The SumThreshold method: technical details

André Offringa

October 15, 2013

In this technical document, the `SumThreshold` method is briefly described, and implementation details are given. The algorithm that is considered to be optimal will be given, and details of how to vectorize it with the SSE instruction set will be discussed.

1 Problem statement

Consider a data set consisting of a sequence of samples. The samples contain noise sampled from some distribution, and occasionally a feature of unknown intensity and length. The `SumThreshold` method is an algorithm for detecting such features, including its start and end position.

The method is introduced in Offringa et al. (2010a), where it is shown to be useful for detection of radio-frequency interference (RFI). For this case, it is applied separately in the time and frequency directions at high resolution. A pipeline using the `SumThreshold` method was described in Offringa et al. (2010b).

Paraphrasing Offringa et al. (2010a), the input of the `SumThreshold` method is a one dimensional sequence of values and its output is a binary mask of samples in which features are detected. If the input contains a consecutive sub-sequence with M samples, for which the average of the sub-sequence exceeds a threshold function $\chi(M)$, this sub-sequence will be selected in the mask. However, an added requirement is that a sample that exceeds some threshold $\chi(M)$, should not be used when testing larger sub-sequences. For example, if $\chi(1) = 1$ and $\chi(2) = 0.7$, then the sequence $[0, 3, 0]$ produces an output mask in which only the number 3 has been flagged, even though the size 2 sub-sequence $0 + 3 > 2\chi(2)$. Because the number 3 will be masked by the threshold limit of $\chi(1)$, it will be replaced by the sub-sequence averages (which is 0 here), and the sub-sequences do not exceed the other thresholds. Table 1 lists a few examples.

Table 1: Example outputs with $\chi(1) = 1$, $\chi(2) = 0.7$ and $\chi(4) = 0.5$

Input	Output mask
`0, 0, 3, 0, 0`	`_ _ X _ _`
`0, 0.9, 3, 0.9, 0`	`_ X X X _`
`0, 0.9, 0.9, 0.9, 0`	`_ X X X _`
`0.5, 0.9, 0.9, 0.9, 0.5`	`X X X X X`

2 Algorithm

The method has some correspondence with the scale-invariant rank (SIR) operator (Offringa et al., 2012). The SIR operator can also be said to detect features, though in a binary mask. It detects sub-sequences in which the ratio of masked values exceed the threshold. On the other hand, the `SumThreshold` method works on continuous values, but otherwise also detects sub-sequences in which the average value exceeds the threshold. If we would assign the values '0' and '1' to the possible binary values in the definition of the SIR operator, the `SumThreshold` seems to select the same sub-sequences.

However, the two methods differ on one important point: the SIR operator is meant to extend the mask beyond the original (binary) features. The purpose of the `SumThreshold` is to exactly select the feature. This difference is expressed by the iterative definition of the `SumThreshold` method: first, the least strict threshold is applied on single samples. Next, a slightly stricter (more sensitive) threshold is applied on the sum of size-two sub-sequences, but individual samples that were already detected in the first round, do not trigger the second iteration. In the third iteration, samples that were detected in the first or second will not trigger size three sub-sequences, etc.

This difference has a major consequence on the algorithm. While the SIR operator can be implemented by three passes over the data, we have not been able to find a similar fast algorithm for the `SumThreshold` method. The `SumThreshold`'s inherent iterative definition requires a pass over the data for each sub-sequence size. Therefore, if every sub-sequence size needs to be tested, one has to test $\mathcal{O}(N^2)$ sub-sequences, which requires the same time complexity. For some applications, a $\mathcal{O}(N^2)$ algorithm might be sufficient, but in the case of LOFAR, such an algorithm would be too slow. To overcome this problem, we allow a slight decrease in accuracy by constraining the number of sub-sequence lengths.

2.1 Constraining the tested sub-sequence lengths

We consider two relaxations in the the tested sub-sequence lengths, that increase the efficiency of the algorithm. Both relaxations are constraints on the tested sub-sequence lengths. The first constraint is to only consider exponentially increasing sub-sequence sizes, e.g. sizes of $[1, 2, 4, 8, 16, \ldots]$. This decreases the time complexity to $\mathcal{O}(N \log N)$, while having only a benign effect on the accuracy. This is because it is likely that features of non-tested sizes will be detected by one of the tested sizes, e.g., a feature of size 3 is likely detected as two features of size 2 if the feature is strong enough. If it is not strong enough, it might still be detected as a feature of size 4. In this case, one falsely detected sample is included, but in the RFI detection case, it is more important to flag possible features, than minimizing false-positives. Features whose total average is larger than $\chi(3)$ and, together with their two neighbouring samples, are smaller than $\chi(4)$, are not detected.

The second constraint is to not consider sub-sequence sizes larger than a given size limit. In the LOFAR pipeline, we only consider features up to 1024 samples. Features larger than that are probably detected by one of the smaller sub-sequence tests, but one should note that a class of very extended and faint (but detectable) features are now ignored. This further optimizes the efficiency of the algorithm such that it has linear time complexity. A linear time complexity is particularly important in real-time environments.

All in all, by only considering exponentially increasing sub-sequence sizes up to

1024 samples in size, we have to perform 11 iterations of the algorithm, and each iteration has linear time complexity. We will now consider how to efficiently perform the individual iterations.

2.2 A single `SumThreshold` iteration

The following steps efficiently implement a single `SumThreshold` iteration:

- Slide a window over the data, with size equal to the sub-sequence size M to be tested in this iteration.
- Maintain the sum and the number of unflagged samples in the window. In particular, when moving the window one sample to the right:
  - If the sample to the right was not flagged in previous iterations, add it to the sum and increase the counter.
  - If the sample to the left was not flagged in previous iterations, subtracted it from the sum and decrease the counter.
- For each window position, the average can be calculated by dividing the sum with the counter. If this average exceeds the threshold χ, flag all samples in the window.

Listing 1 performs a single iteration.

Listing 1: Calculate one iteration of the `SumThreshold` mask

```
Require: $\chi$ is the average value threshold for sub-sequences of size $M$,
    $x(0 \dots N-1)$ is the input sequence of $N$ values,
    $y(0 \dots N-1)$ is the mask generated by the previous iteration.
Ensure: $y(0 \dots N-1)$ contains the output mask
  $z \leftarrow 0, i \leftarrow 0$, count $\leftarrow 0$
  $t(0 \dots N-1) \leftarrow y(0 \dots N-1)$
  while $i \neq M$ do
    if NOT $y(i)$ then
      $z \leftarrow z + x(i)$
      count $\leftarrow$ count $+1$
    end if
    $i \leftarrow i + 1$
  end while
  while $i \neq N$ do
    if $z > \chi \times$ count OR $z < -\chi \times$ count then
      $t(i - M \dots i - 1) \leftarrow$ set
    end if
    if NOT $y(i)$ then
      $z \leftarrow z + x(i)$
      count $\leftarrow$ count $+1$
    end if
    if NOT $y(i - M)$ then
      $z \leftarrow z - x(i - M)$
      count $\leftarrow$ count $-1$
    **end if**
    $i \leftarrow i + 1$
  **end while**
  $y(0 \ldots N-1) \leftarrow t(0 \ldots N-1)$
```

We note that the algorithm is not as fast for all cases. In the case that many large sub-sequences need to be flagged, the statement in line 12 (which will actually extend into a loop) will iterate over all samples in the window to flag those, for each window position. To make the algorithm fast even in such cases, an extra variable can be added to register the most recently flagged sample. If this value is larger than the start of the window to be flagged, only samples after the most recently flagged sample need to be set. Adding such a variable makes the speed of the algorithm less dependent on the number of samples to be flagged, thus is probably recommended in most cases. It proves to be more difficult to solve this in the vectorized algorithm however.

When optimizing and profiling the LOFAR RFI detection pipeline, it was found that this `SumThreshold` algorithm was dominating the computation time of the RFI pipeline. Therefore, the algorithm was vectorized, which will be discussed in the next section.

2.3 Using SSE instructions for vectorization

We have implemented a vectorized version of the algorithm that can compute the `Sum-Threshold` over multiple sequences at once. In the case of RFI detection, this is very useful, as the algorithm needs to be applied on all time steps and frequency channels. The sizes of both these dimensions are typically at least on the order of thousands. While using SSE instructions is a very specific and less portable solution, the algorithms that we use are commonly executed on Intel cluster machines that provide these instructions, and recent CPU's all implement the SSE instruction set. To implement the SSE algorithm, we have used `gcc` intrinsics. These intrinsics are functions that map back to assembly instructions, but one does not need to think about registry allocations, etc., as that is still performed by the compiler. Moreover, unlike literal assembly code, the compiler is able to perform certain optimizations such as instruction pairing and loop unrolling.

Because the algorithm contains multiple branched statements, some care need to be taken when vectorizing the algorithm, as these need to be replaced by conditional moves. Another point of care is the use of booleans that are implemented with 1 byte and the use of floats of 4 bytes. This requires to combine both SSE instructions and 'regular' instructions, but because the SSE instructions have dedicated registers, efficiently sharing data between these requires some work. Because the SSE instruction set contains instructions that perform 4 computations at once, the vectorized algorithm can process 4 sequences at once. However, because of the overhead created by avoiding branching, we see about a 2–3 times speed-up over the normal algorithm. Newer processors also provide the Advanced Vector Extensions (AVX) instruction set, which can simultaneously process 8 floating point computations. The algorithm can easily be extended to AVX instructions, thereby further increasing its speed. However, this extension is only available in recent processors, and because of scarce availability this is not yet used in our implementation.

Listing 2: Vectorized algorithm of a `SumThreshold` iteration

```
Require: $\chi$ is the average value threshold for sub-sequences of size $M$,
    $\mathbf{x}(0 \ldots N-1)$ are 4 input sequences of $i < N$ values stored in a vector,
    $\mathbf{y}(0 \ldots N-1)$ are 4 masks generated by the previous iteration, stored in a vector.
Ensure: $\mathbf{y}(0 \ldots N-1)$ contain the output masks
    $i \leftarrow (0), \mathbf{z} \leftarrow (0,0,0,0), \mathbf{count} \leftarrow (0,0,0,0)$
    $\mathbf{t}(0 \ldots N-1) \leftarrow \mathbf{y}(0 \ldots N-1)$

    {calculate first window sum and count}
    while $i \neq M$ do
        {add sample to the right}
        $\mathbf{isnflagged} \leftarrow$ (NOT $\mathbf{y}(i)$) ? 0xFFFFFFFF : 0x0
        $\mathbf{z} \leftarrow \mathbf{z} + ((\mathbf{x}(i)\ \&\ \mathbf{isnflagged}) \mid (\mathbf{0.0}\ \&\ \neg\mathbf{isnflagged}))$
        $\mathbf{count} \leftarrow \mathbf{count} + (1\ \&\ \mathbf{isnflagged})$
        $i \leftarrow i + 1$
    end while

    {slide window over the data}
    while $i \neq N$ do
        {if threshold exceeded, set mask}
        $\mathbf{exceedsThreshold} \leftarrow$
            $((\mathbf{z} > \chi \times \mathbf{count})$ OR $(\mathbf{z} < -\chi \times \mathbf{count}))$ ? 0xFFFFFFFF : 0x0
        if $\mathbf{exceedsThreshold} \neq (0,0,0,0)$ then
            $\mathbf{byteFlags} \leftarrow \mathbf{exceedsThreshold}$ ? set : unset
            for $s \in t(i-M \ldots i-1)$ do
                $\mathbf{t}(s) \leftarrow (\mathbf{t}(s) \mid \mathbf{byteFlags})$
            end for
        end if

        {add sample to the right}
        $\mathbf{isnflagged} \leftarrow$ (NOT $\mathbf{y}(i)$) ? 0xFFFFFFFF : 0x0
        $\mathbf{z} \leftarrow \mathbf{z} + ((\mathbf{x}(i)\ \&\ \mathbf{isnflagged}) \mid (\mathbf{0.0}\ \&\ \neg\mathbf{isnflagged}))$
        $\mathbf{count} \leftarrow \mathbf{count} + (1\ \&\ \mathbf{isnflagged})$

        {subtract sample to the left}
        $\mathbf{isnflagged} \leftarrow$ (NOT $\mathbf{y}(i-M)$) ? 0xFFFFFFFF : 0x0
        $\mathbf{z} \leftarrow \mathbf{z} - ((\mathbf{x}(i)\ \&\ \mathbf{isnflagged}) \mid (\mathbf{0.0}\ \&\ \neg\mathbf{isnflagged}))$
        $\mathbf{count} \leftarrow \mathbf{count} - (1\ \&\ \mathbf{isnflagged})$
        $i \leftarrow i + 1$
    end while
    $\mathbf{y}(0 \ldots N-1) \leftarrow \mathbf{t}(0 \ldots N-1)$
```

The vectorized version follows the presented scalar version of the algorithm and is given in Listing 2. In the algorithm, some operators are applied on vectors. These operands and their corresponding symbols are add (+); subtract (-); bitwise or (|); bitwise and (&); and conditional move (x ← test ? a : b). When applied on vectors, these symbols denote the element-wise operations. The following SSE intrinsics are used to implement the vectorized algorithm:

_mm_set_ps : sets a vector to constant float values. Returns (a, b, c, d).

_mm_set_epi32 : similar as _mm_set_ps, but for constant integer values.

_mm_set1_ps : sets all values in a vector to one constant value. Returns (a, a, a, a).

_mm_load_ps : loads a vector from (non-constant) values in memory. Returns (a, b, c, d).

_mm_cmpeq_epi32 : element-wise compare of two integer vectors and return all bits set if equal, or all bits unset otherwise. Returns:
$(\mathbf{y} = \mathbf{z})$? 0xFFFFFFFF : 0x0.

_mm_cmpgt_ps and **_mm_cmplt_ps** : element-wise compare of float vectors similar to _mm_cmpeq_epi32, but for "greater than" and "less than" comparisons.

_mm_and_ps and **_mm_or_ps** : bitwise `and` and bitwise `or` between two vectors. Return $(\mathbf{x}$ & $\mathbf{y})$ and $(\mathbf{x} \mid \mathbf{y})$ respectively.

_mm_andnot_ps : bitwise and of a vector with the bitwise negation of another vector. Returns $(\mathbf{x}$ & $\neg\mathbf{y})$.

_mm_add_ps, **_mm_sub_ps** and **_mm_div_ps** : element-wise add, subtract and divide two float vectors.

_mm_cvtepi32_ps : convert integer vector to float vector.

_mm_movemask_ps : creates a 4 bit mask from the most significant bits of a float vector. This allows to store the result of a vector comparison into a single word, that can be used in regular (non-SSE) instructions.

From the vectorized algorithm, we can extract three essential sub-operations: adding a sample at the right of the window to the window (lines 7–9 and 25–27), subtracting a sample at the left from the window (lines 30–32) and testing the current window and setting the corresponding mask if necessary (lines 16–22). The other statements provide the loops and the initialization, and are trivial to implement.

In listing 3, a SSE algorithm is given in the C++ language, that adds a sample to the window. The subtraction can be implemented similarly.

Listing 3: Adding samples to windows with SSE instructions

Requires:
`rowFlagPtr`: a `const bool*` pointer to an array of flags, such that `rowFlagPtr`[x] with $0 \leq x < 4$ is the flag for window x.
`rowValPtr`: a `const float*` pointer to an array of samples, similar to `rowFlagPtr`.
`zero4i`: a `__m128i` vector containing zeros.
`ones4`: a `__m128i` vector containing ones.
`count4` : the number of samples in the windows (integer `__m128i` vector).
`sum4` : the sum of the unflagged samples in the windows (float `__m128` vector).

```
// Assign each integer in the vector to one bool in the mask
// Convert true to 0xFFFFFFFF and false to 0
__m128 conditionMask = _mm_castsi128_ps(
    _mm_cmpeq_epi32(_mm_set_epi32(rowFlagPtr[3], rowFlagPtr[2],
                                  rowFlagPtr[1], rowFlagPtr[0]),
                    zero4i));

// Conditionally increment counters
count4 = _mm_add_epi32(count4,
    _mm_and_si128(_mm_castps_si128(conditionMask), ones4));

// Add values with conditional move
__m128 m = _mm_and_ps(_mm_load_ps(rowValPtr), conditionMask);
sum4 = _mm_add_ps(sum4, _mm_or_ps(m,
    _mm_andnot_ps(conditionMask, zero4)));
```

The remaining algorithm to threshold the window and output the flags is given in Listing 4. This part interchanges between using the C++ boolean type of 1 byte and SSE masks, and therefore applies some tricks to convert between the two, as well as to "or" 4 booleans at a time.

As discussed, due to the `for` loop that sets the flags and that might be executed for each window position, the algorithm is optimized for a low number of positives. Nevertheless, the loop is fast, as it consists of one statement that is not a floating point operations. Therefore, even in cases where a lot of windows need to be flagged, the loop will not excessively slow down the algorithm.

When calculating the average, the `count4` variable is not tested for zero. Therefore, calculating the average might perform a division by zero. However, this can be ignored, since the outcome is not important if all samples are already flagged.

Listing 4: Thresholding the windows with SSE instructions

Requires:
Function `outputMask->RowFlagPtr(x, y)`: returns a `bool*` pointer to an array of flags, such that `RowFlagPtr`(x, y) with $0 \leq x < 4$ is the y-th output flag for window x, ordered in x direction.
`threshold4Pos`, `threshold4Neg`: positive and negative thresholds, χ and $-\chi$.
`count4` : the number of samples in the windows (integer `__m128i` vector).
`sum4` : the sum of the unflagged samples in the windows (float `__m128` vector). `M` : tested sub-sequence size, hence the number of samples in the window.

```
// if sum/count > threshold || sum/count < -threshold
__m128 avg4 = _mm_div_ps(sum4, _mm_cvtepi32_ps(count4));
const unsigned flagConditions =
  _mm_movemask_ps(_mm_cmpgt_ps(avg, threshold4Pos)) |
  _mm_movemask_ps(_mm_cmplt_ps(avg, threshold4Neg));

// The assumption is that most of the values are actually not
// thresholded. If this is the case, we circumvent the whole loop
// at the cost of one extra comparison:
if(flagConditions != 0)
{
  union
  {
    bool theChars[4];
    unsigned theInt;
  } outputValues = { {
    (flagConditions&1)!=0,
    (flagConditions&2)!=0,
    (flagConditions&4)!=0,
    (flagConditions&8)!=0 } };

  for(size_t i=0;i<M;++i)
  {
    unsigned *outputPtr = reinterpret_cast<unsigned*>(
      outputMask->RowFlagPtr(x, yTop + i));

    *outputPtr |= outputValues.theInt;
  }
}
```

Our implementation is written in C++ and uses a template variable for M. By doing so, the compiler creates a specialized version of the algorithm for each M value, and this allows the compiler to fully unwrap the two loops with M limits. The gcc compiler will only do this if the optimization parameter `-funwrap-loops` is specified on the command line. Specifying this compiler option will also partially unwrap the main loop, which speeds up the implementation considerably, because the compiler is now free to optimize between loop iterations and perform an optimization technique called instruction pairing.

Different machine architectures give different speed ups, but we generally see a factor 2–3 increase. Apart from the `-funwrap-loops` option, we also specify `-march=native` to enable the compiler to use any instruction set which the host computer provides. Memory that is used in the `SumThreshold` SSE implementation need to be aligned on 16 byte boundaries. On certain architectures, notably Apple machines, memory return from `malloc()` is already aligned correctly, but other architectures require the use of the `posix_memalign()` function instead.

3 Discussion & conclusions

We have shown how the feature detection accuracy of the `SumThreshold` method can be slightly decreased to create a fast implementation. Using exponentially increasing tested subsequence sizes decreases the time complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$, and limiting the subsequence size decreases the time complexity to $\mathcal{O}(N)$. Our final implementation performs about 10 passes over the data — one for each tested subsequence size — and each pass requires a few floating point calculations per sample.

Using SSE instructions gave another factor of 2–3 increase in speed, without compromising the accuracy. However, such optimizations come at the expense of limiting its portability and increasing its implementation complexity. It is therefore only useful for the most time-critical parts of the software. Some algorithms might get an additional improvement from using GPUs. The down-side of GPUs is, that they require stricter memory access patterns, are not good at branched code and programming them is in the author's opinion somewhat more complex than using SSE instrinsics. GPUs that are efficient at scientific calculations, such as the NVIDIA Tesla machines, are also generally less available. In the LOFAR case, the central processing cluster does not provide them, thus they were not an option. Also astronomers that run the software at home probably have less benefit from GPU code. On the other hand, in the last few months the SSE implementation has processed all LOFAR recorded imaging observations on the LOFAR cluster, and has caused no issues. The SSE implementation is also shipped for some time in the latest AOFlagger package, and no problems have been reported from its users. Using AVX instructions is an attractive future improvement that might give another factor of 2 increase. However, this instruction set is very new and not yet generally available.

References

A. R. Offringa, J. J. van de Gronde, and J. B. T. M. Roerdink. A morphological algorithm for improved radio-frequency interference detection. *A&A*, 539, March 2012.

A. R. Offringa, A. G. de Bruyn, M. Biehl, S. Zaroubi, et al. Post-correlation radio frequency interference classification methods. *MNRAS*, 405:155–167, June 2010a.

A. R. Offringa, A. G. de Bruyn, M. Biehl, and S. Zaroubi. A LOFAR RFI detection pipeline and its first results. *Proc. of RFI2010*, March 2010b.