and sensitivity (red) for the more interesting range from 0-0.5.*

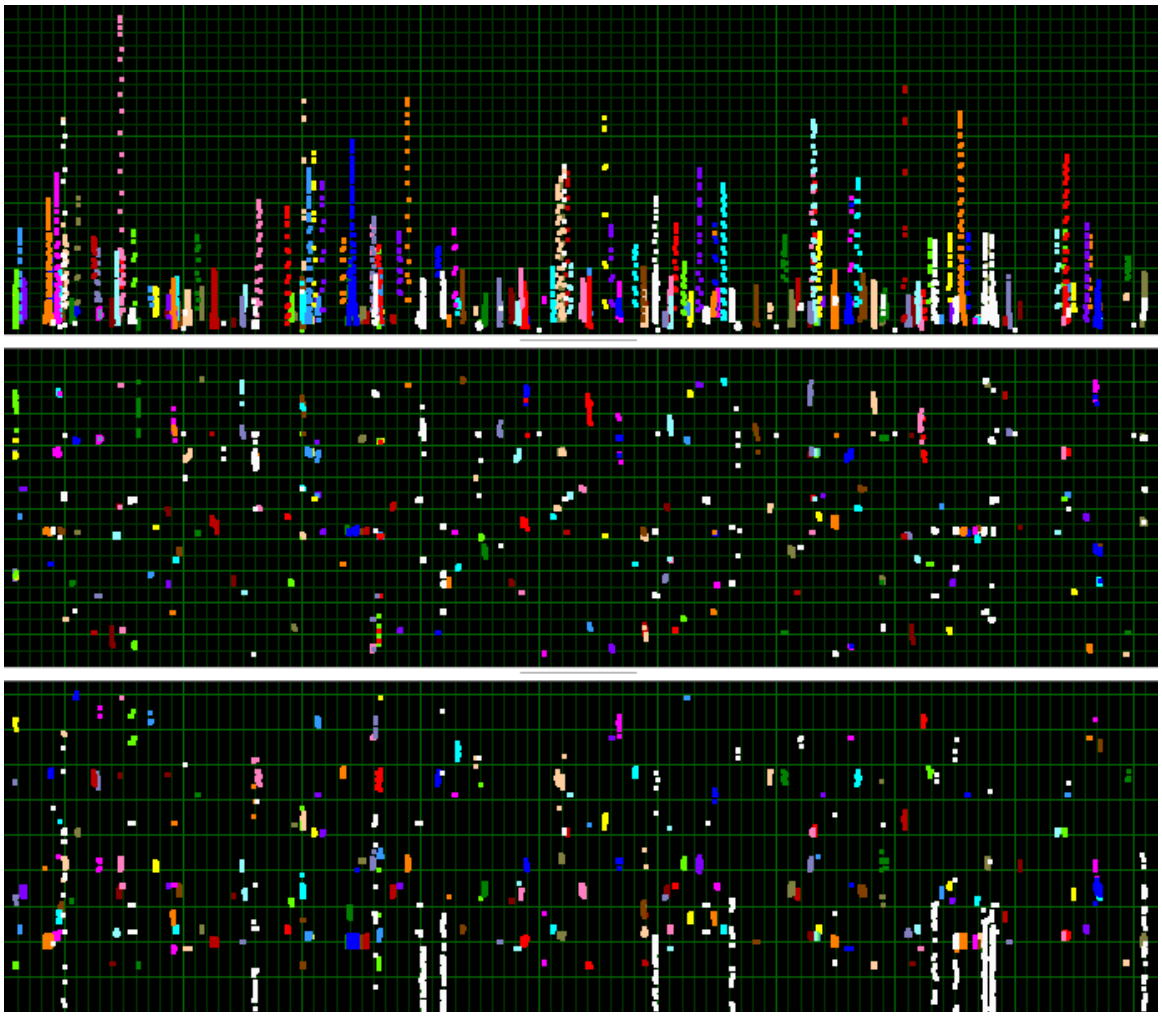

*Figure 4.8 Deinterleaving result for J=0.15.*

Figure 4.8 shows deinterleaving results for J=0.15, choosing a high precision over some extra gain in sensitivity. Figure 4.9 zooms in on one part of Figure 4.8, giving a clearer look at how the deinterleaving goes. The left most part of the top plot of Figure 4.9 shows a lobe with some alternation in colors (perhaps beige and brown) along the lobe. This suggests that the deinterleaving has gone somewhat awry. Apart from that, the results look pretty good. However, zooming in further, as shown in Figure 4.10, reveals that one emission has been split into two halves.

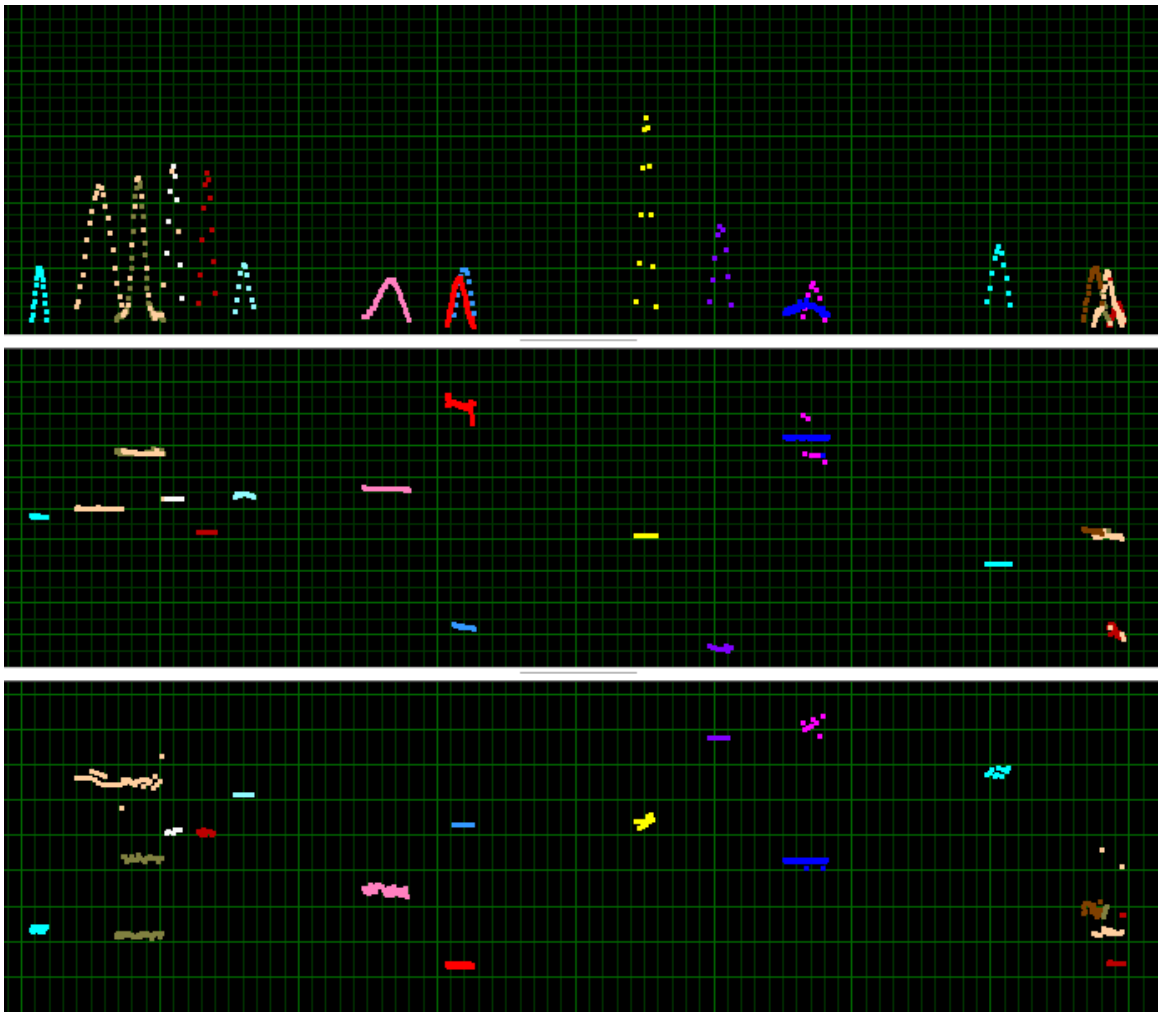

*Figure 4.9 Zoomed in version of Figure 4.8, showing some examples of where LINE_Deint succeeds or fails.*

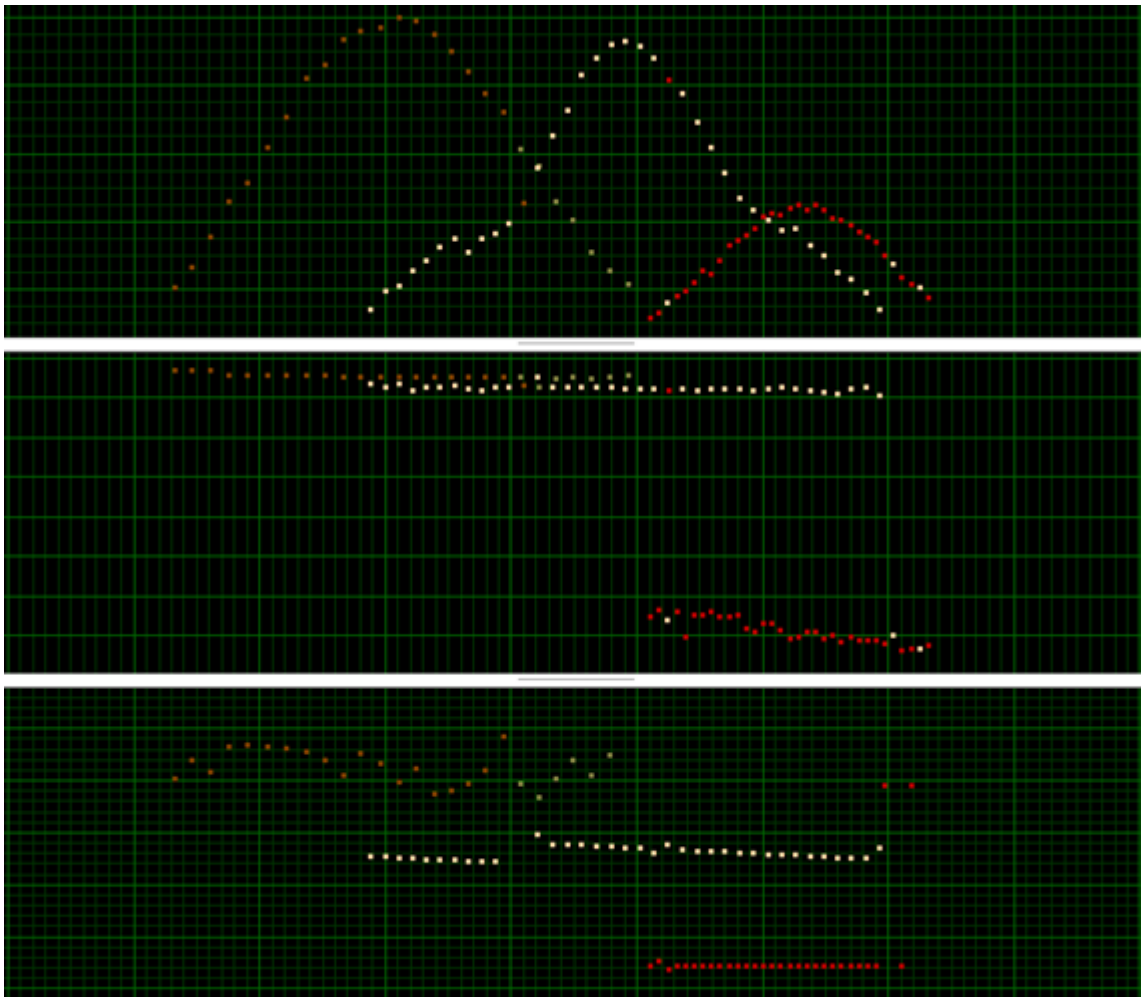

*Figure 4.10 The right most lobes that were shown in Figure 4.9. Notice the transition in color of the brown lobe as it is about to intersect the other (grayish) lobe. In this case, the deinterleaver has split one actual emission into two perceived emissions.*

## 4.2 Testing the GEOIDE deinterleaver on the same data as for LINE_Deint

During the GEOIDE projects, a deinterleaver was developed (here referred to as GEOIDE_Deint, intended to be used by other programs developed in GEOIDE. GEOIDE_Deint used frequency clustering followed by PRI-analysis for validation. It also had other features, which are not relevant to this discussion. For further details, please refer to (2). GEOIDE_Deint has some adjustable parameters, but it was designed to work autonomously, so they are not manipulated here.

When applying GEOIDE_Deint to the example input from this chapter, it gives a result with sensitivity of 0.74, which is not bad, compared to LINE_Deint. However, it should be said that LINE_Deint was developed in a scenario where only time and amplitude was available, whilst GEOIDE_Deint also uses frequency. This means that the algorithms are competing on pretty different terms. Since LINE_Deint is not designed to utilize all data given to it, one really should expect it to perform worse than GEOIDE_Deint, although the opposite is the case, for many values of J.

Unfortunately, the result from GEOIDE_Deint gives a very low precision of 0.07. This is pretty terrible, since it means that only 7% of the pulse pairs that were formed should have actually been formed, thus most pairs are there by some error. However, this has the simple explanation that GEOIDE_Deint does some automated merging. Since the solution was built without merging, this gives a huge number of pairs that may be correct when merging is wanted, but is completely wrong in this context.

A simple solution is to unmerge the results by looking for big time gaps within the data of each emission in turn, within the result produced by GEOIDE_Deint. After unmerging, the sensitivity was reduced to 0.72, but the precision was increased to 0.93. This result is not bad at all, although still not as good as the best results from LINE_Deint. However, the comparison is still somewhat unfair, since LINE_Deint, unlike GEOIDE_Deint, was optimized for this particular data set. Nevertheless, this illustrates how the real data with existing solutions can be used in order to choose between algorithms.

This concludes the comparison with GEOIDE_Deint. The following sections will direct the focus toward testing LINE_Deint in different scenarios.

### 4.3 Using the same data to generate new scenarios

The deinterleaved data from this chapter can be modified in several ways in order to provide different challenges to the deinterleaver. Here are some suggestions for changes that can be made to one individual emission. Note that, each of the manipulations below is made on an individual emission, and not on the entire record:

(1) Change the time the emission arrives by adding a constant to the TOAs of all the pulses.
(2) Change the PRI by multiplying the TOA differences between consecutive pulses by a constant and modifying the TOA values accordingly.
(3) Change the perceived distance between the emitter and the ESM sensor by multiplying all amplitude values by a constant.
(4) Change the radar frequency, by adding a constant to the radar frequencies of all pulses.

The following are more complex manipulations that could also be made to the data:

(5) Change a fixed PRI pattern into a stagger pattern, by adding different constants to different subsets of the pulses, representing each pattern. Whether this would be realistic is not certain, and pulse attributes would also have to be carefully modified in order to account for the changes in the time between pulses.
(6) Dwell switching could be simulated, by similar considerations as for stagger.
(7) Radar frequency patterns could be simulated in similar ways as stagger and dwell switching. This may also be easier, as other parameters may not need adjustment.

All of the above modifications can be implemented on many or all emissions within a recording. Suggestion (1) preserves the realism of the original recording, but still provides plenty of opportunity for creating new scenarios. It will therefore be the focus in the following sections.

### 4.3.1 Randomly moving every emission

An easy way to use the original recording to create new challenges is to apply suggestion (1) to all the emissions, using a different random constant for each emission. This can be done repeatedly, exposing the deinterleaver to lots of different scenarios, sometimes with several emissions received at the same time, sometimes with emissions evenly distributed. Here, the constants were drawn from a uniform distribution, so that all the data stayed within the original time range. J=0.15 was kept from earlier experiments, since it proved pretty successful. Running this 100 times, the following performance statistics:

| | Mean | Standard deviation | Minimum | Maximum |
|---|---|---|---|---|
| Sensitivity | 0.89 | 0.03 | 0.76 | 0.94 |
| Precision | 0.70 | 0.06 | 0.56 | 0.83 |

The worst case precision is relatively poor, probably caused by lots of emissions arriving at approximately the same times. This could be analyzed further. Alternatively, one could automatically retry the algorithm when the result is poor, and then use a different value for J, or other parameter adjustments.

### 4.3.2 Randomly moved emissions on a resized time axis

In the previous section the emissions were moved randomly along the time axis in order to provide new challenges to the deinterleaver. This can be combined with resizing the range of time values of the emissions. Decreasing or increasing this size creates a, respectively denser or sparser signal environment.

Figure 4.11 illustrates how the deinterleaving job gets gradually easier when the emissions are distributed along a larger range of time values. This makes sense since, with a great time range the emissions will not tend to overlap. This makes emissions easy to discover and reduces risk that pulses from one emission get mixed in with pulses from another emission.

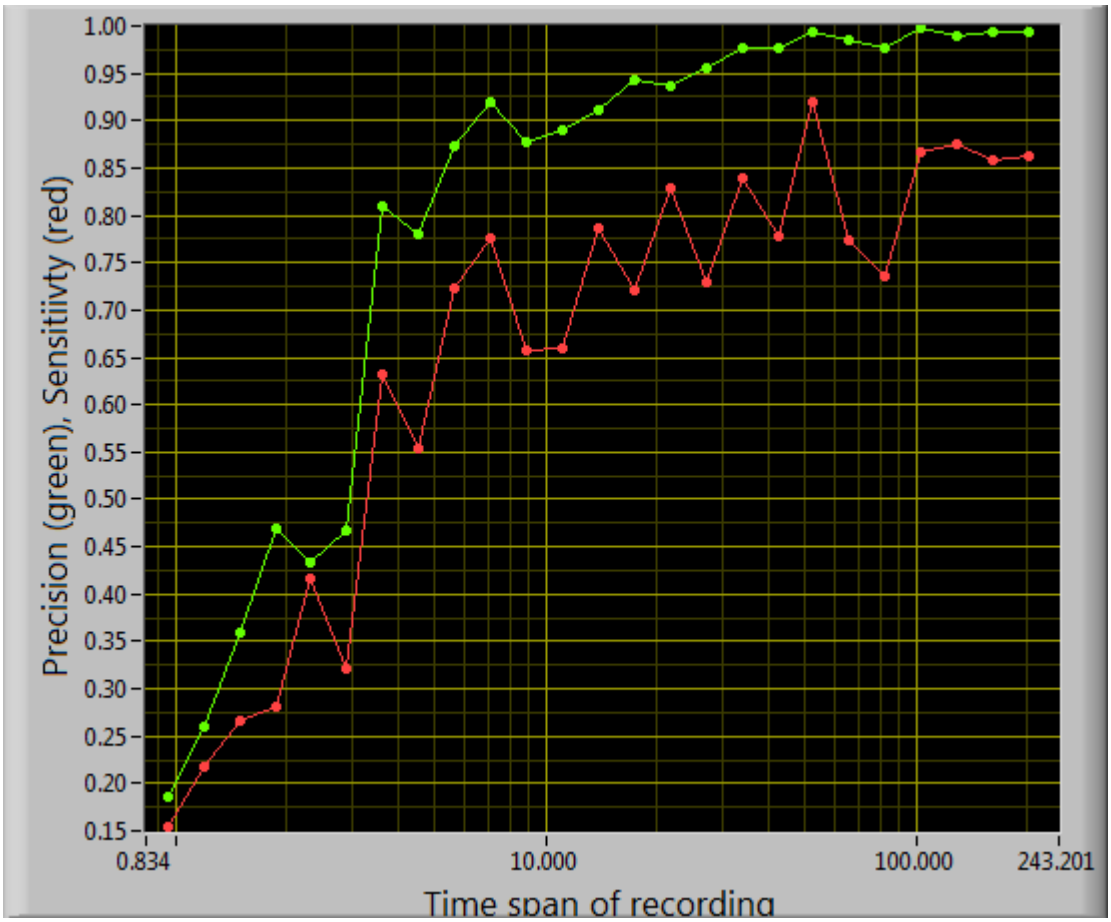


*Figure 4.11 Precision and sensitivity when deinterleaving randomly moved emissions over time spans (ranges) of different size.*

The performance curve in Figure 4.11 oscillates quite rapidly, rather than increasing continuously as the time span is increased. This is probably due to the random positioning of the emissions. Figure 4.12 confirms this by computing the mean precision and sensitivity after deinterleaving ten different, randomly arranged, sets of emissions for each time span. The oscillation is reduced considerably, whilst the increasing trend remains as before.

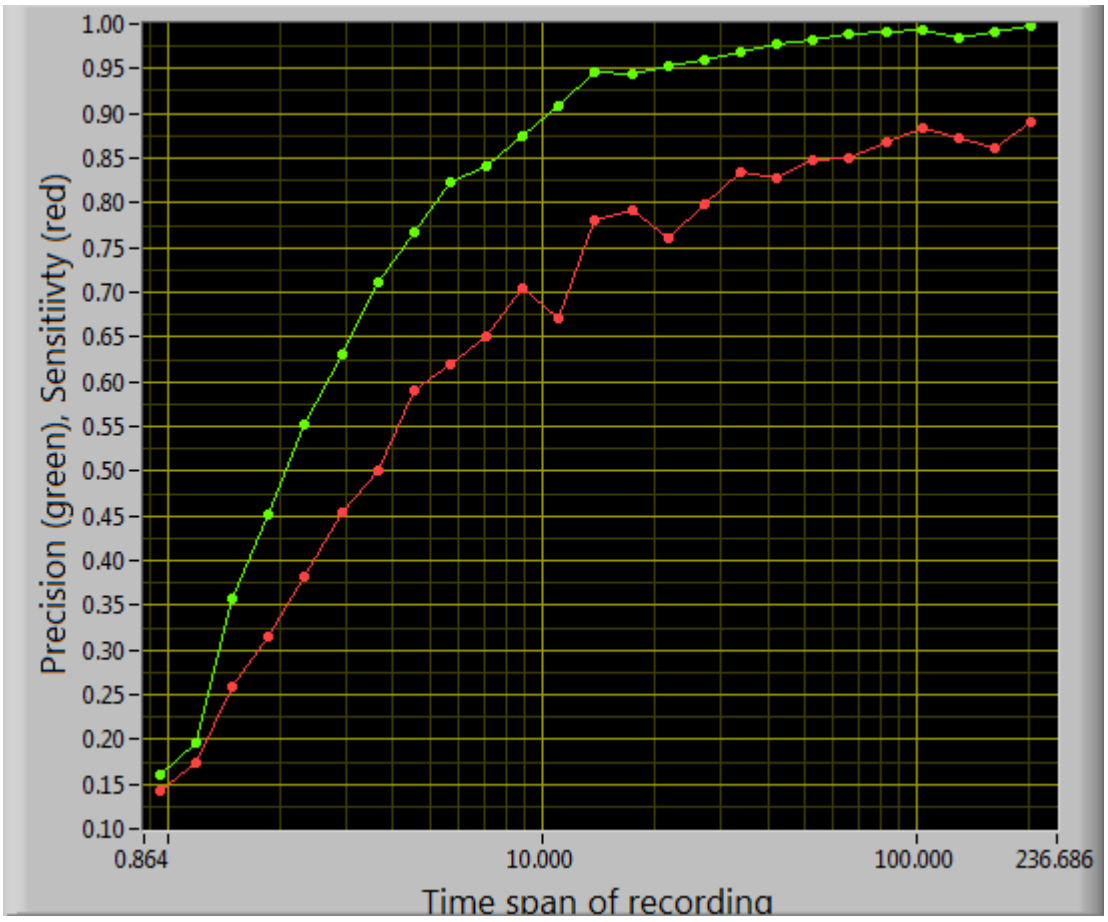


*Figure 4.12 Mean precision and sensitivity when deinterleaving randomly moved emissions over different time spans.*

## 4.4 Further work

Further modifications can be made to the data in order to generate different realistic signal environments. Currently, only suggestion (1) from section 4.3 has been exploited. The other suggestions have different potentials. Particularly suggestions (2) and (3) seem relevant to the deinterleaver, which only exploits the attributes TOA and amplitude, and only supports fixed PRIs.

The random adjustments in this section can be replaced by more targeted testing, in order to better understand the strengths and limitations of particular algorithms. For example, one could measure how the performance changes as two or more emissions approach each other in time.
The methods in this report can also be used for optimizing algorithms with respect to multiple parameters, before deploying the algorithm in a real scenario. Another very useful application of these methods is to build a test suite, which could be used both for the check if new deinterleavers meet certain requirements or that they keep meeting those requirements after modifications that are made in order to meet additional requirements.

Finally, having input data with correct solutions, as well as ways to evaluate candidate solutions against the correct solutions, give the potential of using various supervised learning methods in

future deinterleaving development. This depends on access to sufficiently representative data sets, which can be accomplished by modifying existing data sets, as seen in section 4.3. However, great care must be taken to preserve the realism and representativeness in the modified data, so that the machine does not just learn faulty assumptions one makes about the data.

A challenging, but very powerful future addition, is be to design evaluation criteria that do not depend on having a solution. However, developing such criteria may be almost as difficult as the deinterleaving job itself, at least if the criteria are to be highly reliable. However, given such criteria, one could apply them during an actual operation, to determine when a deinterleaving result is good enough, to choose between different candidate solutions and to automatically adapt the behavior of the deinterleaver during an operation.

# 5 Summary

This report has proposed some powerful criteria for evaluating clustering results against a model solution. These criteria have been applied to deinterleaved navigation data in order to optimize one deinterleaving algorithm and test two different deinterleaving algorithms. Discussion of the results has also identified weaknesses in the criteria, and hence guided the choice of which criteria to use.

Randomized testing was used in order to see how well the algorithm can cope with signal environments of different density. Some possible further work and future applications were discussed.

# Appendix A LINE Deinterleaver Algorithm, LINE_Deint

LINE_Deint uses the following configuring parameters:

- Sequence Length = N = 4
- Relative Jitter Tolerance = J = 0.04
- Missing Pulses Tolerance = M = 2
- Pulses Skipped = S = 0

The following input is given to the algorithm:

- A set, D, of $N_D$ data points, where each data point represents a pulse.

1) Remove half of the pulses – the 50% weakest (smallest amplitude) pulses.
2) Search for a sequence of N+1 consecutive pulses having a fixed PRI with Jitter < J × mean(ΔTOA).
    - Allow a total of S pulses from other emitters between these pulses. If S > 0, the algorithm quickly takes a lot more time. This has also proven unnecessary on relevant data from LINE, so S=0 has been chosen.
3) Now include all pulses, also the ones removed in step 1.
4) Perform a simple sequence search forwards and backwards in time from the sequence that was found. Stop the search whenever M consecutive pulses are missing in the sequence.
5) If a sequence was found in step 2, then the algorithm continues from step 1. Otherwise, the algorithm is terminated. Each sequence found in an iteration of step 4 contains the pulses from one emitter.

The time complexity (see Appendix B) LINE_Deint is $O(N_D(N + \log(N_D)))$, because S is currently set to 0. The time complexity with respect to a variable S is

$$O\left(N_D\left(\frac{(N+S)!}{S!\,(N-S)!} + \log(N_D)\right)\right) \tag{A.1}$$

This effectively means that LINE_Deint is dependent on finding a relatively uninterleaved sequence of pulses after the amplitude filtering. Otherwise, the process becomes extremely inefficient. However, some level of interleaving can be handled ok. For example, if N=4 and S=4, then the term $\frac{(N+S)!}{S!(N)!}$ Becomes a constant, 8!/(4!4!)=70, which is quite manageable. N=7 and S=8, giving a constant of 6435 might also be ok. On the other hand, N=20 and S=20, giving a constant of 138 billion, could easily be a show-stopper.

Step (2), the one leading to all the complexity when S is variable, is quite a powerful step. It can be applied without the amplitude filtering, or after some other filtering, for example based on frequency or bearing. Although it is limited by the complexity of increasing parameters, it can be quite useful, and one option is to first try it with only a small S (even S=0), before progressively trying larger values of S.

# Appendix B Complexity analysis and the big O notation

Complexity measures of an algorithm quantify how much of some resource will be used for the execution of the algorithm. Time complexity and memory complexity are two common examples of such measures. The big O notation, $O(\cdot)$, is frequently used for expressing time complexity and memory complexity in terms of the size of the input(s) to the algorithm. The big O notation excludes every term of an expression, except the one of highest order, and also leaves out constant coefficients. This makes it easy to compare the algorithms in terms of how well they scale to large inputs and whether it is feasible to use an algorithm at all for large inputs.

Example: Consider the task of sorting a list of N names into ascending order. This problem can be solved with the bubble-sort algorithm, which has time complexity $O(N^2)$. However, there are much more efficient algorithms, such as merge-sort, with time complexity $O(N \times \log(N))$, which is proven to be optimal. An $O(N^2)$-algorithm may still be more efficient than an $O(N \times \log(N))$-algorithm for inputs up to a certain size. However, when inputs get sufficiently large, $O(N \times \log(N))$ will be much better.

Memory complexity expresses the largest amount of memory that is required during the execution of an algorithm. For example, an algorithm may depend on building an array with some information about every input during the execution. This would have memory complexity $O(N)$. Another algorithm may require a 2-dimensional array of information relating every input to every other input. This would have memory complexity $O(N^2)$.

So far, all analysis has been in terms of a single input size N, but an algorithm may have several inputs of different quantities and different relevance to the efficiency. In this case, one may get complexities like $O(M \times N+N^2)$ and $O(M+N)$, in which neither term can be excluded because that would suggest that the excluded input size could be increased arbitrarily without influencing the efficiency of the algorithm. Thus, when there are several inputs, exclusion must be done with respect to all variables, and with some more care than for a single variable.

Finally, when analyzing complexity, it can be useful to rank the order of some different expressions. Knowing this ranking makes complexity analysis more efficient. The following list ranks different expressions by the order of each expression, so that expressions of higher order are to the left (separated by commas) or above ones of lower order. N is the variable size of the input, a and b are real numbers greater than one, such that $a > b$:

- $N^N$, $N!$, $a^N$, $b^N$, $N^a$, $N^b$, $N \times \log(N)$, $N$, $N^{b/a}$
- $\log(N)$ (Logarithms of different base are mutually equivalent, differing only by a coefficient)
- $\log(\log(N))$
- $a$ (Here ‘a’ is simply a coefficient of $N^0$, so $O(a)$ can simply be written $O(1)$).

The determining criterion of the ranking is the limiting behavior of the expression as N tends toward infinity. For further reading, please refer to (3) or another good text book on algorithmic complexity analysis.

# Appendix C Algorithm for calculating TP, FN, FP and TN

Here follows a full description for calculating all of TP, FN, FP and TN, sensitivity, specificity and accuracy. Justifications for the different steps of the algorithm were given in chapter 3.

1. Build the arrays Z, C and X as follows:
   1.1. Initialize all elements in $Z_{j,k}$, $C_k$, $X_j$ to zero
   1.2. For each data point, $d_i$:
      1.2.1. $j=A_R(i)$
      1.2.2. $k=A_S(i)$
      1.2.3. Increment $Z_{j,k}$ by 1
      1.2.4. Increment $C_k$ by 1
      1.2.5. Increment $X_j$ by 1
2. Build the array Q as follows:
   2.1. For each j,k
      2.1.1. $Q_{j,k} = X_j - Z_{j,k}$
3. The next equations reveal the values of TP, FN, FP, TN, sensitivity, specificity, precision and accuracy
   3.1. $TP = \sum_{j=1}^{N_R} \sum_{k=1}^{N_S} Z_{j,k} \times (Z_{j,k} - 1)/2$
   3.2. $TP_{Max} = \frac{\sum_{k=1}^{N_S} C_k \times (C_k - 1)}{2}$
   3.3. $FN = TP_{Max} - TP$
   3.4. $FP = \left(\sum_{j=1}^{N_R} \left(\sum_{k=0}^{N_S} Z_{j,k} \times Q_{j,k}\right)\right)/2 + \sum_{j=1}^{N_R} Z_{j,0} \times \left(Z_{j,0} - 1\right)$
   3.5. $TN_{Max} = \sum_{k=1}^{N_S} |s_k| \times (N_D - |s_k|)$
   3.6. $TN = TN_{Max} - FP$
   3.7. $Sensitivity = \frac{TP}{TP_{Max}}$
   3.8. $Specificity = \frac{TN}{TN_{Max}}$
   3.9. $Precision = \frac{TP}{TP+FP}$
   3.10. $Accuracy = \frac{(TP+TN)}{(TP_{Max}+TN_{Max})}$

# References


(1) Høye G et al (2013): "Pulse Descriptor Word (PDW) Simulator – Version 1.0", FFI-rapport 2013/00048

(2) Opland E (2012): "Problemstillinger, vurderinger og erfaringer I 1127 GEOIDE II i forbindelse med pulssortering", FFI-rapport 2012/01607 (Begrenset)

(3) Harel D (1987): "Algorithmics: The Spirit of Computing", Addison-Wesley Publishing Company