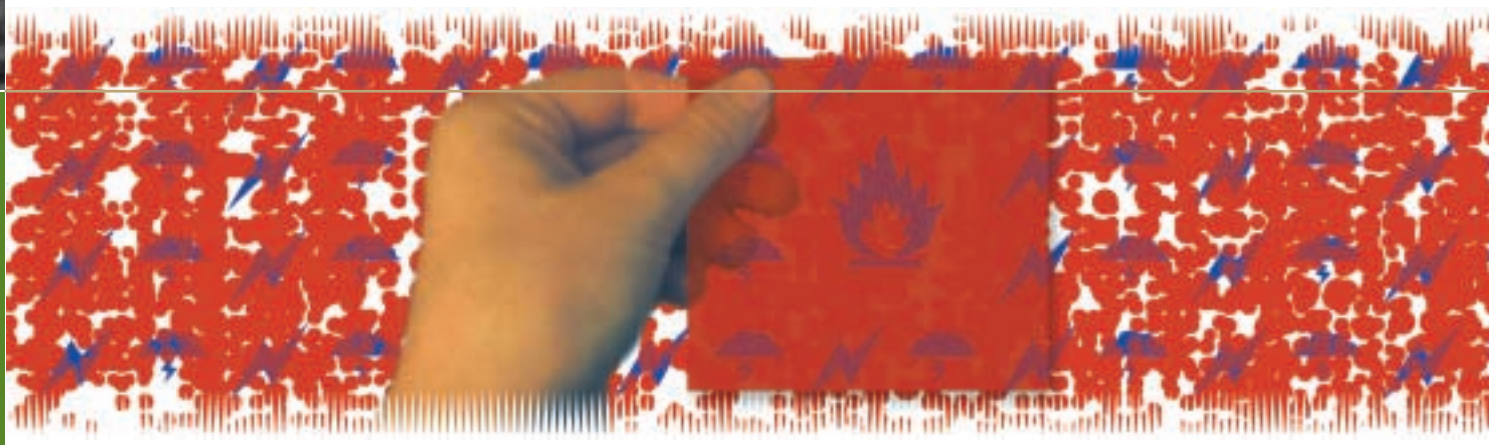
PHIL EKSTROM



Better Than Average

Some averages are better than others. In particular, the median may give more useful results than the mean in some circumstances.

Imagine that you were building a robot to find fire or glowing embers in the dark. You would probably install a couple of sensors to detect red light and command the robot to move in the direction from which the brightest light was coming. Now imagine that you have to use noisy detectors. The usual answer would be to average a number of successive readings, looking for the steady glow behind all that noise.

Now let's make it all work in a lightning storm. You don't know how close you will be either to the fire or to the lightning—either one may be brighter. A flash might be much brighter, or you might be close to the fire and the bolts may be distant, but you know one crucial fact: lightning will flash only now and then; the sky will go dark in between those flashes. Fire may flicker, but it is significantly different in giving substantial light nearly all the time. In the language of statistics, the lightning has hung a fat tail on our probability distribution. My favorite tools for handling that are order statistics like the median.

The problem with a simple running average or any linear filter in this kind of situation is that it has the wrong kind of memory. If its job is to combine *N* observations, then the effect of a single large data value will be significant for something like *N* sample times—longer if the filter does not have finite impulse response, the flash is a bright one, and the firelight is dim. During that time, our robot would go trundling off towards the flash and ignore the fire.

Instead, we should apply a non-linear filter, the running median. To compute the median, create a sorted list of data and pick out the middle entry. The advantage is that the process systematically ignores any unusually large and small data elements. These extreme elements turn up near the ends of the sorted list, and are only counted. If a few of them are unreasonably extreme, it doesn't matter. Once the brightness of a lightning flash exceeds the magnitude of a typical measurement, it can be as bright as it likes and cannot further affect the result. If someone walks between the robot and the fire, blocking the view for a moment, it ignores that too. The wonderful thing is that this all happens automatically. You don't need to make an advance guess about how bright either the glow or the flash will be.

Averages

There are some good reasons why the mean, our familiar average, is ordinarily used instead of the median. One is the relative difficulty of forming that sorted list instead of just

In the language of statistics, the lightning has hung a fat tail on our probability distribution.

adding up the values and dividing by *N*. We will deal with such practical details in a moment. Another, more fundamental, reason calls for a short diversion into theory.

If the data had an underlying statistical distribution with thin tails—something like the classic normal density function shown in Figure 1 (red trace)—the ordinary mean would work better. In this ideal case, the median would be noisier than the mean by a factor near $\sqrt{(\pi/2)} \approx 1.25$.

However, distributions in the real world often have fat tails. Consider the distribution shown in Figure 1 (blue trace). It is 90% the same as before, but 10% of the probability follows a normal distribution 10 times wider. What I'm trying to represent is a composite process: something well-behaved is making the narrow central peak, but something else is going on at a low level and making outliers.

The mean allows the outliers—represented by those fat tails—to affect its estimate unreasonably strongly, and is 3.3 times as noisy with the new data. In contrast, the median focuses its attention on the largely undamaged probability peak in the center, and is noisier by a factor near 1.1. This time, the median is quieter than the mean by a factor near 2.5. Quite a difference for such a subtle change in the shape of the distribution!

So it matters what the underlying statistical distribution is. As is usual, I have used the normal distribution for convenience and familiarity when doing theory. Some real-world situations, like our lightning storm, have even longer tails that are harder to treat analytically, but show off the median to even better advantage. I once saw a really good example at very close range: a pair of

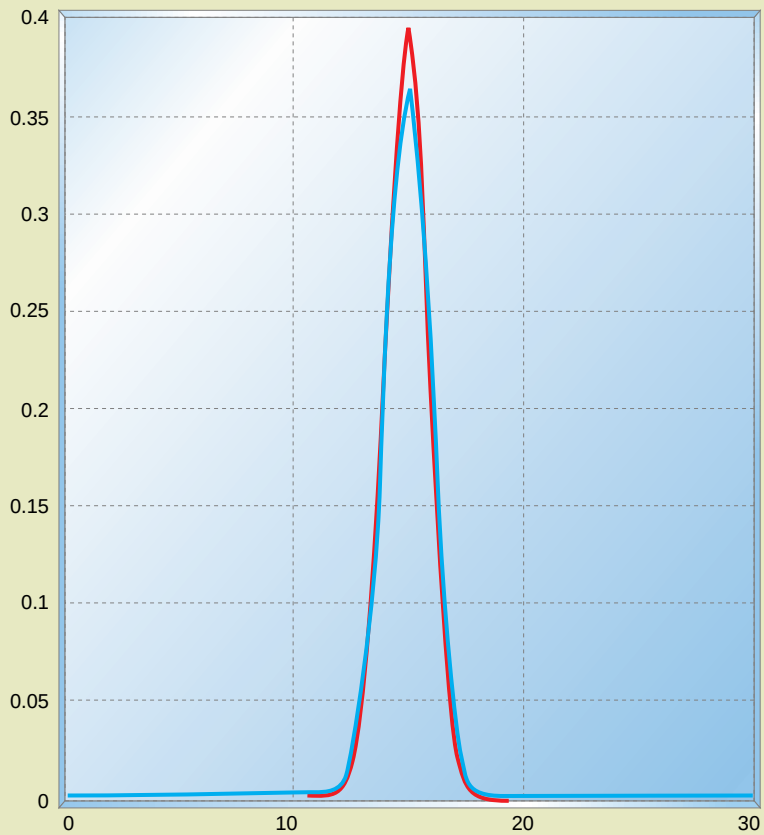
Confidence intervals

So how did I get those theoretical results? Confidence intervals for the mean are based on the variance, s^2, of the composite distribution. To form that, you form the appropriate weighted average of variances of the two distributions that make it up. As always when averaging squared quantities, the larger s quickly takes control of the result.

In sharp contrast, confidence intervals for the median are based on the central density instead[1], and that is where we win. For the two normal distributions of our example, the central density goes like $1/s$ and we take the appropriate weighted average of that quantity. As always when averaging reciprocals, the smaller s takes control! In calculating the median, the most common, organized data behavior dominates any rare large excursions just as we wish it to.

1. The median m is approximately normal with $s^2 = 1/[4\ (N-1)\ f^2(m)]$, f(m) the central density, N odd. For the normal distribution, $f(m) = f(\bar{m}) = 1/s\ (2\pi)$. *Mathematical Statistics*, Freund, Prentice-Hall (1962). I neglect a factor of N/(N-1) in the noise comparisons.

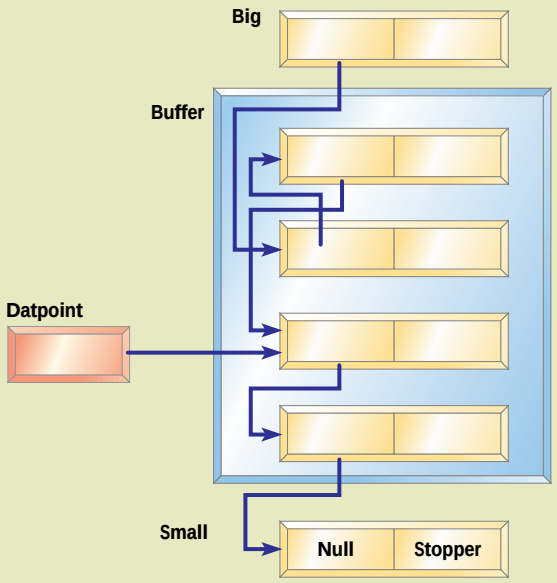
FIGURE 1 Density functions



HANNAH BLAIR

The world of embedded systems is commonly the world of on-the-fly processing, and so it is with our goal-seeking robot.

FIGURE 2 A scheme for a sorted array that will produce the next median value

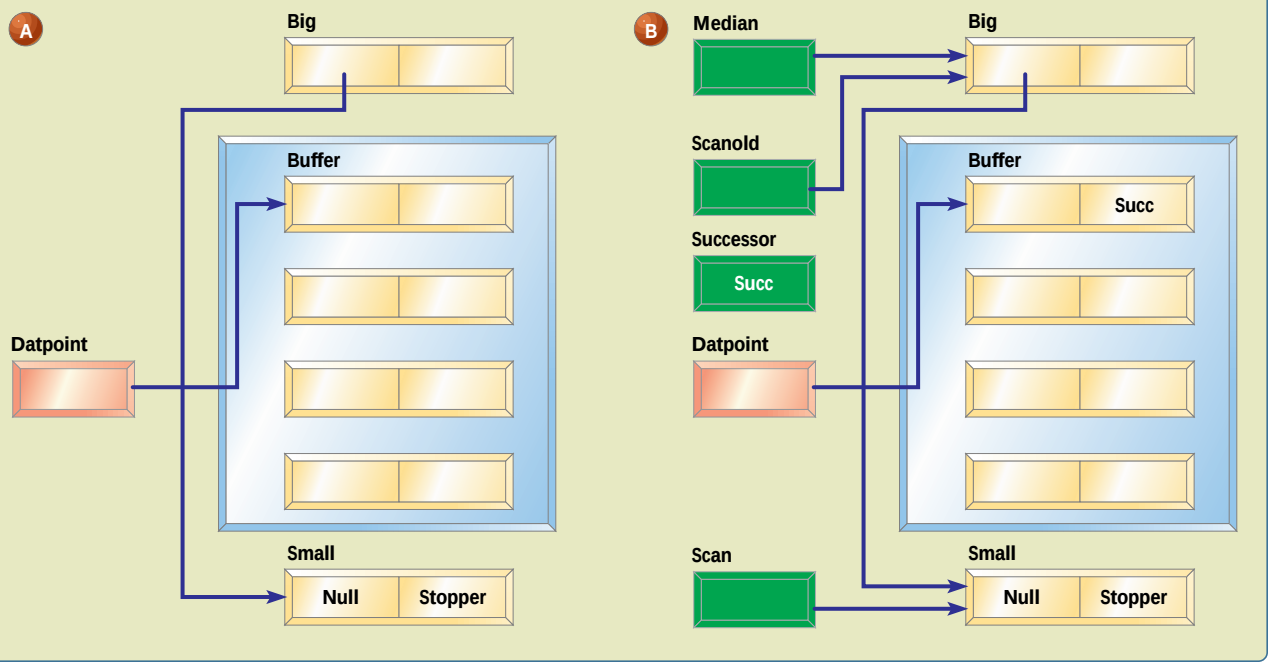


interpenetrating atomic oscillators immersed in a strong electric field. The electric field caused otherwise undetectable random discharges that occasionally upset the oscillators. I was trying to measure the frequency ratio of the two, but the mean of my measurements was so noisy as to be unusable. Fortunately (since this was my thesis experiment), the median of that same data took those upsets in stride.

Median filters

The world of embedded systems is commonly the world of on-the-fly processing, and so it is with our goal-seeking robot. The data comes in at regular intervals and we must produce a stream of median values. We need a median filter. This is a filter that runs in hard real time. We don't care as much about the typical computation time needed to produce the next median value as we do about the worst-case time. One goal for our implementation, then, will be to minimize the worst case computation effort.

FIGURE 3 a) Pointers as initialized before first entry to the filter code. b) The additional pointers needed for a scan



LISTING 1 Median filter in C

```
/* medfilt_new.c    Median filter in C*/

#include <stdio.h>
/* Size of the data buffer; length of the sequence. */
#define NWIDTH 3
/* Smaller than any datum */
#define STOPPER 0

unsigned int medfilter(unsigned int datum)
{
    unsigned int i;
    struct pair
    {
        /* Pointers forming list linked in sorted order */
        struct pair   *point;
        /* Values to sort */
        unsigned int  value;
    };
  /* Buffer of nwidth pairs */
    static struct pair buffer[NWIDTH];
  /* pointer into circular buffer of data */
    static struct pair *datpoint={buffer};
  /* chain stopper. */
    static struct pair small={NULL,STOPPER} ;
  /* pointer to head (largest) of linked list.*/
    static struct pair big={&small,0} ;
  /* pointer to successor of replaced data item */
    struct pair *successor   ;
  /* pointer used to scan down the sorted list */
    struct pair *scan        ;
  /* previous value of scan */
    struct pair *scanold     ;
  /* pointer to median */
    struct pair *median;     ;

  /* No stoppers allowed. */
    if(datum == STOPPER) datum = STOPPER + 1;
  /* increment and wrap data in pointer.*/
    if( (++datpoint - buffer) >= NWIDTH) datpoint=buffer;
  /* Copy in new datum */
    datpoint->value=datum        ;
  /* save pointer to old value's successor */
    successor=datpoint->point    ;
  /* median initially to first in chain */
    median = &big                ;
  /* scanold initially null. */
    scanold = NULL               ;
  /* points to pointer to first (largest) datum in chain */
    scan = &big
  /* Handle chain-out of first item in chain as special case */
;
        if( scan->point == datpoint ) scan->point = successor;
        /* Save this pointer and   */
        scanold = scan ;
Listing 1 continued on p. 108
```

Every median calculation leaves us with a sorted array that is very nearly what we need in order to produce the next median value. All we need to do is to remove the oldest data value from the list and merge in the newly measured one. That is all, I say, but to do it requires accessing the data array in two incompatible ways: in time order (removing the oldest) and in order of magnitude (merge in the next). One scheme for doing that is shown in Figure 2.

The figure shows each measured datum being represented as a two-element structure. These structures contain both the observed value and a pointer to the structure holding the next smaller value. The structures are stored in time order in the circular buffer `buffer`, with `datpoint` indicating the oldest value. Simply incrementing `datpoint` (and wrapping it when necessary) and storing the next datum takes care of replacing the oldest data value by the newest. (Actually it does that replacement only after the array is full. Before then it just fills the array, as we would want it to do.)

The structure `big` holds a pointer to the structure holding the largest datum; its `value` field is unused, but it must be a structure like the others so it can be pointed to by the same kind of pointers. The structure `small` is an end marker. It must hold a null pointer to fail equality comparisons with other list pointers. In its value field it needs a stopper, something smaller than any datum.

Having replaced the old data value, all we need to do is to update the links. There are two tasks involved, and if we add a third we find that the same logic will handle starting up from an empty array. (A chess player might call that "array filling en passant.") The obvious two tasks are to remove the old datum from the linked list and link in the new one in its proper place. The third task is to step a pointer down that list half as fast as we scan for other purposes, and stop our scan at the end

The smallest possible data value, zero for the present case of unsigned integer data, is usurped as a chain stopper.

of the linked list, not necessarily the end of the buffer. That will always leave `median` pointing to the median of the data whether the array is full or not. We can do all three tasks on one pass down the list.

Of course, the median pointer is just right only when there is an odd number of data elements in the list. For an even number of elements there are two middle elements, not one, and the median is defined to be the mathematical average of those two elements. In that case, the pointer ends up pointing to the earlier (larger) of those two and does not quite return the true median. So long as you choose an odd number for the buffer size, this situation will only occur during array filling.

Figure 3a shows the pointers as they are initialized before first entry to the filter code. To initialize the empty list, we need only link the two end structures together as shown. As successive values arrive, they are linked in. When the array finally fills up, a pre-existing link to the element just being replaced will begin to be found in the chain each time, and must be removed.

Figure 3b shows, in green, the additional pointers needed for our scan down the list, initialized as they are at the beginning of the scan. They include `datpoint` (already discussed), `successor` (which points to the old datum's successor in the list), `scan` (which begins equal to `big.point` and which we then chase down the list), `scanold` (which holds the previous value of scan), and `median` (which ends up pointing to the median). The only one of these needing special mention is `successor`.

The new datum might lie either earlier or later in the list than the old datum did, and in particular we may need to link in the new value before we encounter a pointer to the old value and get a chance to link it out. Since both old and new data are associated with the same array element, we need to save a copy of the old datum's successor pointer `datpoint->point` before we begin the scan and risk writing over it. `successor` is the place where we save it.

C code for the filter is found in Listing 1 and is available for download at *www.embedded.com/code.htm*. The code was compiled and tested on a PC

The code is easy to test thoroughly; it quickly exercises all of its branches when tested with any non-monotonic data sequence long enough to fill the buffer and start replacing old values.

under Watcom C. It looks about twice as elaborate as it really is; the code for scanning down the linked list is nearly duplicated. Since the `median` pointer needs to be dereferenced only on odd numbered steps, the first of the near-duplicate copies has code to do that; the second lacks it. The two are executed alternately.

As mentioned previously, the smallest possible data value, zero for the present case of unsigned integer data, is usurped as a chain stopper. In order to protect that marker, if the stopper

LISTING 1, continued Median filter in C

```
       /* step down chain */
        scan = scan->point ;

    /* loop through the chain, normal loop exit via break. */
     for( i=0 ;i<NWIDTH ; i++ )
     {
      /* Handle odd-numbered item in chain  */
       /* Chain out the old datum.*/
         if( scan->point == datpoint ) scan->point = successor;
       /* If datum is larger than scanned value,*/
         if( (scan->value < datum) )
         {
           /* chain it in here.  */
             datpoint->point = scanold->point;
           /* mark it chained in. */
             scanold->point = datpoint;
             datum = STOPPER;
         };
   /* Step median pointer down chain after doing odd-numbered element */
       /* Step median pointer.  */
         median = median->point        ;
       /* Break at end of chain  */
         if ( scan == &small ) break ;
       /* Save this pointer and   */
         scanold = scan ;
       /* step down chain */
         scan = scan->point ;

   /* Handle even-numbered item in chain.  */
         if( scan->point == datpoint ) scan->point = successor;
         if( (scan->value < datum) )
         {
             datpoint->point = scanold->point;
             scanold->point = datpoint;
             datum = STOPPER;
         };
         if ( scan == &small ) break;
         scanold = scan ;
         scan = scan->point;
     };
     return( median->value );
}
```

value appears in the data stream, it will be replaced by a value one larger. In the present case, incoming zeros get replaced by ones. This is ordinarily harmless. Recall that data values other than the median itself are only counted; so long as they stay on the correct side of the median, their actual values cannot affect the value returned by the calculation. Only if the true mean were ever zero would this usurpation affect the result, and in that case the median would be reported as 1, not zero. Ordinarily it is easy to avoid that ever happening, and even more commonly, one would not care. However, in the rare situation where the smallest possible data value must also be a possible median, the program can be changed to suit. Either reverse the sorting order and use the largest possible data value as the flag instead of the smallest, or use some other means (like setting a flag) to prevent generating additional links once the new datum has been linked in.

Worst-case analysis

A microcontroller like the Motorola 68HC12 is a more likely controller than a PC for something like our robot project, so for a time estimate I counted cycles in a hand-coded version of this program for that processor. For a full *N*-element data buffer, there will be *N* steps down the chain, of which $(N + 1) / 2$ will be odd and will include dereferencing the median pointer. There will be one pass through the link-out code, and one through link-in. Counting up the bus cycles gives $86 + N * 26$, or at an 8MHz cycle rate (16MHz crystal) the filter takes $10.75 + N * 3.25$ms: 46.5ms when $N = 11$.

This is, in several respects, a good algorithm for hard real-time application. It has a fixed execution time (approximately so on any machine, and exactly so on the 'HC12). By making use of the sorted array left over from a previous step, that time becomes proportional to *N*, better than *N* log *N* which is the usual optimum when sorting. The code is easy to test thoroughly; it quickly exercises all of its branches when tested with any non-monotonic data sequence long enough to fill the buffer and start replacing old values.

So next time you are hunting for a fire in a lightning storm, or have some other case of fat tails to handle, try this median filter. **esp**

Phil Ekstrom holds a British postgraduate diploma in computer science, and a PhD in physics. He is currently chief physical scientist at Northwest Marine Technology, where he develops field instruments that often involve embedded computation. Contact him at ekstrom@nmt-inc.com.