REAL-TIME SPEECH ENHANCEMENT USING MEDIAN FILTERS

Malcolm B. Jones

Signal Processing Research Centre
School of Electrical and Electronic Systems Engineering,
Queensland University of Technology

ABSTRACT - Considerations involved in using median filters in Digital Signal Processing systems for speech enhancement are discussed. A summary of algorithms available is presented together with measurements of the real-time performance on sample speech corrupted by impulsive noise.

INTRODUCTION

The properties and applications of median filters are well known (Nodes & Gallagher, 1982). A number of articles presenting largely theoretical discussions of the relative performance of the various algorithms for implementing median filters have been published (Juhola, 1991; Pasian 1988), but none to date has seriously discussed the practical considerations involved in using median filtering as a software subsystem within a typical DSP based real-time system. This article presents considerations involved in developing just such a system based around a Texas Instruments TMS320C30 floating point Digital Signal Processor.

MEDIAN FILTER OPERATION

The usual definition of a median filter is given as follows -

$$y[n] = median(x[j] \mid j = n-k, ..., n+k) \tag{1}$$

This yields a median filter window length of 2k+1(i.e. always odd) and note also that it is a non-causal system, as y[n] depends on inputs after the present time. In this discussion we use a slightly modified definition -

$$y'[n] = median(x[j] \mid j = n-m+1, ..., n) \tag{2}$$

Note: for m = 2k+1 (i.e. m odd) we have y'[n] = y[n-k], i.e. y'[n] is a delayed version of y[n], and also we have gained the ability to make m even (if the properties of an even window suit the application).

ALGORITHMS FOR IMPLEMENTING MEDIAN FILTERS

There are a number of algorithms available for implementing median filters in software, and each has its advantages and disadvantages. It has been shown (Juhola 1991) that the theoretically fastest method uses the histogram bin approach (actually a kind of hash table). One key feature of this method is that it relies on the assumption that the input comes from a finite integer domain. In this discussion we are concerned with the implementation of median filters as a general sub-system on a floating point processor. Hence, in the interest of maintaining the routine as general as possible, and considering the practicality of using large histogram tables (which consume large amounts of memory) in a development DSP system with limited memory, we don't consider the histogram method here. Presented here is a general overview of the algorithms considered, including discussions on the theoretical *average* complexity of each. In practice the average complexity does not always provide a good measure of performance, since in discrete-time systems it is usually required to maintain a constant sample rate. In this case, the worst case complexity is of more importance, as it provides an indication of the maximum time it will take to calculate a sample output, which in turn limits the minimum sample period (and hence maximum sample rate).

"BRUTE FORCE" METHODS

The algorithms in this section are grouped together because they don't use any redundancy of information between inputs to help optimize the number of operations, i.e. for each input, they form the array containing $(x[j] \mid j = n-m+1, ..., n)$ and then proceed to find the median of that array.

Sorting based Method

The sorting based method(s) uses sorting to put the window data into order, so the median can be extracted from the middle point(s) of the array. The algorithm is as follows-

```
for each input
    form array W containing $(x[j] \mid j = n - m + 1, ..., n)$
    sort(W)
    if m is odd
        $y[n] = W[(m+1)/2]$
    else
        $y[n] = (W[m/2] + W[m/2+1])/2$
```

Any sorting algorithm can be used, in this discussion we restrict ourselves to the well known QuickSort (Horowitz & Sahni, 1978) algorithm. The best sorting algorithms are $O(m.logm)$, so the overall execution on N inputs is $O(N.m.logm)$. In terms of memory usage, the memory required (other than simple local variables) is an array of size m to store the array W.

Median Search Method

The median search method uses the realisation, that actually there is no need to order all the data, it is only needed to find the kth largest (and k+1th if m is even) value. Thus Hoare's algorithm (Wirth, 1976) can be used to find the median. The algorithm then becomes -

```
for each input
    form array W containing $(x[j] \mid j = n - m + 1, ..., n)$
    if m is odd
        $y[n] = find((m+1)/2, W)$
    else
        $y[n] = (find((m/2), W) + find((m/2+1), W))/2$
```

Where *find(j, D)* uses Hoare's algorithm to find the *j*th largest element from array D. Since find is O(m) the overall theoretical complexity of this method (for N inputs) is O(N . m). As in the sorting method, the storage requirement is for one array of size m.

SMART SORTING APPROACHES

Between each input, the elements within the median filter "window" differ by only one element. It is possible to use this information to reduce the time spent finding the median by sorting or searching. This is achieved by maintaining the invariant in a data structure between successive applications of the median filter. In this way we trade off time spent sorting or searching against extra overhead in maintaining the data structure.

Double Heap Method

(Astola & Campbell, 1989) Showed that a structure called a double heap can be used to perform the median filter operation. The algorithm also requires that an array of pointers (called the time ring) be used to maintain the location of the oldest item in the heap(s). The algorithm used is as follows -

```
initialise data structure
for each input
    find oldest item in doubleHeap
    replace old item with new input
    y[n] = findmedian( doubleHeap )
```

Where *find oldest item in heap* is
use the time ring to find the location of the oldest element in the heap. (O(1)).

replace old item with new input is
substitute the new value for the oldest value and 'bubble' the new value to the correct location within the heap. (O(log m)).

findmedian is

return the root of the double heap (m odd) (O(1))

Thus the overall average complexity (ignoring for the moment the overhead of initialising the data structure) is O(N . log m). This is valid, since, in many applications the time required to initialise the data structure does not effect the continuous system operation. In those applications where the data structure initialisation cost does effect system performance, the complexity of the initialisation is O(m). The storage requirement for data is twice that of the sorting method, as we require m points for the heap, and an m element time ring, but the asymtopic complexity for storage is still O(m)

Delayed Sorting

If the history buffer is maintained in an initially sorted condition, it is realised that if the next input value is on the same side of the median as the old value it replaces, the median does not change. In this case it is not required to completely re-sort the array. This means the sort operation can be delayed until it is necessary, hence the total time spent sorting the array is reduced. The algorithm used is as follows -

```
Initialise sorted array W
for each input
    replace oldest item in W
    if new input is not on same side of median as the old value it replaces
        sort(W)
    end if
    if m is odd
        y[n] = W[(m+1)/2]
    else
        y[n] = ( W[m/2] + W[m/2+1] ) /2
```

In order to maintain the history list correctly, an extra field is added to the elements of W. This field forms a chain of pointers that can be used to maintain the correct history list. The theoretical average case asymtopic complexity of this method is still O(N . m . log m). This is the same as the sorting methods, but since it is expected to only require to sort on average less than 50% of the time, the coefficients in the complexity equation are expected to be smaller for this method. The trade-off in using this method is that since the elements of the array are twice as large, each element swap takes extra time. The storage requirements are again 2m locations (asymtopic O(m)).

Hybrid Sort Method

A different form of sorting that offers better performance Is considered. When a value is replaced in an already sorted array, only the new value violates the ordered property of the array. Hence it is not needed to completely sort the whole array each time a value is replaced, it is sufficient to "bubble" each new value to its correct location in the array W. This is the basis for the hybrid sort approach. The algorithm is -

```
Initialise sorted array W
for each input
    replace oldest item in W
    "bubble" new item to its correct location
    if m is odd
        y[n] = W[(m+1)/2]
    else
        y[n] = ( W[m/2] + W[m/2+1] ) / 2
```

The "bubble" operation take on average m/2 comparisons and element swaps. This yields an overall complexity for this method of O(N . m). This algorithm is effectively a cross between quick sort and insertion sort (if you consider the operations involved in filtering a sequence of input values), hence the name "Hybrid Sort". The algorithm also requires an extra field be added to the elements, so the history list can be maintained (O(m) storage).

Finger Trees

Binary search trees offer the possibility of similar performance to the heap method (O(N . log m)). In order to gain this performance, however, the tree must be maintained in a balanced state. The additional overhead involved in maintaining this balanced tree state can be removed if we let the tree degenerate into just the leaf nodes (i.e. a double linked list). This weakens the theoretical complexity to O(N . m), but in practice if a "median finger" is maintained (a pointer to the current median in the list), and the knowledge that most times, each new input value will be close to the previous input value, we gain a method that works well in practice. In fact using this degenerated tree data structure yields an algorithm very similar to the Hybrid Sort method. In the comparison of methods, the main differences between the implementation of Finger Trees and Hybrid Sort is that the Finger Tree method uses a linked data structure rather than an array (and hence gains overheads associated with the pointer based data structure). The Finger Tree method has the ability to start the search for the location to insert the new item at positions other than the position of the old value it replaces. These differences have the effect that the element swaps do not need to be performed when "bubbling" the new value to its correct position, the number of comparisons required can be reduced (on average), and if the operation is performed optimally, only one set of pointers need be updated.

COMPARISON OF PERFORMANCE DATA

Each of the above algorithms was coded in C and compiled with the Texas Instruments optimising C compiler (for the TMS320C3x/C4x) with all optimisations enabled. The resulting function was incorporated into a program to run on a TI TMS320C30 EVM card running at 30MHz. The input data to the program came from 15 seconds of sampled speech corrupted by random impulsive noise. The host (a 80486 Personal Computer), was used to transfer the speech data to the TMS320C30 for processing, with the output filtered speech available on the EVM D/A converter output in real time. The host was also used to collect the execution times (derived from the second TMS320C30 in-built timer peripheral). Figures 1 shows the time required to process 2048 samples averaged over the 15 second data file, for median filter orders of 1 to 20. This figure clearly indicates that the sorting method, median search method, and delayed sorting method are each (to varying degrees) inferior to either of the remaining three methods (hybrid sort, finger tree, and double heap). An interesting feature shown in this figure is the extra overhead required to find the second value to calculate the median in the case of even window lengths for the median search method (curves B1 and B2). In this figure the hybrid sort method appears to be the best. For most speech applications, where a median filter order of 20 would be considered very large, the performance suggests that this is the case. This is, however, only true if the average execution time is the important performance consideration, if a smaller sample buffer was being used, it would be expected that the worst case performance would limit the system much more.

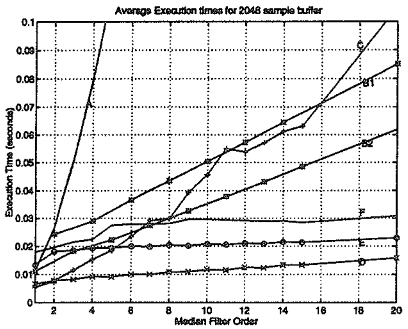

Figure 1: (A) - quicksort ; (B1,B2) - median search ; (C) delayed sorting ; (D) - hybrid sort ; (E) - finger tree ; (F) - double heap

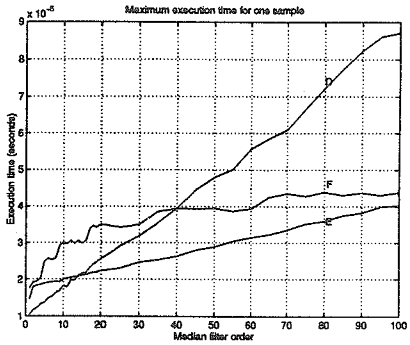

Figure 2: (D) - hybrid sort ; (E) - finger tree ; (F) - double heap

Figure 2 shows the worst case execution time for the three best algorithms, and we can see that the measured data exhibits behavior very close to the expected O(N.m), O(N.m) and O(N.log m) for the three methods hybrid sort, finger tree and double heap respectively. It is interesting to note that even though O(N.m) the finger tree displays the lowest execution time for the window sizes tested. It would be expected that in the two dimensional case, (which is not considered here), the O(N.log m) complexity of the double heap method would reap the most benefit, as in the two dimensional case, the effective filter order is much larger.

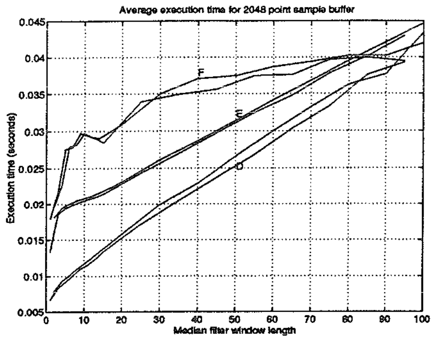

Figure 3: (D) - hybrid sort ; (E) - finger tree ; (F) - double heap

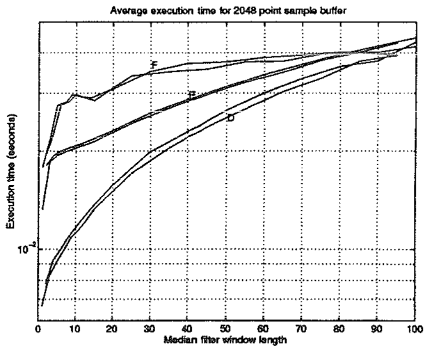

Figure 4: (D) - hybrid sort ; (E) - finger tree ; (F) - double heap

Figures 3 and 4 show the asymtopic behavior, of the three best algorithms with linear and logarithmic time axes respectively. Each algorithm has a separate curve for even and odd filter orders, to demonstrate the extra overhead associated with calculating the average of two values in the even filter order case. These graphs clearly show that for large m, curves D and E (corresponding to hybrid sort and finger tree), as expected, approach the same asymtope. These graphs also show that as m increases beyond 100, the double heap method begins to display its theoretical advantage in the real performance data.

CONCLUSIONS

The measured performance data clearly shows the advantages of using the more sophisticated algorithms for implementing median filters. In practice, although theoretically "weaker" than the double heap method, the hybrid sort and finger tree algorithms showed the best performance in speech enhancement applications. Which of the two algorithms (hyb. sort, fing. tree) best suits a particular application depends on the style of implementation. If the median filter is being used on a sample by sample basis, the maximum execution time to calculate one output is likely to be the most important characteristic, so the finger tree would probably be the algorithm of choice - particularly if filter orders of larger than approximately 15 samples are to be used. On the other hand, if the application is such that the average execution time is more important, then the hybrid sort approach would probably be the best choice. In the case of speech enhancement, large median filter orders are generally not used, however there may be other applications (such as image enhancement) that may benefit from the logarithmic complexity of the double heap method.

REFERENCES

Astola T., & Campbell T. (1989) *On Computation of the Running Median*, IEEE Trans. Acoust., Speech, Signal Processing, vol 37, 572-574.

Horowitze E., & Sahni S. (1978) *Fundamentals of Computer Algorithms*, pp 121-127, (Pitman Publishing: London).

Juhola M., Katajainen J. & Raita T. (1991) *Comparaison of Algorithms for Standard Median Filtering*, IEEE Trans. Sig. Proc., vol 39, 204-208.

Nodes T., & Gallagher N. (1982) *Median Filters: Some Modifications and Their Properties*, IEEE Trans. Acoust., Speech, Signal Processing, vol 30, 739-746.

Pasian F. (1988) *Sorting Algorithms for Filters Based on Ordered Statistics: Performance Considerations*, Signal Processing, vol 14, 287-293.

Wirth N. (1976) *Algorithms + Data Structures = Programs*, pp 82-84, (Prentice Hall: Englewood Cliffs, New Jersey).

SPEAKER CHANGE DETECTION

A. Satriawan and J.B. Millar

Computer Sciences Laboratory
Research School of Information Sciences and Engineering
The Australian National University

ABSTRACT - An approach to speaker change detection based on speaker discontinuity models is described. It builds on the limited evidence of speaker characteristics in individual phone segments to enable a rapid response when a speaker change is detectable. A baseline system based on a speaker identification task is built and tested on the TIMIT corpus.

INTRODUCTION

Automatic speaker monitoring (ASM), defined as automatic techniques to cope with potentially multi-speaker environments, encompasses a number of related spoken language processing techniques. ASM is concerned with such issues as modelling the number of speakers co-talking, the existence of a dominant speaker, the continuity or discontinuity of speaker identity, and speaker identity itself. There are several important practical applications of ASM : speaker assignment for conversational speech in an interview or telephone conversation, quick identification of speakers in senatorial debate in parliament, assignment of speaker in pilot/co-pilot conversation with tower control (Gish, Siu, and Rohlicek, 1991), detection of an intruding imposter (Schalkwyk *et.al*, 1994), speaker recognition in noisy environment, and better segregation of target speech from background speech.

This paper will present our initial work in the study of ASM. It concentrates on the speaker change detection (SCD) problem which is based on models of speaker discontinuity. The next section introduces a framework for ASM based on a speaker recognition paradigm. The following sections describe our SCD approach and the result of our preliminary experiments. The last section presents our conclusions.

A TAXONOMY OF AUTOMATIC SPEAKER MONITORING

The ASM problem can be regarded as a standard pattern recognition problem. Its solution in this paradigm involves two main phases : the training phase and the testing phase. In the training phase, the characteristics of a speaker are extracted from training utterances and stored as templates for reference purposes. In the testing phase, an input signal which consists of mixture of several speech utterances from a multi-speaker environment will be recognised in terms of its constituent speakers, the number of speakers, and the speaker at certain time. An input signal of this kind will be called a multi-speaker utterance (MSU).

The nature of the speaker population defines four possible task classes of varying complexity : The *speaker population* participating in the training and testing phases may be a *closed* or an *open set* and the *number of speakers* participating in the MSU may be *known* or *unknown*.

In a closed-set task, all possible speakers in the MSU are in the training database. A closed-set task can then be subdivided based on the knowledge of the number of speakers in the MSU. The number of speakers in the utterance may or may not be known beforehand. In the first case, the speakers may be known, and the task is only to detect which speakers are speaking at a certain time. The second case is more difficult since beside the need to recognise the speakers, the number of speakers in the MSU also needs to be detected.

In an open-set task, whether all speakers in the MSU are in the training database is unknown. Some speakers may be in the database but some may not be. The problem becomes more complicated if the number of speakers is unknown. However, even in the case of known number of speakers the problem is still difficult. The open-set task with unknown number of speakers is the most difficult one.

Within the above structures, the problem can also be divided into two cases based on the *degree of speech co-occurrence* within the MSU : *sequential speakers*, when there is no co-occurrence of speech from different speakers, and *concurrent speakers*, when more than one speaker is speaking at the same time.