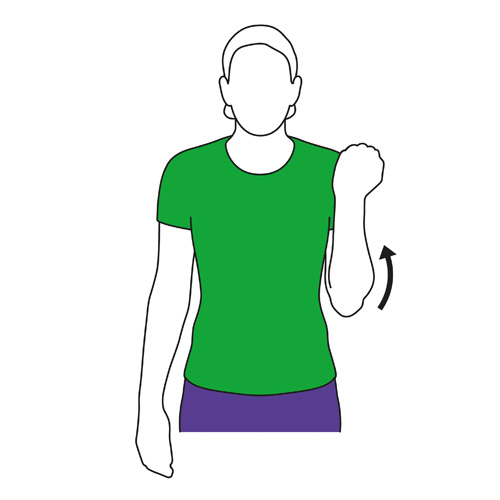


Elbow flexion exercise



# Introduction

In this document an example dataflow diagram is constructed that monitors the flexion of an elbow and counts the number of cycles, i.e. the times the elbow goes from below 10° to above 120°.

It also compares the so-called *movement regularity* by storing the first cycle as a reference and then determining the *deviation* with every subsequent cycle.

This functionality is implemented by first creating a data flow diagram that calculates the joint angle between the upper and lower arm.

In “[*Threshold detection*](#_lcbu651xf57y)” the diagram is then extended to determine whether the joint angle becomes higher than one of the thresholds.

In “[*Cycle detection*](#_vbrfctaqf2pu)” it is then described how this threshold detection can be used to determine the beginning and end of a cycle. This is then used to create a sequence of data-points and count the number of cycles (“[*Generating cycles*](#_p36mah6j5iqo)”).

To be able to compare cycles, i.e. comparing the resulting sequence of data-points, a sequence comparator is constructed in “[*Comparing cycles*](#_wm7o05rwhv77)” which calculates the deviation between two sequences.

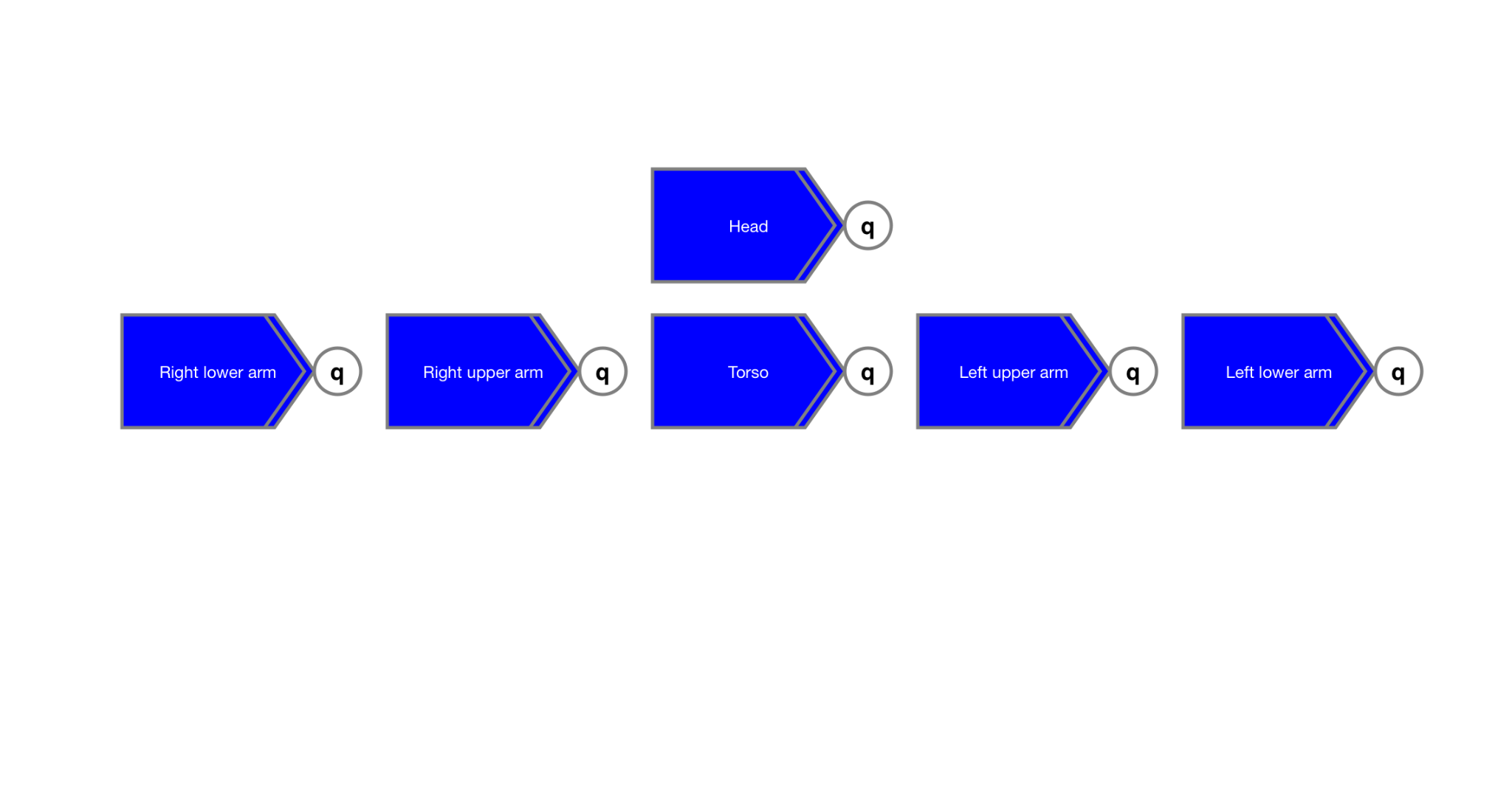
In “[*Storing the first cycle*](#_k18lxp55om38)” the diagram is created which stores the first sequence in order to use it as input for the sequence comparator to compare it with subsequent sequences.

An experiment was performed where motion tracking sensors were attached to the segments of the left arm. The results are given in the chapter “[*Results*](#_5a4u8c50iojl)”.

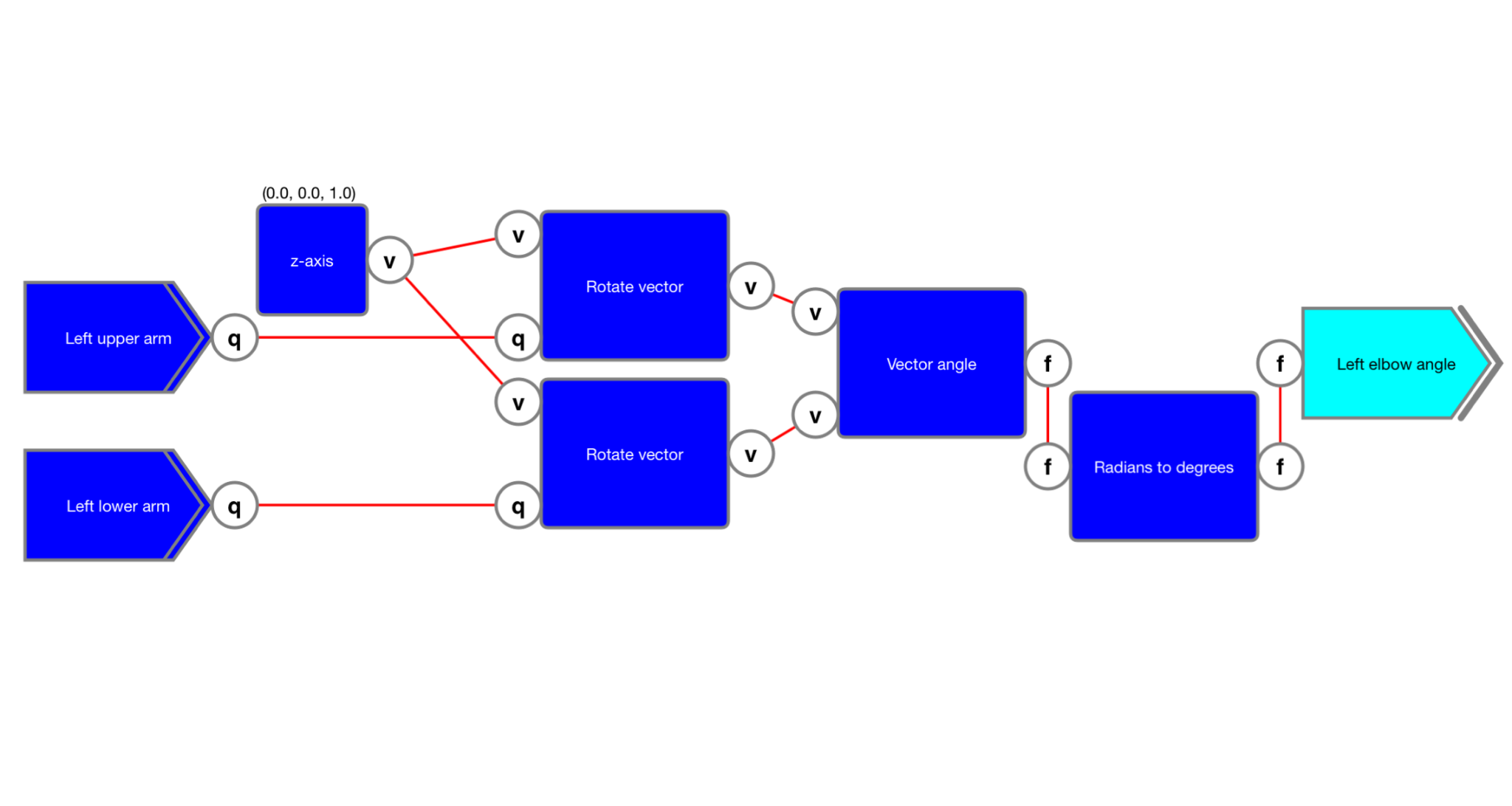
The diagram created in this document can be downloaded via this [link](https://drive.google.com/file/d/1azuf16BJvyqgp58DzHJA4K2E4ZiuwLdT/view?usp=share_link).

# Elbow flexion

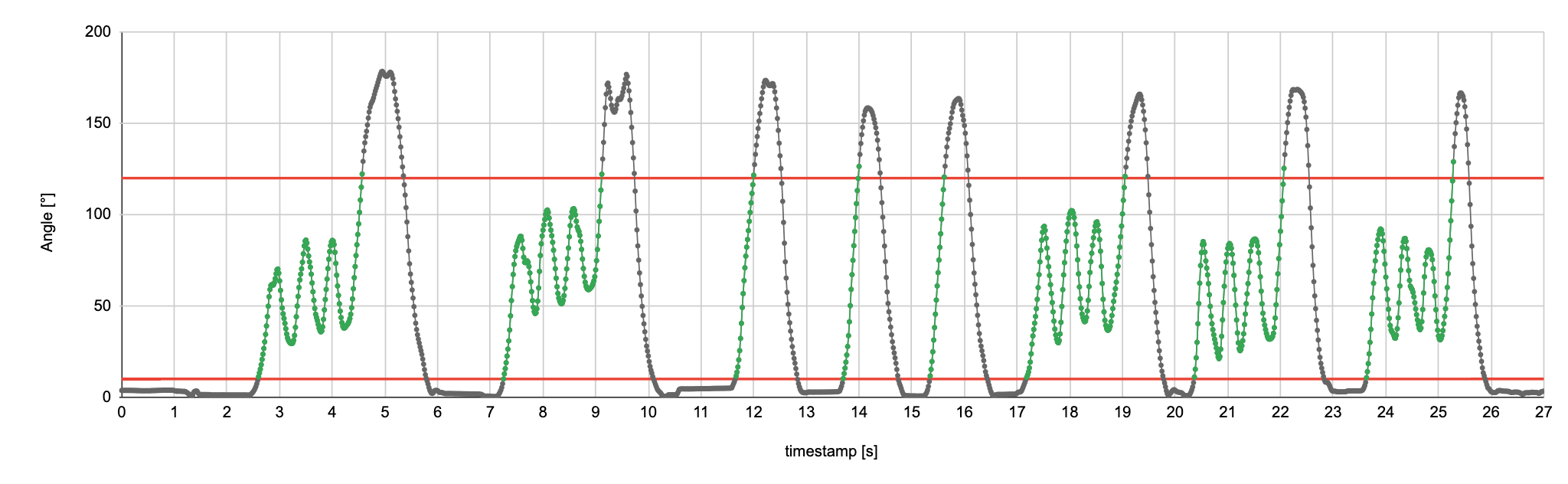
To be able to calculate the flexion, the first step is to add the input streams for the left arm. The torso is added to get a nicer visualization.

******[***Figure 1***](#figur_used_segments)***: Used segments.***

The *elbow angle* between the two segments of the left arm can now be calculated by determining the angle between the longitudinal axes of the segment as shown in [figure 2](#fig_elbow_angle).

******[***Figure 2***](#figur_elbow_angle)***: Calculating the elbow angle.***

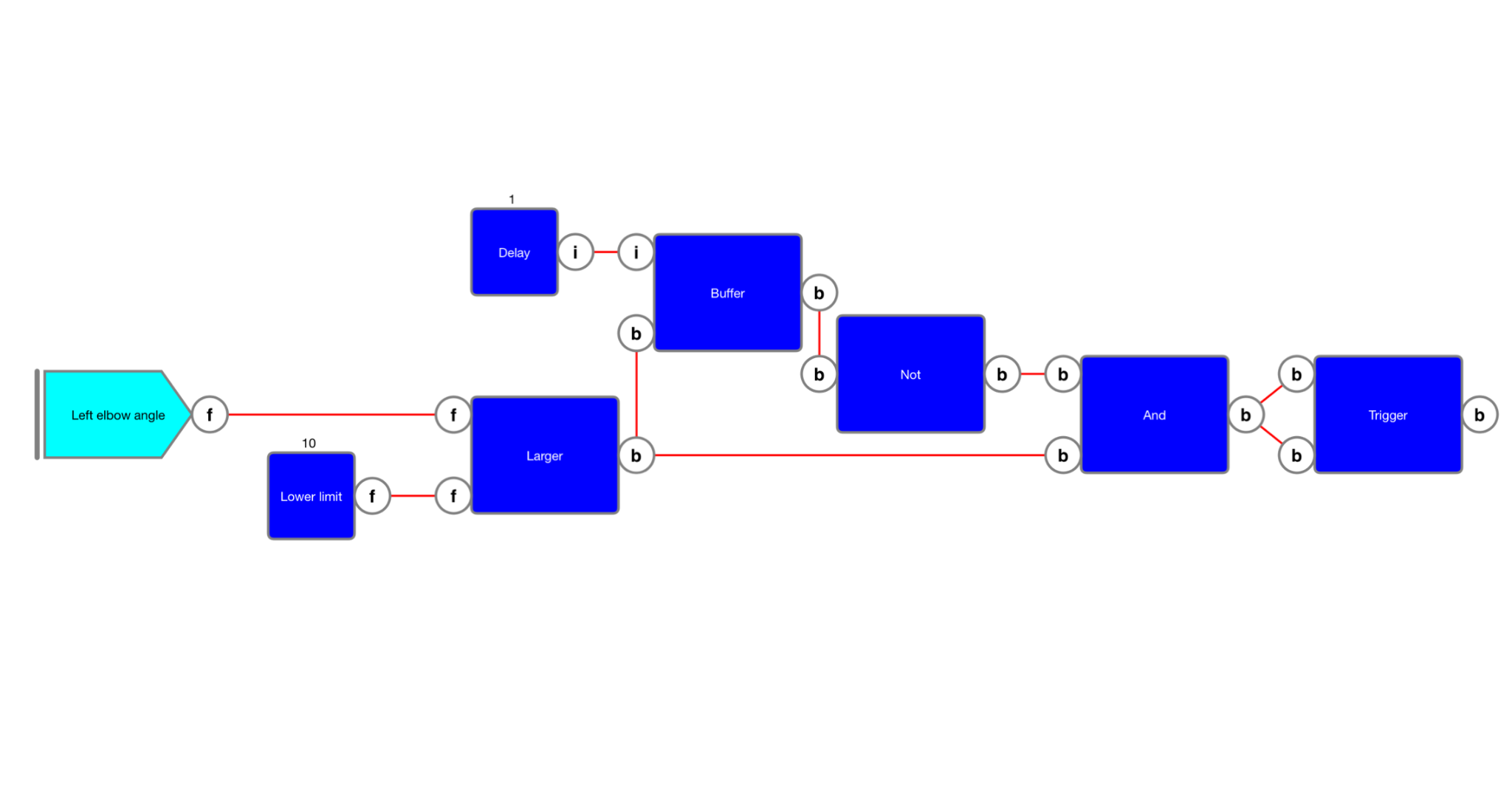
An experiment was performed where 60 Hz motion tracking sensors were attached to the segments of the left arm. The result is given in [figure 3](#fig_elbow_recording) together with the intended thresholds and areas for which the comparison must be performed. The dots in the plot indicate the data-points.

******[***Figure 3***](#figur_elbow_recording)***: Left elbow angle with thresholds.***

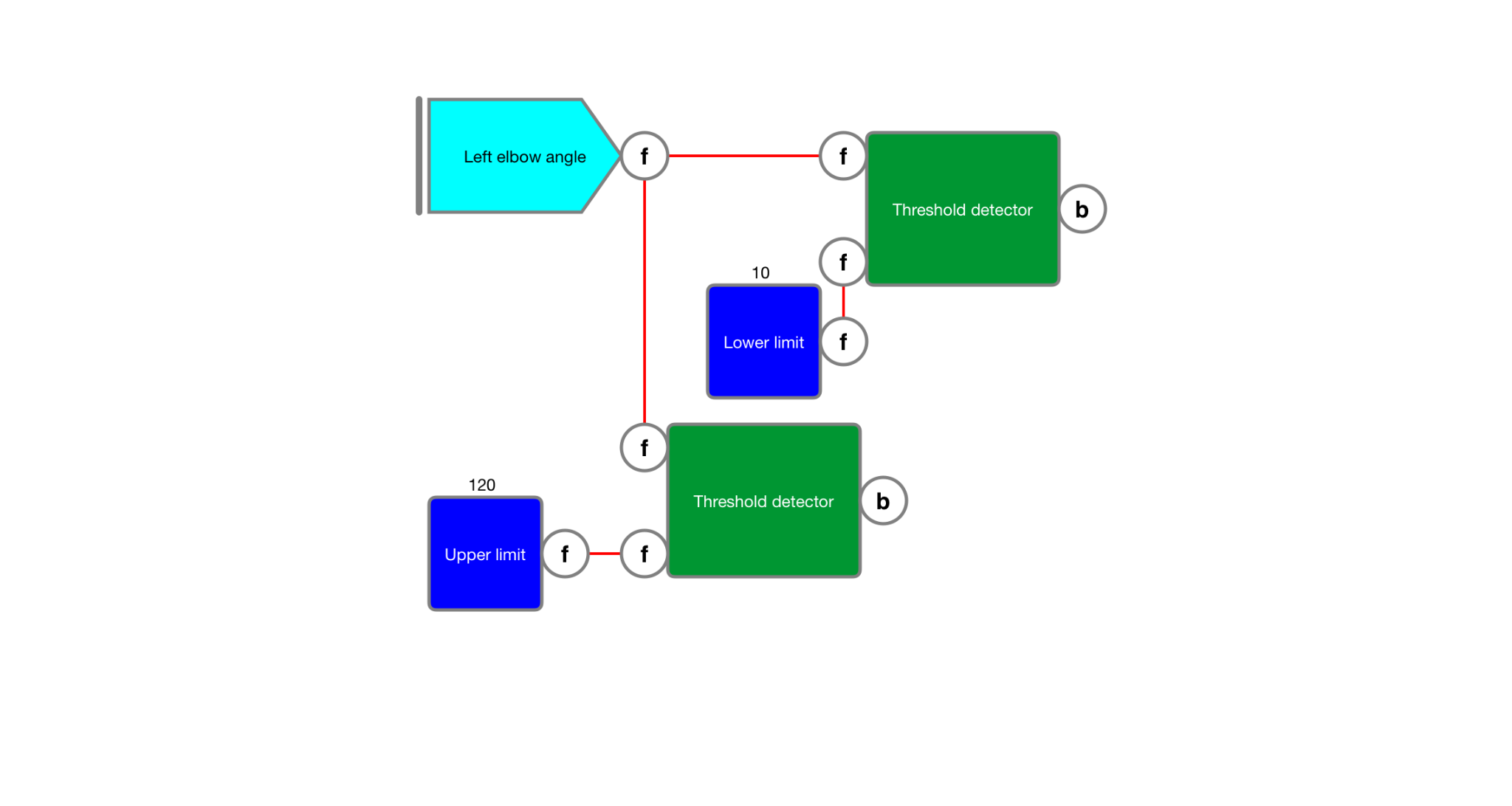
The recording can be downloaded via this [link](https://drive.google.com/file/d/18haGCKj-DjzdqfI9hzYewzDZu50yn8Qh/view?usp=share_link).

# Threshold detection

With the elbow flexion available as a joint angle in degrees, the next step is to extend the diagram with the functionality to detect when the joint angle becomes larger than 10°. This functionality is described in the document “[Threshold detection](https://docs.google.com/document/d/170yHOCNlUd9p27sxratVakVZhff4GVsfhZ5q8bSqI0Y/edit?usp=share_link)”. The sub-diagram to detect the lower threshold is given in [figure 4](#fig_lower_limit_th).

******[***Figure 4***](#figur_lower_limit_th)***: Threshold detection for lower limit.***

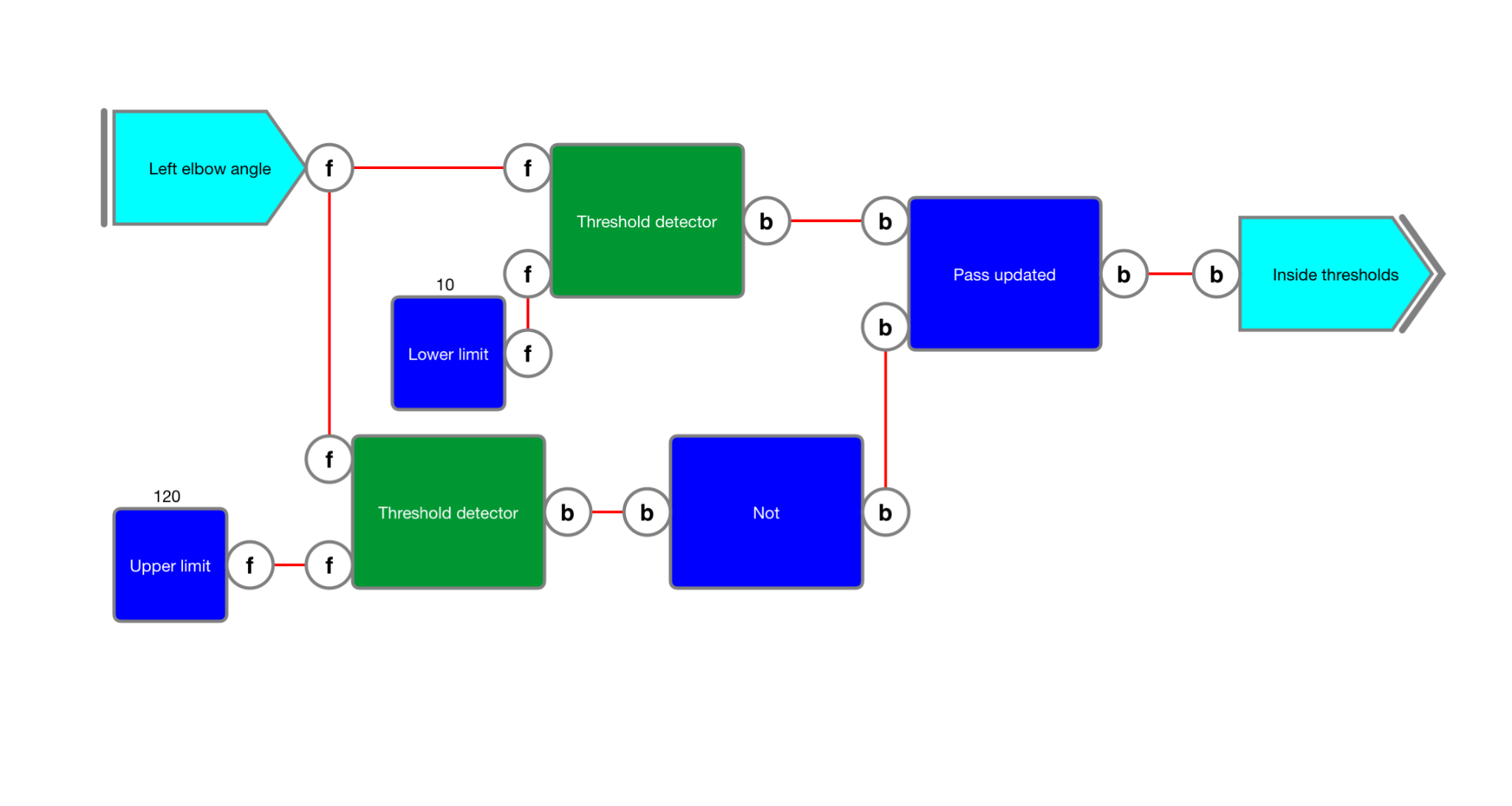
The functionality is collapsed into a single “*Threshold detector*” container and then copied to also detect when the elbow angle passes the upper limit of 120°. The resulting sub-diagram is given in [figure 5](#fig_threshold_detection).

******[***Figure 5***](#figur_threshold_detection)***: Threshold detection.***

In the next chapter the diagram is extended to detect a cycle.

# Cycle detection

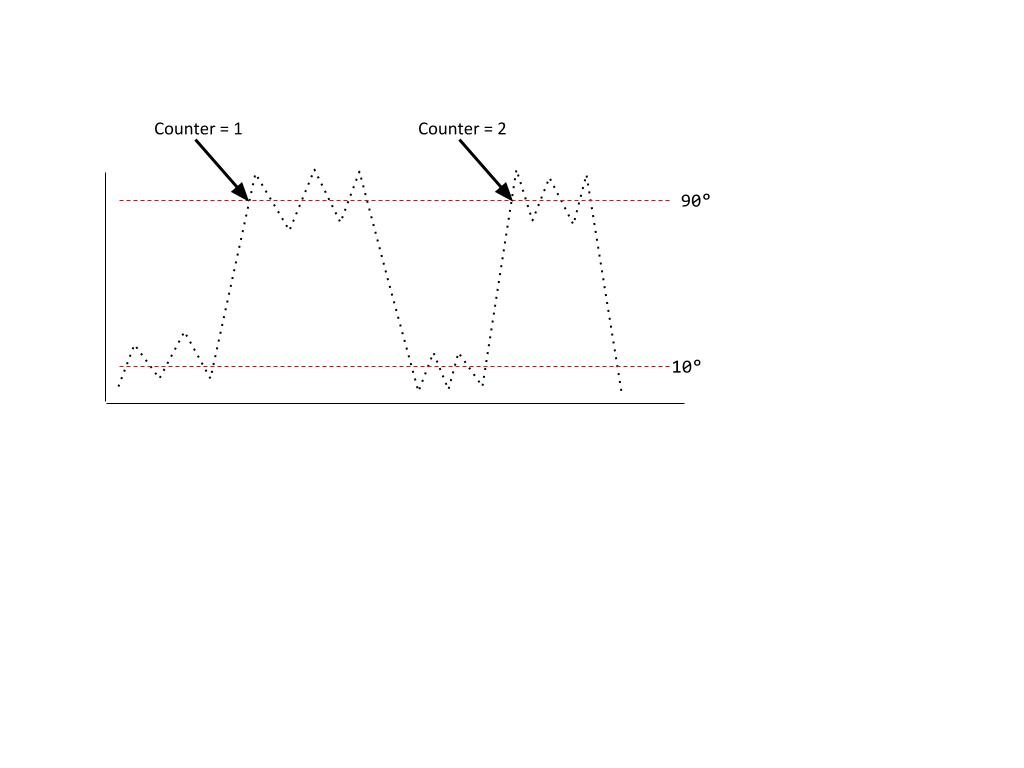
The sub-diagram created in the previous chapter can be extended to create a variable that becomes *true* when the lower limit is passed and returns to *false* when the upper threshold is passed.

******[***Figure 6***](#figur_inside_thresholds)***: Variable to indicate whether inside thresholds.***

The value of the "Inside thresholds" variable output for the measured elbow angles is given below. This can now be used to create a sequence as will be discussed in the next chapter.

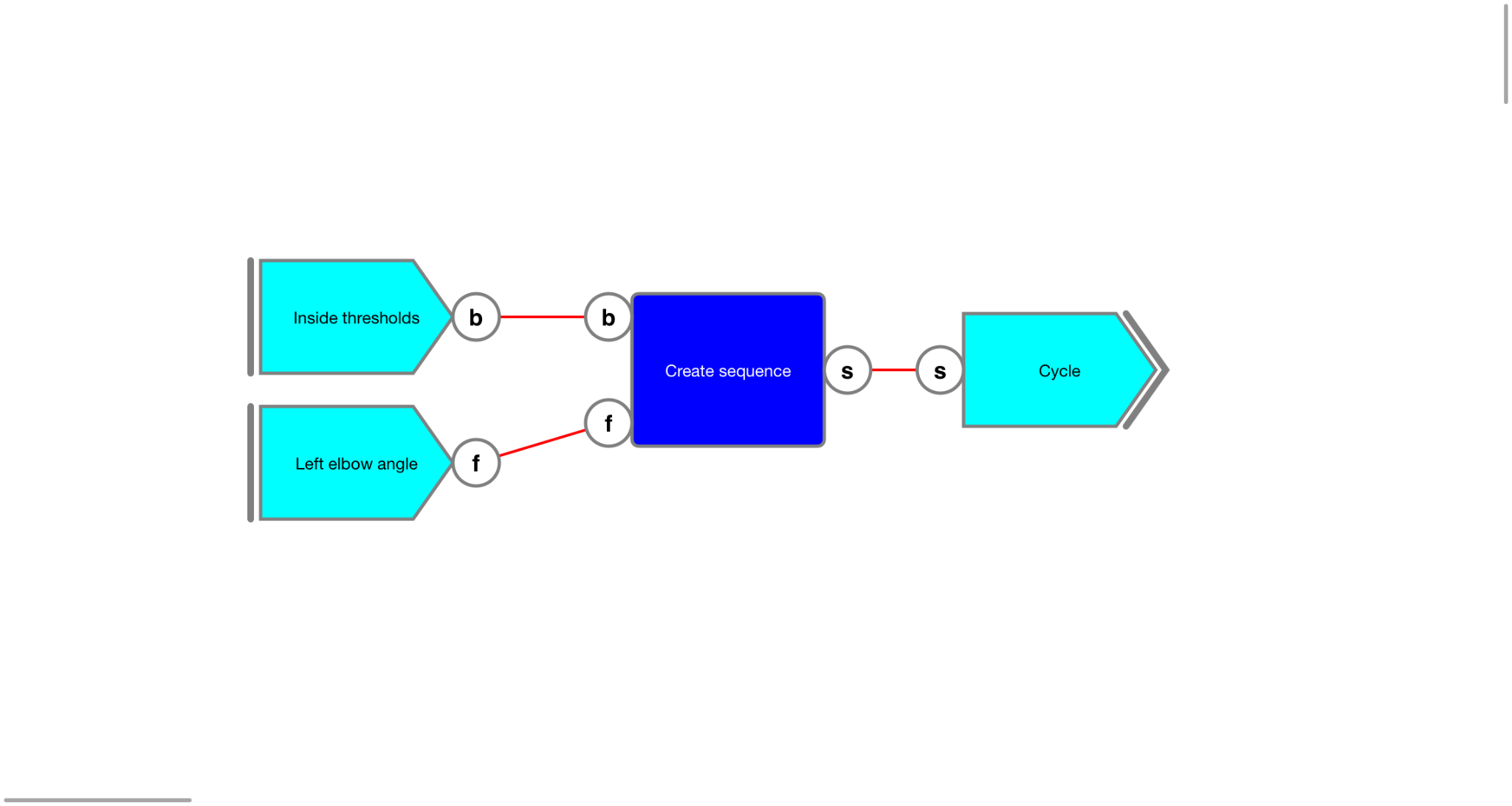
| Timestamp,Inside thresholds  2.600351999979466,True  4.567257999908179,False  7.250594999990426,True  9.116898999898694,False  11.667099999962375,True  12.000239999964833,False  13.700873999972828,True  14.000879999948665,False  15.350756999920122,True  15.617328999913298,False  17.183626999962144,True  19.050830999971367,False  20.36727399996016,True  22.067907999968156,False  23.634505999973044,True  25.284688999992795,False |
| --- |

Note that the implemented diagram also takes care of possible jitter where a boundary might be crossed multiple times before the other boundary is crossed, as illustrated in the [figure 7](#fig_jitter).

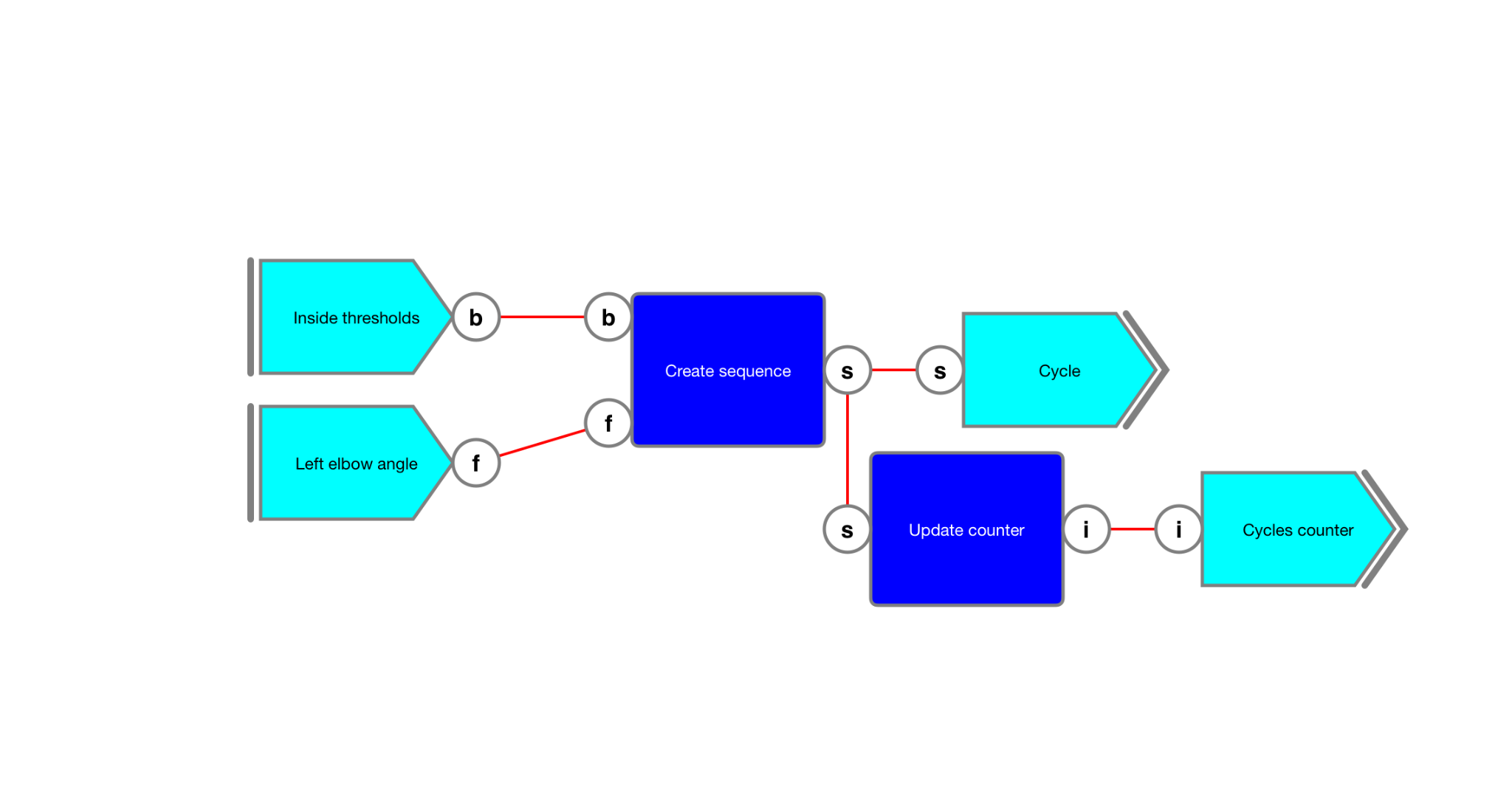
******[***Figure 7***](#figur_jitter)***: Threshold detector can handle jitter.***

# Generating cycles

To create a sequence, the corresponding function is used, which takes a boolean and float as input and generates the sequence when the elbow angle is inside the thresholds. In [figure 8](#fig_generating_cycle) the sub-diagram is given in which the elbow angles are stored after the boolean value is updated to *true*. This continues until the boolean value is updated to *false* after which a sequence output is updated resulting in an update of the "*Cycle*" variable output.

******[***Figure 8***](#figur_generating_cycle)***: Generating a cycle.***

Since a sequence is created at the end of each cycle, the update of the “Create sequence” can also be used to count the number of cycles by using the "Update counter" function as shown in the extended diagram in [figure 9](#fig_counting_cycles).

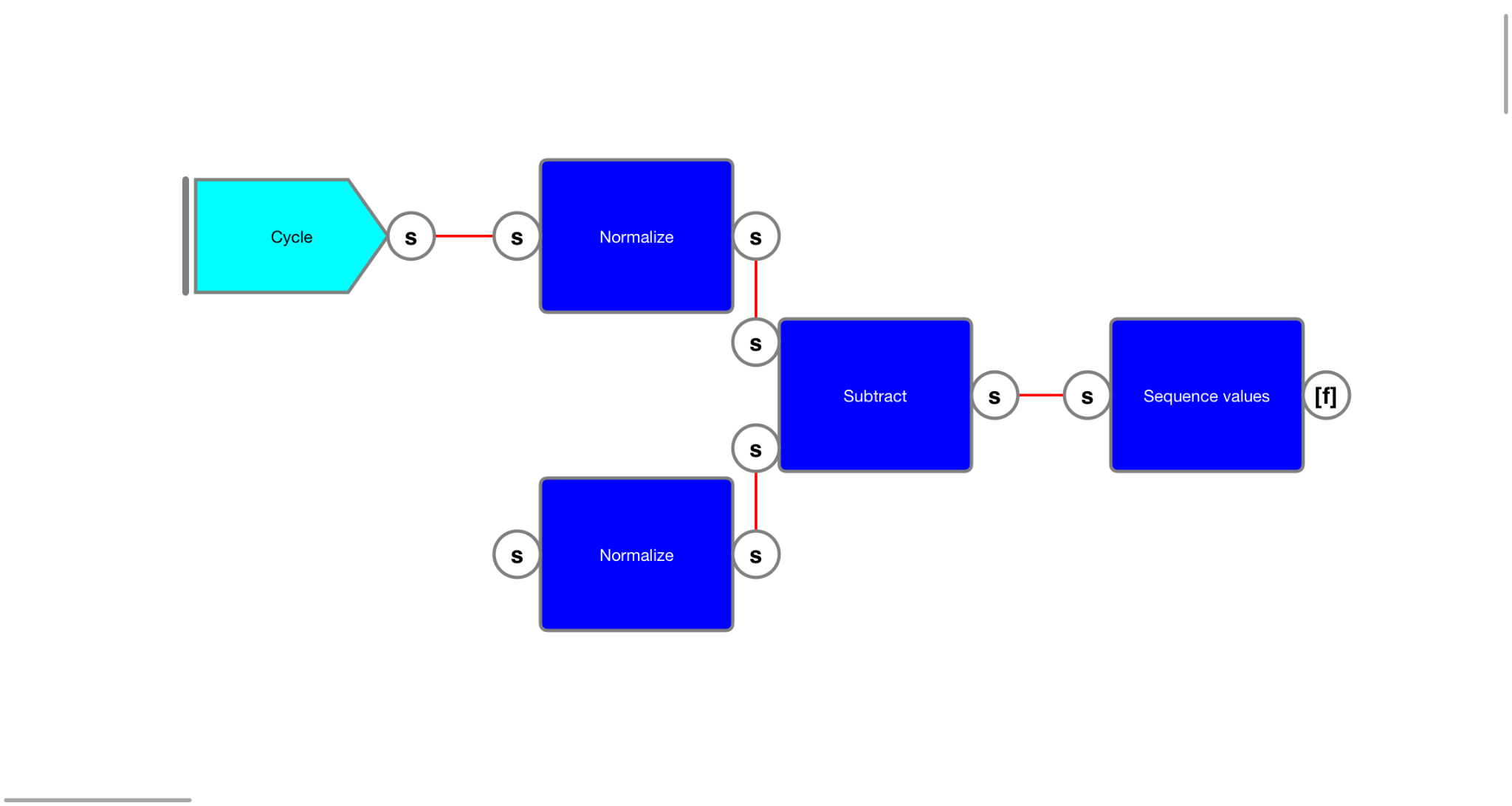
******[***Figure 9***](#figur_counting_cycles)***: Counting the created cycles.***

The expected result is given below.

| Timestamp,Cycles counter  4.567257999908179,1  9.116898999898694,2  12.000239999964833,3  14.000879999948665,4  15.617328999913298,5  19.050830999971367,6  22.067907999968156,7  25.284688999992795,8 |
| --- |

# Comparing cycles

To compare cycles, they must be normalized and then the difference can be simply calculated by using the “*Subtract*” function. This function will output a new sequence in which the samples are the result of subtracting the value of the first sequence from the interpolated value of the other sequence at the same timestamp. After this, we’re interested in processing the resulting values which can be retrieved as an array using the “*Sequence values*” function. The sub-diagram is shown in [figure 10](#fig_cycle_diff). The other input of the "*Subtract*" function is not determined yet as that will be the first cycle as will be discussed later in "[*Storing the first cycle*](#_k18lxp55om38)".

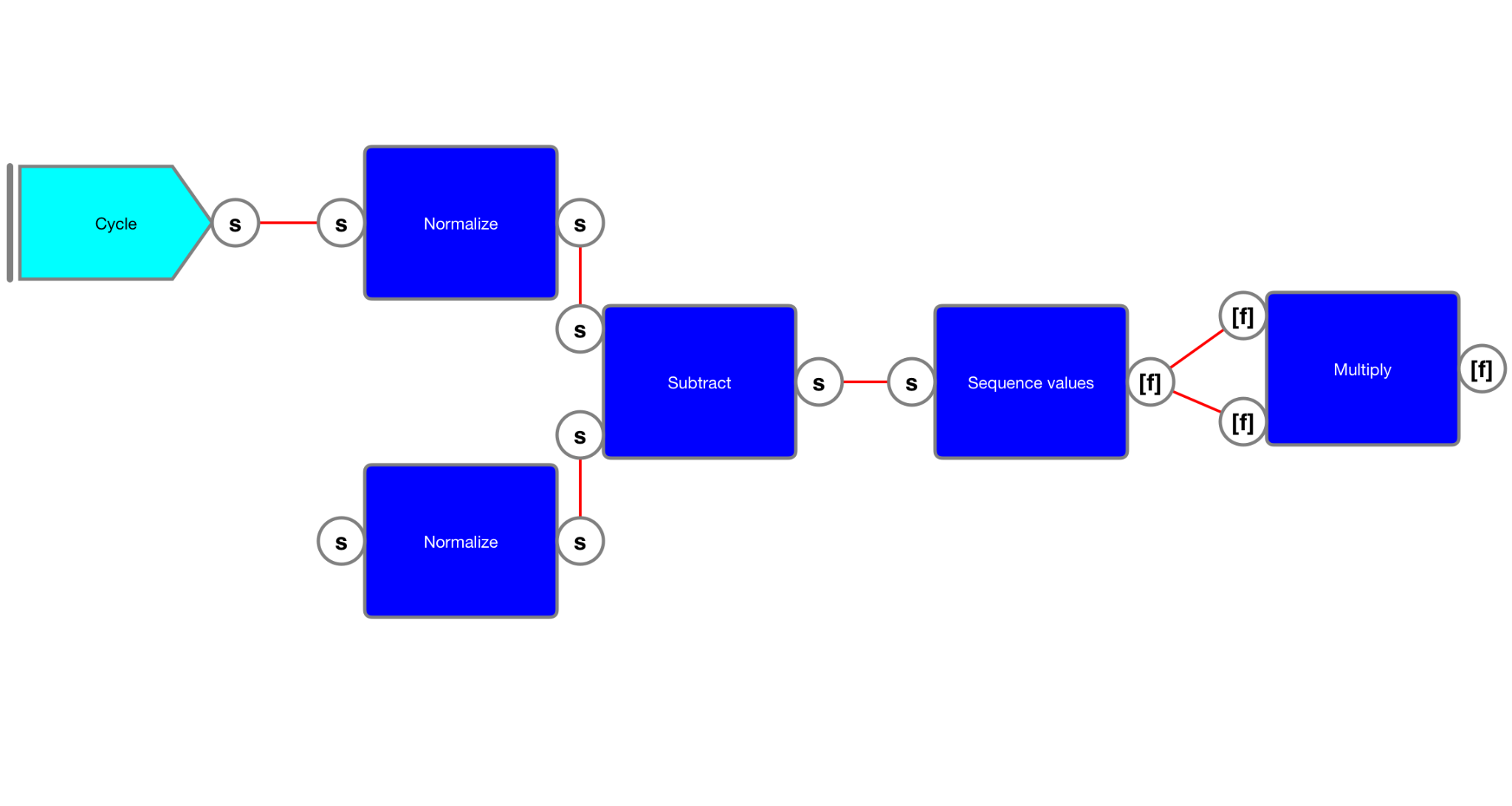
******[***Figure 10***](#figur_cycle_diff)***: Obtaining the difference between two sequences.***

In [estimation theory](https://en.wikipedia.org/wiki/Estimation_theory), the [root-mean-square deviation](https://en.wikipedia.org/wiki/Root-mean-square_deviation) (RMS) of an estimator is a measure of the imperfection of the fit of the estimator to the data. For a set of error values *xi* the RMS is specified by the following equation.

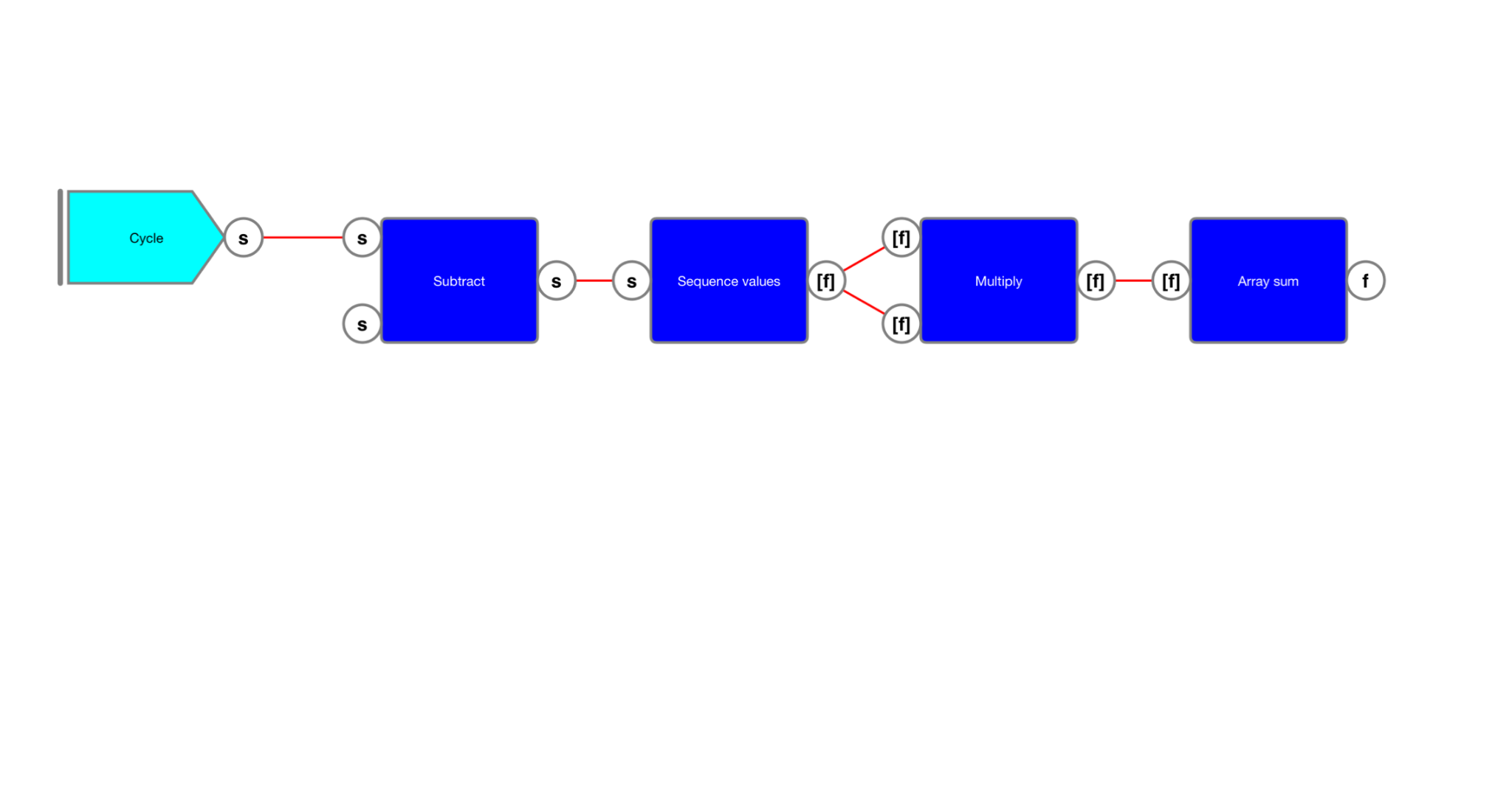
Taking a float array as input, the equation can be implemented in a diagram.

The first step is squaring all the elements in the array and determining .

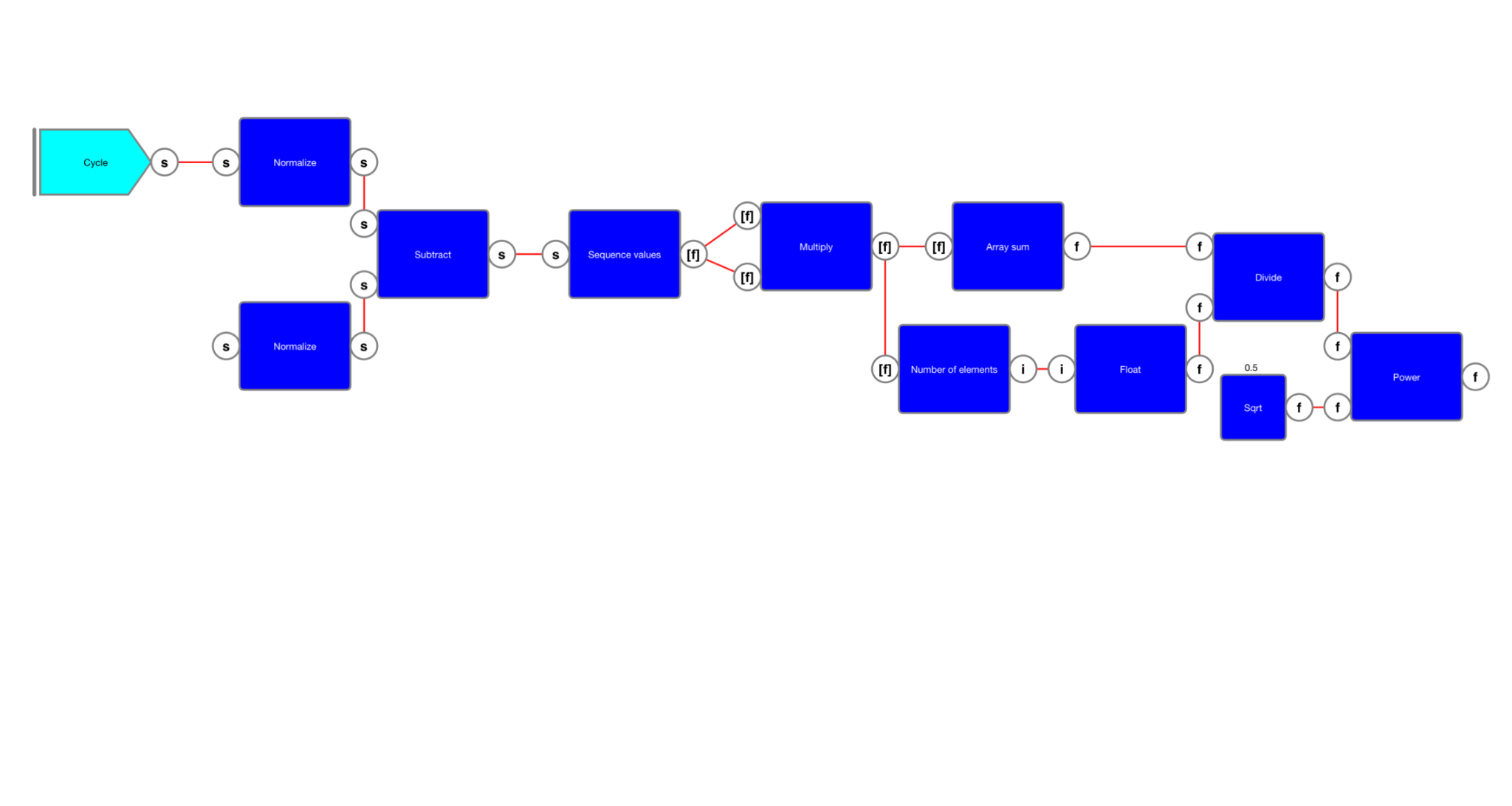
This is simply done by using the generic “*Multiply*” function as shown in the sub-diagram in [figure 11](#fig_multiply).

******[***Figure 11***](#figur_multiply)***: Added the "Multiply" function.***

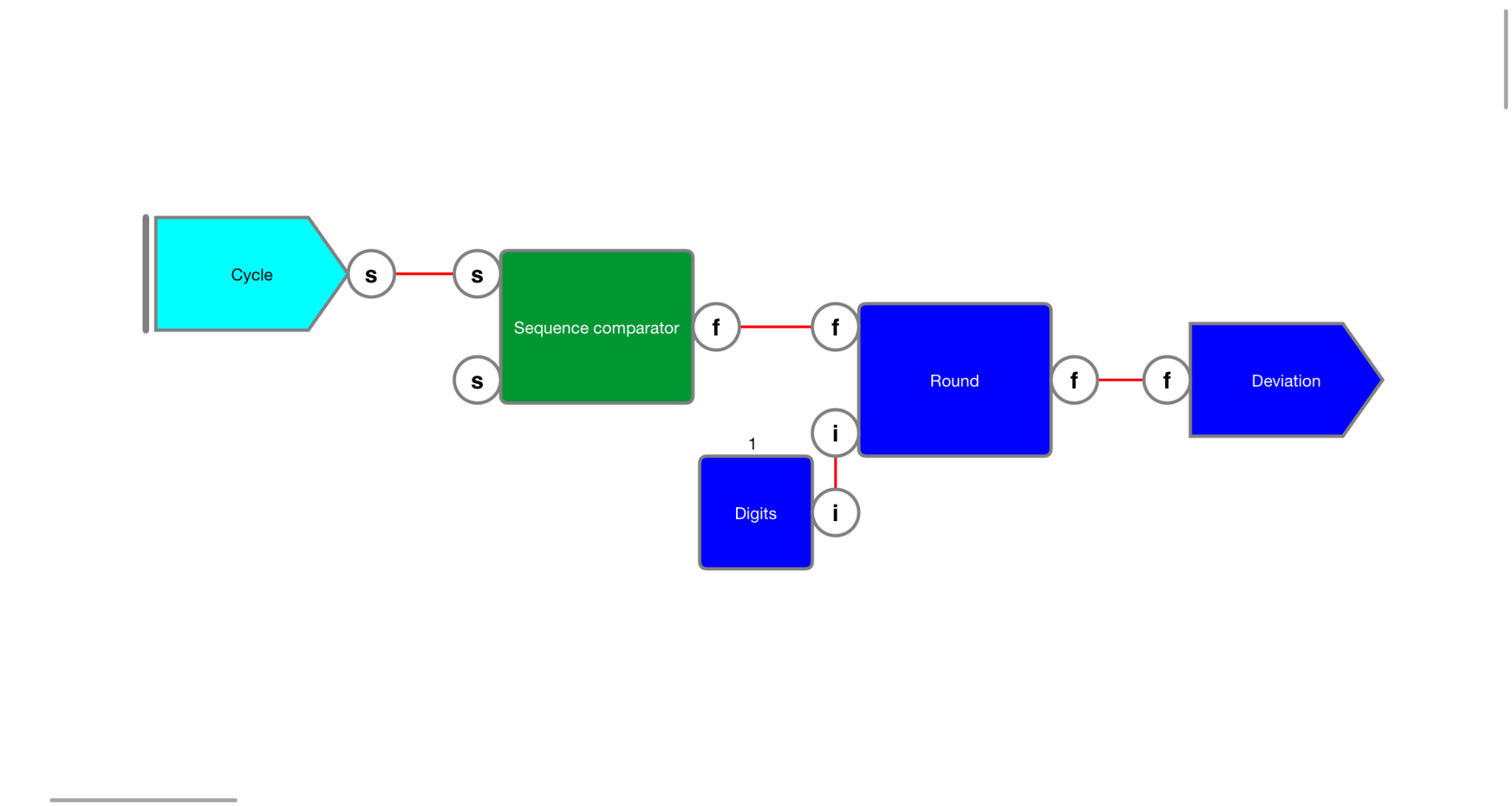
Of the resulting array, the sum can be determined by using the “*Array sum*” function as shown in [figure 12](#fig_added_array_sum).

******[***Figure 12***](#figur_added_array_sum)***: Added the "Array sum" function to obtain the sum of all the elements in the array.***

The resulting sum must be divided by the number of elements in the array, which can be retrieved using the “*Number of elements*” function. This gives an integer which must be converted to a float first before it can be used in the “*Divide*” function. To calculate the square root of the result the “*Power*” function can be used. Raising a value to the power of 0.5 is the same as determining the square-root. This results in the sub-diagram shown in [figure 13](#fig_calculate_rms).

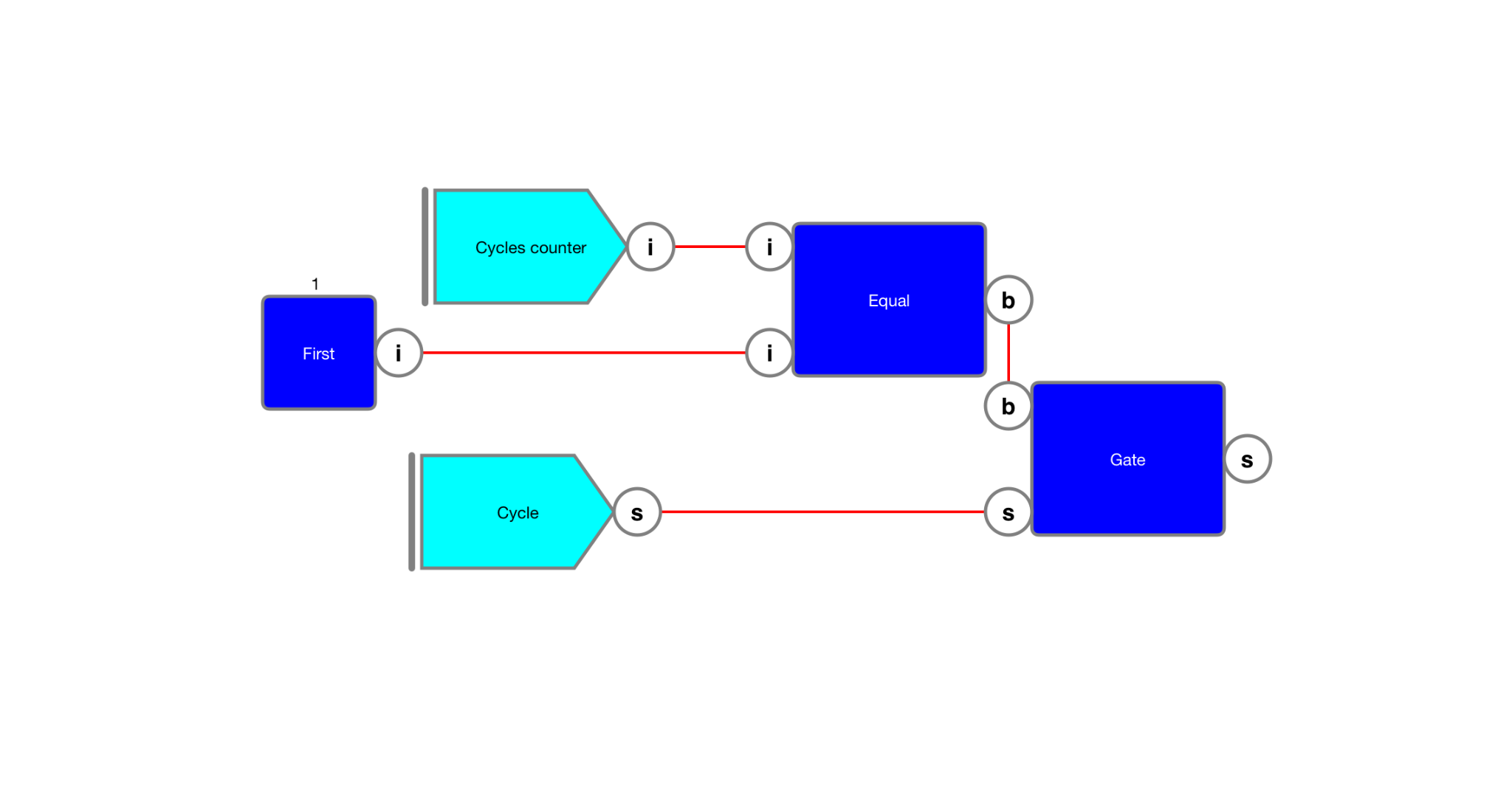
******[***Figure 13***](#figur_calculate_rms)***: Calculate the RMS between two normalized sequences.***

Using the results, the sequence comparator can be created as follows. This can be collapsed to a single “*Sequence comparator*” container. The output of this comparator can be rounded to one decimal place and offered as an output variable as shown in [figure 14](#fig_sequence_comparator).

******[***Figure 14***](#figur_sequence_comparator)***: Sequence comparator container.***

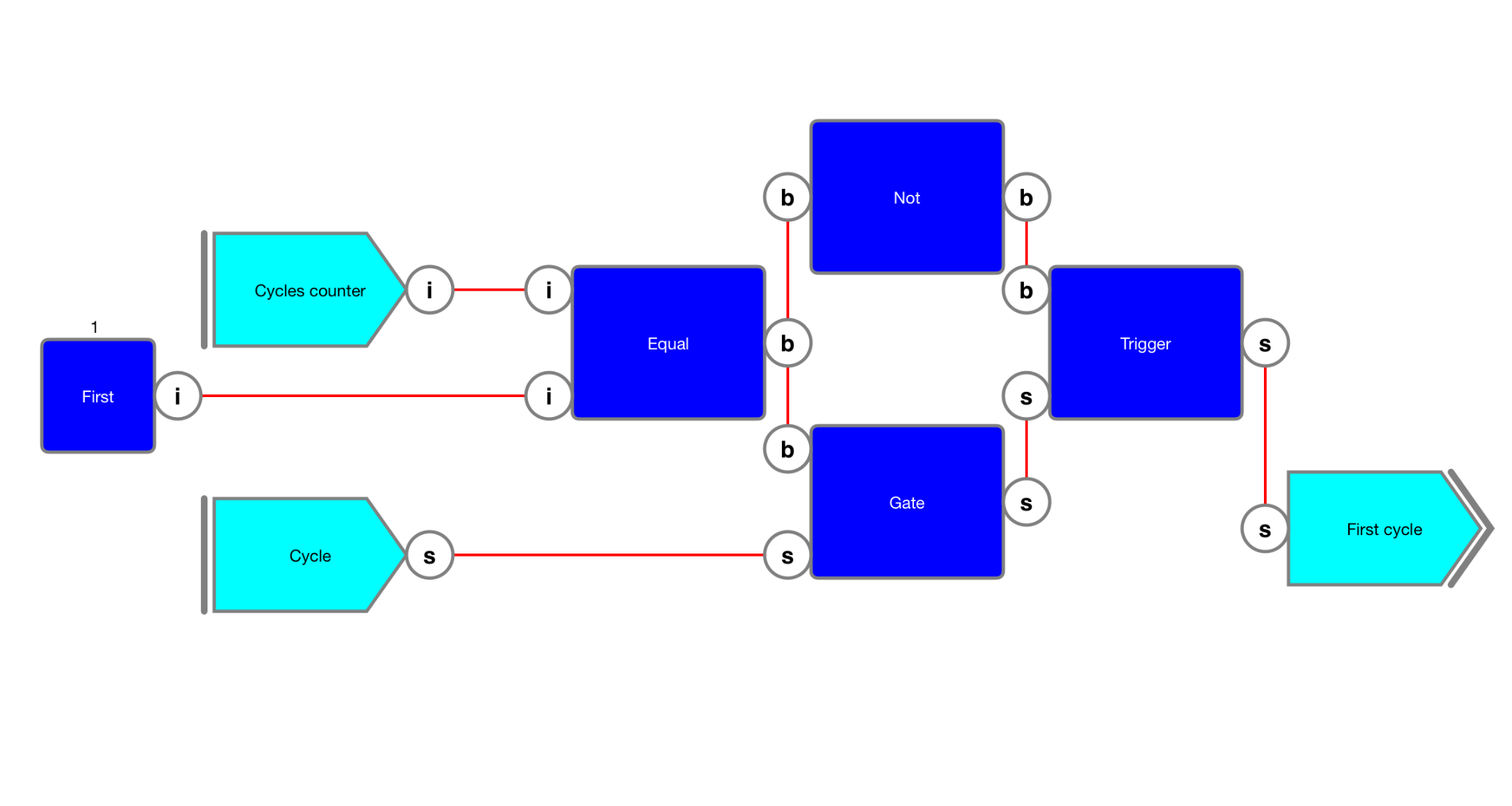
# Storing the first cycle

With the cycle available, the objective is to store the first cycle and use it to determine the deviation with every subsequent cycle. To detect the *first* cycle, the earlier determined values of the “*Cycle counter*” and "*Cycle*" output variables (see "[*Cycle detection*](#_vbrfctaqf2pu)") are looped. To identify the first cycle, "*Cycle counter*" is compared with an integer constant of 1 and to *only* output the first cycle, a “*Gate*” function is used.

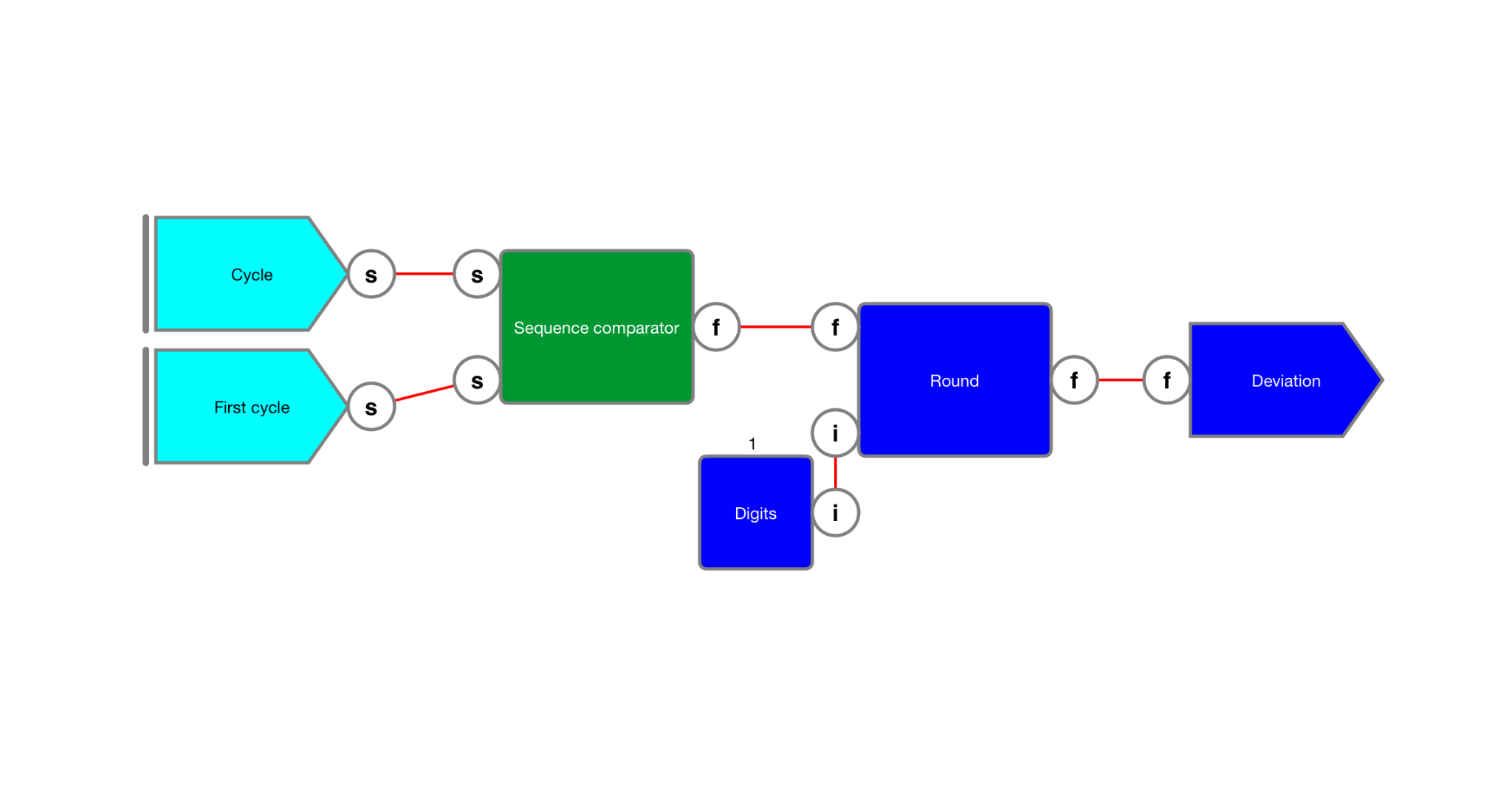
******[***Figure 15***](#figur_first_cycle)***: Capturing the first cycle.***

However, what we actually want is a diagram that offers the first sequence as an updated variable which can be used as input for the sequence comparator constructed in the previous chapter. Note that the “Subtract” function used there will only update when both inputs are updated.

To achieve this, a “*Trigger*” function is used as shown in [figure 16](#fig_first_cycle_2). This will generate an output value when its boolean input value updates to true. The value of this output is equal to the input irrespective of whether this input was updated or not.

******[***Figure 16***](#figur_first_cycle_2)***: Generating the first cycle.***

As shown in the sub-diagram in [figure 17](#fig_compare_first), the output of the “*First sequence*” variable output can now be looped and used as input for the “Cycle comparator” constructed in the previous chapter.

******[***Figure 17***](#figur_compare_first)***: Comparing the first cycle which each consecutive cycle.***

# Results

The result of the sequence comparator is given below.

| Timestamp,Deviation  9.116898999898694,19.2  12.000239999964833,33.3  14.000879999948665,34.1  15.617328999913298,31  19.050830999971367,11.8  22.067907999968156,21.1  25.284688999992795,10.3 |
| --- |

The result is plotted below for reference.

