SVPplus (HORIZON®)

IEC 61131 Library for ACSELERATOR RTAC® Projects

SEL Automation Controllers

Table of Contents

RTAC LIBRARY

SVPplus (HORIZON®)

Introduction

The SVPplus (Synchrophasor Vector Processor Plus) library provides Prony Analysis of signals, referred to here as modal analysis. The overall goal of this library is to encapsulate algorithms that describe the dynamics of a substation or electrical grid.

Modal Analysis (MA)

Similar to Fourier series, modal analysis decomposes a signal into its individual frequencies or modes. In addition, it also provides the damping rate of each frequency. Damping information is vital when monitoring wide area power system stability as it can be used to predict sustained or uncontrolled oscillations.

Sample Timing

The timing requirements of the modal analysis input signal are the responsibility of the user of the library. The library does not check the times of the samples given. The interval between each sample must be within acceptable error of the sample rate given in hertz. For example, if 20 Hz is given, the period between samples should be 0.05 seconds. The recommended error threshold is 5 percent.

Use Cases

Modal Analysis has been proven significantly useful in several areas of power system analysis.

Sometimes power system equipment or their controllers can be configured in such a way that they unintentionally incite growing oscillations. Rapid detection of these forced oscillations is important to guard against excessive mechanical fatigue and system instability. Modal analysis can detect these events by noting the maximum modal amplitudes of a system and triggering an alarm or control action based on larger than normal and/or increasing magnitude of oscillatory modal amplitude.

Disturbances in the power system are often identifiable from the decaying oscillations visible from streaming synchrophasor measurements. These events provide an opportunity to determine the natural system dynamic modes. The dynamic modes inform planners of the inherent stability, or lack thereof, of the power system and help identify areas where additional stabilizing devices could be installed.

Modal analysis performed on ambient data can identify the frequency of the natural dynamic modes. A change in the frequency component of these modes can indicate a system change that may require further investigation.

Operation

The modal analysis function block in this library is responsible for two primary tasks. The first task is to collect and store samples given to it. The second task is to analyze these stored values once enough new samples are given and return the modes of the stored signal.

The samples are stored in a ring buffer. Once a certain percentage of new samples are given, modal analysis is conducted on the stored array of samples. *Figure 1* shows this visually, where blue indicates old samples and red new samples. If the object has just been initialized and there are no stored samples, modal analysis is not complete until the entire buffer has been filled. After the initial filling of the buffer, modal analysis is done once the percentage of new samples has been reached. For example, if the total number of samples specified as the initialization variable *numSamples* is set to 100, and *percentUpdate* is set to 10 percent, then the tenth call to `GiveSample()` will trigger another calculation. This new modal calculation is performed on the most recent 90 old samples and the 10 new samples, with the oldest 10 samples having been overwritten in the ring buffer.

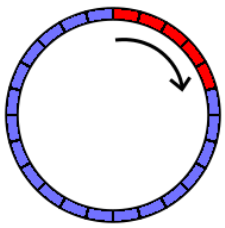

Figure 1 Ring Buffer Used To Store Samples In Modal Analysis Object

Because modal analysis is very computationally expensive, the analysis is done over many task cycles. The `Run()` method of each modal object does part of the analysis each cycle until it is complete. If a new analysis has been triggered and the previous is not yet complete, it is ignored until the previous analysis is finished. Once the current analysis is complete, the ignored trigger causes a new analysis to begin immediately. The ignored trigger is not a queue, but rather is a single request to restart once the previous analysis completes.

Modes

Modes are returned from modal analysis, which is the decomposition of the input signal into a series of modes. These modes consist of sinusoidal decaying signals that closely match the original signal when summed together. The equation used to construct each mode from their values is given in *Equation 1*, where A is the amplitude, α is the decay, f is the frequency, and θ is the angle phase.

$$Ae^{\alpha t}\cos\left(2\pi ft+\theta\right) \quad \text{(Equation 1)}$$

The quality of the signal is determined by the signal-to-noise ratio (SNR) returned with a call to `GetModes()`. *Equation 2* shows how the SNR is calculated, where S is the original signal and N is a signal reconstructed from the modes returned.

$$SNR = 20\log_{10}\left(\frac{\sqrt{S_1^2+S_2^2+...+S_n^2}}{\sqrt{(S_1-N_1)^2+(S_2-N_2)^2+...+(S_n-N_n)^2}}\right) \quad \text{(Equation 2)}$$

The SNR is calculated by using the decibel logarithmic unit on the ratio of the root-mean-square of the original signal over the root-mean-square of the difference between the original signal and reconstructed signal. If the value is 20, then the amplitude of the signal is 10 times greater than the error; when the value is 40, the amplitude of the signal is 100 times greater than the error, and so on.

Damping Ratio

The damping ratio ς is determined from the decay α and the frequency f (as seen in "Modern Solutions for Protection, Control, and Monitoring of Electric Power Systems" ISBN 978-0-9725026-3-4). *Equation 3* is used to compute the damping ratio. A negative damping ratio illustrates an increasing oscillation that can lead to power system instability.

$$\varsigma = -\frac{\alpha}{\sqrt{\alpha^2+(2\pi f)^2}} \quad \text{(Equation 3)}$$

Modal Analysis

Because of the computationally intensive nature of this algorithm, it is very important to consider benchmarks while using this object. See *Benchmarks on page 9* for benchmark information.

Supported Firmware Versions

You can use this library on any device configured using ACSELERATOR RTAC® SEL-5033 Software with firmware version R143 or higher.

Versions 3.5.0.3 and older can be used on RTAC firmware version R132 and higher.

Global Constants

This section lists values of global constants provided for facilitating work with the library.

Name	IEC 61131 Type	Value	Description
g_c_Infinity	REAL	∞	Positive infinity.

Structure Definitions

This section enumerates structures needed for the user to communicate with this library.

struct_Mode

This object contains the information for one sinusoidal mode returned from class_Modal-Analysis.

Name	IEC 61131 Type	Description
Amplitude	REAL	Amplitude of this mode.
Frequency	REAL	Frequency of this mode.
Phase	REAL	Phase of this mode in radians.
Damping	REAL	Decay rate of this mode.
DampingRatio	REAL	Damping ratio of this mode.

Classes

This section enumerates the classes used to access the functionality of this library.

class_ModalAnalysis (Class)

This class conducts modal analysis on an input signal.

Initialization Inputs

Name	IEC 61131 Type	Description
numSamples	UDINT	Total number of samples this modal analysis object uses.
sampleRate	UINT	Rate at which samples are taken in hertz.
percentUpdate	UDINT(10..100)	Percentage of new samples this modal analysis collects before performing analysis.
stepTime	ULINT	The amount of time in microseconds the `Run()` method will run each time it is called.
numModes	UDINT(1..16)	Number of modes that are returned from analysis.

For any meaningful analysis of a digital input signal, a minimum number of sample history must be stored in a buffer. *numSamples* is the size of this buffer for modal analysis.

Outputs

Name	IEC 61131 Type	Description

Outputs

Name	IEC 61131 Type	Description
IsEnabled	BOOL	Flag that states if this class instance is active.
Error	POINTER TO STRING(80)	If class instance failed initialization, this points to a string describing what failed.

These outputs give the status of the modal analysis object as a whole. If the initialization of the class instance was successful, it is flagged and enabled and ready to take new samples to analyze. If initialization failed, it is flagged as not enabled and the error string pointer returns a pointer to a string which described what failed in initialization.

bootstrap_SetFilter (Method)

This initialization routine sets the filter that is used on the raw input provided by `GiveSample()`. This routine is optional to call, and if it is not called no filter is used and the raw samples are fed directly into modal analysis.

Inputs

Name	IEC 61131 Type	Description
filter	I_Filter	Interface pointer to filter that will be used on raw input.

Return Value

IEC 61131 Type	Description
BOOL	Returns TRUE if the filter was successfully added to the object.

Processing

This routine takes the interface to the filter and confirms the pointer to it is real. If the pointer to the filter is zero or the modal analysis object has already been initialized, it will return FALSE as a failure.

GiveSample (Method)

This method gives a new sample as input to the modal analysis block. The sample rate integrity of the signal must be verified by the user, as discussed in *Sample Timing on page 3*.

Inputs

Name	IEC 61131 Type	Description
sample	REAL	New sample.

Processing

This method takes the input sample and stores it within its internal buffer of samples. If enough new samples have been given or the entire buffer of samples has been populated for the first time, the method will trigger the calculation of new modes. This calculation is done over many cycles in the run method.

Reset (Method)

Modal analysis is only effective on a continuous stream of evenly spaced samples. When a gap in the stream has been detected, modal analysis must be reset. This causes it to discard all previous samples. Additionally, a sample with unacceptably low quality is considered a gap in the stream, and therefore, modal analysis must be reset.

Resetting modal analysis delays the next analysis as additional samples will need to be gathered. A sample is considered bad if it does not meet the timing requirements defined in *Sample Timing on page 3* or if the quality of the sample is unacceptable.

Processing

This method will discard all saved samples and start with an empty buffer that needs to be refilled for modal analysis to happen.

Run (Method)

This method must run once per cycle for each modal analysis object that exists. It is responsible for processing modal analysis of the samples. It performs a part of an analysis, if one is pending, each time it is called.

Return Value

IEC 61131 Type	Description
UDINT	Percentage of completion for modal analysis task from 0 to 100.

Processing

Once the output *complete* has reached 100, the analysis is complete and GetModes can be called.

GetModes (Method)

Call this method to get the most recent modal analysis results. If no analysis has yet been done, this method will return an array of structures whose values are all set to zero.

The range of the signal-to-noise ratio can be anywhere from 0 to the maximum number of UINT. Anything under a value of 80 is considered a poor value and the returned modes should be ignored. Increasing the number of modes returned will improve this signal-to-noise ratio.

Inputs

Name	IEC 61131 Type	Description
pt_modes	POINTER TO struct_Mode	Address of array to copy modes to as returned by the `ADR()` function. This array must contain *numModes* structures as specified in the Initialization Inputs of the class.

Return Value

IEC 61131 Type	Description
REAL	Signal-to-noise ratio that determines quality of modes returned.

Processing

If no analysis has yet been done, this method will set all structures to zero. If the number of modes for this object has been initialized with a number *n* that is less than the global value g_p_numModes, then all structures past *n* will be set to zero.

Benchmarks

Benchmark Platforms

The benchmarking tests recorded for this library are performed on the following platforms.

- SEL-3505
  - R134 firmware
- SEL-3530
  - R134 firmware
- SEL-3555
  - Dual-core Intel i7-3555LE processor
  - 4 GB ECC RAM
  - R134-V1 firmware

Benchmark Test Descriptions

Because so much of the performance of this library is defined by the user, these benchmarks attempt to give an overall feel for the cost of performing modal analysis in terms of number of samples and number of modes requested. All results are presented as the number of scans taken to perform the analysis where the ModalAnalysis object is given 5 ms of run time each task cycle.

The following benchmarks are tested:

- 4, 8, and 16 modes with 500 samples
- 4 modes with 600 samples
- 4, 8, and 16 modes with 5000 samples

Benchmark Results

Operation Tested	Platform (time in μs)		
	SEL-3505	SEL-3530	SEL-3555
4 Modes 500 Samples	27	16	2
8 Modes 500 Samples	114	64	6
16 Modes 500 Samples	462	253	21
4 Modes 600 Samples	34	17	3
4 Modes 5k Samples	155	93	2
8 Modes 5k Samples	460	276	23
16 Modes 5k Samples	1586	952	80

Examples

These examples demonstrate the capabilities of this library. Do not mistake them as suggestions or recommendations from SEL.

Implement the best practices of your organization when using these libraries. As the user of this library, you are responsible for ensuring correct implementation and verifying that the project using these libraries performs as expected.

Calculating Modes

Objective

The objective of this example is to calculate the modal frequencies and damping coefficients of a synchrophasor frequency measurement for oscillation stability analysis at an update rate of once per second. This example will cover the preconditioning of incoming time stamps and how to call the ModalAnalysis methods in the appropriate sequence.

Assumptions

This example assumes the following are true:

- The RTAC project incorporates both the SVPplus and Analog Conditioning libraries.
- The RTAC is collecting synchrophasor measurements from one synchrophasor measurement device, referenced here as a C37_118 RTAC client called PMU1.
- The RTAC is performing no tasks other than those referred to in this example.
- The nominal frequency of the power system is 60 Hz.
- The synchrophasor message rate is 60 samples per second.
- The RTAC Main Task cycle time, under which all programs are being run, is set to 8 ms.
- The Deadband and Zero Deadband settings for the PMU1.FREQ tag are set to 0.
- There are only four modes of interest in the system. Each is between 0.3 Hz and 10 Hz.
- Data samples are filtered with a 10 Hz cutoff low-pass filter before being processed with MA. The ArmaFilter class from the Analog Conditioning library is used to implement this filter. This is also demonstrated in the example.
- The global variable list and sample flagging programs shown in *Code Snippet 1* and *Code Snippet 2*, respectively, are included in the project.

Code Snippet 1 Global Variable List

```
VAR_GLOBAL CONSTANT
    g_c_NumModes : UINT := 4;
    //Using coefficients for a 10Hz cutoff low pass filter.
    g_c_Acoeff : ARRAY[1..g_p_MaxFilterOrder] OF REAL :=
                         [-1.96299, 1.40000, -0.34641];
    g_c_Bcoeff : ARRAY[0..g_p_MaxFilterOrder] OF REAL :=
                         [0.011325, 0.033975, 0.033975, 0.011325];
END_VAR

VAR_GLOBAL
    g_SampleQuality, g_SampleUpdated, g_SampleTimeValid, g_SampleValid :
        BOOL;
    g_SNR : REAL;
    g_ModeResults : ARRAY [1..g_c_NumModes] OF struct_Mode;
    g_DampingAlarm : ARRAY [1 .. g_c_NumModes] OF BOOL;
END_VAR
```

Code Snippet 2 prg_SampleFlagging

```
PROGRAM prg_SampleFlagging
VAR
    dTime : DINT; //current day and time of PMU1_C37_118
    us, timeNow, timeLast, timeDiff : LREAL;
END_VAR
```

Code Snippet 2 prg_SampleFlagging (Continued)

```
//Establish the quality of the sample.
g_SampleQuality := NOT(DINT_TO_BOOL(PMU1_C37_118.FREQ.q.validity)) AND
       NOT(PMU1_C37_118.FREQ.t.quality.clockNotSynchronized);
//Calculate the time difference between the current and previous samples.
dTime := DT_TO_DINT(PMU1_C37_118.FREQ.t.value.dateTime);
us := UDINT_TO_LREAL(PMU1_C37_118.FREQ.t.value.uSec) / 1000000;
timeNow := dTime + us;
timeDiff := (timeNow - timeLast)*1000; //Sample time difference in ms
timeLast := timeNow;
//Determine if the sample has updated.
IF (timeDiff > 0) THEN
    g_SampleUpdated := TRUE;
ELSE
    g_SampleUpdated := FALSE;
END_IF
//Validate the sample time difference against +-2.5 % of a 60Hz period.
IF (timeDiff > 17.08 OR timeDiff < 16.25) THEN
    g_SampleTimeValid := FALSE;
ELSE
    g_SampleTimeValid := TRUE;
END_IF
//Assign overall sample validity.
IF (g_SampleQuality AND g_SampleTimeValid) THEN
    g_SampleValid := TRUE;
ELSE
    g_SampleValid := FALSE;
END_IF
```

Solution

Having flagged the incoming samples for quality and continuity, the samples can be filtered and processed with MA as shown in *Code Snippet 3*.

Code Snippet 3 prg_ModeCalc

```
PROGRAM prg_ModeCalc
VAR
    modal : class_ModalAnalysis( numSamples := 600, sampleRate := 60,
                                 percentUpdate := 10, stepTime := 4000,
                                 numModes := g_c_NumModes);
    inputSample : REAL;
    filter : class_ArmaFilter(g_c_Acoeff, g_c_Bcoeff, 3, 4);
    analysisComplete : R_TRIG;
    doneLatch : BOOL;
    init : BOOL := TRUE;
END_VAR
```

Code Snippet 3 prg_ModeCalc (Continued)

```
//Update inputSample.
inputSample := PMU1_C37_118.FREQ.instMag;
//Initialize: Load filter into the MA object.
IF init THEN
    modal.bootstrap_SetFilter(filter);
    init := FALSE;
END_IF

//Load the sample into MA only if it has updated and is valid.
IF(g_SampleUpdated) THEN
    IF(g_SampleValid) THEN
        modal.giveSample(inputSample - 60); //Factor out the 60Hz offset
    ELSE
        modal.reset();
    END_IF;
END_IF;

(*If Run() method is at 100 % completion, populate g_ModeResults and
    return SNR.
Use an edge trigger to guarantee that getModes() is only called once per
    analysis.*)
doneLatch := modal.Run() = 100;
analysisComplete(CLK := doneLatch);
IF analysisComplete.Q THEN
    g_SNR := modal.getModes(ADR(g_ModeResults));
END_IF
```

Analyzing Modes

Objective

The objective of this example is to demonstrate how the output of the ModalAnalysis class can be processed to flag a necessary control action.

Assumptions

This example extends the previous example. Thus, it is assumed that the `getModes()` method has been called successfully. It is further assumed that the program in this example will accesses the global variable list defined in the previous example.

Solution

The code below shows a simple example which asserts a per-mode alarm bit if the given mode meets the user-specified criteria. The signal-to-noise ratio output of the `getModes()` method is used to validate the accuracy of the modal estimation.

Code Snippet 4 prg_ModeAnalyze

```
PROGRAM prg_ModeAnalyze
VAR
    dmpThr : REAL; //Define damping ratio threshold here.
    ampThr : REAL; //Define oscillation amplitude threshold here.
    SNRFail : BOOL;
    i : UINT;
END_VAR
```

```
IF g_SNR > 80 THEN
    SNRFail := FALSE;
    FOR  i := 1 TO g_c_NumModes DO
        IF g_ModeResults[i].DampingRatio < dmpThr
              AND g_ModeResults[i].Amplitude > ampThr
              AND g_ModeResults[i].Frequency > 0  //Factor out DC modes.
        THEN
            g_DampingAlarm[i] := TRUE;
        ELSE
            g_DampingAlarm[i] := FALSE;
        END_IF
    END_FOR
ELSE
    SNRFail := TRUE;
END_IF
```

Release Notes

Version	Summary of Revisions	Date Code
3.5.1.0	➤ Allows new versions of ACSELERATOR RTAC to compile projects for previous firmware versions without SEL IEC types "Cannot convert" messages. ➤ Must be used with R143 firmware or later.	20180921
3.5.0.3	➤ Improved precision of entire algorithm. ➤ Improved performance for large matrices. ➤ Hid internal variables of all function blocks.	20150718
3.5.0.0	➤ Initial release.	20141101