Fast, efficient algorithms for 3x3 ranked filters using finite-state machines

Frederick M. Waltz\textsuperscript{a}, Ralf Hack\textsuperscript{b}, and Bruce G. Batchelor\textsuperscript{b}

\textsuperscript{a} 2095 Delaware Avenue, Mendota Heights, MN 55118-4801 USA
\textsuperscript{b} Department of Computer Science, University of Wales Cardiff, Cardiff, Wales, UK

\textbf{ABSTRACT}

Median filters and ranked filters of ranks other than median have often been proposed or used to remove image noise as well as for other reasons. These are nonlinear operations, and often have relative long execution times, making them unsatisfactory for many speed-critical industrial applications.

This paper builds on the earlier work of Mahmoodi and Waltz\textsuperscript{1} to provide efficient implementations of 3x3 ranked filters of ranks 1 (minimum), 2, 3, 4, 5 (median), 6, 7, 8, and 9 (maximum). These implementations are based on a partial realization of the SKIPSM (Separated-Kernel Image Processing using finite-State Machines) paradigm. A full SKIPSM realization is not possible because, except for the filters of ranks 1 and 9, these operations are not separable. This paper shows that, in spite of this lack of separability, the finite-state machine aspect of SKIPSM can be used to advantage. The emphasis is on software implementations, but implementations in pipelined hardware have also been demonstrated.

In addition, a fast “full-SKIPSM” implementation of a slightly different ranked filter, sometimes called the “separable median” filter, is presented. This filter guarantees that the output pixels are of rank 4, 5, or 6. For typical noise-reduction applications, it is difficult to find a convincing argument that this filter is inferior in any meaningful way to the true median filter.

\textbf{Keywords:} median filter, ranked filters, separable median filter, separation, finite-state machines

\section{1. INTRODUCTION}

There are three commonly-used measures of central tendency: the mean, the mode, and the median. The ordinary mean, or uniformly-weighted average, is an easily-computed linear operation involving only addition, scaling, and rounding. The weighted average, usually called a convolution, is a variation of this. Both are sensitive to extreme values, and hence may give poor results in situations with “shot” noise or “salt-and-pepper” noise.

The mode, or most commonly observed value, requires even more computation than the median and often fails to give a unique value. For example, the set \{1, 1, 4, 5, 6, 9, 9, 9\} has both mean and median equal to 5, but it is “bi-modal” – it has two modes, 1 and 9. The set \{1, 1, 5, 5, 5, 9, 9, 9\} has both mean and median equal to 5, but is tri-modal. For normal distributions on large sets, the mean, mode, and median give essentially the same value. However, the mode is generally erratic on small sets (the kind most commonly used in image processing), and should usually be avoided.

The median, or middle value of an ordered set of values, has the great advantage of being totally unaffected by one or a few extreme values. It has the disadvantage of requiring considerably more computation than the mean. Median filters and ranked filters of ranks other than the median have often been proposed or used to remove image noise as well as for other reasons. These are nonlinear operations, and often have relative long execution times, making them unsatisfactory for many speed-critical industrial applications. Therefore, if a faster method of computing the median filter and other ranked filters can be developed, these filters can be expected to be used in a wider range of applications.

This paper builds on the earlier work of Mahmoodi and Waltz\textsuperscript{1} to provide efficient implementations of 3x3 ranked filters of ranks 1 (minimum), 2, 3, 4, 5 (median), 6, 7, 8, and 9 (maximum). For ranks 2 through 8, these implementations are based on a partial realization of the SKIPSM (Separated-Kernel Image Processing using finite-State Machines) paradigm. Detailed discussions of SKIPSM have been presented in numerous previous works\textsuperscript{2-21}, and will not be repeated here, except to say that extreme speedups of many 2-D and 3-D neighborhood image-processing operations have been obtained, and significant speedups have been achieved for many other operations.

A full SKIPSM realization is not possible for ranked filters because these operations are not separable, except for the filters of ranks 1 and 9. Separability in this context is not limited to the familiar separability of a subset of the linear-convolution kernels. Instead, it has been shown\textsuperscript{2-21} that a wide range of 2-D operations can be separated into two 1-D operations – a row operation followed by a column operation, or a column operation followed by a row operation, or two orthogonal diagonal
operations. The speedups provided by SKIPSM result primarily from this separation, as well as the pipelining of the operation and the use of recursive finite-state machines.

To see why the median operation is not one of those which can be separated, let us first consider a few operations which are separable:

- Addition or, equivalently, averaging – If we divide a set of numbers to be added into subsets, and add the members of the subset separately, then the overall sum is the sum of the subset sums. For averaging, the same conclusion holds if weighting is taken into account.
- Maximum (rank 9) – The overall maximum is the maximum of the subgroup maximums.
- Minimum (rank 1) – The overall minimum is the minimum of the subgroup minimums.
- Binary erosion – If the various parts of the structuring element “fit” the corresponding parts of the image, then the overall structuring element “fits” the image.

Now consider the median of a set of numbers: The overall median is, in general, not the median of the subset medians. Here is a counter-example: Let the subsets be \([2, 2, 1], [4, 8, 9],\) and \([5, 8, 9]\). The subset medians are 2, 8, and 8. The median of these numbers is 8. But the median of the overall set \([1, 2, 2, 4, 5, 8, 8, 9, 9]\) is 5. Similar conclusions hold for ranks between 2 and 8 (of 9-element sets). Separability rules for arbitrary operators are given in 1.

This paper shows that, in spite of this lack of separability, the FSM (finite-state machine) aspect of SKIPSM can be used to advantage for ranked filters on 3x3 neighborhoods. In particular, this approach retains the pipelined aspects – each input pixel is fetched once, and recursive finite state machines are used for the row and column operations – even though the column-machine state is more complex and leaves more computations to be performed each time than has been the case for most other SKIPSM applications.

In addition, a fast “full-SKIPSM” implementation of a slightly different ranked filter, sometimes called “the separable median filter,” is presented. This filter guarantees that the output pixels are of rank 4, 5, or 6. For typical noise-reduction applications, it is difficult to find a convincing argument that this filter is inferior in any meaningful way to the true median filter.

2. A Preliminary Case – the “Separable Median” Filter

This is precisely the filter used, two paragraphs prior to this one, to prove that the true median filter is not separable. In spite of those pathological examples, which can, of course, happen sometimes in actual practice, this is a useful operation, guaranteeing that the output value will be of rank 4, 5, or 6.

From a philosophical or theoretical standpoint, there is seldom a justification for insisting on one particular operation in preference to others. There are of course exceptions, such as in the use of Fourier or wavelet transforms, so that the powerful body of theoretical results associated with these operations can be applied. But the typical use of median or other ranked filters is for “cleaning up” images subject to “dropouts,” “snow,” “salt and pepper” noise, etc. For these applications, it is difficult to find a convincing argument that a rank 4-5-6 filter is significantly inferior in any meaningful way to the true median filter in most cases. Thus, where speed of operation is at all important, this “nifty” little filter – just eleven program steps in the main loop – should at least be considered.

Figure 1 shows a functional diagram for a “separable median” filter, in which the row medians (the medians of the three pixel values in the three rows of the 3x3 neighborhood) are formed first, and then the median of these is taken as the output. At this stage, the algorithm has not yet been put in pipelined form. The only sequential relationship here is that the operations in the top row must be performed before the lower operation can be carried out. The rectangular boxes in this diagram are 3-element sorters, the middle output of which is the median of the three inputs. Also shown is a way of realizing 3-element sorter using three 2-element (min-max) sorters.

Figure 2 shows the same algorithm in pipelined form - each pixel is fetched only once, and all information necessary to compute the output is carried along as the “state” of a finite-state machine (FSM). Two FSMs are used here, a row machine and a column machine. Two registers, RS0 and RS1, are needed to...
store the state of the row machine. These must be set to zero at the beginning of each row. Two column state buffers, CS0[i] and CS1[i], each with a number of addresses equal to the number of pixels in an image row, must also be established. These must be set to zero at the start of the overall operation. “Input” is the input image, and “Output” is the output image. If desired, the output may be written back into the input buffer because of the “pipelined” nature of the algorithm.

Because the output is written to an address one row above and one pixel to the left of the current pixel, to compensate for pipeline latency, the first row of the image must be handled as a special case, to avoid writing to an area outside the image buffer. For the same reason, the first pixel in each row must be handled as a special case. One way to handle these cases is to execute all the steps of the algorithm, but for the first row and column simply omit writing the output. To save time, these should be done as a preliminary step or loop, rather than by inserting tests in the main loop. In fact, these can easily be done as part of the initialization of the state buffers.

Sample main loop code is given below. To conserve space in this paper, code conventions used for appearance (as opposed to functionality) are omitted; e.g., indenting, extra carriage returns, and unnecessary “curly brackets.”

```
1. \textbf{// Row machine for “separable median” filter}
2. \textbf{if} (RS1 > RS0) \{ Min = RS0; Med = RS1; \} // Compare “k” with “l"
3. \textbf{else} \{ Min = RS1; Med = RS0; \} \textbf{// “Med” is a temporary variable}
4. RS1 = RS0; \textbf{// Update row state register RS1}
5. RS0 = Input[j][i]; \textbf{// Fetch the next pixel from the input image}
6. \textbf{if} (RS0 < Med) Med = RS0; \textbf{// “Med” is a temporary variable}
7. \textbf{if} (Min > Med) Med = Min; \textbf{// “Med” is the row machine output}
8. \textbf{// Column machine for “separable median” filter}
9. \textbf{if} (CS1[i] > CS0[i]) \{ Min = CS0[i]; Out = CS1[i]; \}
10. \textbf{else} \{ Min = CS1[i]; Out = CS0[i]; \} \textbf{// “Out” is a temporary variable}
11. CS1[i] = CS0[i]; \textbf{// Update column state register CS1[i]}
12. CS0[i] = Med; \textbf{// Save the row machine output in CS0[i]}
13. \textbf{if} (CS0[i] < Out) Out = CS0[i]; \textbf{// “Out” is a temporary variable}
14. \textbf{if} (Min > Out) Out = Min; \textbf{// “Out” is the overall filter output}
15. \textbf{Output}[j-1][i-1] = Out \textbf{// Write output, compensating for latency}
```

Note that, once state buffer initializations have been completed, these eleven steps are executed once for each pixel in the image. No other steps are required. Note also that the variables “Med” and “Out” are used in such a way as to eliminate the “else” clause from four of the six “if” statements (steps 4, 5, 9, and 10 in the figure), so as to save execution time.

It is interesting to compare this 11-step 6-comparison SKIPSM “separable median” algorithm with non-pipelined implementations of the same operation: For each output value to be computed, one must fetch nine pixels from the image, sort these in a process requiring at numerous “compare-and-perhaps-interchange” operations, and then write the result into a different image buffer, so as not to destroy information needed for computing results for as-yet-unprocessed rows and columns. The well-known “bubble-sort” algorithm requires 22 comparison operations, in addition to 9 image reads and 1 write. The non-pipelined version of Mahmoodi’s algorithm requires 12 comparisons, plus 9 image reads and 1 write.

### 3. The True Median Filter on a 3x3 Neighborhood

Using Mahmoodi’s algorithm\(^1\), the (true) median of nine values can be computed by a “row” sort followed by a “column” sort, followed in turn by a “diagonal” sort, as shown in Figure 3 using the “sorting modules” defined in Figure 1. For comparison, a “bubble sorting” algorithm for the median is also included. Note that the center “column” sorter (indicated by...
an asterisk) produces precisely the “separable median” described above. This suggests that the true median can be obtained using an approach analogous to that used for the “separable median,” but with additional steps. Such an approach is described in this section.

Whereas a non-pipelined bubble sorter requires 30 comparison modules, Mahmoodi’s algorithm requires only 19. There may be median sorting algorithms more efficient than this one, but Mahmoodi’s was chosen because it lends itself so beautifully to pipelining and to the recursive SKIPSM approach. Note: The shaded sorters in Figure 3 are those eliminated due to the pipelining of the operation, resulting in only 13 comparison modules for the median calculation.

An aside: Mahmoodi’s algorithm was demonstrated about 10 years ago at the 3M Company using first-generation Datacube pipelined image processing boards. At that time, Datacube did not offer pipelined median filter modules, so ordinary neighborhood min-max (SNAP), convolver (VFIR), and pixel processing (MaxSP) boards were used. Mahmoodi’s algorithm “mapped” rather nicely onto this hardware, whereas a bubble sorter would have been highly impractical and expensive.

Figure 4 shows the same algorithm in pipelined form - each pixel is fetched only once, and all information necessary to compute the output is carried along as the “state” of a finite-state machine (FSM). Two FSMs are used here, a row machine and a column machine. As with the “separable median” filter, two registers, RS0 and RS1, are needed to store the state of the row machine. These must be set to zero at the beginning of each row. Six column state buffers, CS0[i] through CS5[i], each with a number of addresses equal to the number of pixels in an image row, must also be established. These must be set to zero at the start of the overall operation. “Input” is the input image, and “Output” is the output image. If desired, the output may be written back into the input buffer because of the “pipelined” nature of the algorithm. As in the previous case, the first row and column of the image must be handled as special cases.
Sample main loop code for the median filter is given below.

```c
// Row machine for median filter — “row sort”
1 if (RS1 > RS0) { Tmp = RS0; Med = RS1; } // Compare “k” with “l”
   else { Tmp = RS1; Med = RS0; } //“Med” is a temporary variable
2 RS1 = RS0; // Update row state register RS1
3 RS0 = Input[j][i]; //Fetch the next pixel from the input image
4 if (RS0 < Med) { Max = Med; Med = RS0; } // “Max” is the row maximum
   else Max = RS0;
5 if (Med < Tmp) { Min = Med; Med = Tmp; } // “Med” is the row median
   else Min = Tmp; // “Min” is the row minimum

// Column machine part 1 — finding MaxMin = the maximum of the minimums
6 if (CS5[i] > CS4[i]) MaxMin = CS5[i];
   else MaxMin = CS4[i]; //“MaxMin” is a temporary variable here
7 CS5[i] = CS4[i]; // Update column state register CS5[i]
8 CS4[i] = Min; // Save the row machine minimum in CS4[i]
9 if (CS4[i] > MaxMin) MaxMin = CS4[i]; //MaxMin = max of row minimums

// Column machine part 2 — finding MedMed = the median of the medians
10 if (CS3[i] > CS2[i]) { Tmp = CS2[i]; MedMed = CS3[i]; }
    else { Tmp = CS3[i]; MedMed = CS2[i]; } //“MedMed” is temporary here
11 CS3[i] = CS2[i]; // Update column state register CS3[i]
12 CS2[i] = Med; // Save the row machine median in CS2[i]
14 if (Tmp > MedMed) MedMed = Tmp; //“MedMed” = median of row medians

// Column machine part 3 — finding MinMax = the minimum of the maximums
15 if (CS1[i] < CS0[i]) MinMax = CS1[i];
   else MinMax = CS0[i]; //“MinMax” is a temporary variable here
16 CS1[i] = CS0[i]; // Update column state register CS1[i]
17 CS0[i] = Max; // Save the row machine maximum in CS0[i]
18 if (CS0[i] < MinMax) MinMax = CS0[i]; //MinMax = min of row maximums

// Column machine part 4 — finding the median of Parts 1, 2, & 3 above
19 if (MedMed > MaxMin) { Median = MedMed; Tmp = MaxMin; }
    else { Median = MaxMin; Tmp = MedMed; }
20 if (MinMax < Median) Median = MinMax;
21 if (Tmp > Median) Median = Tmp; //“Median” = desired filter output
22 Output[j-1][i-1] = Median; // Write output, compensating for latency
```

While this is obviously a lot more code than the “separable median” filter above (twice as many steps), it is still highly efficient, and produces the true median, for cases where this is considered to be important.

### 4. Filters of Rank 4 and 6 on a 3x3 Neighborhood

Using Waltz’s extension of Mahmoodi’s algorithm, filters of ranks 1, 2, 3, 4, 6, 7, 8, and 9 can be computed. Each of these take fewer steps than the median filter. In general, the closer the rank is to the median, the more steps are required. In this section, filters for ranks 4 and 6 are presented. Figure 5 shows the static diagram for these. Note the left-right symmetry of the diagrams. Comparison of these diagrams with Figure 3 shows that these differ from the median filter only in the last few steps. Again, the shaded sorters are those eliminated due to the pipelining of the operation. Figure 6 shows the last three steps of the corresponding programming diagrams and main loop code. The earlier steps (1 through 18) are the same as for the median filter.
4. Filters of Rank 2, 3, 7, and 8 on a 3x3 Neighborhood

Static diagrams for the remaining 3x3 ranked filters, again using Waltz’s extension of Mahmoodi’s algorithm, are shown in Figure 7. As above, the shaded sorters are those eliminated due to the pipelining of the operation. The implementations of these are similar to but simpler than those already described above. The main loop code for Rank 1 is given below, to show just how simple a ranked filter algorithm can be. The Rank 1 filter is of course just the neighborhood minimum, which can be obtained easily in various ways.

```
// Row machine for minimum (Rank 1) filter — obtaining the row minimum
1 if (RS0 < RS1) Min = RS0;
2 else Min = RS1; // "Min" is a temporary variable here
3 RS1 = RS0; // Update row state register RS1
4 RS0 = Input[j][i]; // Fetch the next pixel from the input image
5 if (RS0 < Min) Min = RS0; // "Min" is the row minimum

// Column machine — finding Rank1 = the minimum of the row minimums
6 if (MaxMin < MinMax) Rank1 = MaxMin;
7 else Rank1 = MinMax; // "Rank1" is a temporary variable here
8 CS1[i] = CS0[i]; // Update column state register CS1[i]
9 CS0[i] = Min; // Save the row machine output in CS0[i]
10 if (Min < Rank1) Rank1 = Min; // "Rank1" = neighborhood minimum
11 Output[j-1][i-1] = Rank1; // Write output, compensating for latency
```

```
// Column machine part 4, for Rank 4
19 if (MedMed < MinMax) Rank4 = MedMed;
20 else Rank4 = MinMax;
21 Output[j-1][i-1] = Rank4;

// Column machine part 4, for Rank 6
19 if (MedMed > MaxMin) Rank6 = MedMed;
20 else Rank6 = MaxMin;
21 Output[j-1][i-1] = Rank6;
```

5. Other Related Filters

Measures of variation about the central value are useful in locating image regions where pixel grey levels are changing rapidly, such as edges (high variation), or nearly monotone regions, such as the background or the faces of an object (low or zero variation). In general technical work, the most commonly used measures of this type are the variance and its square root, the standard deviation. Even more than the mean that they are usually used with, they are very sensitive to extreme values (i.e., noise). They also require a lot of computation, and tend to be erratic on small sets. Thus, they are less likely to be used in image processing.

A widely-used alternative, especially suitable for small sets, is the range. This can be either the neighborhood maximum minus the neighborhood minimum (Rank9 – Rank 1), or some other measure of spread such as (Rank8 – Rank2). Static (non-pipelined) diagrams for both of these are shown in Figure 8. Comparisons of these figures with previous static diagrams suggests that these can be computed similarly to the other filters described above, using pipelining and row and column
machines. Sample main loop code for the (Rank9 – Rank1) filter is given below.

With relatively few additional steps (compared to the median), it is even possible to compute all nine ranked values simultaneously, as shown in Figure 9. From these, any desired combination of ranked values can be obtained. This ranked set of values could then be used to find the mode, or to determine if the set is multi-modal (a good indication of a sharp edge), or to compute a specified weighted sum of the rank values - an operation available in the PIP image processing environment and its predecessors, and included there because it was found to be useful (although slow). The SKIPSM approach should therefore be able to speed up that operation to some degree also.

**6. Larger Neighborhoods**

In general, the complexity of median calculations and other sorting operations increases as the square of the number of values. For example, the number of compare-and-perhaps-switch modules of the type used here required by an all-ranks bubble sorter is \( N(N-1)/2 \), where \( N \) is the number of values to be sorted. For the odd-\( N \) bubble-sort median filter, the

---

**Figure 8.** Two functional diagrams for static computation of the range of pixel values.

**Figure 9.** Functional diagrams for static computation of all nine ranks, plus a weighted ranked filter
The corresponding formula is $3 \times (N-1) \times (N+1)/8$. The actual numbers for bubble-sort median and all-ranks filters of various sizes are as follows:

<table>
<thead>
<tr>
<th>$N$</th>
<th>Number of values to be sorted</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
<th>11</th>
<th>13</th>
<th>15</th>
<th>17</th>
<th>19</th>
<th>21</th>
<th>23</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of modules for median filter</td>
<td>3</td>
<td>9</td>
<td>18</td>
<td>30</td>
<td>45</td>
<td>63</td>
<td>84</td>
<td>108</td>
<td>135</td>
<td>165</td>
<td>198</td>
<td>234</td>
<td></td>
</tr>
<tr>
<td>No. of modules for all-ranks filter</td>
<td>3</td>
<td>10</td>
<td>21</td>
<td>36</td>
<td>55</td>
<td>78</td>
<td>105</td>
<td>136</td>
<td>171</td>
<td>210</td>
<td>253</td>
<td>300</td>
<td></td>
</tr>
</tbody>
</table>

The rules for other algorithms are more complex, but follow the same general trend. Therefore, the number of calculations needed to compute the median for a 5x5 neighborhood (25 values) can be expected to be roughly $(25/9)$ squared = 7.71 times as great as the number for a 3x3 median. Of course, the number required for a pipelined algorithm in the style of Waltz's 3x3 median filter will always be less than this for the corresponding case – sometimes considerably less – but all in all the results for larger neighborhoods are not encouraging if high speed operation is desired. Reference 1 contains some discussion of this point. Considerable additional work was done by Waltz on large-neighborhood sorting algorithms, but this work was reported only in an internal 3M Company technical report.

### 7. Speed Comparisons for 3x3 Median Filter Implementations

Comparative speed tests were run on various platforms for three algorithms: a median filter computed by the bubble-sort process, the median filter computed using the SKIPSM algorithm given above, and the “separable median” filter computed using the SKIPSM algorithm given above. The table below gives the total times in seconds for completing these operations 10, 100, or 1000 times on the full 512x512 test image.

<table>
<thead>
<tr>
<th>System</th>
<th>Compiler &amp; command</th>
<th>Total execution times, in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Bubble Sort Median</td>
</tr>
<tr>
<td></td>
<td></td>
<td>number of repeats</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>PowerMac 7100 80Mhz</td>
<td>Codewarrior</td>
<td>22</td>
</tr>
<tr>
<td>SGI Indigo R5000</td>
<td>cc -mips2</td>
<td>11</td>
</tr>
<tr>
<td>Sun Sparc 10</td>
<td>cc -O</td>
<td>17</td>
</tr>
<tr>
<td>Linux Pentium 133Mhz</td>
<td>gcc</td>
<td>14</td>
</tr>
<tr>
<td>Sun UltraSparc</td>
<td>cc -O</td>
<td>9</td>
</tr>
<tr>
<td>Mac PowerBook G3 250 MHz</td>
<td>Codewarrior</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 10 presents the results for the 1000-repeats tests in reciprocal form; i.e., images per second. Note that the fastest machine in this test, the 250 MHz Apple Macintosh PowerBook G3® nearly equals the standard European video frame rate (25 images per second) when executing the SKIPSM implementation of the “separable median” operation. The newer

![Graph showing speed comparisons for three algorithms on various platforms.](image)

Figure 10. Speed comparisons for three algorithms on various platforms.
Median filters (and ranked filters of ranks other than median) are useful for noise removal and other functions, but they often have relative long execution times, making them unsatisfactory for many speed-critical industrial applications. This paper has provided complete main-loop C-code for implementations of 3x3 ranked filters of ranks 1 (minimum), 2, 3, 4, 5 (median), 6, 7, 8, and 9 (maximum). These implementation, which are based on a partial realization of the SKIPSM paradigm, are believed by the authors to be highly efficient. A full SKIPSM realization is not possible because, except for the filters of ranks 1 and 9, these operations are not separable. This paper shows that, in spite of this lack of separability, the finite-state machine aspect of SKIPSM can be used to advantage. In addition, a fast “full-SKIPSM” implementation of a rank 4-5-6 filter, sometimes called the “separable median” filter, has been presented. Speed comparisons on various platforms show that the SKIPSM median filter is typically faster than a bubble-sort median filter by a factor of about 5, and that the SKIPSM “separable median” filter is typically faster by a factor of 10 or 11, regardless of operating system or platform.

9. Bibliography


