Burst Detection using Wavelets for Elastic Windows

T. H. Merrett*

McGill University

April 3, 2008

[1]

This is taken from [SZ04, Chap. 7]. It is the third of three applications they develop based on their review of timeseries techniques in the first four chapters.

1. *Bursts* are sudden occurrences of high values over a continuous duration of time in a timeseries. For example, in

```
1   3   5  11  12  13   0   1
```

if we specify thresholds for different durations, d, such as

d	1	2	3	4
threshold	15	25	35	45

then we see two bursts: one of length 3 starting at time 3 (counting from 0)

$$11 + 12 + 13 = 36 \geq 35$$

and, contained in that, one of length 2 starting at time 4

$$12 + 13 = 25 \geq 25$$

The new aspect of this problem is that we must discover not only the burst but also the duration length. This sounds something like a variable window size: [SZ04] call it "elastic windows", and so they speak of "elastic bursts".

However, it does not require us to redefine **redwin**() (week7p1) for elastic windows, as we shall see, so I'll speak of elastic *durations* and use d for the length of the duration.

We need to look at wavelets first [SZ04, pp.41–2,34–5].

2. The simplest wavelet technique uses pairwise averages to characterize a set of values such as the timeseries above.

The author gratefully acknowledges support from the taxpayers of Québec and of Canada who have paid his salary and research grants while this work was developed at McGill University, and from his students (who built the implementations and investigated the data structures and algorithms) and their funding agencies.

1						
3	2	−1				
5						
11	8	−3	5	−3		
12						
13	12.5	−0.5				
0						
1	0.5	−0.5	6.5	6	5.75	−0.75

The first column is the timeseries itself. The second, fourth and sixth columns are averages: of pairs of the timeseries, of pairs of the averages in column 4, and of pairs of the averages in column 6, respectively. The third, fifth and seventh columns are differences between the averages and the values making up the averages: note that we need only one difference for each average because it is an average of only two numbers, and so the average plus or minus the difference gives back both of these numbers.

We see from this that we can reconstruct the entire sequence from a sequence made up of the final average and all the differences: eight numbers in all and so exactly the same size as the original.

1	3	5	11	12	13	0	1
5.75	-0.75	−3	6	−1	−3	−0.5	−0.5

For example, 5.75 ± -0.75 gives back the two next-level averages, the first of these, 5, ± -3 gives back the two first-level averages, 2 and 8, and so on.

We can see that this "wavelet transform" can be done, either way, in linear time: better than the Fourier transform, even than FFT.

3. Wavelet transforms in Aldat require a little extension to the user-definable **funop** computations of week2p1. We must be able to add further parameters.

Let's practice first on just finding the first set of pairwise averages.

comp funop *average2*(*value, accum*; *seq*) **is**
{ **if** *seq* **mod** 2 = 0 **then**
  { *sum* <− *value*;
    accum <−$\mathcal{DC}$ }
  else
  { *sum* <− *sum* + *value*;
    accum <− *sum*/2 };
}

Ths would be invoked on the relation

TS(*sq*	*ts*)	*av2*
0	1	$\mathcal{DC}$
1	3	2
2	5	$\mathcal{DC}$
3	11	8
4	12	$\mathcal{DC}$
5	13	12.5
6	0	$\mathcal{DC}$
7	1	0.5

as

let *av2* **be fun** *average2*(*sq*) **order** *sq*;

(And we could do it for groups of any size, *gp* by replacing 2 by *gp* in the two places it occurs, and making *gp* a fourth parameter.)

Here is specialized code for finding the wavelet transform of the above timeseries. It is specialized to timeseries of length 8. There is one further trick: the *accum*ulator of the **funop** is a nested

relation.

```
comp funop averge Wave(value, accum; seq) is
{ state sq2 <− 4, sq4 <− 2, sq8 <− 1, sum2, sum4, sum8;
  if seq mod 2 ≠ 1 then
  { sum2 <− value }
  else
  { sum2 <− sum2 + value
    let sqW be sq2;
    let tsW be sum2/2 − value;
    accum <+ [sqW, tsW] in DC;
    sq2 <− sq2 + 1;
    if seq mod 4 ≠ 3 then
    { sum4 <− sum2/2 }
    else
    { sum4 <− sum4 + sum2/2;
      let sqW be sq4;
      let tsW be sum4/2 − sum2/2;
      accum <+ [sqW, tsW] in DC;
      sq4 <− sq4 + 1;
      if seq mod 8 ≠ 7 then
      { sum8 <− sum4/2 }
      else
      { sum8 <− sum8 + sum4/2;
        let sqW be sq8;
        let tsW be sum8/2 − sum4/2;
        accum <+ [sqW, tsW] in DC;
        sq8 <− sq8 + 1;
        let sqW be 0;
        let tsW be sum8/2;
        accum <+ [sqW, tsW] in DC;
      }
    }
  }
}
```

(Ordinary recursion will generalize this to timeseries of length any power of 2, and an arbitrary timeseries can be padded to make its length a power of 2.)

The value of the accumulator corresponding to the highest value of *seq* is the final result. The invocation is

```
let wavelets be fun averge Wave(sq) of ts order sq;
ans <− (where sq = red max of sq in [sq, wavelets] in TS)/wavelets
```

4. This wavelet transform and its inverse can be written as matrices

$$\mathbf{W} = \underline{\underline{W}}\mathbf{T}$$

and

$$\mathbf{T} = \underline{\underline{W}}^{-1}\mathbf{W}$$

where **T** is the timeseries and **W** is its wavelet transform.

You can check for the example that

$$\underline{\underline{W}} = \begin{pmatrix} 1/8 & 1/8 & 1/8 & 1/8 & 1/8 & 1/8 & 1/8 & 1/8 \\ 1/8 & 1/8 & 1/8 & 1/8 & -1/8 & -1/8 & -1/8 & -1/8 \\ 1/4 & 1/4 & -1/4 & -1/4 & & & & \\ & & & & 1/4 & 1/4 & -1/4 & -1/4 \\ 1/2 & -1/2 & & & & & & \\ & & 1/2 & -1/2 & & & & \\ & & & & 1/2 & -1/2 & & \\ & & & & & & 1/2 & -1/2 \end{pmatrix}$$

and

$$\underline{\underline{W}}^{-1} = \begin{pmatrix} 1 & 1 & 1 & 1 & & & \\ 1 & 1 & 1 & -1 & & & \\ 1 & 1 & -1 & & 1 & & \\ 1 & 1 & -1 & & -1 & & \\ 1 & -1 & & 1 & & 1 & \\ 1 & -1 & & 1 & & -1 & \\ 1 & -1 & & -1 & & & 1 \\ 1 & -1 & & -1 & & & -1 \end{pmatrix}$$

In fact, these are usually *normalized*: the first two rows of $\underline{\underline{W}}$ are multiplied by $2\sqrt{2}$ and the first two columns of $\underline{\underline{W}}^{-1}$ are divided by $2\sqrt{2}$, the third and fourth rows/columns by 2, and the last four rows/columns by $\sqrt{2}$.

Then both transforms are orthonormal and have good properties such as preserving the "energy" (the sum of the squares of the terms of either **T** or **W**).

Thus, the wavelet transform can be used to compress a timeseries, at the cost of becoming only an approximation, by omitting all but the most significant terms in **W** after **W** has been found.

Under normalization, the table of averages and differences above becomes

1						
3	2.8284	−1.4142				
5						
11	11.3137	−4.2426	10	−6		
12						
13	17.6777	−0.7071				
0						
1	0.7071	−0.7071	13	12	16.2635	−2.1213

making the final **W**

1	3	5	11	12	13	0	1
16.2635	-2.1213	−6	12	−1.4142	−4.2426	−0.7071	−0.7071

If we pick the four terms of **W** with the largest magnitudes, namely 16.2635, 12, −6 and −4.2426, we get an energy of 462.5 instead of the true energy of 470.

We can also approximate in another way, by taking so many coefficients at the *beginning* of **W**, say the first four. This has the effect of showing the timeseries at reduced *resolution*.

Going back to the unnormalized version, the approximate **W** is

5.75	-0.75	−3	6	0	0	0	0

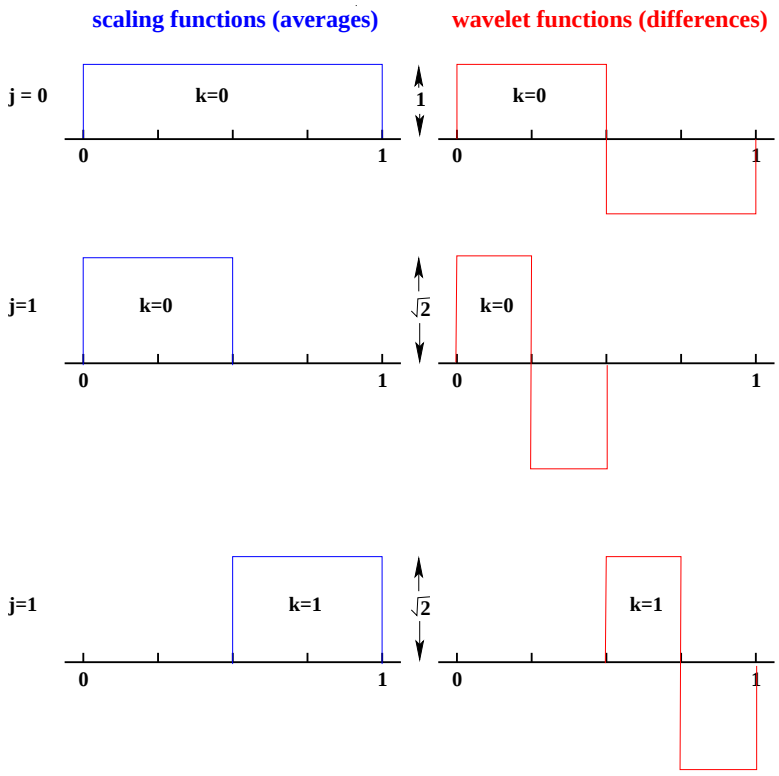


Figure 1: Haar wavelets (cont. next page)

and the timeseries $\mathbf{T}'$ that we get by transforming this back again is just the first set of averages:

$$2 \quad 2 \quad 8 \quad 8 \quad 12.5 \quad 12.5 \quad 0.5 \quad 0.5$$

We will take advantage of this variable resolution capability when we get back to looking at elastic bursts.

5. To prolong the discussion of wavelets just a little further, we should see why they are called "wavelets".

Compare figures 1,2 with the rows of the normalized matrix $\underline{\underline{W}}$: match the first blue figure with the first row, the first red figure with the second row, and the remaining red figures with the remaining rows. Compare the normalizations. The equations of the blue figures are $\phi_{jk}(x) = 2^{j/2}\phi(2^j x - k)$ where

$$\phi(x) = \begin{cases} 1 & \text{if } 0 \le x < 1 \\ 0 & \text{otherwise} \end{cases}$$

and of the red figures are $\psi_{jk}(x) = 2^{j/2}\psi(2^j x - k)$ where

$$\psi(x) = \begin{cases} 1 & \text{if } 0 \le x < 1/2 \\ -1 & \text{if } 1/2 \le x < 1 \\ 0 & \text{otherwise} \end{cases}$$

We can see that the scaling functions (blue) are orthonormal within each value of j: try integrating the products of pairs of functions (i.e., multiply the two magnitudes and check the area under the

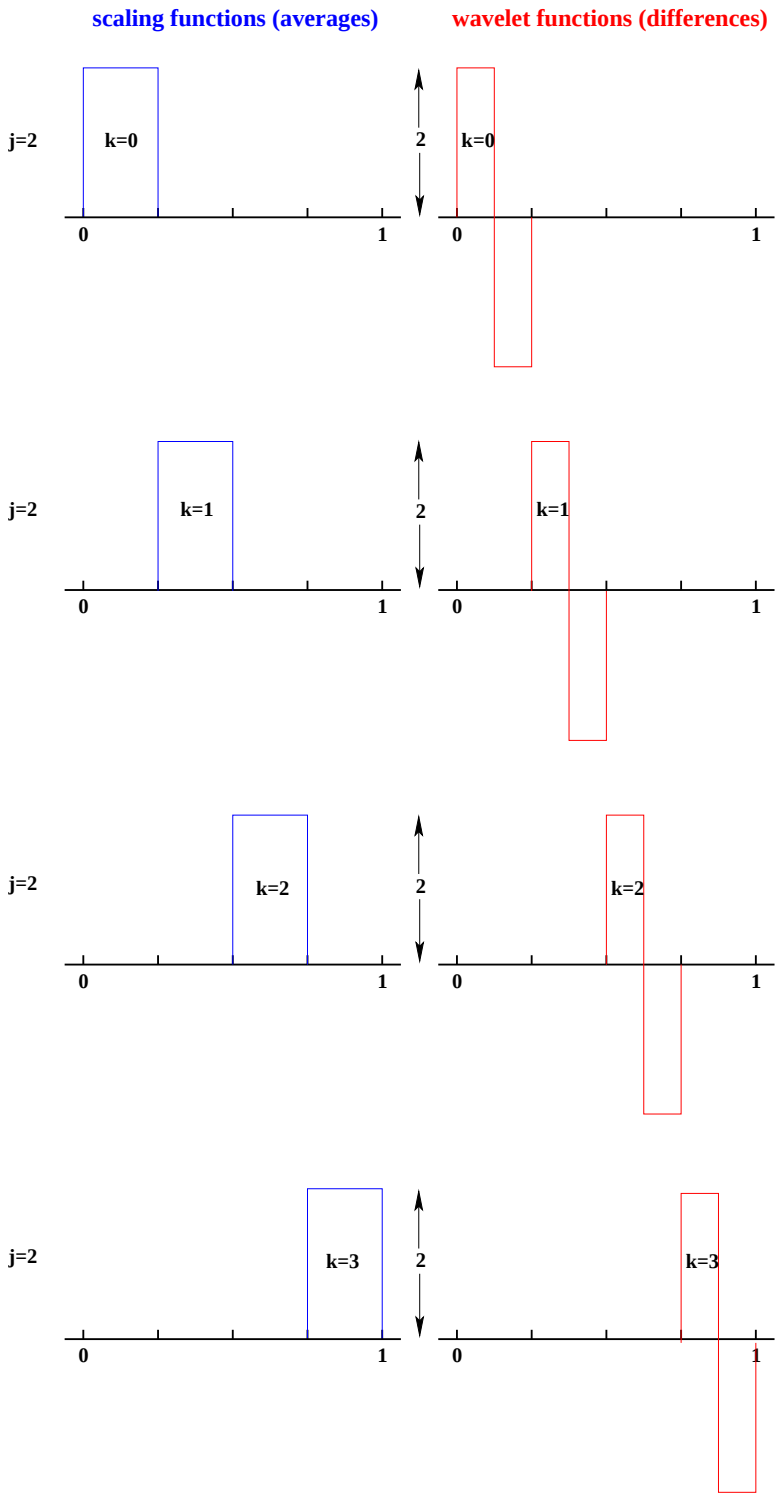
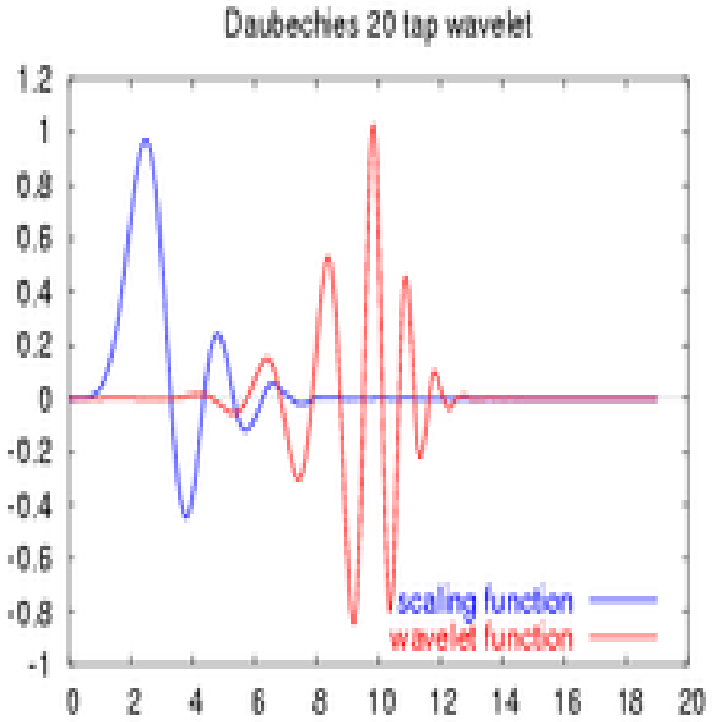


Figure 2: Haar wavelets (cont.)

result). We can also see that the wavelet functions are orthonormal over all values of j and k, and are also orthogonal to all of the scaling functions.

These step-function wavelets are called Haar wavelets. There are many other kinds. Here, for example, is one of the family of Daubechies wavelets (from en.wikipedia.org/wiki/Daubechies_wavelet).



6. Now we can return to elastic bursts. We needed an ability to inspect durations of many different sizes simultaneously and the hierarchical, multiresolution nature of wavelets seems to be helpful.

What wavelet multiresolution gives us is durations of sizes that are powers of 2. How many durations of arbitrary size can we cover with these?

Let's characterize durations by their size and starting position, (s, p).

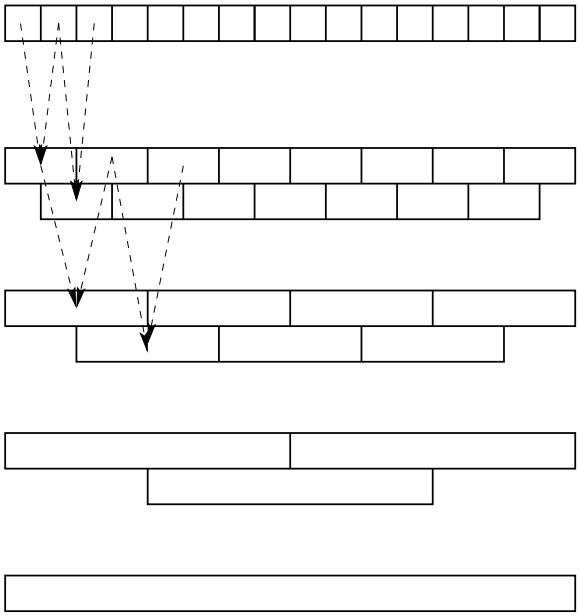
A wavelet of size 2^0 will give any duration of size 1, $(1, p)$ for any p.

A wavelet of size 2^1 will give any duration of size 1 but only those durations of size 2 that start at odd-numbered positions, $(2, p)$ where $p \bmod 2 = 1$ (counting from 1).

A wavelet of size 2^2 will give any duration of size 1 but only those durations of size 2 that do not start at multiples of 3, $(2, p)$ where $p \bmod 3 \neq 0$, and only those durations of size 3 that start at multiples of 3 plus 1, $(3, p)$ where $p \bmod 3 = 1$.

We need not go further to see that this is not good enough. There are no durations of size > 1 that can all be contained in any of these wavelets. Shasha and Zhu introduce a shifted aggregate tree (they call it a "shifted binary tree") to allow containment and reasonable approximations.

The idea is almost to double the tree size by adding shifted rows. Here are the original rows (the wider rows, and, where paired, the upper row of each pair) together with the shifted rows (the lower row of each pair) for a timeseries of 16 points.



(The dashed arrows show how a cell in each row can be constructed from cells in the previous original row. This applies to each cell in the row.)

Now we can see that durations of any size at any position is covered by at least one of these (pairs of) rows.

A wavelet of size 2^0 will give any duration of size 1, $(1, p)$ for any p, using the first row.

A wavelet of size 2^1 will give any duration of size 2, $(2, p)$ for any p, using the second row (pair).

A wavelet of size 2^2 will give any duration of size 3, $(3, p)$ for any p, using the third row (pair). It will also give some, but not all durations of size 4: those starting at even-numbered positions will be missed.

A wavelet of size 2^3 will give any duration of sizes 4 and 5, $(4, p)$ and $(5, p)$ for any p, using the fourth row (pair). (It will also give an increasingly small proportion of durations of sizes 6 to 8, so we had better not try to take a chance on them.)

The final wavelet, of size 2^4, contains all remaining durations, of sizes 6–16, for any p, using the final, single-aggregate row.

Instead of calculating averages and differences, as for wavelets, we calculate sums for burst detection. The third and fifth columns are the two shifted "rows" (they were rows in the figure above).

1					
3	4				
5		8			
11	16		20		
12		23			
13	25			41	
0		13			
1	1		26		46

Let's use this to detect the burst, 11–12–13. Our threshold sums for specified durations, d, were, in Note 1

d	1	2	3	4
threshold	15	25	35	45

We see nothing ≥15 in the first column. There is one entry ≥25 in the second and third columns, so we've identified a burst of duration 2. Finally, the 41 in columns 4 and 5 includes 35, the threshold for a burst of duration 3.

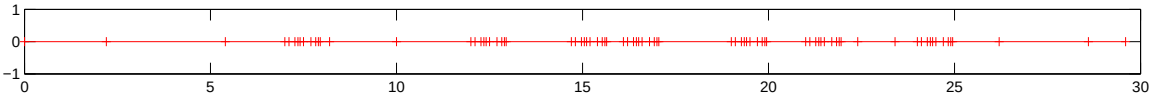
When we check the original timeseries we find that the 3-burst contains the 2-burst, so we report only the longer burst.

7.Other aggregates besides sum can be used. The criterion that admits an aggregate is that it must be *monotonic*. This is true for sums only of positive numbers, which we have because bursts are likely to be counts of occurrences. (Note that the timeseries being aggregated must therefore *not* be normalized.)

Because the sum is monotonic, above, we know that we will not miss a burst by checking a wavelet which is longer than the duration we are looking for: there are no negative values among the neighbours to reduce the sum indicating a burst to a smaller sum which we might ignore. Thus there are no false negatives (failed alarms).

Other monotonic quantities are counts, maxima, minima (which are monotonically *de*creasing) and spreads (max − min).

Here is an example of counts: eighty events happen in 30 seconds.



We will look for bursts with thresholds (durations, d, in seconds)

d	1	10
threshold	10	50

So we must convert the counts to monotonic sums anyway.

We do this by tallying the events in a histogram with bin width of one second. The result is

```
1 0 1 0 0 1 0 10 1 0 1 0 10 0 3 7 8 2 0 10 0 10 1 1 10 0 1 0 1 1
```

We can immediately see five 1-second bursts, each just at the threshold of 10.

We can build a shifted aggregate tree to see if there are any 10-second bursts. (This is built horizontally this time, so we are back to rows. To keep the spacing simple, I've written "a" instead of "10" in the first row. Note that I've padded the 30 seconds with two 0-entries to make a power of 2.)

```
1 0 1 0 0 1 0 a 1 0 1 0 a 0 3 7 8 2 0 a 0 a 1 1 a 0 1 0 1 1 0 0

  1   1   1   10  1   1   10  10  10  10  10  2   10  1   2   0
    1   1   1   11  1   10  3   15  2   10  11  11  1   1   1

      2       11      2       20      20      12      11      2
          2       11      11      20      20      12      3

              13              22              32              13
                      13              40              23

                              35                              45
                                              54

                                                              80
```

We must look for bursts of 10-second durations in the last level. (Check that the next-to-last level can only reliably detect 6- to 9-second durations.) However, most of the 10-second durations can be found in that next-to-last level (check which ones get missed) and so that 54 in the shifted part of level 5 is worth looking into, and reveals a 10-second burst from times 12–21, including three of the five 1-second bursts we saw at the beginning.

8. Two dimensions Two-dimensional wavelets may also be built, using a base unit of four wavelets in a square (which may be combined in pairs, say first horizontal then vertical, in order to continue writing only single numbers, e, for the differences $\pm e$ between the average and two components).

Here we look at one example of two-dimensional burst detection. We'll use the same thresholds as in our first example

d	1	2	3	4
threshold	15	25	35	45

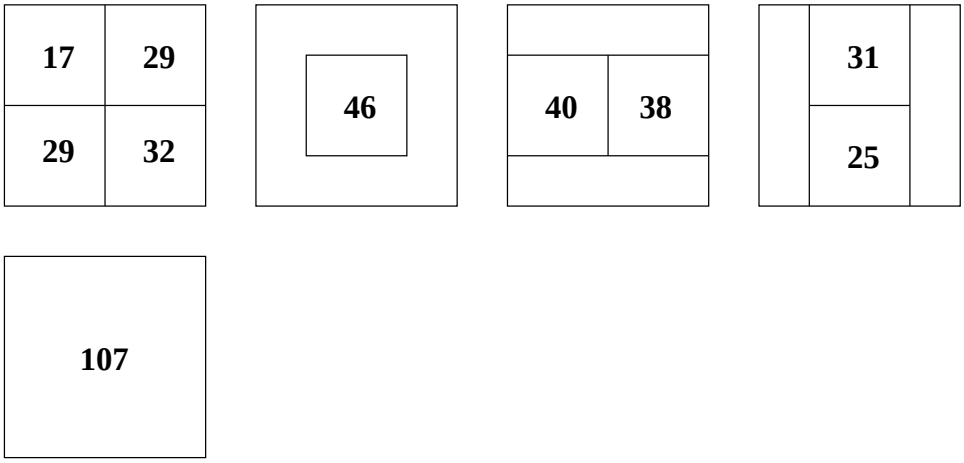
and we'll be looking for *rectangle*-shaped bursts.

Here is the data to be summed. The bursts we should discover are marked in red: one square of 4 and two rectangles of 3.

0	**5**	**3**	**1**
1	**11**	**12**	**13**
16	**12**	**11**	**2**
1	**0**	**2**	**17**

We already see two 1-cell bursts of 16 and 17.

Here is the shifted aggregate tree. Note that there are three shifts at each level (except top and bottom levels) instead of one: one horizontal, one vertical and one both horizontal and vertical.



In the first row (which is actually the second row, since the previous figure should count as the first row), the first, unshifted, grid shows that we may have three 2-cell bursts. Checking confirms only

two: 12,13 and 16,12. These 2 by 2 squares cannot contain any 3-cell bursts (which would have to be 3 by 1 or 1 by 3 rectangles), and do not in this case contain any 4-cell bursts.

Of the three shifted grids in the first row, the horizontal-vertical shift gives 46 for the central square, which must be a 4-cell burst. The vertical shift may have two 2-cell bursts: it does, but we already found them. The horizontal shift may also have 2-cell bursts, but they turn out to be part of the 4-cell burst we found already.

For 3-cell burst, we are obliged to check the whole original data (the bottom level in the shifted aggregate tree), and we find the 3-cells 11–12–13 and 16–12–11. These incorporate the 2-cell bursts we found before, which we forget about. They also overlap the 4-cell burst in the centre, but we cannot do anything about that since we must report rectangles, not arbitrary hexonimos.

So we report one 4-cell and two 3-cell bursts, as we expected we should.

Note that this 2D discusssion is really a 3D discussion, because neither dimension is the time dimension, whereas in the 1D case the one dimension is the time dimension. There is an intermediate "1D" (really 2D) case with one spatial and one time dimension.

9. Bursts in Streams With an indefinite-sized stream of data, we can do the computing of Note 3 in the context of **redwin**(). But it would be nice not to redo it all at every window position.

In fact, we need only update the last entry at each level each time we slide the window (i.e., acquire one more time point).

	window						
seq	(*sq*	*ts*)					
0	371	1					
1	372	3					
2	373	5					
3	374	11					
4	375	12					
5	376	13					
6	377	0					
7	378	1	1	13	26	41	46
8	379	4	5	1	18	36	49
	380	2	6	5	7	26	48

where the vlaues of the current window are given in `typewriter font` and *seq* is the virtual attribute defined **let** *seq* **be** (**fun** + **of** 1 **order** *sq*) − 1 within this window.

Here is how we get these new numbers. Suppose the initial set of entries is given by

entries		
(*pow2*	*shift*	*val*)
1	0	1
1	1	13
2	0	26
2	1	41
3	0	46

and the *seq* positions in the window whose values must be added to or subtracted from this current set of *val*ues are

entries			
(*pow2*	*shift*	*add*	*subtr*)
1	0	8	6
1	1	7	5
2	0	8	4
2	1	6	2
3	0	8	0

Then we can define

toAdd <− [*pow2, shift, ts*] **in** (*positions*[*add*:**icomp**:*seq*][*seq, ts*] **in** *window*);
toSubtract <− [*pow2, shift, ts*] **in** (*positions*[*subtr*:**icomp**:*seq*][*seq, ts*] **in** *window*);

giving

toAdd			*toSubtract*		
(*pow2*	*shift*	*ts*)	(*pow2*	*shift*	*ts*)
1	0	4	1	0	0
1	1	1	1	1	13
2	0	4	2	0	12
2	1	0	2	1	5
3	0	4	3	0	1

This combines with *window*

let *tsp* **be** *ts*;
let *tsm* **be** *ts*;
let *val′* **be** *val* + *tsp* − *tsm*;
let *val* **be** *val′*;
entries <− [*pow2, shift, val*] **in** [*pow2, shift, val′*] **in** (*entries* **ijoin** [*pow2, shift, tsp*] **in** *toAdd*
ijoin [*pow2, shift, tsp*] **in** *toSubtract*);

giving

entries		
(*pow2*	*shift*	*val*)
1	0	5
1	1	1
2	0	18
2	1	36
3	0	49

To reconcile all ths with **redwin**(), which is a domain algebra operator, we must recast relations *entries, toAdd* and *toSubtract* as nested virtual relations: the <− assignments above are replaced by **let .. be**. Then *window* can be defined, also as a virtual nested relation

let *window* **be redwin**(9) **ujoin of relation**(*sq,ts*) **order** *sq*;

and the above code used.

The whole calculation can be done in a computation of two **alt**-blocks, the first **alt**-block containing the code of Note 3, generalized and extended to create the first *entries*, and the second **alt**-block containing the above code, using *entries*.

(Relation *positions* can be a top-level relation, and can be generated from the single parameter *p2* giving the power of 2 that is the size of the sliding window (in this case $p2 = 3$), remembering to extend it to $2^{p2} + 1$ for the update process so that item 0 can be subtracted from the 2^{p2} entry.)

Of course, we do not necessarily need to calculate or recalulate the entries for all levels. A restricted range of elastic durations limits the number of levels needed. For instance, durations $d = 1$, 2 and 3 only for our 8-item window would not need level 2^3.

This is [SZ04]'s "online" algorithm, which guarantees a response time of one unit. They also describe a "batch" algorithm which waits until all input data is available for the entry at any given level. This does less calculating at the expense of a longer response time—but the response time amortizes to two time-units.

References

[SZ04] Dennis Shasha and Yunyue Zhu. *High Performance Discovery in Time Series: Techniques and Case Studies*. Springer Verlag, New York, 2004.