Median Filtering

Ziv Yaniv

School of Engineering and Computer Science
The Hebrew University, Jerusalem, Israel.

In this summery we will discuss several algorithms for computing the median filter and related order filters. The summery is implementation oriented.

1 Introduction

The ith order statistic of a set of n elements is defined as the ith smallest element. Assuming n is odd the median is the $(n+1)/2$ smallest element. Median filtering is done by replacing the value of each element by the median found in a window around the element. Why is this beneficial?

This question leads us to the issue of *robust estimation*. When estimating a location value we would like our estimate to be robust. An indicator of an estimator's robustness is its *breakdown point*. The breakdown point is defined as the smallest fraction of the observations which when replaced with outliers will throw the estimator outside of reasonable bounds. For example the breakdown point of the mean is 1. More formally [5]: given n data points $Z = \{\mathbf{x_1} \ldots \mathbf{x_n}\}$, an estimator $T(Z)$, and considering all possible corrupted samples Z', samples in which m of the original points have been replaced by arbitrary values. The estimators *breakdown point* is:

$$\min_{m} \left(\sup_{Z'} \|T(Z') - T(Z)\| \rightarrow \infty \right)$$

Example:
Given the following observations $\{0.2, 0.2, 0.2, 0.3, 0.3, 0.3, 300\}$. The mean is $\simeq 43$ which obviously does not describe the data correctly.

The median on the other hand has a breakdown point of 0.5, according to its definition. This is exactly the reason to perform median filtering. Assuming we have data contaminated with noise, outlying observations, we want to remove the noise while retaining most of the behavior present in the original data. The median filter does just that and does not introduce values that are not present in the original data.

```
randomizedSelect(data, left, right, i)
1 if(left==right)
2   return(data[left])
3 pivot = partition(data,left,right);
4 if(pivot < i)
5   return(randomizedSelect(data,pivot+1,right,i));
6 else
7   return(randomizedSelect(data,left, pivot-1, i));
```

Figure 1: Randomized Select

Example:
Given the following observations $\{0.2, 0.2, 0.2, 0.3, 0.3, 0.3, 300, 0.3\}$.
Applying a mean filter with window size 3 gives the following series:

$$\{x, 0.2, 0.23, 0.26, 0.3, 100.2, 100.2, x\}$$

Applying the median to the series yields:

$$\{x, 0.2, 0.2, 0.3, 0.3, 0.3, 0.3, x\}$$

where x denotes an uncomputed value.

From the example it is apparent that the median filter is preferable to the mean filter. Given this motivation we would like to apply median filtering in the most efficient way possible. This is the main subject of this lecture.

2 Algorithms

As my concern is mainly with 2D data, images, the median is always computed within a window of size n where $n = (2p + 1)^2$. Algorithm evaluation is on the basis of cost per element.

2.1 Naive Algorithm [1]

The most naive approach to computing an order statistic is to sort the data and then select the ith smallest value. Once the data is sorted finding any order statistic is an operation with cost $O(1)$. Sorting the data leads to a computation time of $O(n \log n)$ in general and a cost of $O(n)$ when dealing with discrete data. Thus the cost of this approach is the cost of sorting the data.

[1] Always implement the naive approach as it is probably the easiest to code and serves as a gold standard for more complex approaches.

```
pixSort(a,b)
1. if(a>b)
2.   swap(a,b)

sort3(data)
1 pixSort(data[0],data[1]);
2 pixSort(data[1],data[2]);
3 pixSort(data[0],data[1];
```

Figure 2: Minimal Sort Network

2.2 Randomized Select

This algorithm is due to C.A.R. Hoare and is similar to the *quick sort* algorithm [1] which is also due to him. The important observation is that the ith order statistic divides the data in two, elements which are greater and elements less than it. The algorithm randomly selects a pivot and partitions the data in two around this pivot. If the pivot is the ith element of the data then we are done, otherwise recurse on the sub array which contains the ith element. The algorithm pseudo code is given in Figure 1.

This approach is similar to quick sort but the recursion is done only on one of the sub arrays and not on both. The worst case for this algorithm is $\theta(n^2)$ and occurs when each partition creates two subarrays of size 1 and $n-1$ respectively. This rarely occurs as we choose the pivot randomly. The average case is $O(n)$ expected time. For detailed proofs see [1].

An algorithm with worst case $O(n)$ is also described in [1].

2.3 Minimal Sort Network

For small window sizes $n \leq 25$ it is possible write the minimal sort network. This allows for linear data flow without branching (ifs) or procedure calls. Figure 2 describes this approach for three elements. After the sorting stage any order statistic is immediately available.

2.4 Median of Medians

This approach is due to [4] and is an approximation of the median filter. If we apply a two dimensional filter in the spatial domain we can accelerate its performance if it is separable. This is a common approach where instead of applying a two dimensional filter we can apply two one dimensional filters. In [4] this approach is applied to median filtering, taking the median of medians. This does not yield the median in most cases, but has similar effects on the data, eliminating outliers, with lower computational costs.

Given a window of size n where $n = (2p + 1)^2$ the algorithm proceeds as follows:

1. Perform median filtering on the rows with a window size of $2p + 1$.
2. Perform median filtering on the columns from the previous step with a window size of $2p + 1$.

The computational complexity of this algorithm depends on the median computation for the rows and columns. We can achieve a constant cost computation using the running median histograms described in 2.5.1. Applying this approach when filtering the rows requires two histogram updates per element, without regard to the window size. The columns are equivalent thus giving an overall cost of $O(1)$ per element.

2.5 Running Medians

Computation of a median filter of a long series of values can benefit from the observation that most values in the running window do not change when the window is translated. The following two subsections describe algorithms which utilize this observation yielding lower computation cost per element. The first algorithm is applicable to continues data while the second is only applicable to discrete data with a finite set of values.

2.5.1 Running Histogram

I am not sure if the algorithm I describe here is due to [3] as I could not obtain this paper. The idea seems straight forward so I will attribute it to [3] anyway. The algorithm is limited to discrete data with a finite set of values, not surprisingly most images fit these restrictions.

If our data fits these restrictions then we can utilize histograms to compute a running median. This reason being that the median computation takes into account only the element values and not their location in the sliding window. Simply maintaining a valid histogram at each location of our sliding window will enable finding the median with a cost of $O(1)$.

Given a window of size n where $n = (2p + 1)^2$ the algorithm proceeds as follows:

1. Initialize a histogram at location (0,0), a cost of $(2p + 1)^2$. Find the median from the histogram, a cost of $O(1)$.
2. Traverse the image and update the histogram, a cost of $2 * (2p + 1)$. Find the median from the histogram, a cost of $O(1)$.

The image is traversed from top left to bottom right as shown in Figure 3. This approach gives us a cost of $O(\sqrt{n})$ per element.

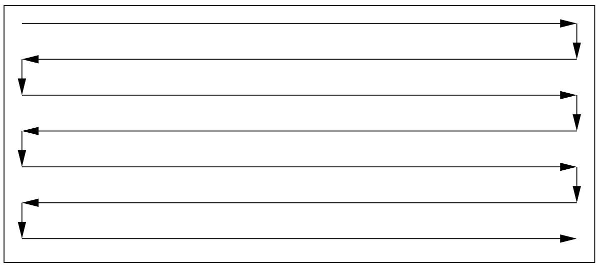

Figure 3: Image traversal path for histogram based running median.

2.5.2 Partition Heaps

This algorithm is due to [2]. Given a window of size n where $n = (2p + 1)^2$ the algorithm maintains data structures such that the cost of finding the median per element is $O(\sqrt{n} \log n)$. The main data structure the authors propose is the partition heap. A partition heap is a tree with the median found at the root. The left child of the root is a max heap containing all values less than the median and the right child is a min heap containing all values greater than the median. Figure 4 is an example of a seven element partition heap. When the current window is translated we have to update $\sqrt{n}$ elements. The data structures used to maintain and update the partition heap appear in Figure 5, the current data window, indexes from the window location into the heap and from the heap to the data window.

The most important aspect of this data structure is the insertion and removal of data from the partition heap. Element removal and insertion are an atomic action. There are seven options for update (all operations maintain the heap structures):

1. *minOutMinIn* - Inserted and removed element both come from the min heap.
   (a) Place the inserted element at the location of the removed element.
   (b) Push the new value towards the leaves as far as possible.
   (c) Push the new value towards the root as far as possible.
2. *minOutMaxIn* - Inserted element goes into max heap, removed element comes from min heap.
   (a) Place the inserted element at the location of the removed element and push it to the min heap root.
   (b) Set the root of the min heap to be the previous median.
   (c) Push the new value from the median position towards the leaves of the max heap as far as possible.
3. *maxOutMaxIn* - Same as *minOutMinIn*.
4. *maxOutMinIn* - Same as *minOutMaxIn*.

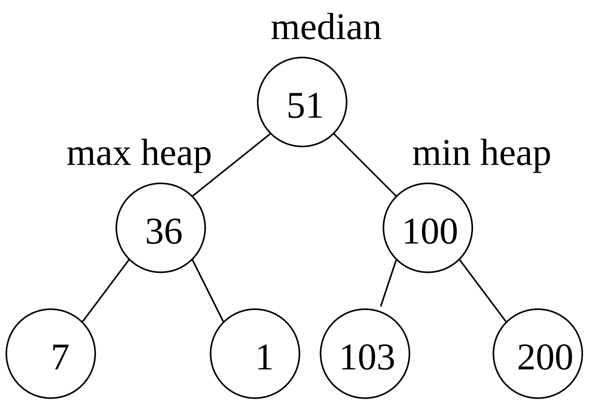

Figure 4: Example of a partition heap.

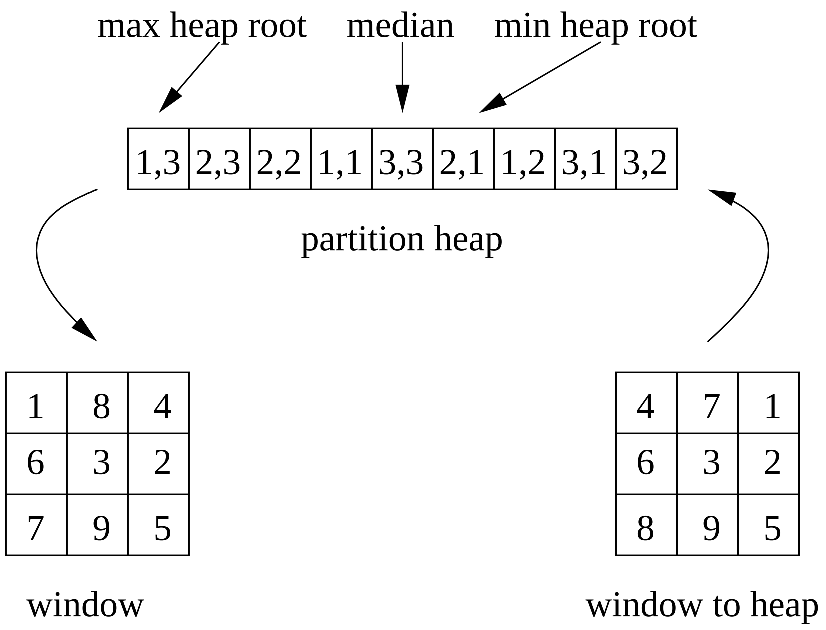

Figure 5: Data structures used when computing running median with a partition heap.

5. *medianOutMinIn* - Inserted element goes into min heap, removed element is the median.
    (a) Swap the new value from the median position and the root of the min heap.
    (b) Push the new value towards the leaves of the min heap as far as possible.
6. *medianOutMaxIn* - Same as *medianOutMinIn*.
7. *medianOutMedianIn* - do nothing.

3 Conclusions and Caveats

This summary has described several median filtering techniques. What we want is fast execution times for a specific window/kernel size. From empirical experience: (a) When using small sized kernels apply the minimal sort network scheme. (b) When using large sized kernels apply the running median scheme.

A caveat of median filtering is its effect on corners. If you require corner detection be aware that the median filter will round existing corners. Figure 6 is an example of applying a median filter which does just that.

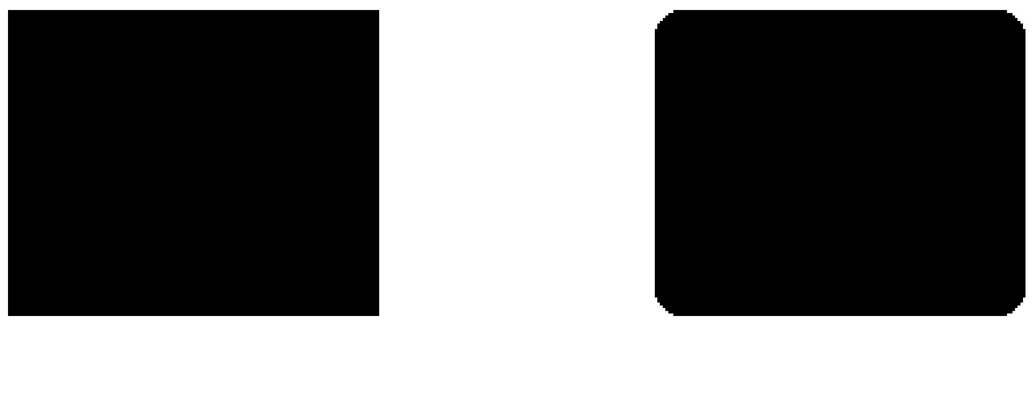

Figure 6: (a) before median filtering. (b) after applying a median filter of size 7×7 the square's corners have been rounded.

This summary has only dealt with the computation of the one dimensional median. For an overview of multi-variate medians, definitions and computational issues see [6].

A Resources and Code

The following resources and implementations are available on the web:

1. Median Filtering (my implementations of most of these algorithms).
   http://www.cs.huji.ac.il/˜zivy/#software
2. Fast Median Search: an ANSI C implementation
   http://ndevilla.free.fr/median/
3. CVonline: Image Transformations and Filters (This is a compendium of filter related links, includes a link to the previous entry).
   http://www.dai.ed.ac.uk/CVonline/transf.htm

References

[1] Cormen T.H., Leisorson C.E., Rivest R.L., “Introduction to Algorithms”, pp. 185–191, MIT Press, 1990.

[2] Hardle W., Steiger W., “Optimal Median Smoothing”, 1994.

[3] Huang T.S., Yang G.J., Tang G.Y “A fast two-dimensional median filtering algorithm”, IEEE transactions on acoustics, speech and signal processing, Vol. 27(1), 1979.

[4] Narendra P.M., “A Separable Median Filter for Image Noise Smoothing”, IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI), Vol. 3(1), pp.20–29, 1981.

[5] Rousseeuw P.J., Leroy A. M., Robust Regression & Outlier Detection, Wiley, 1987.

[6] Small C.G., “A Survey of Multidimensional Medians”, International Statistical Review 58, p.263, 1990.