Python Class PMU Documentation

This document outlines the design and use of the Python Phasor Measurement Unit (PMU) classes developed by Dylan Tarter for Pacific Northwest National Labs under the Open Energy Data Initiative (OEDI) summer internship program (July/August 2022).

The python classes outlined in this document are for simulating PMU data when given timestamped data. The classes in this document demonstrate the standard algorithms in IEEE C37.118 (2018), provide a framework for using alternate PMU estimation, and perform the necessary tests from the standard to ensure compliance. An implementation of James Follum’s Non-Linear Least-Squares estimation method was also implemented, requiring the PMU framework to work with windowed data.

The python files used by this documentation are named as follows:

|  |  |
| --- | --- |
| File Name | Description |
| pmuClass | The pmu base class function. Used to pass data to the estimators. Use “from pmuClass import pmuClass” to load it, and use “pmuClass.run(…)” to create a class instance. |
| ieeeClass | An estimator class for the IEEE C37.118 standard PMU. Use “from ieeeClass import ieeeClass” to load it, and use “ieeeClass.run(…)” to create a class instance. |
| nllsClass | An estimator class for the Non-Linear Least-Squares method. Use “from nllsClass import nllsClass” to load it, and use “nllsClass.run(…)” to create a class instance. |
| PMUTestLib | The IEEE C37.118 standard testing library. Use “import PMUTestLib as PMUTests” to load it. Use “PMUTests.run(…)” to run the standard tests. Use “PMUTests.writeJson(…)” and “PMUTests.readJson(…)” to save and load test results. Use “PMUTests.plot(…)” to plot result dictionaries. |
| StandardTestDemo | This file should be in the same folder as the other named scripts. When ran, it will perform all the standard tests on NLLS and IEEE, then plot results. Note that NLLS takes a very long time to run with ROCOF so ROCOF was disabled. Without ROCOF it can take around 100s to run. With ROCOF, over 500s. |

# The PMU Base Class

The PMU base class is intended to create a framework for passing the input, timestamped, data to the different types of estimators. To create a PMU base class, one must specify the sampling rate (fs), nominal frequency (f0), report rate (RR), method (filtered/windowed), and estimator (a custom class later discussed). The sampling rate must be an integer multiple of the nominal frequency and the report rate, otherwise undefined behavior may occur. The method variable refers to how the data is handled when it is given to the run function. Filtered data is passed to the estimator, then downsampled to the report rate. Windowed data is cut into windows with the reports at the center of each window.

## Filtered Data

The provided examples include the IEEE P and M class phasor estimation algorithm which uses their P and M class filters to reduce harmonics after complex demodulation is applied to the input waveforms. If the input data is of size N, the IEEE PMU will output N phasor estimates. This behavior was defined as a ‘filtered’ PMU method. Filter based PMUs apply filters to the input data and must be decimated following the estimation process to achieve the desired reporting rate. For that reason, if a pmuClass is given the method of ‘filtered’ when run, it will simply fix the orientation of input arrays and pass them along to the estimator. It will expect to receive N estimates and so it will handle the decimation.

## Windowed Data

For the NLLS example, the data must be windowed. This means that for the provided “W” cycles, the data must be split into windows of size M.

The windows are insured to be odd so that there is a center point which the report rates will be tied to. The report rate indices (RRidx) are obtained first, and each window is placed with their center at an index, spanning +/- W/2 points. There is additional code to make sure that the windows do not contain NaN values, which indicate unknown or invalid data due to latency, filter startup, or group delay. The valid windows are passed to the estimator and expect 1 estimate as a return, which are then returned to the user directly, with no decimation.

## Decimation

The decimation block is recommended to have some filtering, however in this implementation it was found that integer decimation was suitable for the estimator. It performs the decimation and assigns timestamps based on the report rate indices (RRidx).

## Report Rate Timestamps

In all cases, timestamps must be generated at the window rates. This can be likened to integer downsampling from fs to RR, however the reports much lie on the top of a second. That means that the first report, if the data runs from 0 to 1s, must lie on 0.0s and the last report must lie on 1.0s. In the simulated data it is assumed that timestamps are evenly spaced based on the sampling rate, and lie on integer multiples of the report rate. To do this, the first whole second is assumed to be the ceil of the first timestamp. If the timestamps are from 0.2s to 0.6s, that value would be 1s. If this value is not contained in the timestamps, it extrapolates timestamps from the end of the input up until the time. From that initial value, indices are taken in even integer steps, expanding to the start and end of the timestamp array.

# The Estimator Class

The estimator class has a few guidelines for variables and how data should be inputted/outputted but can do any signal processing needed on the data. The required input variables are sampling frequency (fs), nominal frequency (f0), report rate (RR), and PMU class (p/m). A name can also be provided, otherwise it will use the name of the class when the \_\_name\_\_ property is called (estimator classes must contain this property). The PMU class can either be P or M. This will determine what tests are done to it in the standard test library and helps specify what kind of behavior it may have. P classes are typically faster response but less tolerable of noise and changes. M classes are typically slower response with bigger filters or windows but are more tolerant to noise and changes.

The estimator class must contain a function named “run” which takes an input of timestamps “t” and measurements “x”. The timestamps and measurements must be of same length N. Measurements can either contain one, three, or six measurements (M). These correspond to a single phase, three phase, (voltage) and three + three phase test (voltage + current). The measurements come from the PMU Base Class and will be oriented as a N x M array, where each row is a measurement, and each column is a channel.

If the PMU Base Class is set to filtered mode, the estimator is expected to output N estimated. If mode is set to windowed, the estimator is expected to output 1 estimate. The estimator must output a phasor, frequency, and rate of change of frequency estimate. The phasors must be an NxL array where L is defined based on M.

If only 1 phase is inputted, it should only output 1 phasor estimate. If 3 phases are inputted, it should output the 3 phasor estimates followed by a positive sequence phasor estimate. If 3 + 3 phases are inputted it should output 3 phasor estimates, followed by a positive sequence phasor estimate, followed by the next 3 phasor estimates, followed by the last positive sequence phasor. The frequency should be estimated based on the positive sequence, and any subsequent estimates should be based on the original frequency estimate. For example, if 3 phase voltage and 3 phase current is provided. The frequency estimate should come from the 3 voltage phasors, and the 3 current phasors should be estimated based on the already calculated frequency. ROCOF should follow the same rules as frequency.

If any calculations are not possible, the convention is to pass back NaN values to indicate an unknown value. For example, the IEEE PMUs are not capable of estimating frequency for single phases, so if only one phase is presented to the estimator, it will output all NaNs for frequency and ROCOF.

# The IEEE Standard PMU

The IEEE C37.118 (2018) pmu from Annex D was implemented to fit the estimator class specifications. When the class is created, either a P or M class filter is calculated and saved for later use. The filters are defined in D.6 at equation D.5 and D.7 at table D.1. The P class filter is calculated based on the samples per cycle, and will work for any combination of fs,f0,and RR. The M class filter has a range of values based on f0 and RR, and thus only works for specific values. In addition to basic filter values, the complex demodulation coefficients are also precalculated at this step.

When it is called by the PMU base class, it follows the rules defined in section 2. If the inputted measurements only have 1 phase, frequency and ROCOF are set to NaN. If there are 3 phases, the 3 phasor measurement arrays are calculated by filtering the input with the precalculated filter, and appropriate startup and group delay are applied. The frequency is calculated based on the central difference of the angles of the positive sequence estimate, which is calculated using a symmetrical component transform of the separate 3 phase estimates. ROCOF is calculated using the 2nd order central difference of the same values. For P class filters, all of the phasor’s magnitude gain is adjusted according to equation D.6.

# The NLLS PMU

The Non-Linear Least-Squares PMU algorithm was implemented to fit the estimator class specifications. The NLLS algorithm was developed during this project by James Follum and Dylan Tarter and is meant to be a curve-fitting type estimation method. NLLS assumes that the input data is a sinusoid with an unknown frequency, amplitude, and phase. Those parameters are then solved for using a non-linear least-squares approach to find the frequency, followed linear algebra to calculate the amplitude and phase. The theory is similar to the SEMPR algorithm, but does not include ROCOF in the model, and results in a faster and simpler solution.

Signal Model:

The NLLS algorithm uses a method like gradient descent to estimate the frequency of a set of measurements. Then from that frequency the magnitude and phase can be estimated. For this reason, a windowed method is used. For each window, the time used by the algorithm must be stationary and centered around 0, otherwise phase angles will be inaccurate. The tolerance ranges for the frequency estimate must also be restricted based on the class. For P class all standard tests lie within 58-62Hz and for M class all standard tests lie within 55-65Hz, thus the values are set just above and below that.

## ROCOF Estimation

To calculate ROCOF a variety of methods were tested. It was found that using the phasor angles of the positive sequence, as with IEEE, resulted in values with far too much random deviation. To fix this, a central difference of the frequency was used. This required there to be 3 NLLS estimates of frequency, using a centered window, a forward shifted window, and a backwards shifted window. Using the central difference of the 3 frequencies resulted in a more accurate and stable ROCOF estimate, however it did not pass every test. To further improve the stability, an average of central differences was used. That is, 5 frequencies are estimated and the average of the 3 central difference measurements was used for ROCOF. This managed to pass all standard tests. In addition, the outputted frequency was taken as the average of all 5 estimates, which improved performance during the harmonics test. If the estimator is made with the option “doROCOF=0”, it will greatly speed up computation time by only doing 1 frequency estimate, and reporting NaN for ROCOF.

1. Pre-Filtering

While implementing the PMU standard tests, it was found that the NLLS algorithm performed poorly on the harmonics and out-of-bound interference test. To supplement this pre filtering is used. A function called preFilter is called by the PMU Base Class if it exists. The filters are Parks-McClellan ideal filters implemented using the scipy library’s remez function. The values for the filter were calculated in Matlab, because the firpmord function is not available. These hardcoded values are only valid for a sampling rate of 960Hz, although this could be fixed by implementing the firpmord function. The P class filter is a low pass filter with 31 points and a cutoff frequency at 130Hz. The M class filter is a band pass filter with 115 points and has band cutoffs at 30Hz and 90Hz to help with the out-of-bounds interference test specifications. The filters add additional group delay to the estimator, but the filter sizes were chosen to have similar delays as the IEEE standard filters. It was also found that the NLLS estimation method can pass all P class tests without the use of the filter, if a window size of at least 2 cycles is used. The use of this can be disabled by optionally specifying doPreFilter=0.

# The PMU Standard Test Library

The PMU Standard Test library implements all of the tests in IEEE C37.118 (2018) and can generate data and plots that resemble the 2014 NIST Assessment of PMUs document. The library can perform all of the tests, or just a select few. It can plot the results or save them to dictionaries. The dictionaries can be combined and plotted together for comparisons. The dictionaries can also be written and read to and from .json files. The libraries use and implementation details are contained in this section.

## Running the Standard Tests

The PMUTestsLibrary has 4 user functions. The first user function is “run”. This function can run the tests and returns the results on a dictionary that is organized in such a way that it can be used in the plotting function. The function has 3 inputs: pmu, doPlot, and tests. The first input is the pmu base class that will be used during the tests. The tests done will depend on the related estimator class’s filter class (P or M). The second input is a boolean value which determines whether the tests are plotted. If you just want to quickly see the results of a test set this value to 1. If you want to compare the results of one pmu to another, set it to 0. The last field contains what tests are to be performed. Naturally, some tests are slower than others. The best test to perform to test a PMU’s general performance is the Positive Frequency Ramp Test. The tests variable can either be the string ‘all’ to run all tests, or an array of strings that specify which tests to be done. The names of the tests are as follows:

|  |  |
| --- | --- |
| Test Name | Input String |
| Steady State Frequency | frequency |
| Steady State Magnitude | magnitude |
| Steady State Harmonic | harmonic |
| Out-of-Bounds Interference | oob |
| Amplitude Modulation | aModulation |
| Phase Modulation | pModulation |
| Positive Frequency Ramp | pRamp |
| Negative Frequency Ramp | nRamp |
| Positive Magnitude Step | pMagStep |
| Negative Magnitude Step | nMagStep |
| Positive Phase Step | pPhaseStep |
| Negative Phase Step | nPhaseStep |

When inputting the test names, they can have any capitalization and come in any order. For example, to perform a positive frequency ramp test, steady state harmonic test, and positive phase step tests on an P class IEEE pmu with fs = 960, f0 = 60, and RR = 60 the following code would be used.



## Standard Test Dictionary

The standard tests output a dictionary that can be saved to variables for later use. This dictionary has a uniform structure. Each test dictionary contains the fs, f0, RR, test class used (class) and the name of the estimator. Each new tests has its own entry label that matches the test name inputs. Each test will contain fields named range, tve, fe, rfe, and limits. Some tests have an additional field labeled inputs. The range field refers to the x axis of the plot to be made. For the frequency test it will range from 58Hz to 62Hz for P class tests. The tve, fe, and rfe fields contain the respective values corresponding to the range field’s value. So the first tve value is the Total Vector Error of the PMU data when an off nominal frequency of 58Hz is used. The FE is the frequency error, and the RFE is the ROCOF error. The limits field contains the maximum tve, fe, and rfe respectively. The inputs field contains some necessary information for plotting additional information, or determining what test was used.

## Writing / Reading Test Data

The standard test dictionary can be saved by using the writeJson function and read using the readJson function. The writeJson function takes the first input as the test result dictionary, and the second input as a string for the name of the json file. The readJson file takes one input that is the name of the json file and outputs the test result dictionary.

## Plot Function

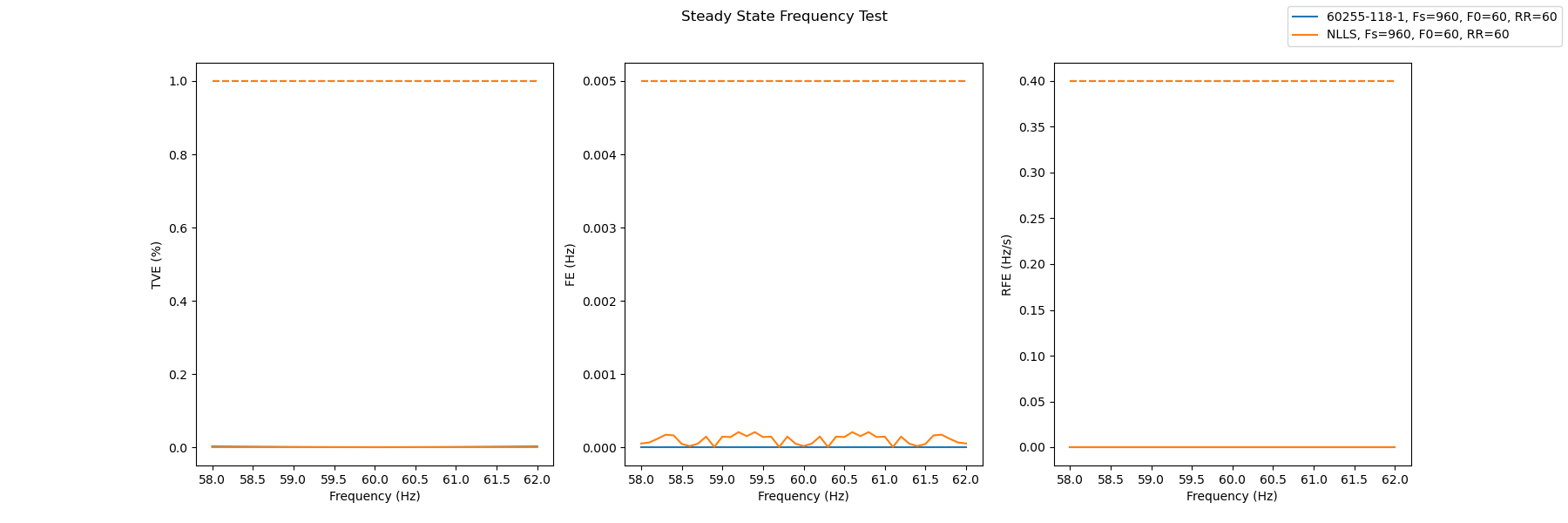
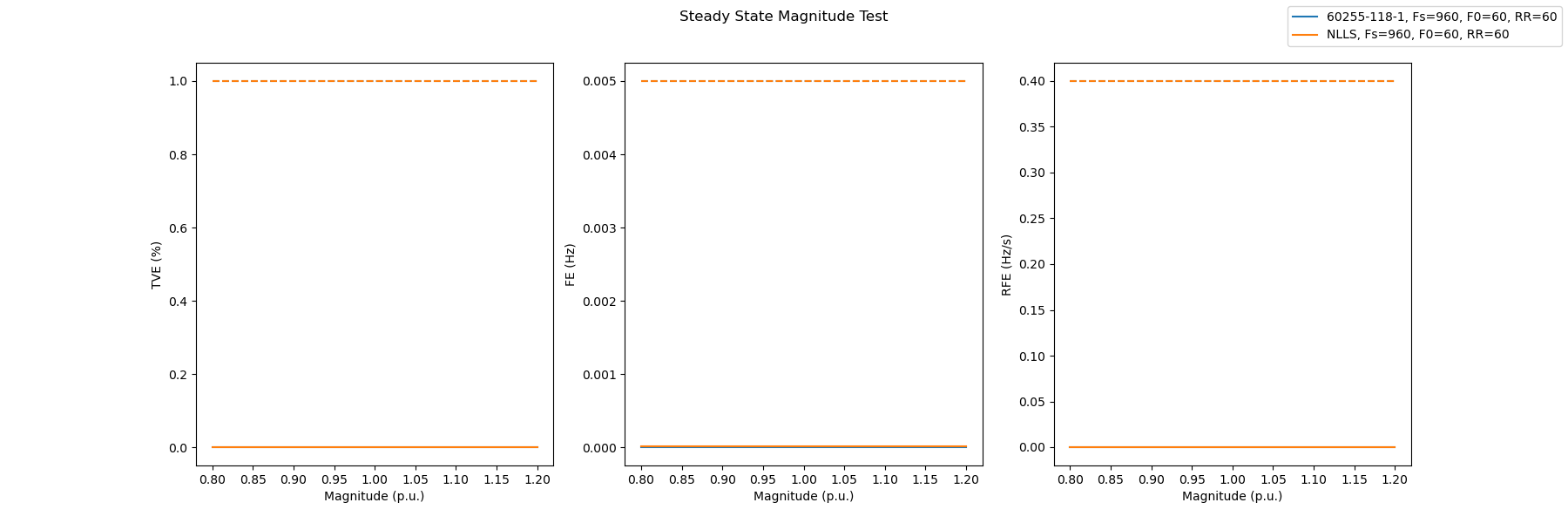
The plot function is used to generate and combine plots from different tests, in order to compare results. To do this, run a set of tests and save the test result dictionary to a variable. Do this for each pmu that is to be tested. To plot the combined results run the plot function with the tests as the only input on an array. The order will dictate the color of the tests. The function can handle situations where both p and m classes are used and will only plot the results of the tests contained in the dictionary. For example, if one test contained a frequency and harmonic test for a p class NLLS, and another test contained just the frequency test for the p class IEEE PMU, the plot function would show both the NLLS and IEEE results on the same plot for the frequency test, but would only plot the NLLS for the harmonic test. In addition to results, the maximum values are also plotted using dashed lines. They may overlap, so the color will usually be whatever the last test’s color is. For all plots besides the step plot, there are horizontal limit lines. As long as the results lie below any dashed line, they passed the test. In step tests, a vertical dashed line is placed at the point in which the value crosses a horizontal limit line. As long as the results stay below the limit line before the following vertical line it has passed the response time requirements of that test.

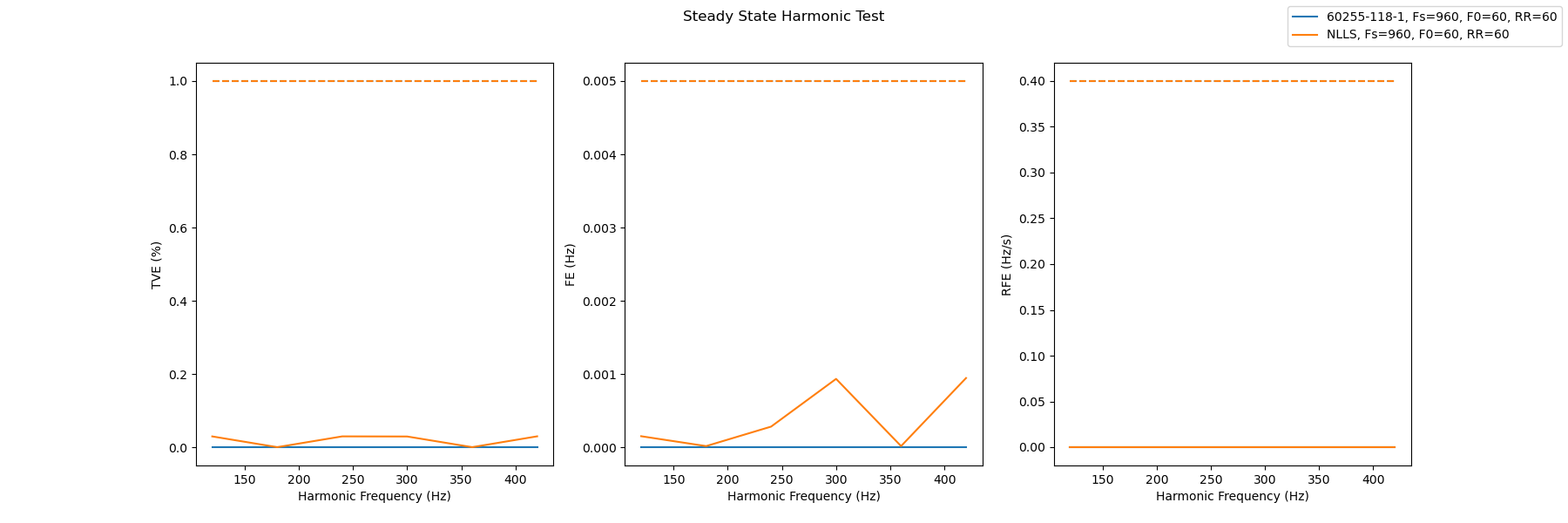
# Standard Test Results

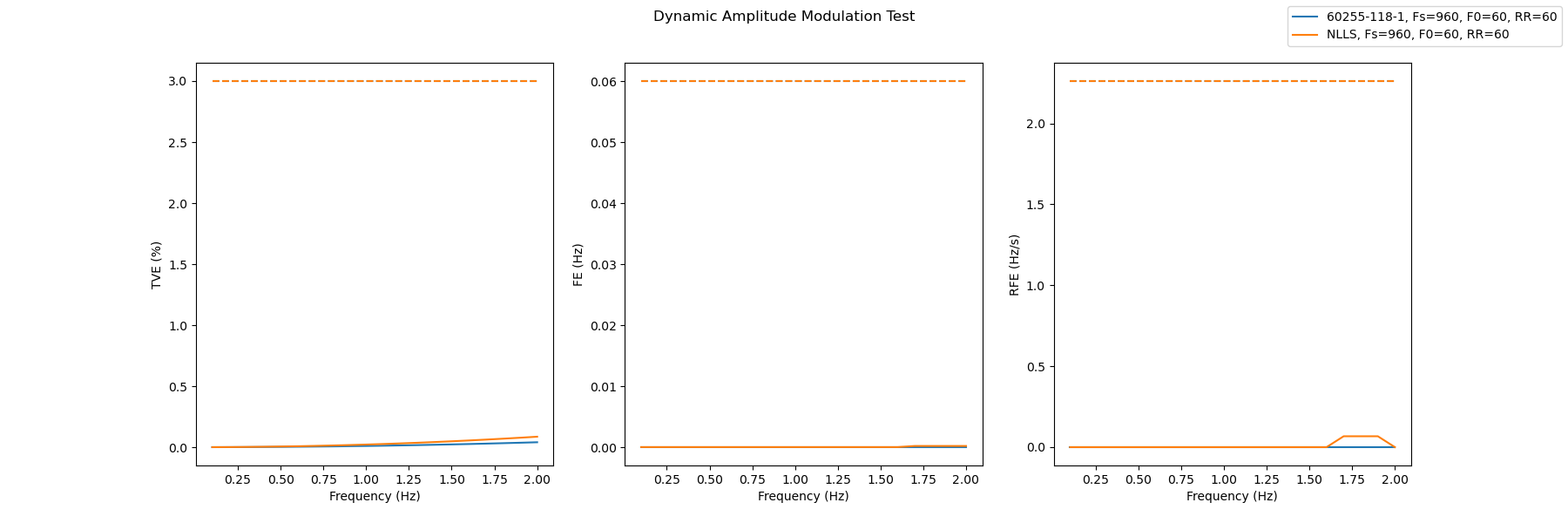
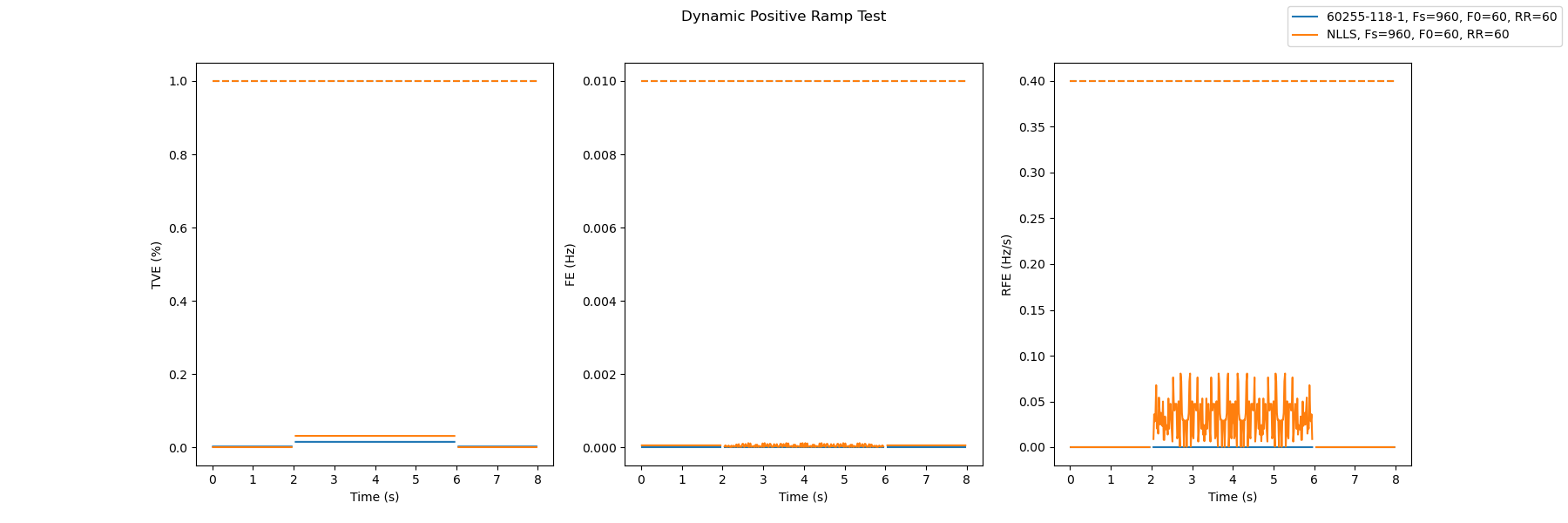
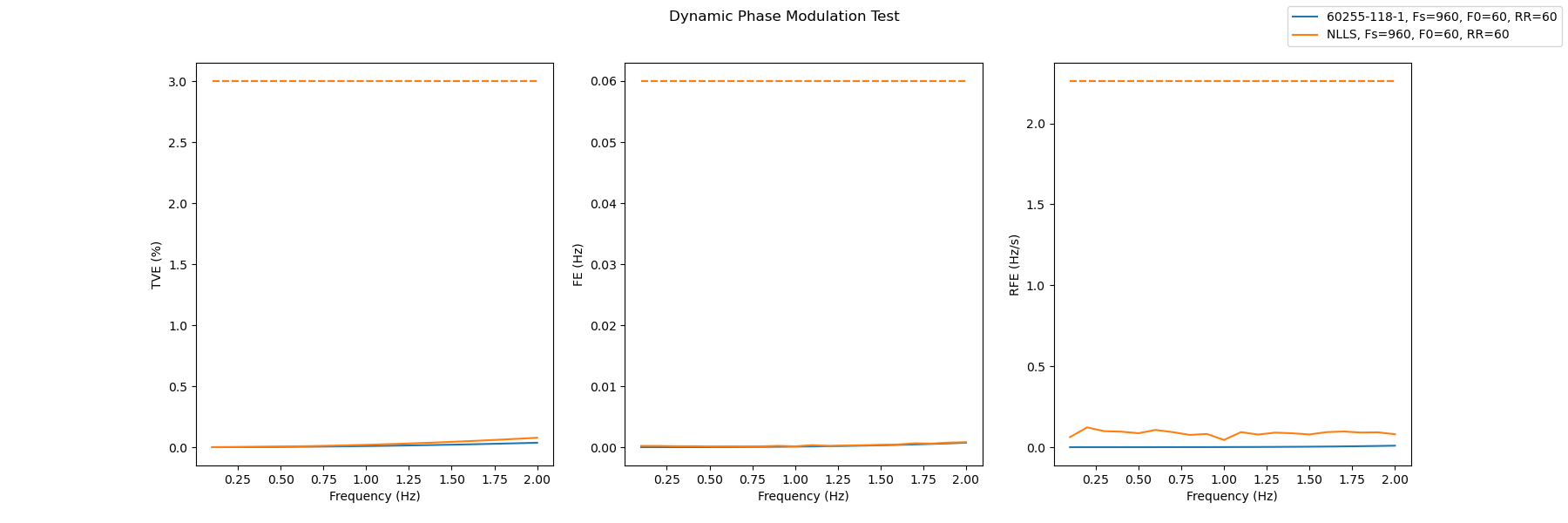
The ideal implementation of NLLS was tested against the IEEE P and M class standard algorithms. The tests were ran with a sampling rate of 960Hz, a nominal frequency of 60Hz, and a report rate of 60Hz and were plotted for comparison.

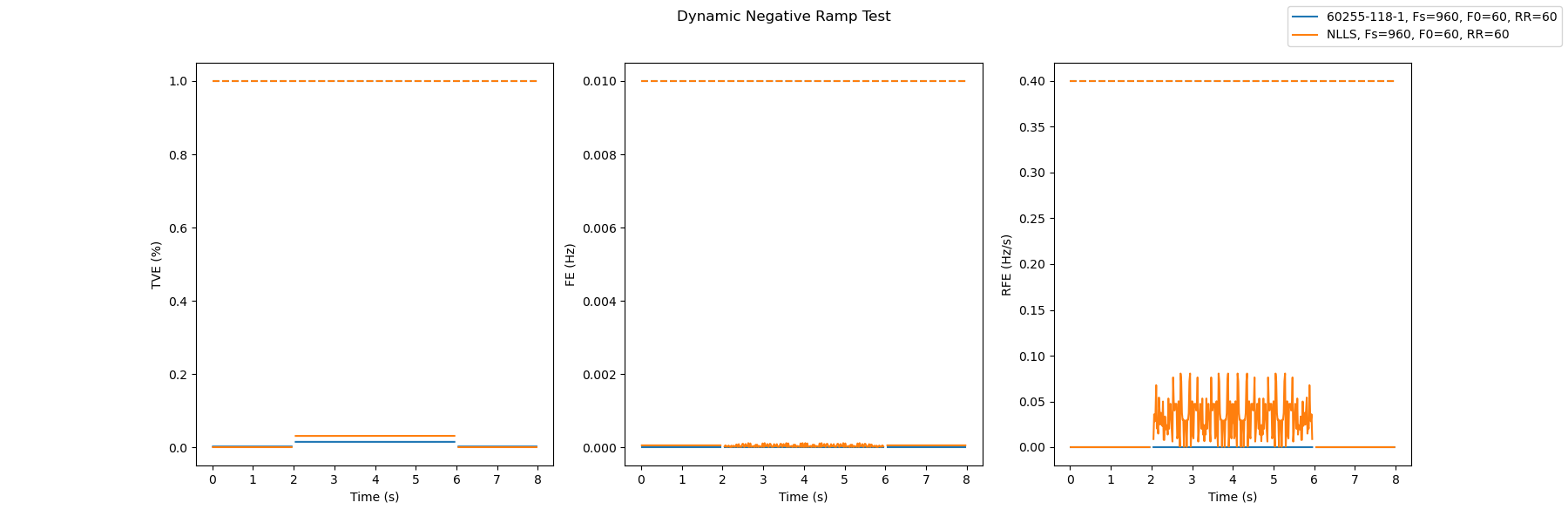
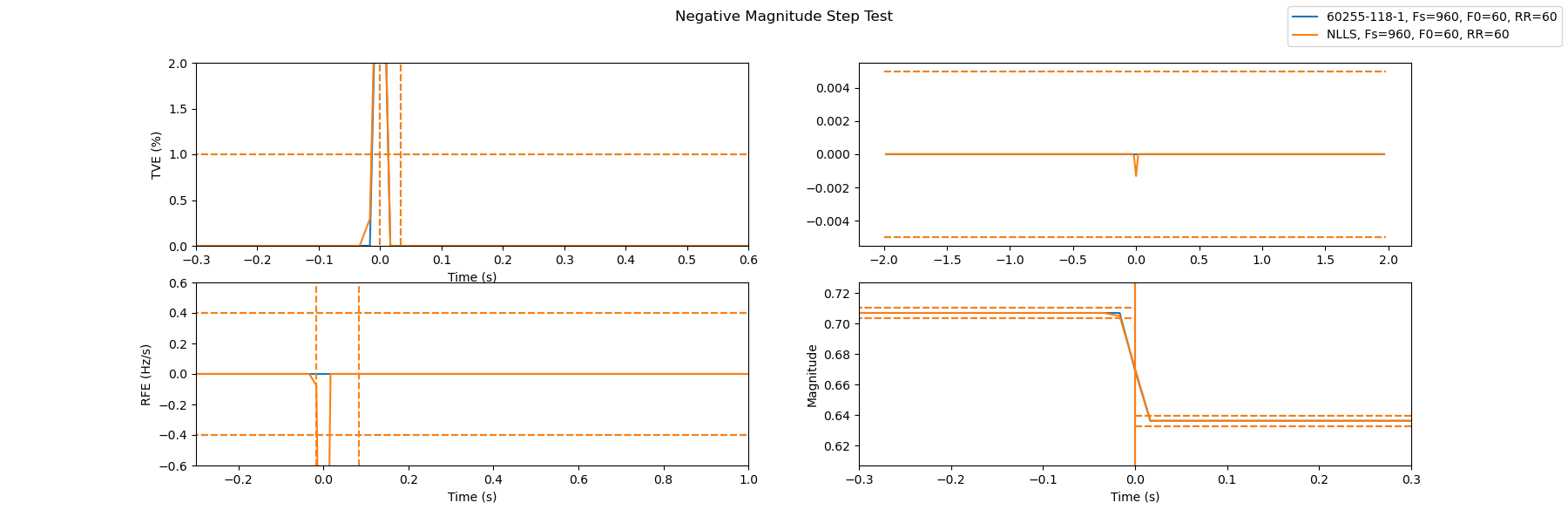
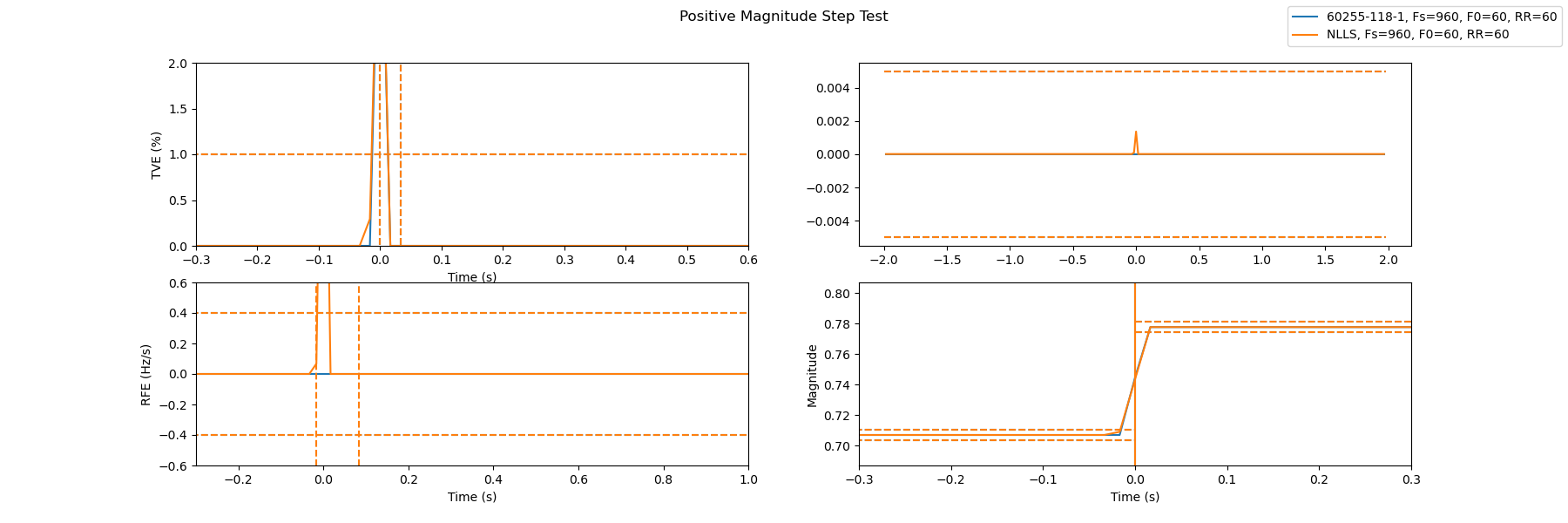
## P Class Results

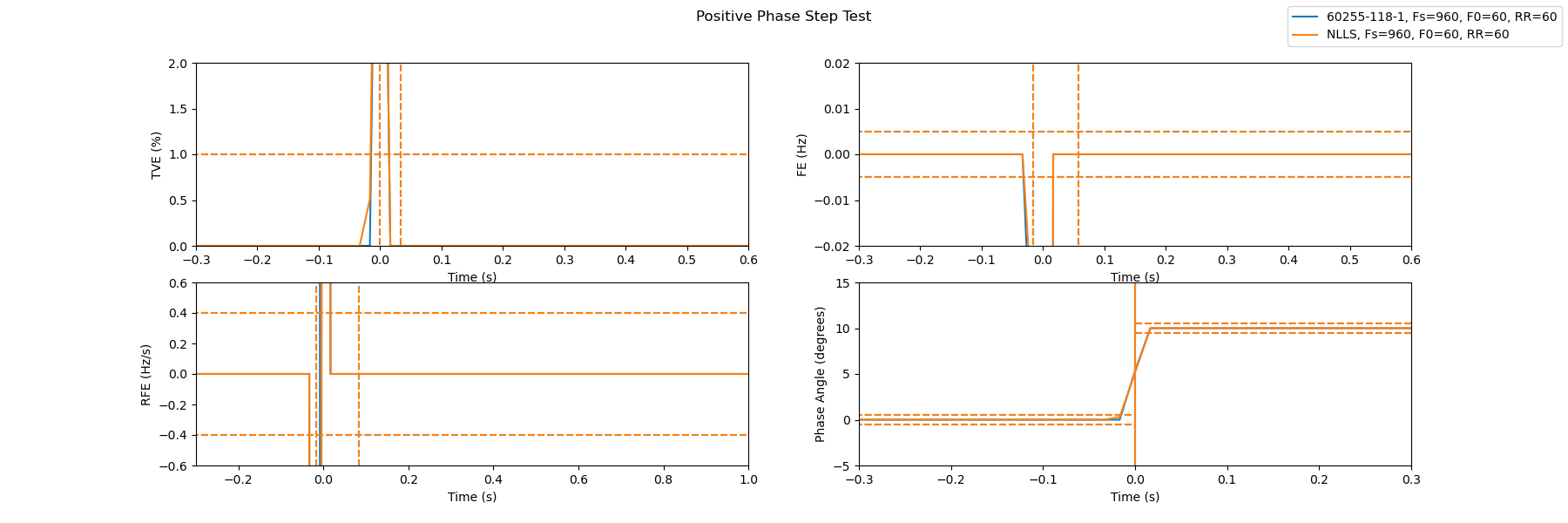
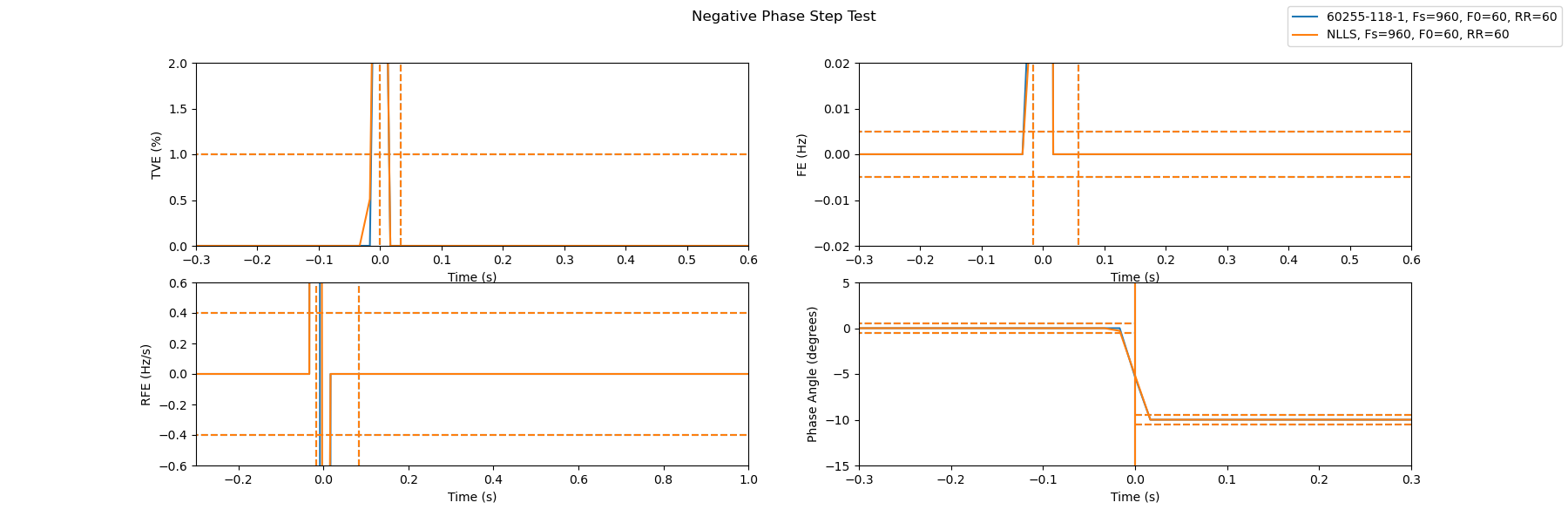
The P class NLLS algorithm was determined to get the best results when ran without a filter and with a window size of 2 cycles. Both PMUs pass all tests, and NLLS had good results but proved to have lower quality ROCOF results.





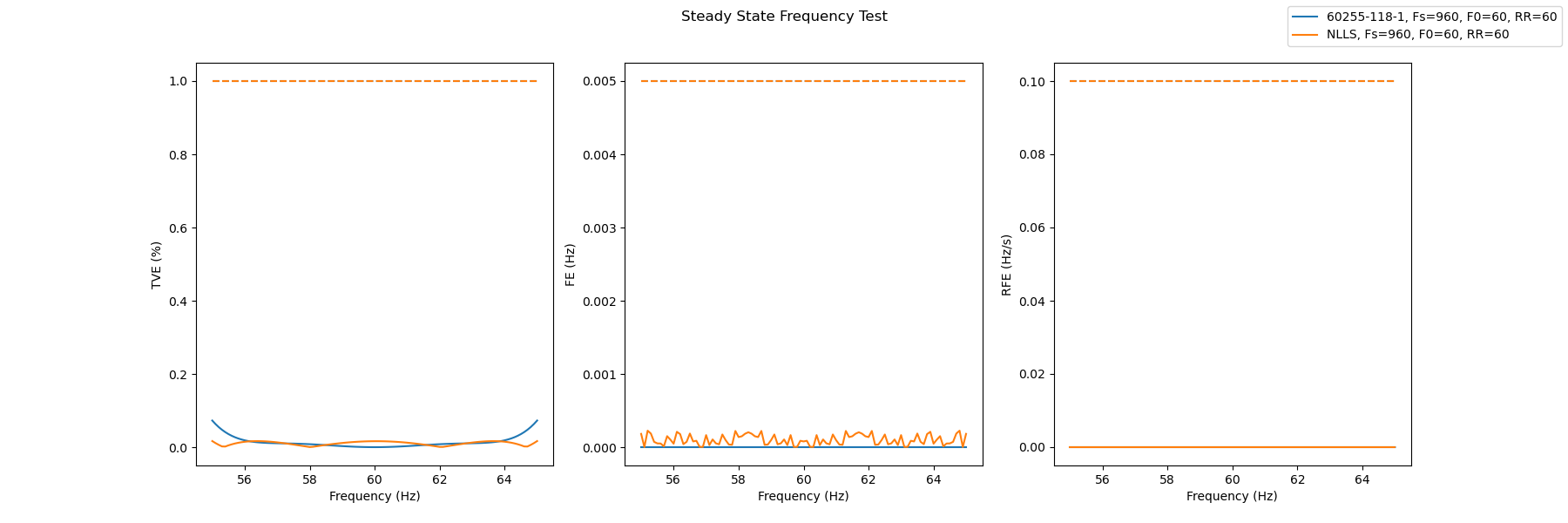
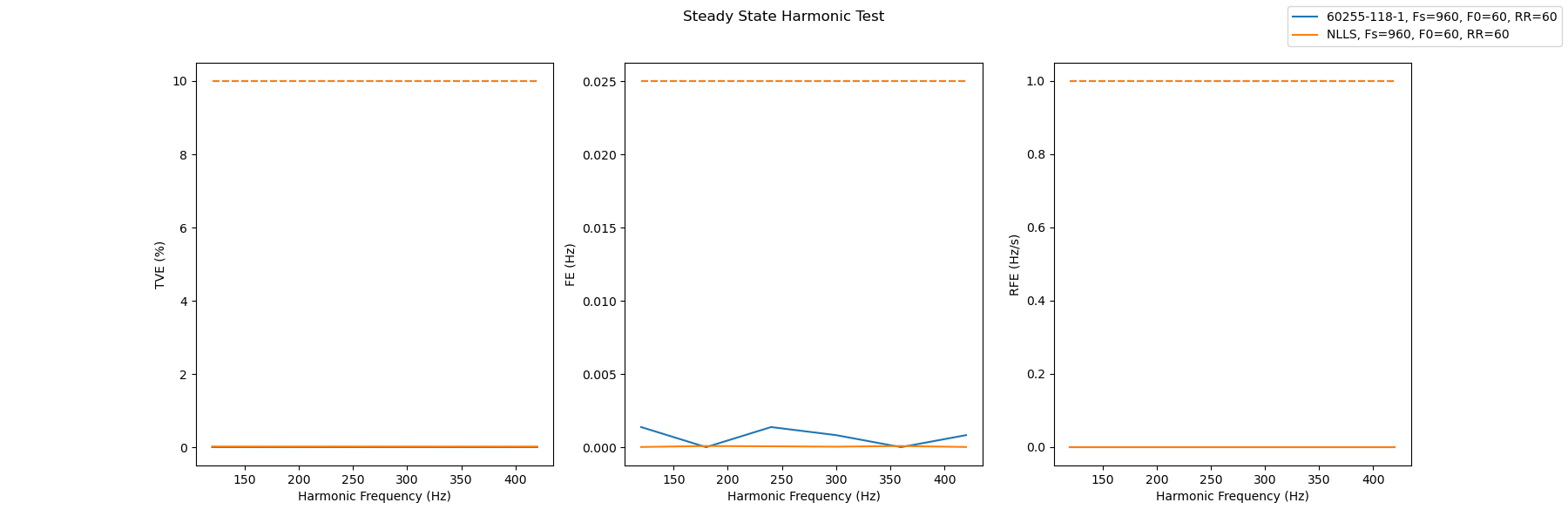
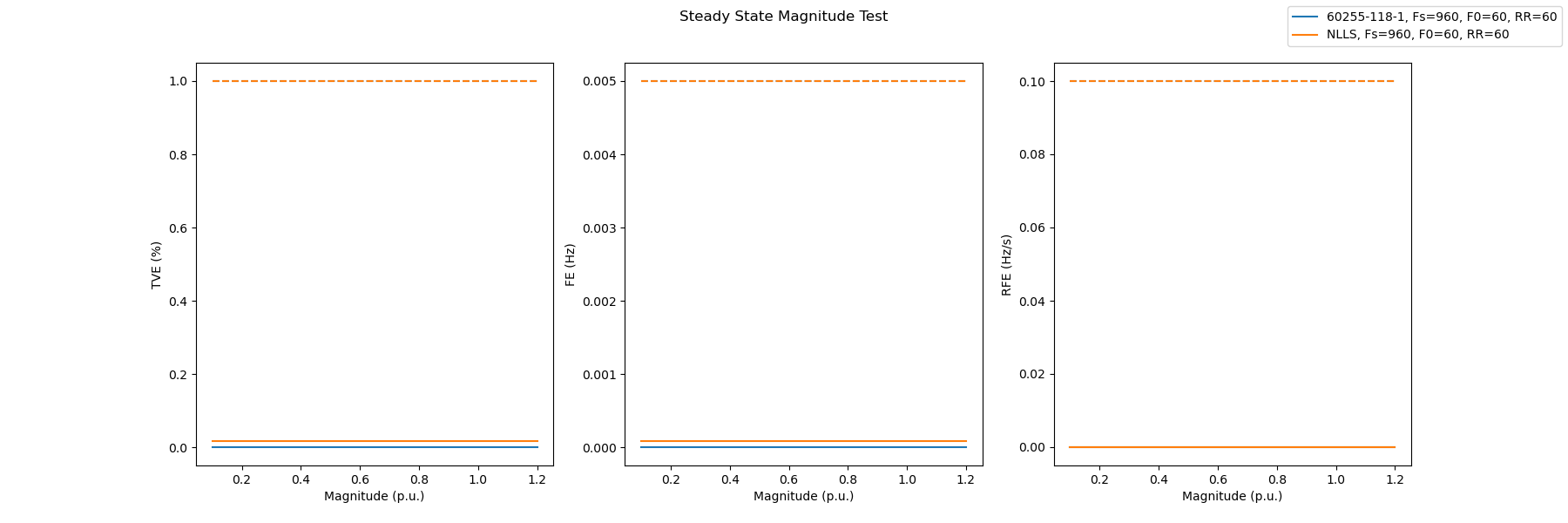


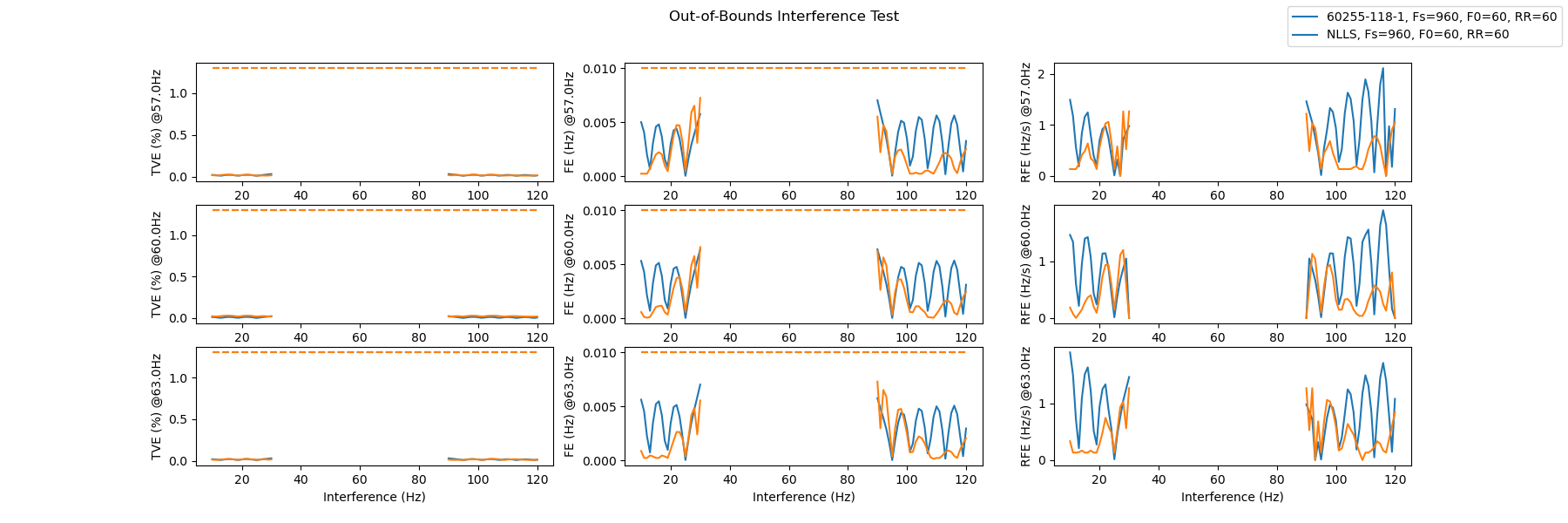
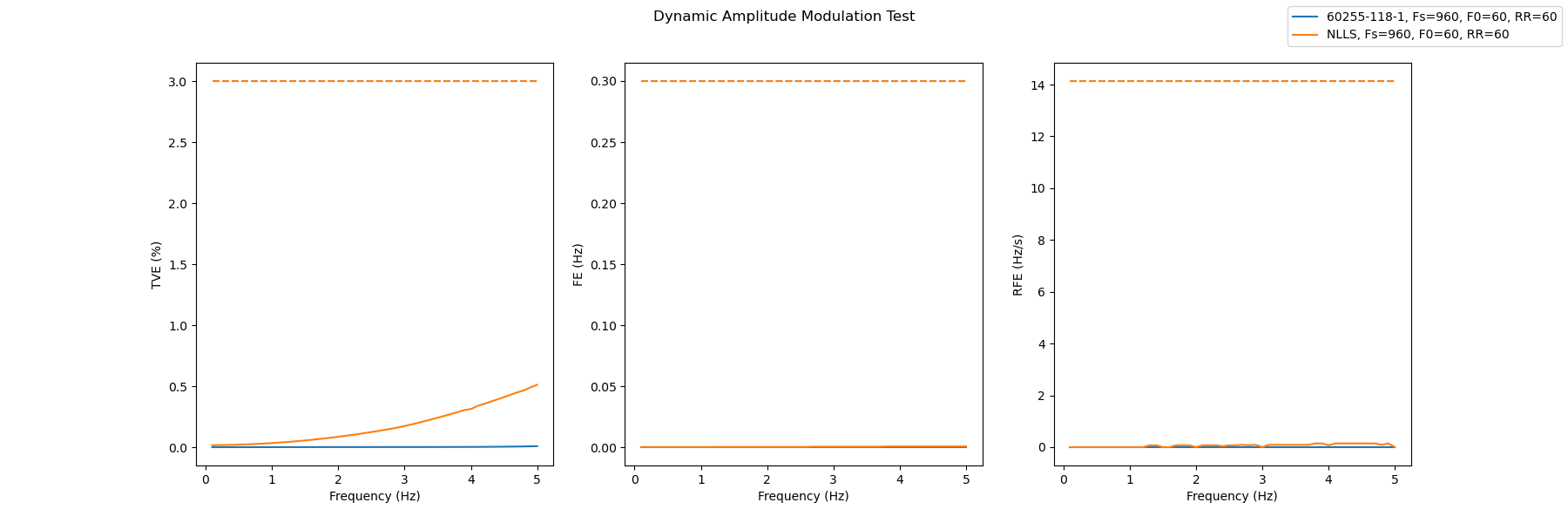
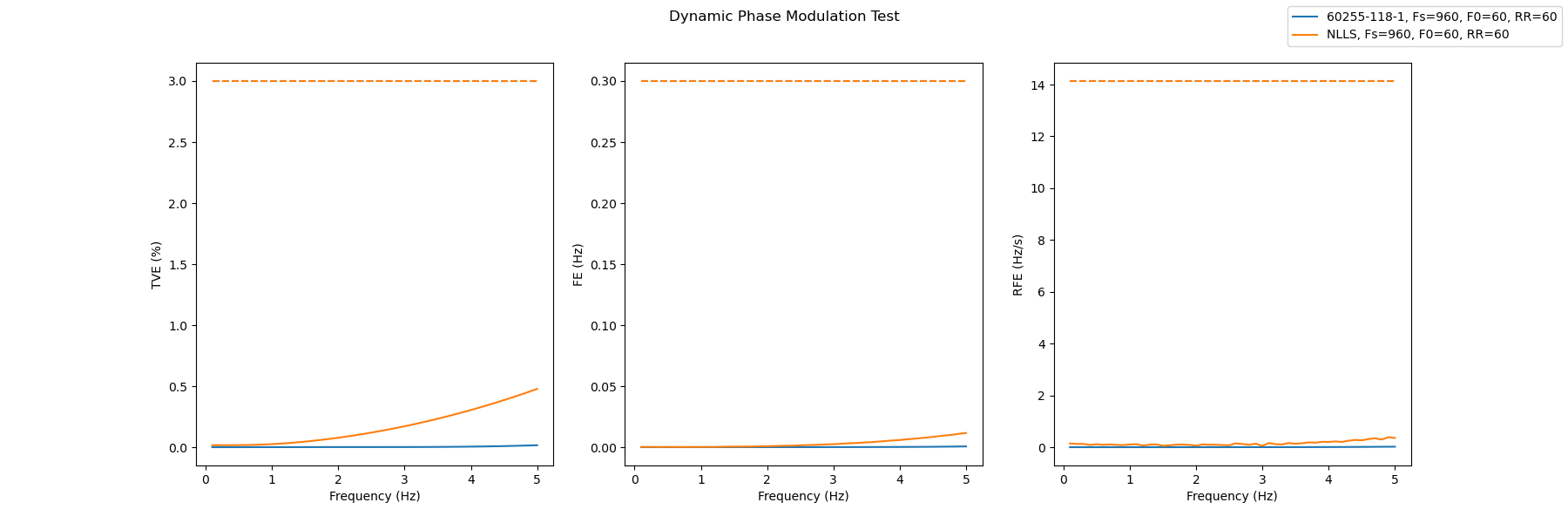


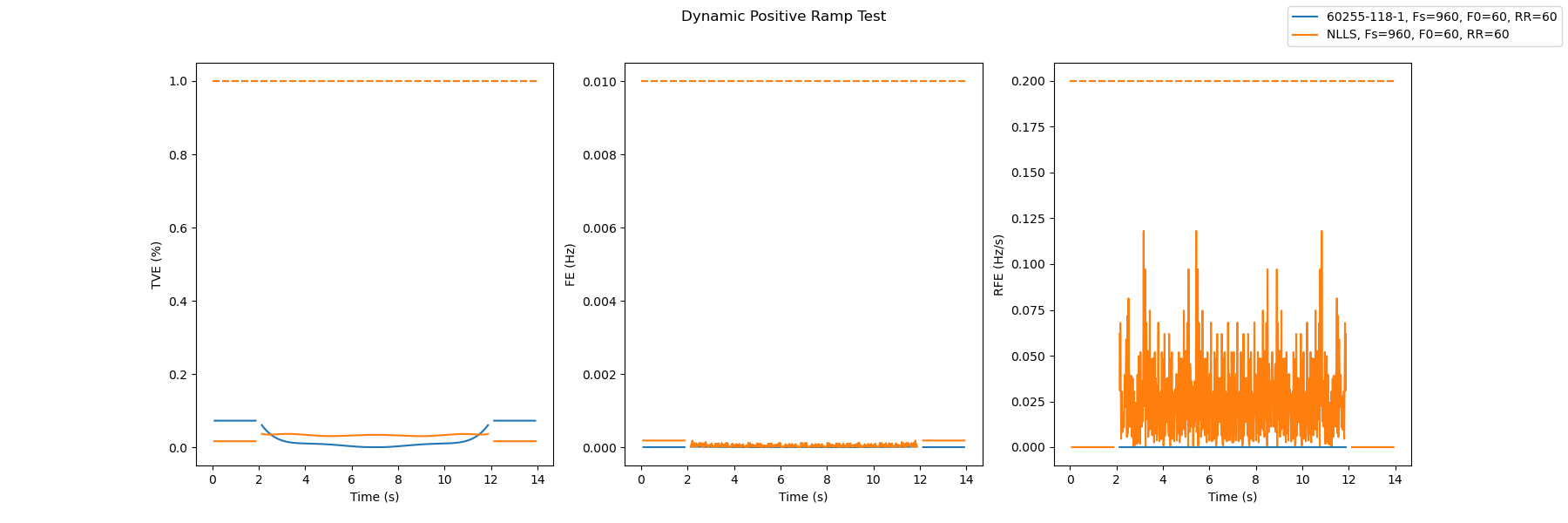
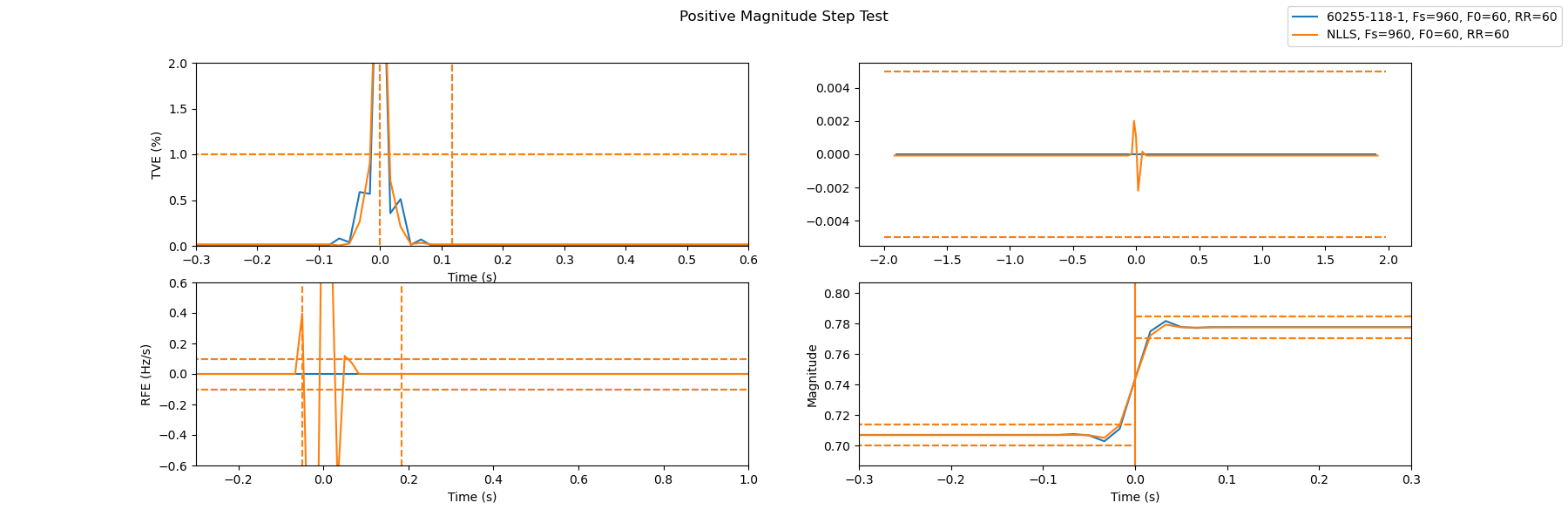
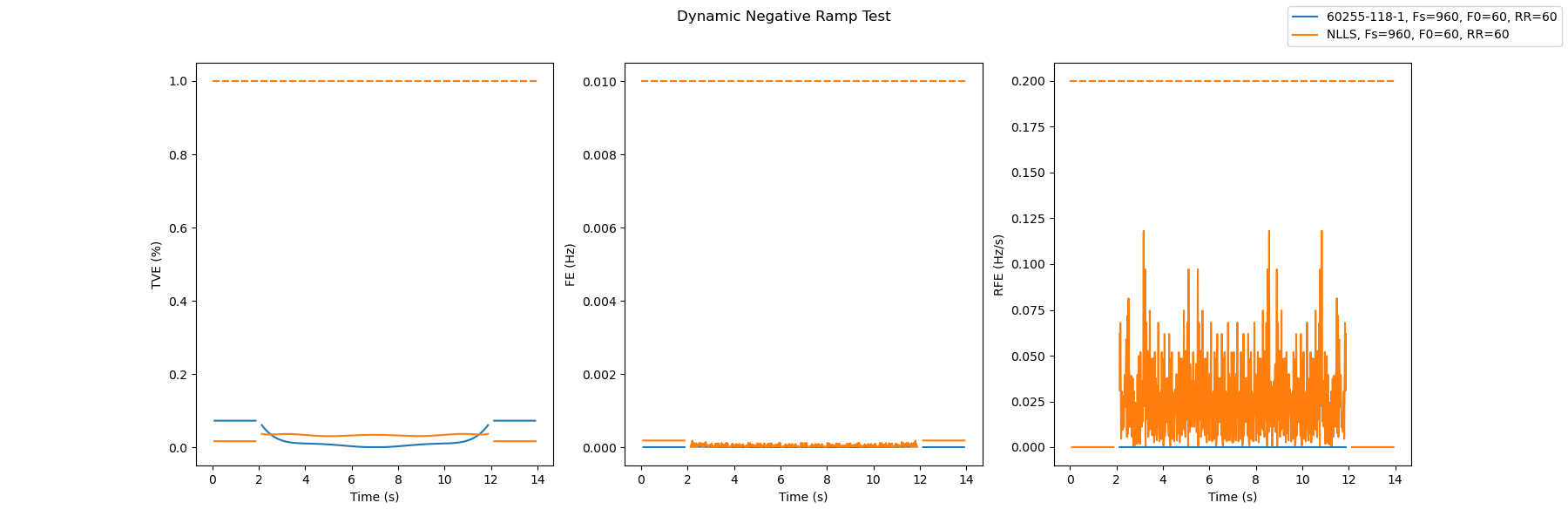


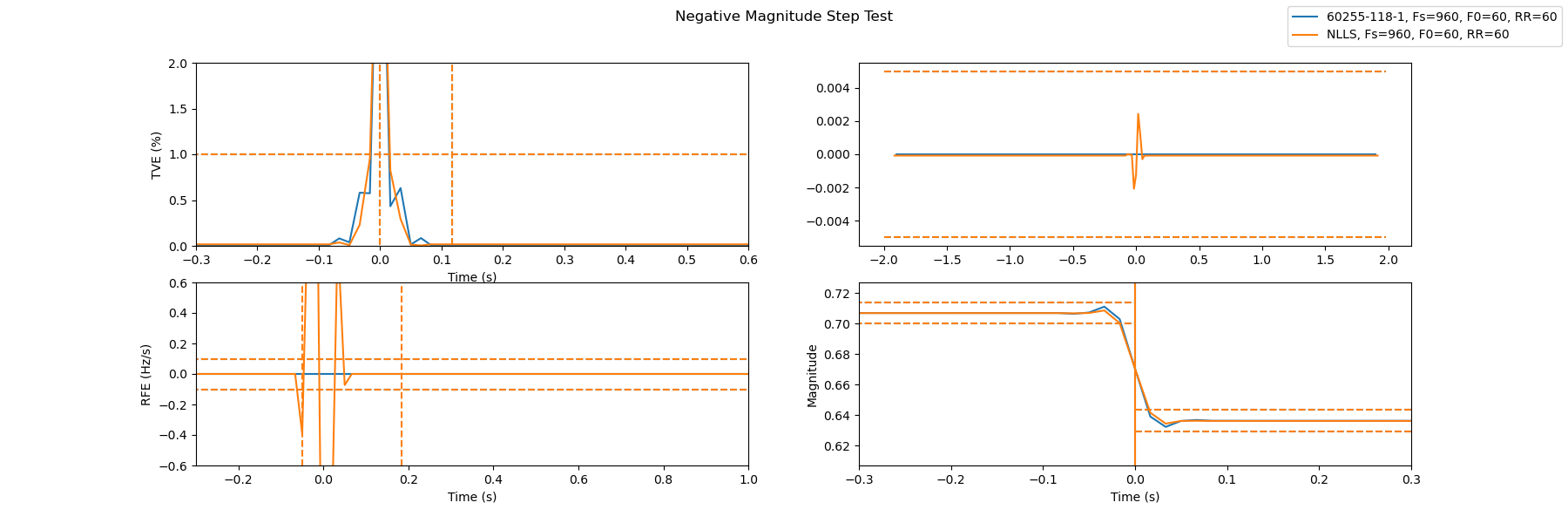
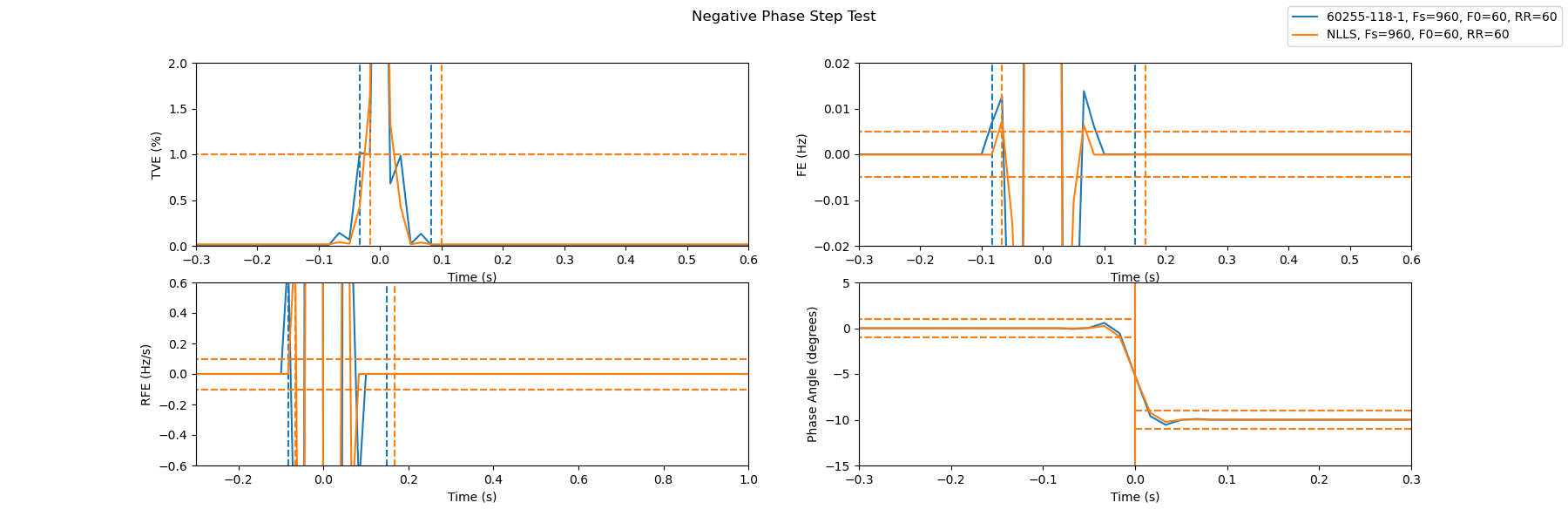
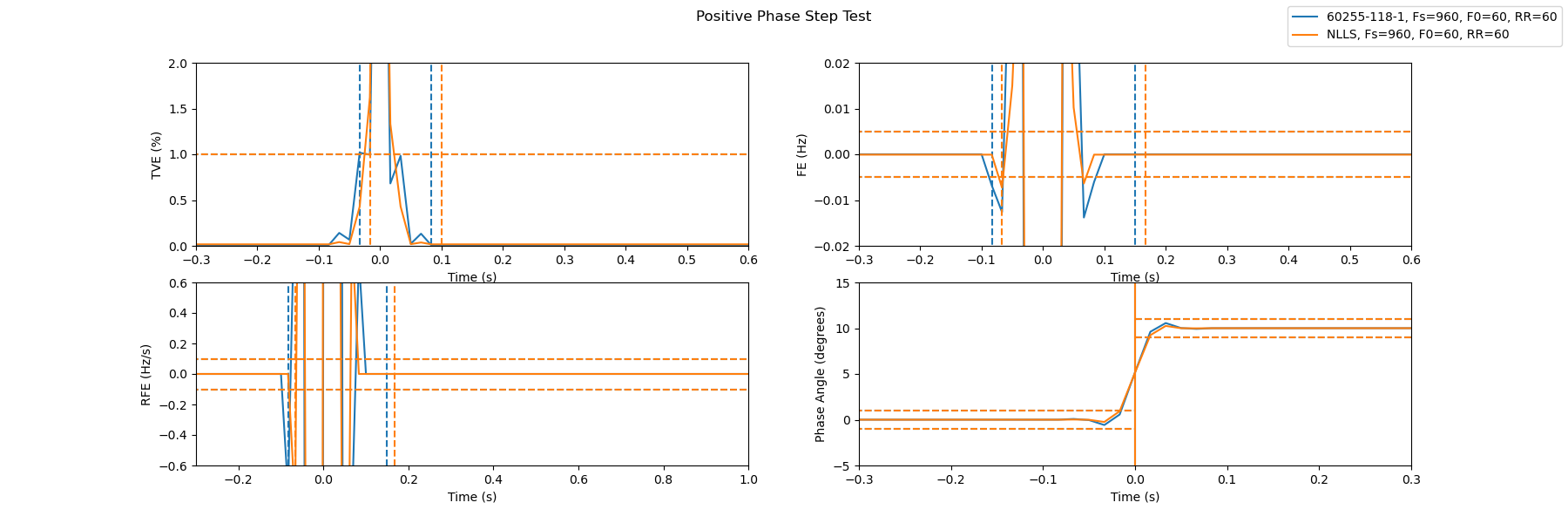
## M Class Results

The M Class NLLS algorithm was determined to get the best results when ran using a the pre-filter with a window size of 2 cycles. The difference between NLLS and IEEE are similar to that of the p class test, yielding acceptable results but poor ROCOF stability during the ramp test.









# Conclusions

The PMU Base Class and Estimator Classes were fully implemented in python, and the Standard Test Library successfully recreates the tests and plots shown in the NIST assessment paper. The framework leaves some room for adding new estimators and is well organized for future research and development.