Shape Monitor

Program implements algorithm to identify deviations from expected pattern in data.

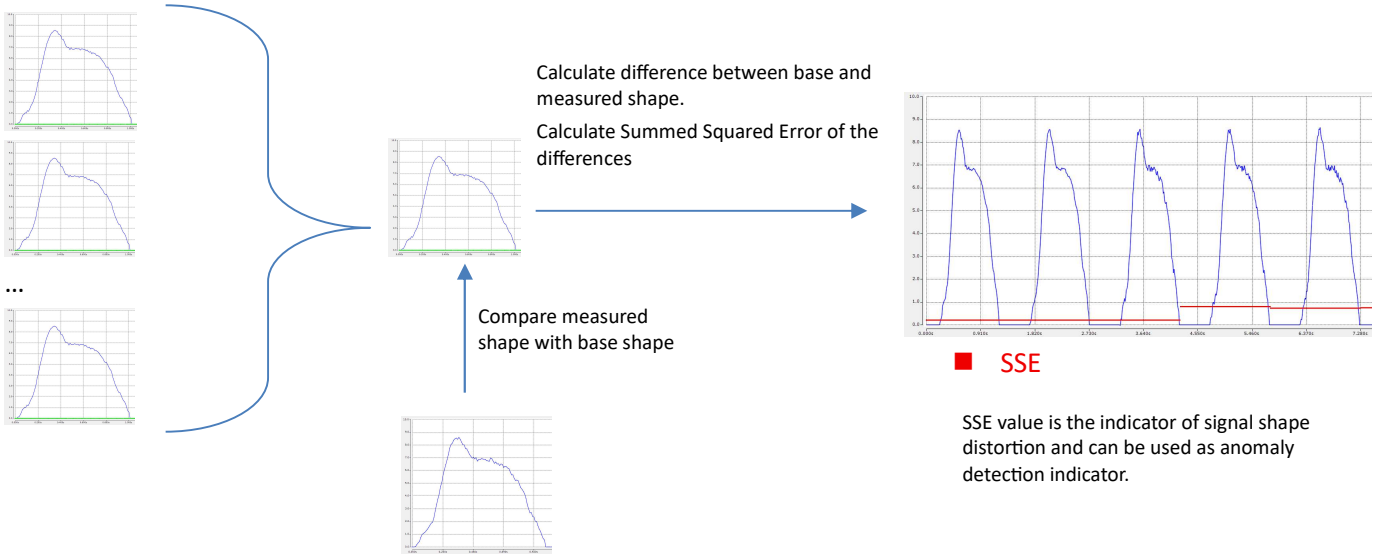
It evaluates measured analog data during trigger sensor status is `true`.

This enables to analyse periodically repeating signal shapes like motor inrush current, angle position of shifting device etc and find anomaly in their signal curves.

General algorithm overview

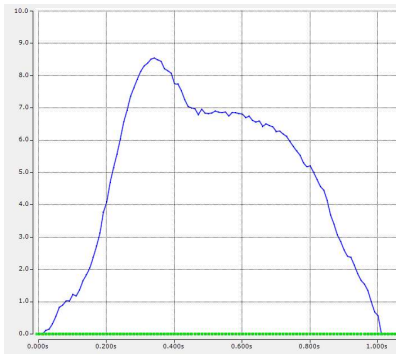
Base Shapes

Calculate Avarage Base Shape

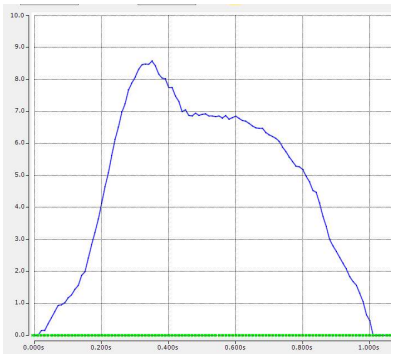


SSE value is the indicator of signal shape distortion and can be used as anomaly detection indicator.

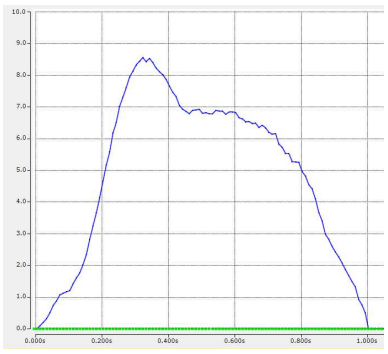
1. In first step, algorithm collects 10 shapes for calculation of *base shape*.



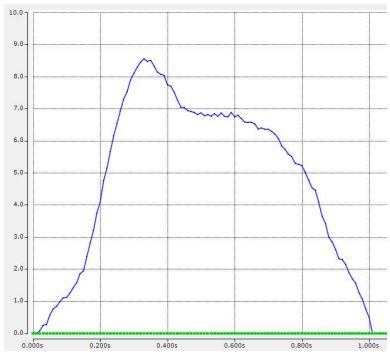
1st shape



2nd shape

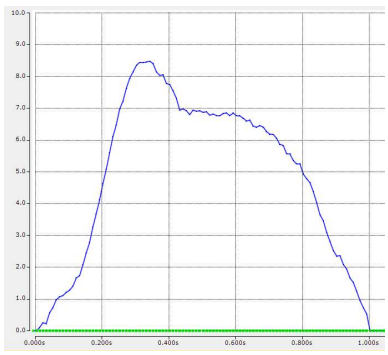


3rd shape



4th shape

.....

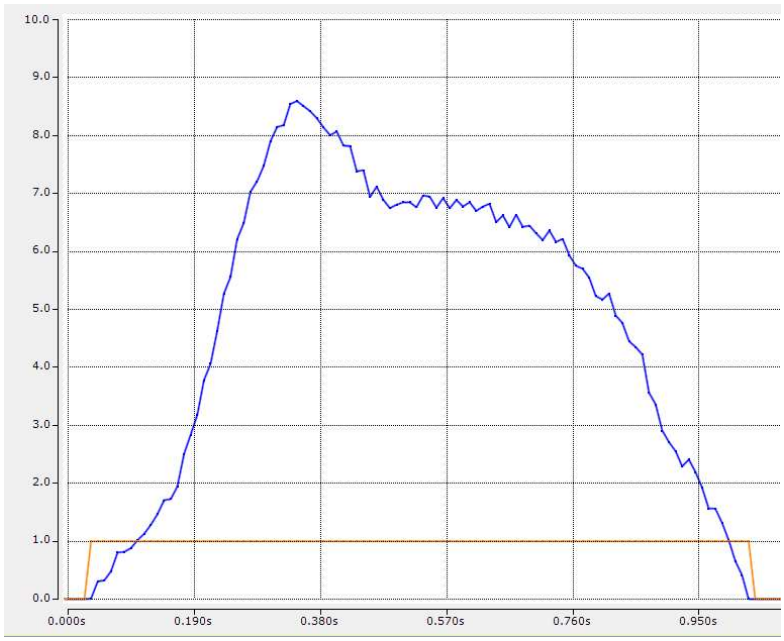


10th (or nth base shape)

2. Next it equalizes length of all collected base shapes into longest collected shape. Missing samples in shorter shapes are filled with last sample from respective shape.

3. Get new measured shape and equalize its length to the base shape length.

Missing samples in shorter shapes are filled with last sample from respective shape.

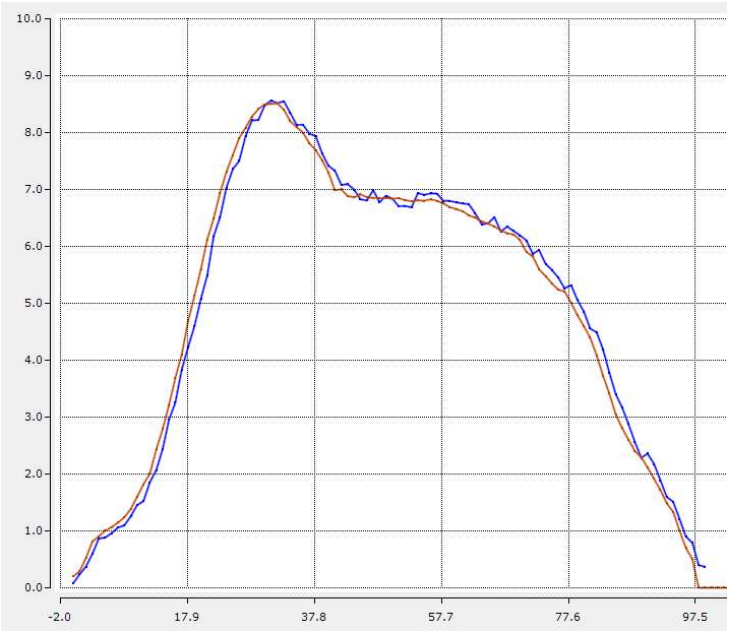


-- measured signal shape

-- trigger signal

4. Calculate samples difference between base and measured shape.

Calculate Summed Squared Error of the differences

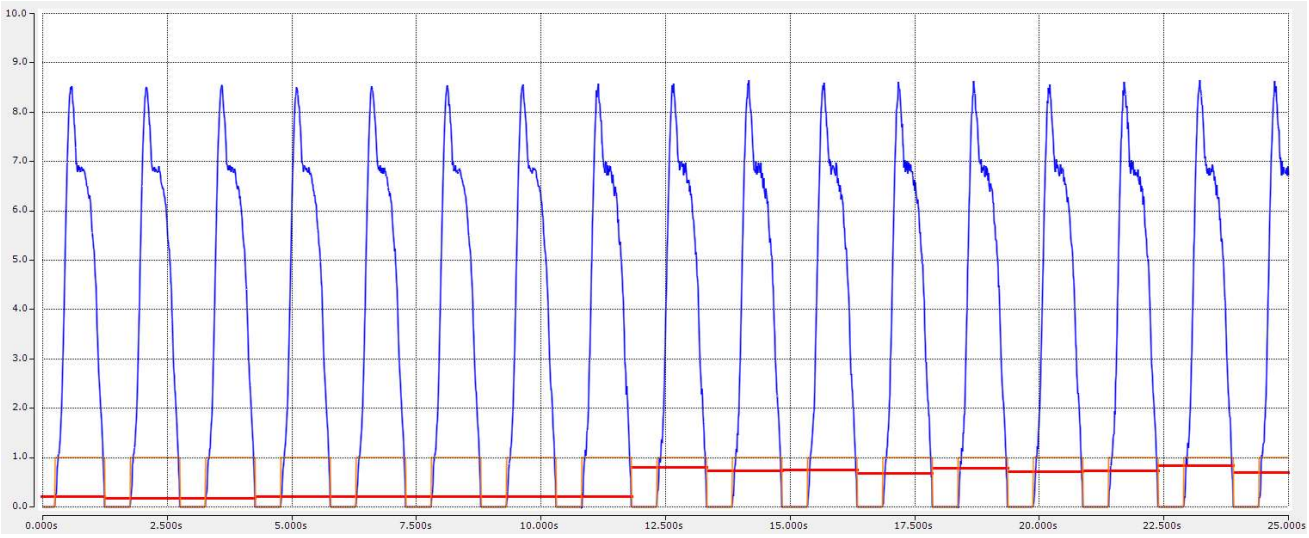


-- base shape

-- measured signal shape

e.g. SSE = 0.771

SSE can be used as anomaly detection predictor.



--- SSE

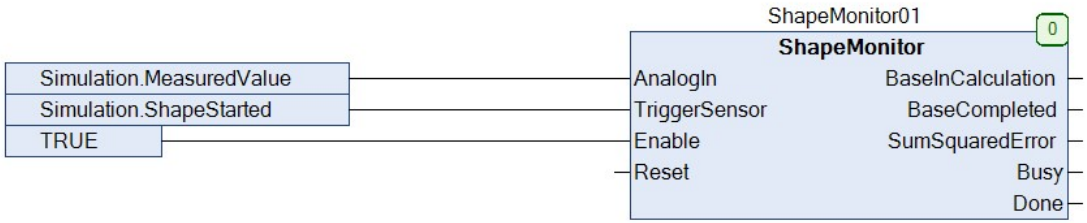
--- measured signal shape

--- trigger signal

```
PROGRAM Monitoring
VAR

END_VAR

VAR RETAIN PERSISTENT
	ShapeMonitor01 : ShapeMonitor;
END_VAR
```




```
FUNCTION_BLOCK ShapeMonitor
VAR_INPUT
	AnalogIn : REAL;
	TriggerSensor : BOOL;
	Enable : BOOL;
	Reset : BOOL; // reset algorithm
END_VAR
VAR_OUTPUT
	BaseInCalculation : BOOL;	// base model during calculation
	BaseCompleted : BOOL;	// model base calculation completed
	SumSquaredError : REAL;	// SSE - sumed squared error
	Busy : BOOL;	// shape evaluation in progress
	Done : BOOL;	// shape evaluated
END_VAR


VAR
	rtTriggerSensor : R_TRIG;	// detect rising edge of trigger sensor
	ftTriggerSensor : F_TRIG;	// detect falling edge of trigger sensor
	rtReset : R_TRIG;	// detect rising edge of reset signal

	State : (IDLE, TRIGGER_STARTED, TRIGGER_STOPPED ); // state machine

	arrShapeSamples : ARRAY [0..SAMPLES_IN_SHAPE] OF REAL;	// real-time analog samples catched at each PLC scan

	IdxMeasureSample : UINT;	// index of real-time measured sample
```

```
        arrBaseSamples                : ARRAY[0..SHAPES_IN_BASE] OF ARRAY [0..SAMPLES_IN_SHAPE] OF REAL;  // collected real-time signal shapes in base shapes


        arrNoSamplesInBaseShape       : ARRAY[0..SHAPES_IN_BASE] OF UINT;          // number of samples in each base shape
        arrBase                       : ARRAY[0..SAMPLES_IN_SHAPE] OF REAL;        // averaged base shapes array

        arrResult                     : ARRAY [0..SAMPLES_IN_SHAPE] OF REAL;       // array with squared diff. between base and incoming new shape

        NoBaseShapes                  : UINT;                                      // number of collected shapes in 'base'
        IdxBaseShape                  : UINT;                                      // idx for shape iteration
        IdxSample                     : UINT;                                      // idx for sample iteration

        MaxBaseLength                 : UINT;                                      // max length of shapes in base model
        SampleLength                  : UINT;                                      // length(num of samples) of evaluated shape
        Total                         : REAL;                                      // temporary var

END_VAR


VAR CONSTANT
        SHAPES_IN_BASE                : UINT := 10;                       // number of shape samples in base
        SAMPLES_IN_SHAPE              : UINT:= 199;                       // max number of samples in one shape
END_VAR
```

```
// detect rising trigger from reset input
rtReset(CLK:=Reset , Q=> );
IF rtReset.Q THEN
        resetAlgorithm();
END_IF;


IF ENABLE THEN

        // detect rising edge of trigger sensor
        rtTriggerSensor(CLK:=TriggerSensor , Q=> );

        // detect falling edge of trigger sensor
        ftTriggerSensor(CLK:=TriggerSensor , Q=> );

        CASE State OF
        IDLE:
                // clear array with measured data samples
                MEMSET(ADR(arrShapeSamples), 0, SIZEOF(arrShapeSamples));
                IdxMeasureSample              := 0;
                Busy                          := FALSE;
                Done                          := FALSE;

                IF rtTriggerSensor.Q THEN
                        State := TRIGGER_STARTED;
                END_IF
```

```
TRIGGER_STARTED:
// trigger sensor entered start position
        Busy                            := TRUE;
        Done                            := FALSE;

        // fill the array with incoming measured data
        arrShapeSamples[IdxMeasureSample] := AnalogIn;
        IdxMeasureSample := IdxMeasureSample + 1;

        // trigger sensor reached stop position OR max number of samples already collected
        IF ftTriggerSensor.Q OR IdxMeasureSample > SAMPLES_IN_SHAPE  THEN
                State := TRIGGER_STOPPED;
        END_IF

TRIGGER_STOPPED:
// trigger sensor left start position or max no of samples collected

        // collect shapes for 'Base'
        IF NoBaseShapes < SHAPES_IN_BASE  THEN
                BaseInCalculation := TRUE;
                arrBaseSamples[NoBaseShapes] := arrShapeSamples;
                NoBaseShapes := NoBaseShapes + 1;

        // shapes for base already collected
        ELSE
                // base calculation not completed yet
                IF NOT BaseCompleted THEN

                        // find max length of each shape in 'Base'
                        FOR IdxBaseShape := 0 TO SHAPES_IN_BASE-1 BY 1 DO
                                FOR idxSample := 0 TO SAMPLES_IN_SHAPE BY 1 DO
                                        IF arrBaseSamples[IdxBaseShape][idxSample] = 0 THEN
                                                arrNoSamplesInBaseShape[IdxBaseShape] := idxSample-1;
                                                EXIT;
                                        END_IF
                                END_FOR
                        END_FOR

                        // find max length of all base shapes
                        MaxBaseLength  :=      0;
                        FOR IdxMeasureSample := 0 TO SHAPES_IN_BASE-1 BY 1 DO
                                IF arrNoSamplesInBaseShape[IdxMeasureSample] > MaxBaseLength THEN
                                        MaxBaseLength:= arrNoSamplesInBaseShape[IdxMeasureSample];
                                END_IF
                        END_FOR

                        // equalize signal length for all collected base shapes (add last point)
                        FOR IdxBaseShape := 0 TO SHAPES_IN_BASE-1 BY 1 DO
                            IF arrNoSamplesInBaseShape[IdxBaseShape] <  MaxBaseLength THEN
                                FOR IdxSample := arrNoSamplesInBaseShape[IdxBaseShape] TO MaxBaseLength BY 1 DO
                                    arrBaseSamples[IdxBaseShape][IdxSample] := arrBaseSamples[IdxBaseShape][ arrNoSamplesInBaseShape[IdxBaseShape]];
                                END_FOR
                            END_IF
                        END_FOR
```

```
                // calculate avarage of all samples in BaseSamples -> create one 'Base' array
                FOR IdxSample := 0 TO MaxBaseLength BY 1 DO
                        Total := 0;
                        FOR  IdxBaseShape := 0 TO SHAPES_IN_BASE-1 BY 1 DO
                                Total := Total + arrBaseSamples[IdxBaseShape][IdxSample];
                        END_FOR
                        arrBase[IdxSample] := Total / (SHAPES_IN_BASE);
                END_FOR

                BaseCompleted  :=      TRUE;

        ELSIF BaseCompleted AND
                // ensure that valid number of samples was collected
                (UINT_TO_REAL(IdxMeasureSample) < 1.1 * UINT_TO_REAL(MaxBaseLength) AND
                 UINT_TO_REAL(IdxMeasureSample) > 0.9 * UINT_TO_REAL(MaxBaseLength) ) THEN

                 BaseInCalculation := FALSE;

                // find length of incoming shape
                FOR IdxSample := 0 TO SAMPLES_IN_SHAPE BY 1 DO
                        IF arrShapeSamples[idxSample] = 0 THEN
                                SampleLength := idxSample;
                                // exit FOR loop when element with value 0 found
                                EXIT;
                        END_IF
                END_FOR

                // adjust incoming shape length equal to base length by adding last value
                IF SampleLength < MaxBaseLength THEN
                        FOR IdxSample := SampleLength TO MaxBaseLength BY 1 DO
                                arrShapeSamples[IdxSample] := arrShapeSamples[SampleLength-1];
                        END_FOR
                END_IF

                // calculate difference between base and measured shape => error and square error value
                FOR IdxSample := 0 TO MaxBaseLength BY 1 DO
                        arrResult[idxSample] := LREAL_TO_REAL(EXPT((arrBase[idxSample] - arrShapeSamples[idxSample]),2)) ;
                END_FOR

                // calculate sum of squared errors (SSE)
                SumSquaredError := 0;
                FOR IdxSample := 0 TO MaxBaseLength BY 1 DO
                        SumSquaredError := SumSquaredError + arrResult[idxSample];
                END_FOR


        // new samples number is outside of limits -> restart algorithm
        ELSIF BaseCompleted AND (UINT_TO_REAL(IdxMeasureSample) > 1.2 * UINT_TO_REAL(MaxBaseLength)) OR
                  (UINT_TO_REAL(IdxMeasureSample) < 0.8 * UINT_TO_REAL(MaxBaseLength)) THEN
                  // withdraw this shape because too many/too less samples collected
                  resetAlgorithm();
        END_IF

END_IF
```

```
            Busy                            := FALSE;
            Done                            := TRUE;
            State                           := IDLE;

        END_CASE

ELSE
        ;
END_IF
```