The effective algorithm for constant state detection in time series.

Andrei Keino

February 22, 2025

Abstract

The article introduces a simple and effective algorithm for constant state detection in the time series. The algorithm, based on the sliding window of variable length, finds all the sections of the time series which have length greater or equal than some given minimal length and have difference between maximal and minimal value in the section less or equal than some given value. It is shown that the computational complexity of the aforementioned algorithm is $\mathcal{O}(N \log N)$, where N is the length of time series.

Introduction.

Some technical applications of signal processing require the steady state detection in time series. This problem has been investigated in many articles. For example, in [3] described a computationally efficient method for identification of steady state in time series data. But there could exist applications for straightforward method, described in this article. The proposed algorithm was described already in [1] , but the algorithm description in this article seems not to be clear enough. The aim of the present article to give a clear description for the presented algorithm for constant state detection in time series.

Author introduces a very basic and simple algorithm for constant state detection, based on the sliding window with variable length, which have good computational complexity. The aim of this article is to propose a simple and effective algorithm for identifying the nearly - constant sections in time series.

The introduction to the algorithm.

The algorithm described in the article keeps count of all the unique values for the moving window in the associative array (AA) structure [2] and updates these values and their count while moving and resizing the sliding window. The keys in AA are the unique values in the moving window, the value which corresponds to the key is the count of this unique value (key value) in the moving window. After every movement or resizing of the moving window, the algorithm takes minimal and maximal key values in the AA and checks the difference between them. If this difference is less or equal to the minimal window height required, the algorithms append the beginning and ending indexes of the window to the array of detected accepted window positions. As AA have computational complexity of lookup, deletion, insertion $\mathcal{O}(logM)$, and on each step for every point in the time series we have at most two lookups, one insertion and one deletion in our AA, this is already evident that the algorithm have $\mathcal{O}(N \log N)$, complexity in the worst case, where N is the length of time series.

The description of the axiliary functions.

Let us intoduce some axiliary functions (pseudocode):

Function add_value_to_map.

This function adds value to the moving window. If the new value exists already in the window, it increments the value count, in other case it adds the new value as a key to AA and set its value count to 1.

```
// function add_value_to_map adds value to our AA
// i.e. to the associative array
add_value_to_map(win_map, val_to_add)
{
        // here <win_map> is the AA;
        // <val_to_add> - value to add to AA
        // add to the AA the value corresponding
        // to the value of <val_to_add>
        // complexity is the O(log(win_map.size())
        key = win_map.find_key(val_to_add);
        if (key.exists())
        {
                key.value += 1;
        }
        else
        {
                key = win_map.add_key(val_to_add);
                key.value = 1;
        }
}
```

Function remove_value_from_map.

This function removes value from the moving window. It finds the key that corresponds to the value to remove from AA and decrements its counter. After that, if the counter is equal to zero (i.e., there is no such value in the moving window left), the key is removed from AA.

```
// function remove_value_from_map removes value val_to_remove
// from our AA i.e. from the associative array
remove_value_from_map(win_map, val_to_remove)
{
        // here <win_map> is the AA; <val_to_remove> - value
        // to femove from AA
        // remove from the AA one value corresponding
        // to the value of <val_to_remove>
        // complexity is the O(log(win_map.size())
        key = win_map.find_key(val_to_remove);
        if (key.exists())
        {
                key.value -= 1;
                if key.value <= 0)
                {
                        win_map.remove(key);
                }
        }
        else
        {
                // something went wrong
```

```
                raise exception("remove_value_from_map: no such value");
        }
}
```

Function append_bounds.

The simplest version of this function appends found moving window start and finish indexes in time series to the arrays of start indexes and finish indexes correspondingly. In more sophisticated version of this function we can do some checks to get the better results of the constant state detection.

```
// function append_bounds(start, finish, start_idx, end_idx)
// appends the boundary indexes for the new section
// The new constant section  was found,
// appending the boundary indexes for the new section.
append_bounds(start, finish, start_idx, end_idx)
{
        // append the steady section bounds to the output arrays:
        // <start> and <finish> are arrays to hold
        // the start and finish position of the constant section

        // <start_idx>, <end_idx> are start and end indexes (constant section bounds)
        // to append to the output arrays
        // appending the start and end points

        start.append(start_idx);
        finish.append(end_idx);
}
```

The algorithm description.

Let we have a time series $y[i]$, $y \in R$, $i = \{1, \ldots, N\}$, where N is the time series size. Let us specify the minimal window acceptable length L_W and its maximal acceptable height H_W.

As we can see from the pseudocode below, at the every step of algorithm AA contains only the keys for all the values that exist in the moving window. So, to find the maximal and minimal values in the moving window, it's enough to get the maximal and minimal key values in AA and as we know, that operation in the associative array has $\mathcal{O}(\mathbf{1})$ computational complexity.

The proposed algorithm works as follows (pseudocode):

```
// H_W is the maximum window acceptable height
// L_W is the the minimal window acceptable length
win_height = H_W
win_map = AA();
win_len = L_W
start = Array();
finish = Array();

// initialize the window of length L_W
```

```
for i = 0..win_len - 1
{
        add_value_to_map(win_map, y[i]))
}

start_idx = 0;
end_idx = win_len - 1;


while (1)
{
        max = win_map.max_key();
        min = win_map.min_key();

        if ((max - min) > win_height)
        {
                if (end_idx - start_idx + 1 > win_len)
                {
                        // moving the left boundary of the moving window
                        // one point to the right
                        remove_value_from_map(win_map, y[start_idx]);
                        start_idx += 1;
                }
                else
                {
                        // moving the right and left boundaries of the moving window
                        // one point to the right
                        if (end_idx >= y.size() - 1)
                        {
                                // the end of time series is reached,
                                // stopping the algorithm
                                break;
                        }
                        remove_value_from_map(win_map, y[start_idx]);
                        start_idx += 1;
                        end_idx += 1;
                        add_value_to_map(win_map, y[end_idx]);
                }
        }
        else // case ((max - min) <= win_height)
        {
                append_bounds(start, finish, start_idx, end_idx);
                // moving the right boundary of the moving window
                // one point to the right
                end_idx += 1;
                if (end_idx >= y.size())
                {
                        // stop the cycle
                        break;
                }
                // add value to a map
```

```
                add_value_to_map(win_map, y[end_idx]);
        }

} // end of cycle while (1)
```

The algorithm complexity.

Theorem

Complexity of aforementioned algorithm is $\mathcal{O}(\mathbf{N}\log\mathbf{N})$.

Proof:

Complexity of lookup, deletion, insertion, of key - value pair in AA is $\mathcal{O}(logM)$, where M is the count of items in AA [2]. As AA is ordered, the complexity of finding key maxima and minima is $\mathcal{O}(1)$. The complexity of appending the boundaries is $\mathcal{O}(1)$ also. Let $N^{(look)}$ and $C^{(look)}$ are the number of lookups for the key value in AA and some constant value respectfully, $N^{(del)}$ and $C^{(del)}$ are the number of deletions from AA and some constant value respectfully, $N^{(ins)}$ and $C^{(ins)}$ are the same for operation of insertion in AA, $M_i^{(AA)}$ is the i - th count of keys in AA, $N^{(op)}$ is the overall number of operations.

Then, neglecting the operations with complexity $\mathcal{O}(1)$, we have:

$$\begin{aligned} N^{(op)} &\leq \sum_{i=1}^{N^{(look)}} C^{(look)}\log(M_i^{(AA)}) + \sum_{i=1}^{N^{(del)}} C^{(del)}\log(M_i^{(AA)}) + \sum_{i=1}^{N^{(ins)}} C^{(ins)}\log(M_i^{(AA)}) \\ &\leq \left(2C^{(look)} + C^{(del)} + C^{(ins)}\right) N\log(N) \end{aligned}$$

The last line of the upper equation means that the complexity of the presented algorithm is $\mathcal{O}(N\log N)$.

Some results.

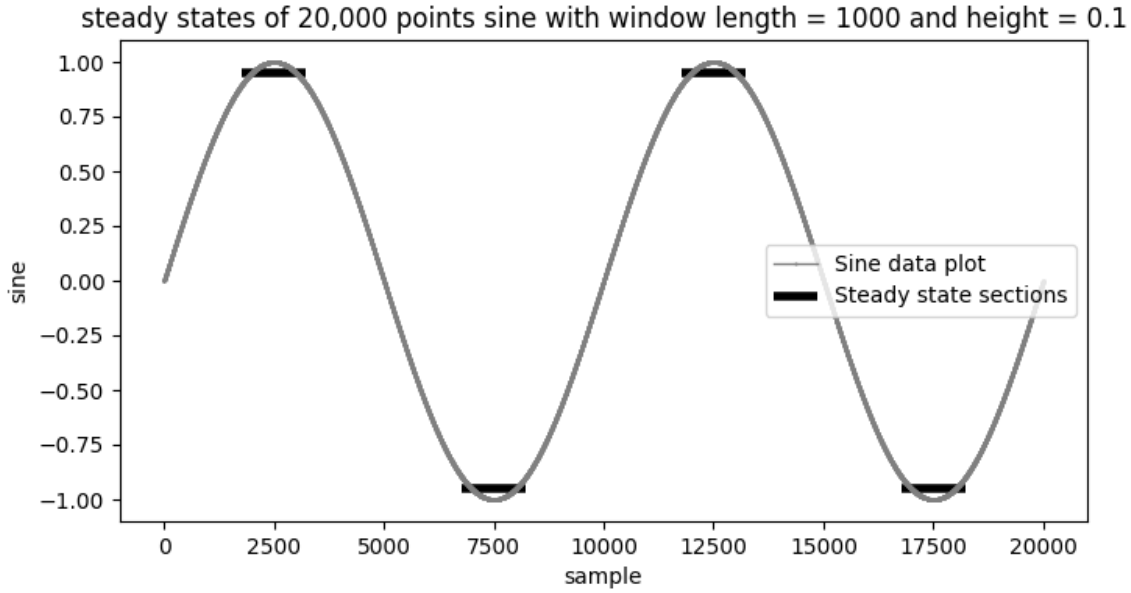

Figure 1: Results of steady state detection for sine time series. Number of sine periods is 2, number of points in time series is 20,000, minimal window length is 1,000, maximal window height is 0.1.

Discussion.

As it's easy to suppose, the sections found can overlap. This issue can be resolved by post-processing the sections found, with the computational complexity $\mathcal{O}(N)$. Also, this issue can be partially resolved by enhancing the function

```
append_bounds(start, finish, start_idx, end_idx)
```

The enhanced version of this function was used for obtaining results shown in Figure 1. For some post-processing techniques, online versions of the described algorithm with the same computational complexity $\mathcal{O}(\mathbf{N} \log \mathbf{N})$ can be implemented.

References

[1] Andrei Keino. Simple and effective algorithm for constant state detection in time series. URL: `https://vixra.org/abs/2110.0094`.

[2] Associative arrays. URL: `https://en.wikipedia.org/wiki/Associative_array`.

[3] Øyvind Øksnes Dalheim; Sverre Steen. A computationally efficient method for identification of steady state in time series data from ship monitoring. URL: `https://www.sciencedirect.com/science/article/pii/S2468013320300103`.