SAND2022-14356C

SDynPy: A Structural Dynamics Python Library

Daniel P. Rohe

Sandia National Laboratories*
P.O. Box 5800 - MS0557
Albuquerque, NM 87123
dprohe@sandia.gov

ABSTRACT

The Python programming language has rapidly become one of the most popular programming languages. Its interpreted nature, clean syntax, and large amount of available open-source libraries gives users the ability to rapidly prototype, deploy, and share software to address engineering challenges. While there are several Python packages available that target specific structural dynamics techniques, there is not yet a general structural dynamics framework available in Python. SDynPy aims to fill that gap and introduces several classes to represent the typical data structures used in structural dynamics test and analysis. SDynPy defines Geometry objects to store and plot nodes, elements, and coordinate systems, Coordinate objects to store degree-of-freedom information, Data objects to store functions like time histories, spectra, or frequency response functions, Shape objects to store mode shapes, and System objects to store mass, stiffness, and damping matrices, as well as their associated transformation matrices to physical space. All SDynPy objects are built on the standard NumPy arrays and therefore natively support arbitrary dimensionality, broadcasting, and many of the NumPy functions. SDynPy also includes readers and writers for common structural dynamics data formats, curve fitters for fitting modes to test data, and various other signal processing functions.

Keywords: Python; Open Source; Structural Dynamics; Modal Analysis; Plotting;

1 Motivation

When performing structural dynamics analyses, it is often necessary to read data into a programming language, analyze those data with custom scripts, and report on those data. Matlab traditionally has been used for performing such analyses, but with the growth of open-source programming languages such as Python, it has become more feasible to use a free and open ecosystem to analyze structural dynamics data. Indeed, several structural dynamics-based Python packages have been and are being developed to tackle specific analysis problems: for example, extracting motions from images in `pyidi`[1], performing substructuring and transfer path analysis in `pyFBS` [1], or running vibration control in `Rattlesnake` [2].

One initial challenge in performing structural dynamics calculations in Python is the lack of a consistent framework and objects to represent the common data types encountered in structural dynamics. One can obviously use more standard objects such as NumPy `ndarrays` or Pandas `DataFrames` to store and manipulate data, but this can easily lead to channel or other bookkeeping errors if the wrong entries in these arrays are chosen, and common operations such as animating mode shapes or plotting large

[1] `pyidi` Github Repository: `https://github.com/ladisk/pyidi`

datasets can be challenging.

To overcome these issues, the SDynPy package was created. This paper will give a brief overview of the features available in SDynPy and then present an example problem that demonstrates some of SDynPy's functionality.

2 SDynPy Overview

SDynPy aims to offer a convenient framework to perform structural dynamics analyses. The typical workflow is the analysis-test-analysis loop where one might use a finite element model to inform a test setup, then run a test with that setup, then use the results of the test to update or otherwise compare to the model results. SDynPy is therefore well-suited to analyzing both test and analysis datasets. Additional functionality has also been added to perform more niche analyses such as substructuring or reduction/expansion.

2.1 Core Data Objects

SDynPy provides objects for common data types used in structural dynamics, namely:

- `CoordinateArray` – Representation of degrees of freedom defined by a node id and local coordinate system direction, e.g. `101X+`, used to aid in bookkeeping or visualizing measurement locations and directions
- `NDDataArray` – Representation of common data types from tests or finite element analyses. Subclasses of this class represent specific types of data, for example Time Histories (`TimeHistoryArray`), Frequency Response Functions (`TransferFunctionArray`), and others
- `ShapeArray` – Representation of mode shapes or deflection shapes
- `Geometry` – Representation of test or finite element geometry, consisting of nodes (`NodeArray`), coordinate systems (`CoordinateSystemArray`), tracelines (`TracelineArray`), and elements (`ElementArray`)
- `System` – Representation of mass, stiffness, and damping matrices defining a dynamic system, and allows for "reduced" systems in which a transformation is defined between the internal state and the physical state (e.g. a modal model or constrained substructure model)

SDynPy array objects are generally built using subclasses of NumPy's `ndarray`[2], and are therefore able to use the nice features of the `ndarray` type, including arbitrary dimensionality, broadcasting, and many of the NumPy functions that operate on `ndarrays` such as `intersect1d`, `concatenate`, `unique`.

2.2 Loading Test Data

In order to be useful for structural dynamics testing, SDynPy must have nice ways to load data into its objects from data acquisition software. SDynPy's data objects are heavily inspired by the Universal File Format[3] (UFF), and therefore SDynPy is able to read and write to a variety of UFF datasets, including:

55 - Data at Nodes, which corresponds to the `ShapeArray` in SDynPy

58 - Function at Nodal DOF, which corresponds to the `NDDataArray` in SDynPy

82 - Tracelines, which corresponds to the `TracelineArray` in SDynPy

151 - Header, which is not currently used in any SDynPy objects

164 - Units, which is not currently used in any SDynPy objects

[2] `ndarray` Documentation: `https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html`
[3] Universal File Format Specifications: `https://www.ceas3.uc.edu/sdrluff/`

1858 - Qualifiers for Dataset 58, which is not currently used in any SDynPy objects

2400 - Model Header, which is not currently used in any SDynPy objects

2411 - Nodes, which corresponds to the `NodeArray` in SDynPy

2412 - Elements, which corresponds to the `ElementArray` in SDynPy

2420 - Coordinate Systems, which corresponds to the `CoordinateSystemArray` in SDynPy

SDynPy will generally read in data using its `readuff` function, which will output a Python `dict` where they key is the dataset number and the value is the information inside the dataset. Many SDynPy objects have `from_uff` methods that will construct objects from this dictionary, for example:

```
# Import SDynPy
import sdynpy as sdpy
# Read in the data from the UFF file
uff_dict = sdpy.uff.readuff('path/to/uff/file.uff')
# Parse the data in the dictionary into a SDynPy Geometry
geometry = sdpy.Geometry.from_uff(uff_dict)
```

Note that while datasets 151, 164, 1858, 2400 can be read, they are not used in any SDynPy objects. This means that users can read these data into their workflows, but SDynPy will not automatically use this data in any way. Users must apply information from these datasets to SDynPy objects. For example, if the user wants to work in an inch-pound-second system, they can read in the Units Dataset 164 from the universal file, identify in which unit system the universal file is written, and scale the SDynPy objects returned by `readuff` appropriately. SDynPy will not yet automatically scale the information in the universal file.

SDynPy can also directly read time data from Rattlesnake's netCDF output [3] using the `read_rattlesnake_output` function. This function will return a `TimeHistoryArray` containing the test data as well as a Pandas `DataFrame` object representing the channel table.

Finally, SDynPy can also read data from Correlated Solutions' VIC3D Digital Image Correlation software [4]. It assumes the data has been exported to `.mat` files from the software, and will automatically generate time data and test geometry from the VIC files.

```
import sdynpy as sdpy
from glob import glob
# Get all mat files in the current directory with glob
files = glob('*.mat')
# Read in time and displacement data
geometry,time_data = sdpy.vic.read_vic3D_mat_files(files)
```

2.3 Finite Element Models

SDynPy also has capabilities to work with finite element models and data. SDynPy has readers and writers for the Exodus [5] file format, which is commonly used in the Sandia finite element codes. Similar to the UFF files, SDynPy objects can be created from these results using `from_exodus` methods found in those objects.

```
# Import SDynPy
import sdynpy as sdpy
# Read in the data from the UFF file
exo = sdpy.Exodus('path/to/exodus/file.exo')
# Parse the data in the dictionary into a SDynPy Geometry
geometry = sdpy.Geometry.from_exodus(exo)
```

SDynPy can also create small beam finite element models using its `beam` package, which can be used quickly for small academic studies. The `System` object also has a beam helper function that will easily make a simple beam `System` and `Geometry`.

```
# Create a beam system (mass, stiffness, and damping) and geometry
beam_system,beam_geometry = sdpy.System.beam(
    length=1.0,width=0.25,height=0.4,num_nodes=20,material='steel')
```

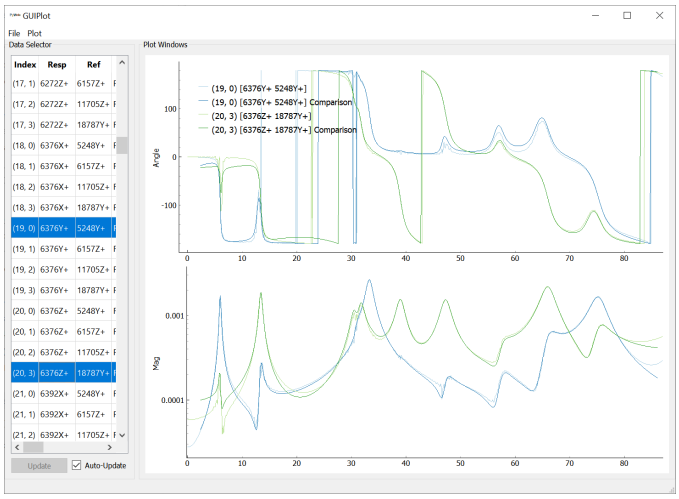


Figure 1: Interactive Plotting using `GUIPlot`

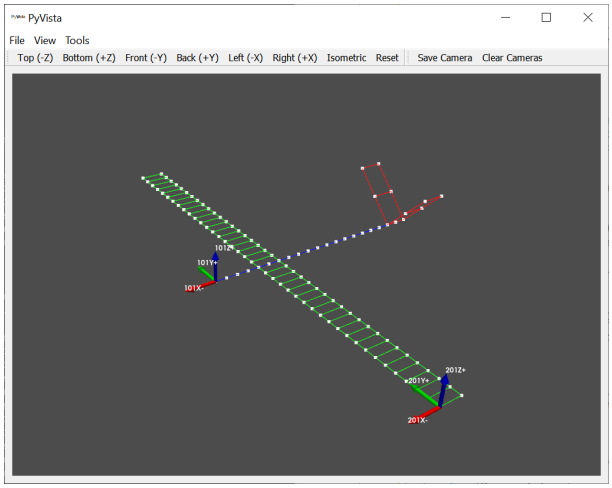


Figure 2: Visualizing coordinates on geometry

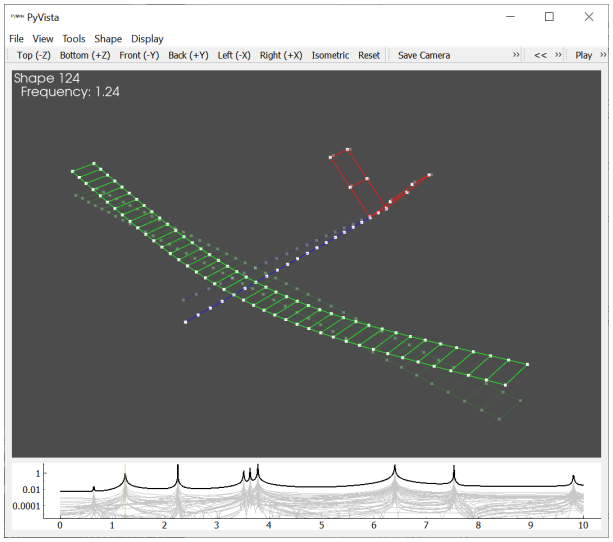


Figure 3: Visualizing deflection shapes with interactive plot

2.4 Experimental Modal Analysis

SDynPy has two modal curve-fitter implementations. The Synthesize Modes and Correlate (SMAC) [6] algorithm is a modal-filter based curve-fitter. A multi-reference Z-domain curve fitter is also included in SDynPy [7]. Both curve fitters can be run purely with code, allowing automation, or through a graphical user interface, allowing more interaction with the data. Figures 23–25 in the demonstration workflow of Section 3 show example usage.

2.5 Data Visualization

SDynPy has a rich selection of interactive data visualization tools. For plotting 2D datasets such as frequency response functions, the `GUIPlot` is useful for interactively selecting which functions to plot, as well as how to plot them (real/imaginary, magnitude/phase, etc.). Two datasets can also be passed simultaneously to compare functions. Figure 1 shows an example.

SDynPy can also interactively plot 3D geometry and degrees of freedom (Figure 2), as well as animated deflections from mode shapes, deflection shapes (Figure 3), and time data.

2.6 Documentation

SDynPy can automatically generate documentation for a test report. Using its `sdynpy.doc.ppt` subpackage, it can generate PowerPoint files populated with mode tables, shape animations, and data plots. Alternatively, the `sdynpy.doc.latex` can generate portions of a LaTeX source document that can be `included` in a full test report document.

2.7 Signal Processing

Finally, SDynPy makes a large variety of signal processing functions available to the user. These are included in submodules:

- `camera` – Functions for working with images and pinhole camera geometry
- `complex` – Functions for working with complex numbers
- `correlation` – Functions for computing correlation metrics such as the Modal Assurance Criterion
- `cpsd` – Functions for computing cross-power spectral density matrices
- `frf` – Functions for computing frequency response functions
- `generator` – Functions to generate common signals such as pseudorandom, burst random, and chirp
- `harmonic` – Functions for dealing with sinusoidal signals
- `integration` – Functions to help perform time integration of dynamic systems
- `rotation` – Functions for dealing with rotation representations

3 SDynPy Demonstration

This example will demonstrate a typical modal analysis workflow using SDynPy. A model will be used to inform instrumentation locations, and then (simulated) test data will be acquired to compare back to the model. In this example, we will:

1. Load in a finite element model and use it to select instrumentation locations
2. Simulate a modal test on the test article to collect time data, including typical data checkouts
3. Compute frequency response functions from the time data
4. Fit modes to the FRF data
5. Compare modal fits back to finite element model results
6. Perform finite element expansion using the System Equivalent Reduction Expansion Process (SEREP) [8]
7. Create a quicklook report using the automatic documentation features

3.1 Importing the Required Modules and Setting up Plotting

Installing and setting up Python, SDynPy, and its required dependencies is outside the scope of this paper, so we will begin by importing the required dependencies. We will import SDynPy along with NumPy for numeric calculations and matplotlib for 2D plotting. We will set up some default parameters for the 3D plotting of geometry, mode shapes, and deflection shapes.

```
# Import required modules
import numpy as np # Used for numeric calculations
import matplotlib.pyplot as plt # For 2d plotting
import sdynpy as sdpy # Used for structural dynamics features

# Since we will be plotting a lot of shapes, we will set up some options to use
# See the documentation for sdpy.Geometry.plot for these options.
plot_options = {'node_size':0,'line_width':1,'show_edges':False,
                'view_up':[0,1,0]}
```

3.2 Load the Finite Element Model

The next step is to load the finite element model into the software. To reduce data sizes, the finite element model will be reduced down to just the exterior surfaces, as accelerometers can only be placed on surfaces of the structure, not on the inside of material volumes. This step will eliminate all of the interior nodes in the model and transform 3D volume elements to 2D surface elements.

```
# Specify the filename of our finite element model
filename = 'airplane-out.exo'

# Load the finite element model and reduce it to just exterior surfaces
exo = sdpy.Exodus(filename)
fexo = exo.reduce_to_surfaces()
```

3.3 Converting Finite Element Data to SDynPy Objects

At this stage, we will take the finite element data and transform it into SDynPy objects that include more advanced functionality. This will involve the extraction of the mesh geometry into a SDynPy `Geometry` object and the finite element mode shape results into a SDynPy `ShapeArray` object.

3.3.1 Extracting the Geometry

The first step is to create a `Geometry` object from the node positions and element connectivity in the finite element model. This can be done easily using the `from_exodus` function.

The purpose of the present analysis is to determine instrumentation positions, so we would ideally like to load in the geometry with local coordinate systems oriented the same way that accelerometers might be positioned. Accelerometers are most often oriented normal to the surface to which they are attached, and often pointed towards some "preferred" direction of the test article (e.g. towards the front). SDynPy offers the ability to automatically create local coordinate systems oriented normal to the test geometry and towards some preferred directions by passing the `local=True` argument to the `from_exodus` function, and also passing these preferred directions.

```
# We will specify a preferred direction in the nose-wise direction (0,0,1)
# The secondary preferred direction will be "up" (0,1,0)
geometry = sdpy.geometry.from_exodus(fexo,local=True,
                                     preferred_local_orientation=[0,0,1],
                                     secondary_preferred_local_orientation=[0,1,0])
```

We may wish to visualize these coordinate systems to ensure they were created correctly. We can do this by creating a `CoordinateArray` of the degrees of freedom in the model and plot them on the model, which is shown in Figure 4. We can see that the local z axis, drawn in blue, is pointed away from the surface, and the local x axis (drawn in red) is always pointing towards the nose of the airplane (as much as it can given the constraint on the z axis), which was the defined preferred direction.

```
# First define a set of all of the degrees of freedom in the model
coordinates = sdpy.coordinate.from_nodelist(geometry.node.id)
# Now we plot the geometry with the coordinates overlaid, scaling the arrows to
# be smaller to accommodate the mesh density.
plotter = geometry.plot_coordinate(coordinates,arrow_scale=0.005,
                                   plot_kwargs=plot_options)
```

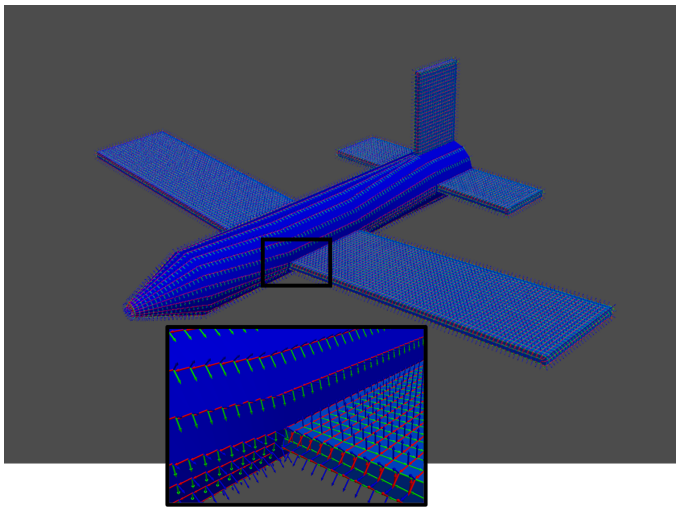


Figure 4: Local coordinate systems drawn on the airplane geometry, with inset close-up showing local coordinate systems oriented to the surfaces of the model.

3.3.2 Extracting the Mode Shapes

We will now extract the mode shape results from the finite element model. We will use these to inform sensor selection for our modal test. We can immediately extract shape data from the model using the `from_exodus` class method in the `ShapeArray`. We will also remove the small negative frequencies for the rigid body modes that occurred from the parallel eigensolution used in the finite element model by indexing the shapes' frequency vector and setting those values less than zero equal to zero.

```
# Now we will extract the shapes.  Note that these shapes will be in the "global"
# coordinate system from the exodus file, so we will need to transform them.
shapes_global = sdpy.shape.from_exodus(fexo)
# One thing to note is that our exodus file came out with slightly negative
# rigid body frequencies, so let's correct that now
shapes_global.frequency[shapes_global.frequency < 0] = 0
```

We should note that the shape data stored in the finite element model is defined in the global coordinate system, and therefore cannot be plotted directly on the geometry we just created with local coordinate systems. If we tried to plot this shape on our geometry, they will be distorted due to incompatible coordinate systems. The shape in Figure 5 should show a rigid body rotation, but instead it looks like a combination of torsion of the body and ballooning of the tail.

```
# If we were to plot the shapes with the geometry now, the results would look
# bad due to the incorrect coordinate system
plotter = geometry.plot_shape(shapes_global,plot_options)
```

SDynPy has the ability to perform coordinate system transformations on shape data using the `transform_coordinate_system` method of `ShapeArray` objects, so we will do that next. To perform this transformation, we will need two copies of the geometry. The first copy will have coordinate systems defined that the shapes will be transformed *from*; in this case, the shapes are currently defined in the global coordinate system. The second copy will have coordinate systems defined that the shapes will be transformed *to*, which are the local coordinate systems in which we wish to place our sensors. The latter geometry we have just created in Section 3.3.1. For the former, we can simply re-load the geometry *without* the `local` argument set to `True`, and SDynPy will instead create a geometry using only the global coordinate system.

Once we have both sets of geometry, we can pass them as arguments to the `transform_coordinate_system` method of our global mode shapes `shapes_global`. This will return a set of shapes transformed to the local coordinate systems defined in `geometry`, and if we now plot those shapes with that geometry, they will appear correctly as a rigid body rotation, as shown in Figure 6.

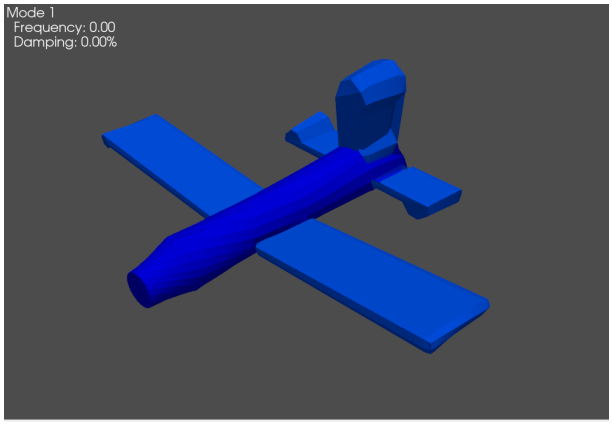


Figure 5: Distorted rigid body rotation shape due to global shapes plotted with local geometry

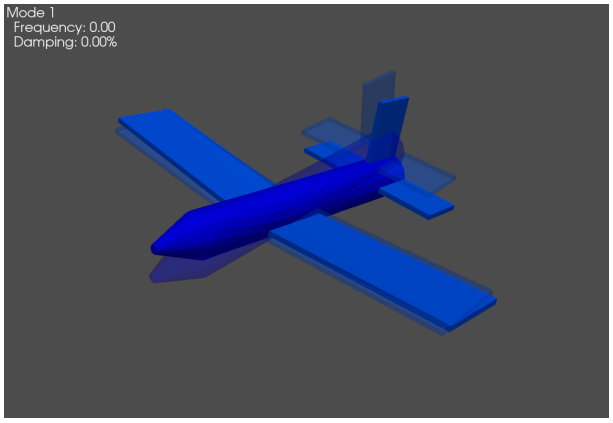


Figure 6: Rigid body rotation shape shown correctly after shapes are transformed into local coordinate systems

```
# Get the geometry with global coordinate systems by setting local=False
geometry_global = sdpy.geometry.from_exodus(fexo,local=False)
# Now we can simply transform the shapes using the global and local geometry.
# The shapes are currently in the global coordinate system, so that is our
# "from" geometry.  The geometry we wish to have the shapes in is the "to"
# geometry
shapes = shapes_global.transform_coordinate_system(geometry_global, geometry)
# Now if we plot the shapes we will see that they look more reasonable
plotter = geometry.plot_shape(shapes,plot_options)
```

3.4 Optimizing Instrumentation

Now that we have geometry and mode shapes, we will perform instrumentation optimization using the effective independence algorithm [9]. In SDynPy, users can give a set of candidate degrees of freedom, and the algorithm will downselect the degrees of freedom to keep. Users can also "group" degrees of freedom into triaxial accelerometers (or any other convenient grouping), so groups of degrees of freedom are kept or discarded as one. We will use this approach here.

First, we will create a candidate set of degrees of freedom. We will reduce the 10,000+ nodes in the model to a more manageable candidate set by using a grid to select a subset of points.

```
# First we will create our candidate set of degrees of freedom.  We will begin
# by selecting a subset of nodes based on a grid of points
grid_size = 0.25
candidate_nodes = geometry.node.by_grid(grid_size)
# Get the ID numbers to create a coordinate array
candidate_node_ids = candidate_nodes.id
```

One nice thing about SDynPy's integration with NumPy arrays is that SDynPy objects can use broadcasting. For example, if we take a set of nodes as a $n_{nodes} \times 1$ array and a set of directions x, y, z as a 1×3 array, the resulting array will be broadcast into an array of shape $n_{nodes} \times 3$ where all combinations of nodes and directions are represented. This quickly allows us to create a set of candidate degrees of freedom already grouped as triaxes that we can use for the sensor optimization. We can plot this set of degrees of freedom on the geometry to visualize the candidate set of sensors, as shown in Figure 7.

```
# Now use broadcasting to construct a set of degrees of freedom grouped by triax
# Here the nodes are the node ids in the geometry, and the directions are
# 1,2,3 corresponding to X+,Y+,Z+.  We pass in a (n_nodes x 1) array for the
# nodes and a (3) array for the direction, which will be expanded to a
# (n_nodes x 3) array output where the rows correspond to the node id and the
# columns correspond to each direction
candidate_dofs = sdpy.coordinate_array(candidate_node_ids[:,np.newaxis],[1,2,3])
# Plot the candidate degrees of freedom on the geometry for visualization
geometry.plot_coordinate(candidate_dofs,arrow_scale=0.01,plot_kwargs = plot_options)
```

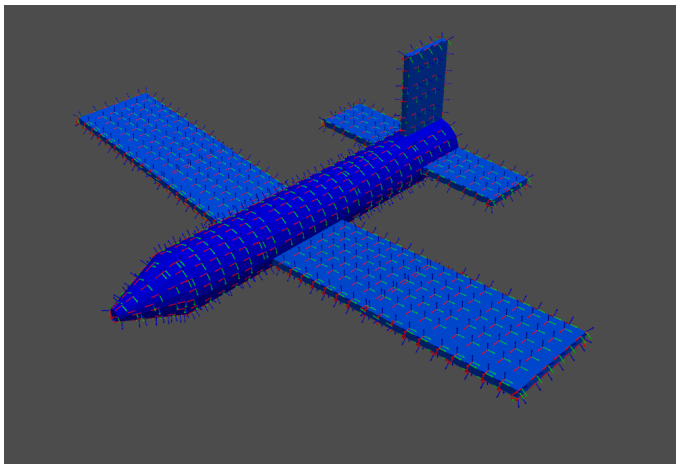


Figure 7: Candidate degrees of freedom used for sensor selection.

With candidate degrees of freedom selected, we can now select the shapes used for the instrumentation selection. For this test, we will be considering the bandwidth below 200 Hz, so we will use modes below 300 Hz to optimize sensors. We will extract a shape matrix from these shapes. To aid in bookkeeping, we can actually use the set of candidate degrees of freedom to index `ShapeArray` objects, which will return a shape matrix with the correct degree of freedom ordering and sign-flipping.

```
# Set the bandwidth used for shape optimization
shape_bandwidth = 300
# We can then select our target shapes by comparing the frequency of each shape
# to our bandwidth
target_shapes = shapes[shapes.frequency < shape_bandwidth]
# We need the shape matrix that we will be targetting.  We can get this by
# indexing the target_shapes with our candidate dofs
shape_matrix = target_shapes[candidate_dofs]
# Note that the default configuration of the shape matrix is to have the degrees
# of freedom as the last dimension and the shape of the shape array as the
# first dimension(s).  While this is generally transposed from the shape
# matrices we are used to (ndof x nmode), it is more natural this way with
# numpy's broadcasting capabilities.  However, the effective independence wants
# the following order (dof_group x dofs_in_group x mode) so we will need to move
# the mode axis (currently index 0) to the end (index 2 or -1)
shape_matrix = np.moveaxis(shape_matrix,0,-1)
```

We will finally pass this information into the `by_effective_independence` function in the `dof` subpackage, as well as a sensor budget to keep. This will return a set of indices into the original candidate `CoordinateArray` that specify which degrees of freedom should be kept in the test. Figure 8 shows the effective independence of the shape matrix compared to sensors kept, and Figure 9 shows the final optimized instrumentation set.

```
# Now let's use effective independence to select which degrees of freedom to
# keep.  Let's pretend our sensor budget is 30 triaxes
sensors_to_keep = 30
# Now we can run the effective independence scheme.  We will let it return
# additional information so we can interrogate the sensor selection
keep_indices,efi = sdpy.dof.by_effective_independence(
sensors_to_keep, shape_matrix,return_efi=True)

# Plot the effective independence vs dofs
fig,ax = plt.subplots(num='EFI vs DoF')
ax.plot(shape_matrix.shape[0]-np.arange(len(efi)), efi)
ax.set_yscale('log')
ax.set_xlim(ax.get_xlim()[::-1])
ax.set_ylabel('EFI')
ax.set_xlabel('DoF Remaining')

# Plot the kept dofs on the model
keep_dofs = candidate_dofs[keep_indices]
geometry.plot_coordinate(keep_dofs,arrow_scale=0.01,plot_kwargs = plot_options)
```

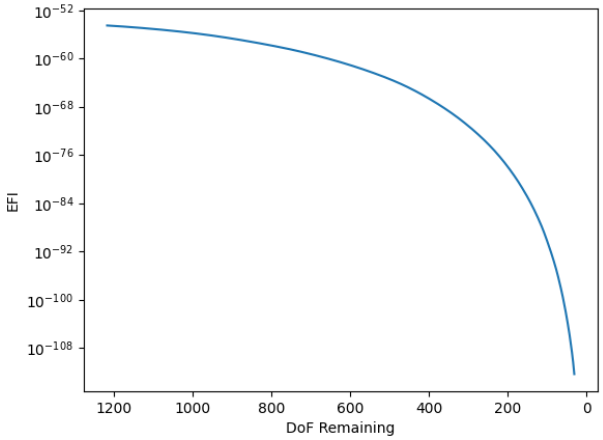


Figure 8: Visualizing effective independence compared to sensors remaining

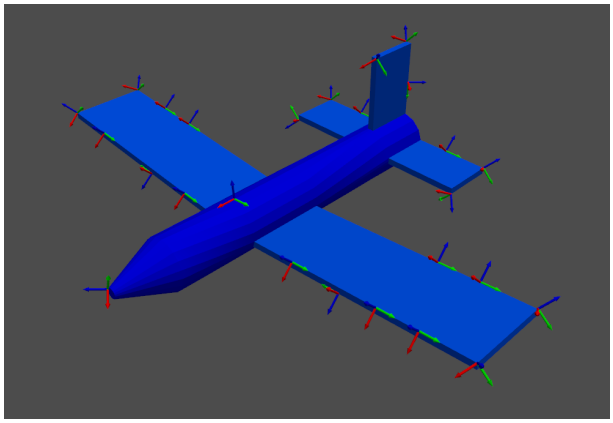


Figure 9: Final optimized triaxial instrumentation set for the modal test.

3.5 Reducing to the Test Geometry

Now that we have selected the sensors to keep in this test, we will need to create a test geometry which only contains those degrees of freedom. We can then add lines to aid in visualization between these degrees of freedom. We can immediately create a reduced geometry by reducing the original geometry to the kept nodes. This will automatically strip out any elements or tracelines that do not contain all the elements in the list. We can then plot that geometry with coordinates labeled, and add tracelines between the nodes. The final test geometry is shown in Figure 10.

```
# Now that we have our selected degrees of freedom, let's create a test
# geometry and shapes to plot on the test geometry
test_geometry = geometry.reduce(np.unique(keep_dofs.node))

# This last step just removed all of the elements, so we want to draw some
# tracelines on the model.  Let's plot the coordinates with the dofs labeled to
# aid us in creating the tracelines
test_geometry.plot_coordinate(keep_dofs,arrow_scale=0.02,label_dofs=True)

# Now let's draw some tracelines, we will get node IDs from the plot we just created
# Fuselage
test_geometry.add_traceline([5248,2796],color=1)
# Wings
test_geometry.add_traceline([6172,6272,6214,6157,6376,6392,6405,8160,8143,6172],color=7)
test_geometry.add_traceline([11909,11892,13647,13660,13603,11705,11722,11735,11664,11764,11909],color=7)
# Tail
test_geometry.add_traceline([19579,19651,19665,19579],color=13)
test_geometry.add_traceline([17573,17563,17107,17573],color=13)
test_geometry.add_traceline([18787,18416,18331,18787],color=13)

# Now plot the geometry to see the tracelines
test_geometry.plot_coordinate(keep_dofs,arrow_scale=0.02,label_dofs=True,plot_kwargs=plot_options)
```

We can also reduce the shapes to the kept degrees of freedom so we can plot them on the test geometry. Here we will also assign a damping factor to our finite element shapes which we will use when simulating experiments. We will plot the modal assurance criterion matrix to show if any shapes look similar due to the degree of freedom reduction, which is shown in Figure 11.

```
# Now get the test shapes
test_shapes = target_shapes.reduce(keep_dofs.flatten())
# We'll need to add damping to the model too
test_shapes.damping=0.02
# Plot the geometry to see what it looks like
test_geometry.plot_shape(test_shapes,plot_options)
# Look at the mac for the target shapes with our set of degrees of freedom
sdpy.correlation.matrix_plot(sdpy.shape.mac(test_shapes),text_size = 6)
```

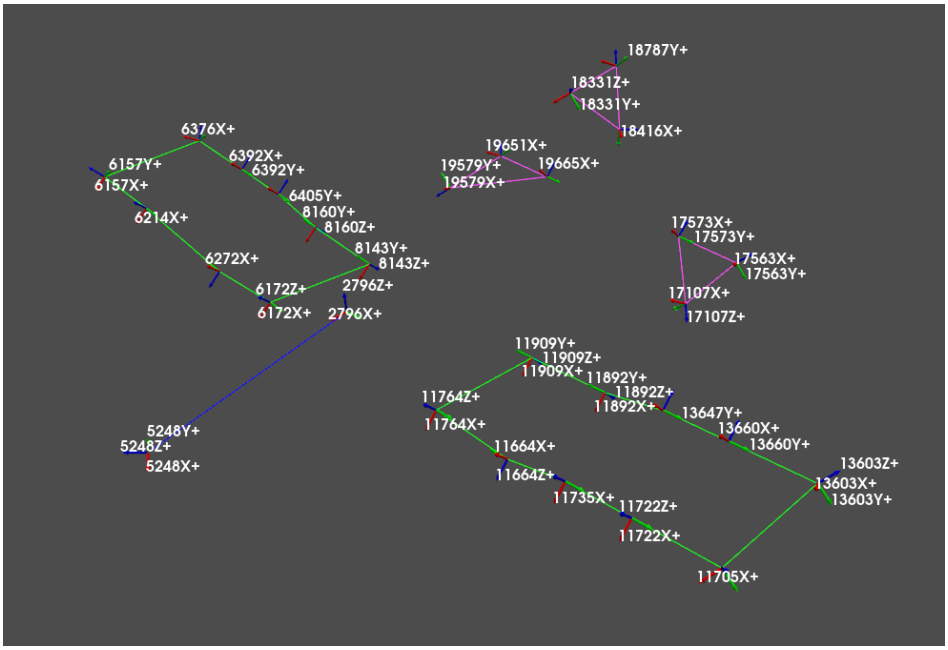


Figure 10: Test geometry with degrees of freedom plotted and labeled.

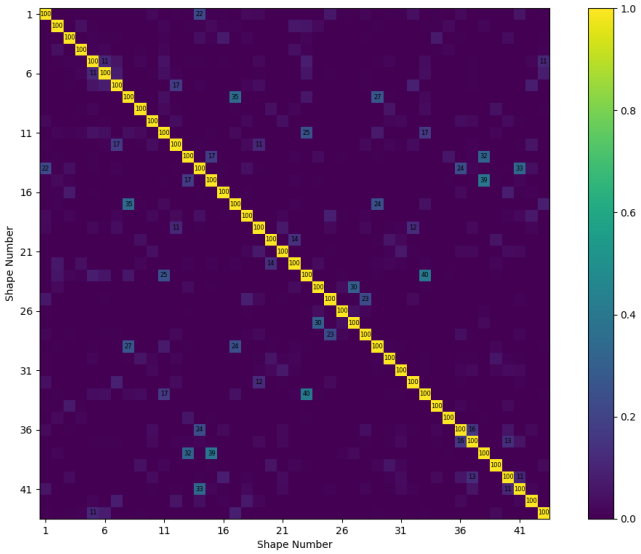


Figure 11: Modal assurance criterion matrix for the test shapes

3.5.1 OOPS! An Instrumentation Error!

As test practitioners are well aware, it is very easy to accidentally swap sensors or place gauges in the wrong orientation. Therefore, we will simulate an issue with instrumentation where we accidentally defined one sensor in the global coordinate system while putting the sensor in a local coordinate system on the part. We will define one of the coordinate systems in our geometry to the global coordinate system. We will later use SDynPy's analysis capabilities to identify and fix this bad channel.

```
# Copy the geometry so we don't overwrite our correct version
test_geometry_error = test_geometry.copy()
# Change the displacement coordinate system of the 5th node to the global
# coordinate system
test_geometry_error.node.disp_cs[4] = test_geometry_error.coordinate_system.id[-1]
```

3.6 Running a Virtual Experiment: Rigid Body Checkouts

For high-value tests, it is often useful to do data checkouts to ensure there aren't errors in the measured data. In particular, because channel tables can often have errors, a rigid body checkout is useful. In this data check, an excitation force is applied to the system well below the first elastic mode of the system. The response of the system to this excitation should therefore be rigid, and we can examine the deflection shapes to ensure that it is.

3.6.1 Creating a System from the Mode Shapes

To simulate this test, we will create a `System` object, which consists of mass, stiffness, and damping matrices which can be integrated in time to produce test data. `System` objects will also track a transformation matrix between internal state degrees of freedom and physical degrees of freedom. We can construct a `System` directly from mode shapes. This `System` will be a modal model, so mass, stiffness, and damping matrices will be the diagonal modal mass, modal stiffness, and modal damping matrices, and the transformation between the modal state degrees of freedom and the physical degrees of freedom is the mode shape matrix. The structure of the `System` object can be visualized with the `spy` method, as shown in Figure 12.

```
# In order to create a measurement, we will want mass, stiffness, and damping
# matrices from our test article.  We can do that easily by getting the
# System object from our shapes.  This will create modal mass, stiffness, and
# damping matrices, and also store the transformation back to physical
# coordinates (i.e. the mode shapes)
test_system = test_shapes.system()
test_system.spy()
```

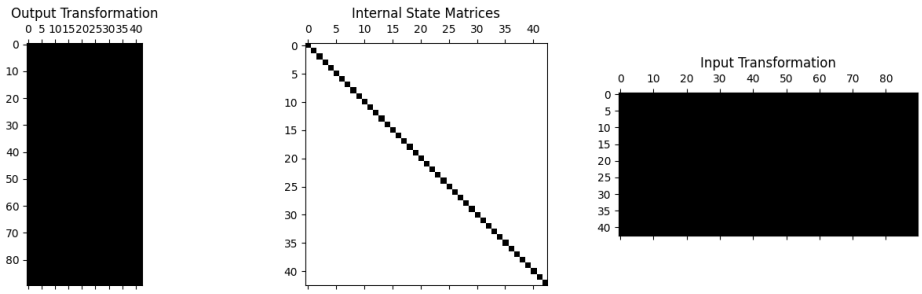


Figure 12: Structure of the modal system, showing full transformation matrices (i.e. the mode shape matrix) and the diagonal internal system matrices (i.e. the modal mass, stiffness, and damping).

3.6.2 Creating a Rigid Body Excitation

We will now create the excitation used for the rigid body check. From our finite element model, the estimate of the first elastic natural frequency is approximately 6 Hz, so we will excite the system at a frequency well below that at 0.5 Hz. We will set up our test bandwidth and other parameters, noting that we will oversample the integration at 10 times the test bandwidth to ensure accurate integration. We will measure 10 averages of the signal. SDynPy's `generator` subpackage will be used to create the sinusoidal signal, which is shown in Figure 13.

```
# First let's set up our general sampling parameters for our test.
test_bandwidth = 200 # Hz
integration_oversample = 10 #x
sample_rate = test_bandwidth*2*integration_oversample
dt = 1/sample_rate
df = 0.125 # Hz
samples_per_frame = int(sample_rate/df)
rb_frames = 10
# Now we will create a sine signal that we can use for rigid body checkouts
rb_frequency = 0.5 # Hz
force = sdpy.generator.sine(rb_frequency, dt, samples_per_frame*rb_frames)
# Plot the sine wave to make sure it is correct
fig,ax = plt.subplots(num='Sine Force Signal')
ax.plot(np.arange(samples_per_frame*rb_frames)*dt,force)
```

We will then excite the structure at degrees of freedom that are approximately in-line with the center of gravity of the structure to excite mostly rigid body translations. We can reference degree-of-freedom plots such as Figure 10 to aid in this selection.

```
# To run the rigid body tests, we select degrees of freedom approximately
# through the CG of the part.
rb_coordinates = sdpy.coordinate_array(string_array=['2796Z+','11722Y+','2796X+'])
```

We will now use the signal to excite the structure. Time integration can be easily performed using the `time_integrate` function of the `System` object. We simply call this function in a `for`-loop, with each iteration of the loop exciting at a different degree of freedom. For each simulated test, the initial transient start-up period is truncated to ensure steady state sinusoidal response by using the `extract_elements_by_abscissa` method. Frequency response functions are created from the integrated time data, and the frequency line corresponding to the excitation frequency is selected as the deflection shape. These are then concatenated into a set of rigid body shapes, one for each excitation location. Note these shapes are complex, but should be dominated by the real part of the FRF due to the excitation being below the first mode of the system. If significant imaginary parts are found in the shapes, this can signify the excitation is not far enough away from the first elastic mode, or it can indicate timing issues in the data acquisition system resulting in phase shifts of the response.

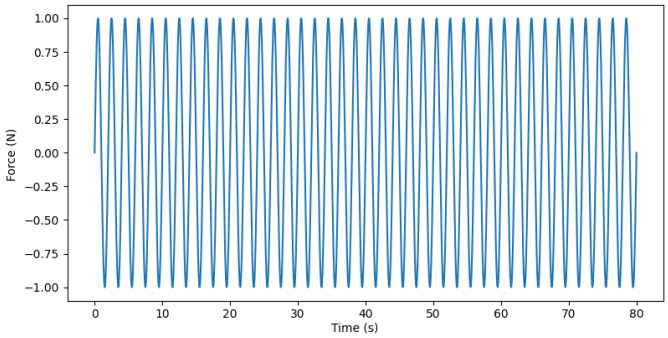


Figure 13: Sinusoidal signal applied to the system for rigid body checkouts.

```
# Now lets perform each of our rigid body checkouts
# Create a list to hold our shapes
rb_shapes = []
# Loop through each of our excitation locations
for rb_coordinate in rb_coordinates:
    # Perform time integration to get the responses to our sine wave
    print('Integrating Rigid Body Excitation at {:}'.format(str(rb_coordinate)))
    responses,references = test_system.time_integrate(
        force,dt,references=rb_coordinate)
    # Plot the responses and references
    fig,ax = plt.subplots(2,1,sharex=True,
                          num='Rigid Body Test {:}'.format(str(rb_coordinate)))
    responses.plot(ax[0])
    ax[0].set_ylabel('Acceleration')
    references.plot(ax[1])
    ax[1].set_ylabel('Force')
    # Truncate the initial portions of the functions so we eliminate the
    # transient portion of the response
    responses = responses.extract_elements_by_abscissa(1,np.inf)
    references = references.extract_elements_by_abscissa(1,np.inf)
    # Now we want to create an FRF from the references and responses
    frf = sdpy.TransferFunctionArray.from_time_data(references,responses,samples_per_frame)
    # Now we want to get the value at our frequency line because the rest will
    # be noise
    frequency_index = np.argmin(abs(frf[0,0].abscissa - rb_frequency))
    shape_matrix = frf.ordinate[...,frequency_index]
    # Now let's create a shapearray object so we can plot the shapes
    rb_shape = sdpy.shape_array(frf[:,0].response_coordinate,shape_matrix.T,
                                rb_frequency,comment1=str(rb_coordinate))
    rb_shapes.append(rb_shape)

# Combine all rb_shapes into one shape array
rb_shapes = np.concatenate(rb_shapes)
```

3.6.3 Investigating and Correcting Channel Table Errors

At this point, we will now investigate the data to see if we can find the channel table error introduced in Section 3.5.1. The first and most obvious solution is to simply plot the deflection shapes. When animated it can be obvious to the viewer when one of the sensors is not moving rigidly with the rest of the sensors. A snapshot of this motion is shown in Figure 14.

```
# Now let's plot those shapes on our (incorrect) geometry
test_geometry_error.plot_shape(rb_shapes,plot_options)
```

While plotting deflection shapes can reveal sensors that are out-of-family, these sensors can be trickier to spot when there are uniaxial sensors that cannot move rigidly with the rest of the model, as they do not have the necessary degrees of freedom. Therefore, a more rigorous approach is included in SDynPy that computes the projection of the measured deflection shapes Φ_m through a set of rigid shapes analytically constructed from the test geometry Φ_a. This effectively removes non-rigid portions of the response. Subtracting this projected shape from the original shape leaves only the non-rigid motions as the residual Φ_r. This is shown in equation (1). Suspect degrees of freedom can therefore be identified as those with large values in this residual shape matrix Φ_r. SDynPy's `rigid_body_check` function accepts as its arguments the test geometry and rigid shapes and computes and plots this residual to highlight suspect channels that can be investigated. The residuals are shown in Figure 15. Note that because this is a least-squares fit to the rigid shapes, large errors in one sensor can make it seem like other sensors are also in error, though those residuals will generally not be as large. Therefore, it is generally advised to investigate sensors with the highest residuals first.

$$\Phi_r = \Phi_m - \Phi_a {\Phi_a}^{+} \Phi_m \tag{1}$$

```
# It looks like there is an error in the shapes (go figure!).  Let's perform
# a more quantitative analysis on the shapes to see what is wrong
suspicious_dofs = sdpy.shape.rigid_body_check(
    test_geometry_error, rb_shapes)
```

The findings from this analysis show that there is likely an issue with the `6214Y+`, `6214Z+`, and `6214X+` degrees of freedom. This is expected given that we introduced an error in our test geometry for that node's coordinate system. If such an error is found, it

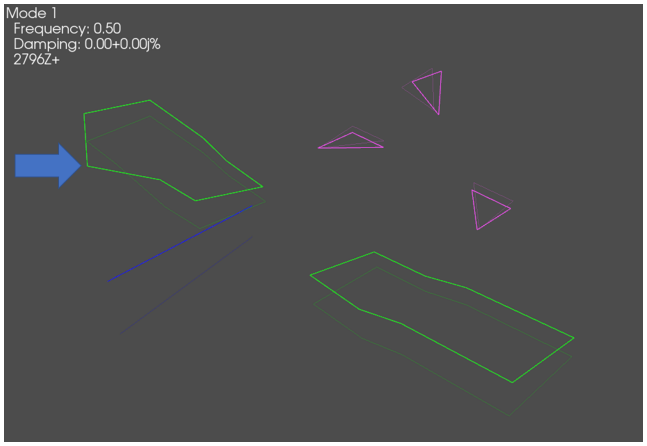


Figure 14: Deflection shape from the rigid body test highlighting a sensor that is not moving rigidly with the rest of the geometry.

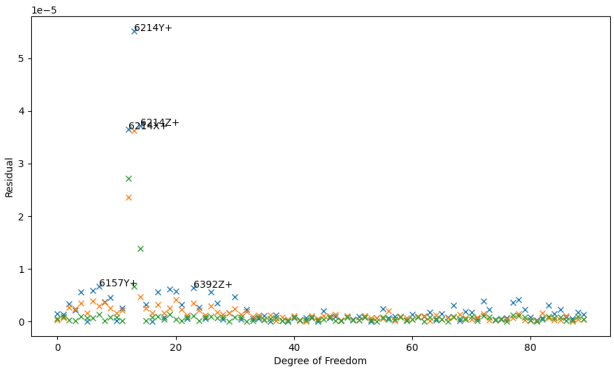


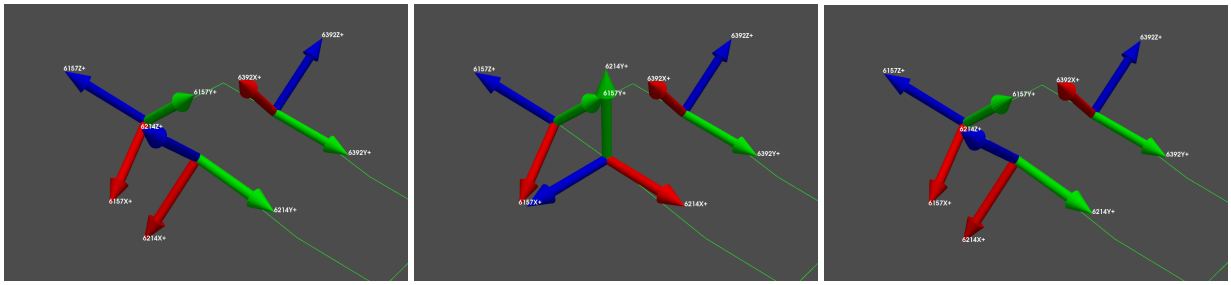
Figure 15: Residuals from the projection of the measured shapes through analytical rigid body shapes. Large residuals occur for the sensor that was incorrectly placed.

is often useful to further investigate the sensor to see if it is oriented incorrectly. Unfortunately, such a sensor may sometimes be located inside a test body and therefore inaccessible, so it can be useful to attempt to correct the sensor using just the rigid body data. SDynPy has a `rigid_body_fix_node_orientation` function that attempts to do just that. It uses a nonlinear optimizer to update the coordinate systems of suspect nodes to reduce the residual error.

```
# Pretty clearly there is an issue with the degrees of freedom at node 6214.
# Let's see if we can't let SDynPy figure out the correct orientation for that
# sensor in the geometry given the data.
suspicious_nodes = np.unique(suspicious_dofs.node)
test_geometry_corrected = sdpy.shape.rigid_body_fix_node_orientation(
    test_geometry_error, rb_shapes,suspicious_nodes)
```

This orientation correction process took the sensor that was erroneously oriented in the global coordinate system (shown in Figure 16b) and modified its orientation to reduce the residual error in the rigid body shapes (shown in Figure 16c). Notice this agrees very well with the true orientation of the sensor, shown in Figure 16a. Now when we plot the rigid deflection shapes on this corrected geometry, they appear to move rigidly, as shown in Figure 17.

```
# Let's see what the fix looks like compared to the way the sensor is actually
# oriented
test_geometry_error.plot_coordinate(
    sdpy.coordinate.from_nodelist(suspicious_nodes),label_dofs=True,
    plot_kwargs=plot_options)
test_geometry_corrected.plot_coordinate(
    sdpy.coordinate.from_nodelist(suspicious_nodes),label_dofs=True,
    plot_kwargs=plot_options)
test_geometry.plot_coordinate(
    sdpy.coordinate.from_nodelist(suspicious_nodes),label_dofs=True,
    plot_kwargs=plot_options)

# We can also look at the coordinate system matrices
print('Coordinate System for Error Geometry')
print(test_geometry_error.coordinate_system(
    test_geometry_error.node(suspicious_dofs[0].node).disp_cs).matrix)
print('Coordinate System for Corrected Geometry')
print(test_geometry_corrected.coordinate_system(
    test_geometry_corrected.node(suspicious_dofs[0].node).disp_cs).matrix)
print('Coordinate System for Correct Geometry')
print(test_geometry.coordinate_system(
    test_geometry.node(suspicious_dofs[0].node).disp_cs).matrix)

# Plot the rigid shapes on the corrected geometry
plotter = test_geometry_corrected.plot_shape(rb_shapes,plot_options)
```



(a) True orientation of the sensor

(b) Erroneous sensor orientation in the initial test geometry

(c) Corrected orientation of the sensor in the new test geometry

Figure 16: Correction of the improperly oriented sensor by minimizing rigid body error

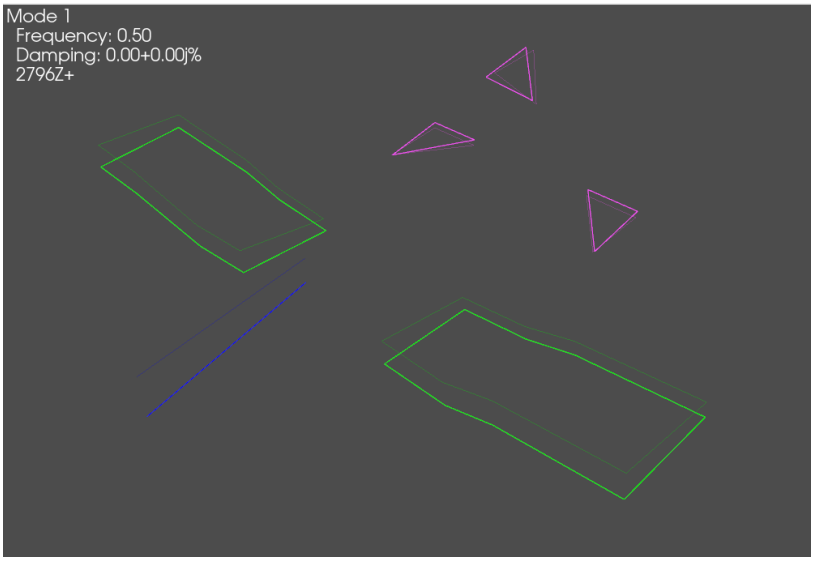


Figure 17: Rigid body mode shape with sensor orientation corrected, which shows the entire body moving rigidly.

3.7 Running a Virtual Experiment: Modal Testing

With the channel table validated, we can now run a modal test. We will simulate a multiple-input, multiple-output modal test with 4 shakers playing random excitation.

3.7.1 Creating the Modal Excitation and Response

We will first select the excitation degrees of freedom. Selected positions are the two wingtips, the tail tip, and the nose of the airplane, which are shown in Figure 18.

```
# First let's select drive points to get all the modes.  Here we will chose the
# wing tip, tail tip, and nose.
drive_points = sdpy.coordinate_array(string_array=[
    '6157Z+',
    '11705Z+',
    '18787Y+',
    '5248Y+',
    ])
test_geometry.plot_coordinate(drive_points,plot_kwargs=plot_options,
                              label_dofs=True)
```

We will now create the random signal to play into shakers using the `generator` subpackage. We will create 30 averages. Note that we must supply a maximum frequency cutoff to ensure that aliasing doesn't occur due to the oversampling required for accurate integration. If a maximum frequency is not specified, the signal would contain content up to the Nyquist frequency of the oversampled integration rate; then, when the signal would be subsequently downsampled to the actual sample rate, aliasing would occur. The signals and their frequency responses are shown in Figure 19.

```
# Now let's create a force.  We will do a random excitation
modal_frames = 30
random_forces = sdpy.generator.random(
    drive_points.shape,modal_frames*samples_per_frame,dt=dt,
    high_frequency_cutoff=test_bandwidth)
# Look at the signal statistics
rms = np.sqrt(np.mean(random_forces**2,axis=-1))
fig,ax = plt.subplots(2,1,num='Random Excitation')
ax[0].plot(np.arange(random_forces.shape[-1])*dt,
           random_forces.T)
ax[0].set_ylabel('Force')
ax[0].set_xlabel('Time')
freq = np.fft.rfftfreq(random_forces.shape[-1],dt)
fft = np.fft.rfft(random_forces,axis=-1)
ax[1].plot(freq,abs(fft.T))
ax[1].set_ylabel('Force')
ax[1].set_yscale('log')
ax[1].set_xlabel('Frequency')
```

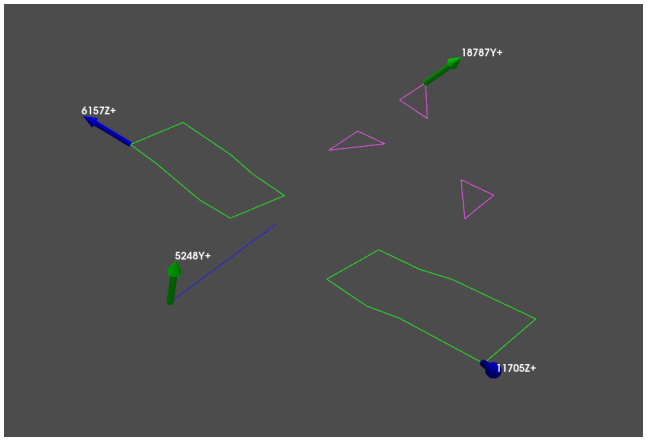


Figure 18: Drive point locations for the modal testing.

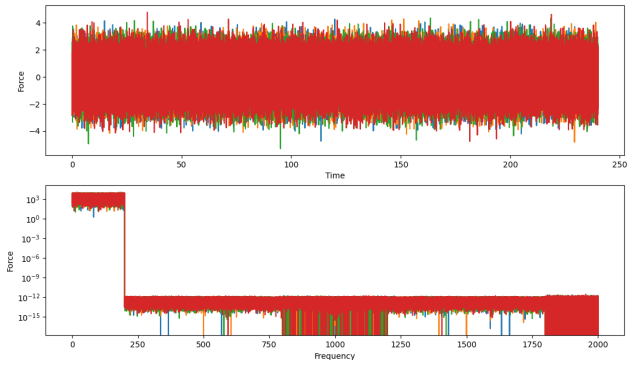


Figure 19: Excitation signals used for the simulated modal test. Note that there is no content past the test bandwidth to prevent aliasing.

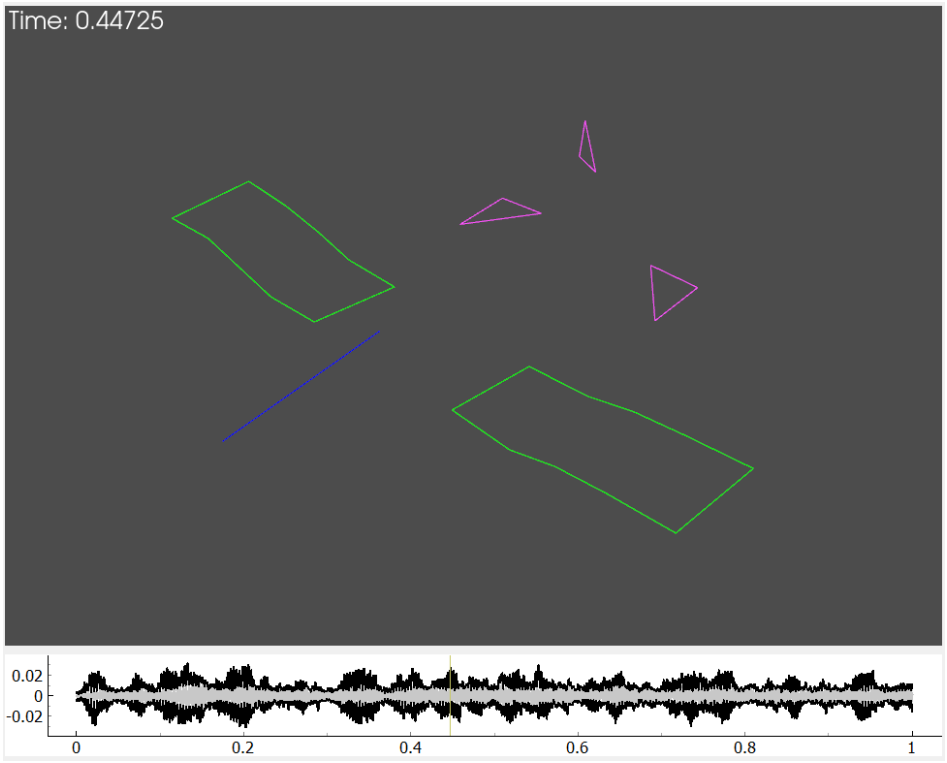


Figure 20: Interactive plot showing transient response of the system due to modal excitation.

We can then again integrate the `System` object using the `time_integrate` method. Transient responses can be interactively investigated using the `plot_transient` method of the `Geometry` objects, which is shown in Figure 20.

```
# Now let's run the modal test
responses_modal,references_modal = test_system.time_integrate(
random_forces,dt,references=drive_points)

test_geometry.plot_transient(responses_modal.extract_elements_by_abscissa(0,1),plot_kwargs=plot_options)
```

The first thing we will do with this output data is to downsample back to the desired (not oversampled) sample rate. Now that we have time history responses to the excitation, we can compute frequency response functions. In this case, we will also specify an overlap fraction and a window function due to the random data excitation. We can quickly look through the frequency response functions by using SDynPy's graphical plotter `GUIPlot` (Figure 21), or plot deflection shapes using `plot_deflection_shape` (Figure 22).

```
# Now let's downsample to the actual measurement (removing the 10x integration
# oversample)
responses_sampled = responses_modal.extract_elements(slice(None,None,integration_oversample))
references_sampled = references_modal.extract_elements(slice(None,None,integration_oversample))
# Compute FRFs from the time data
frf_sampled = sdpy.TransferFunctionArray.from_time_data(
    references_sampled,responses_sampled,samples_per_frame//integration_oversample,
    overlap=0.5,window='hann')

# Now let's use GUIPlot to look at the functions
plotter = sdpy.GUIPlot(frf_sampled)
# Visualize deflection shapes
test_geometry.plot_deflection_shape(frf_sampled[:,0],plot_options)
```

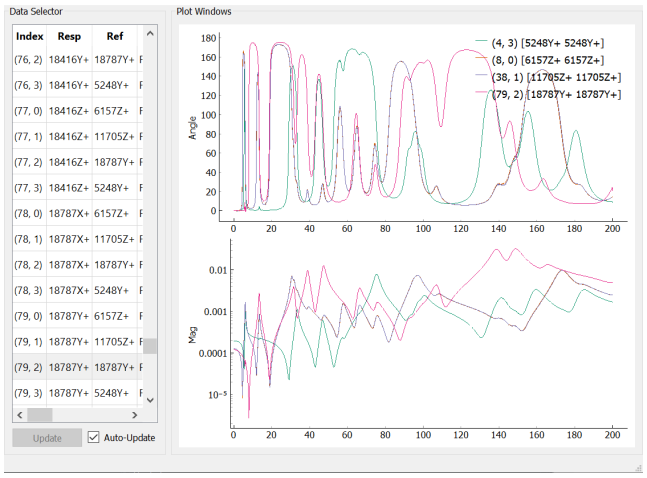


Figure 21: `GUIPlot` showing the four drive point accelerometer FRFs.

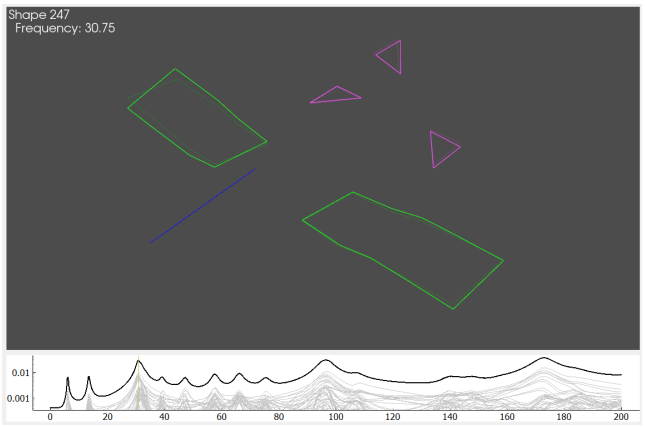


Figure 22: Interactive deflection shape visualizations

3.7.2 Fitting Modes

Now that frequency response functions have been computed, we can fit mode shapes to the data. We will use SDynPy's Z-domain multi-reference curve fitter [7] to perform this analysis. The implementation in SDynPy consists of two parts. The first is to select frequency ranges to compute stability diagrams (Figure 23), and the second is to select the stable poles (Figure 24). The curve fitter can analyze multiple frequency bands separately and combine the shapes into a final set. The curve fitter will then resynthesize frequency response functions which can be displayed in multiple formats, such as Figure 25.

```
# Now that we have FRFs we can go fit modes.  We will use PolyPy
pp = sdpy.PolyPy_GUI(frf_sampled)
pp.set_geometry(test_geometry)
```

From the graphical user interface, we can save the shapes to a file, and then load them back into our script for further analysis. SDynPy offers a number of approaches to compare shapes. For example, the modal assurance criterion matrix can be plotted (Figure 26), a shape comparison table can be printed (Table 1), or geometry and shapes can be overlaid and plotted together (Figure 27). Obviously the comparisons in this case are very good due to the fact that the test data was simulated directly from the finite element model, though damping errors are prevalent in the lower-frequency modes due to the random excitation and application of the Hann window during the frequency response function creation process.

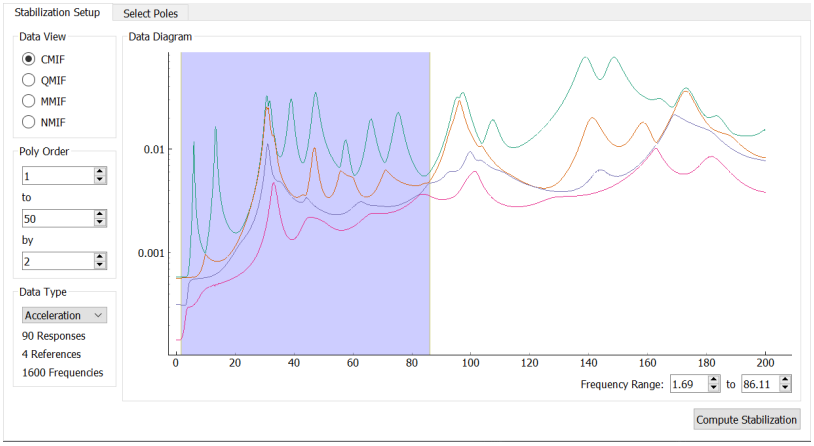


Figure 23: Selecting frequency range and polynomial orders.

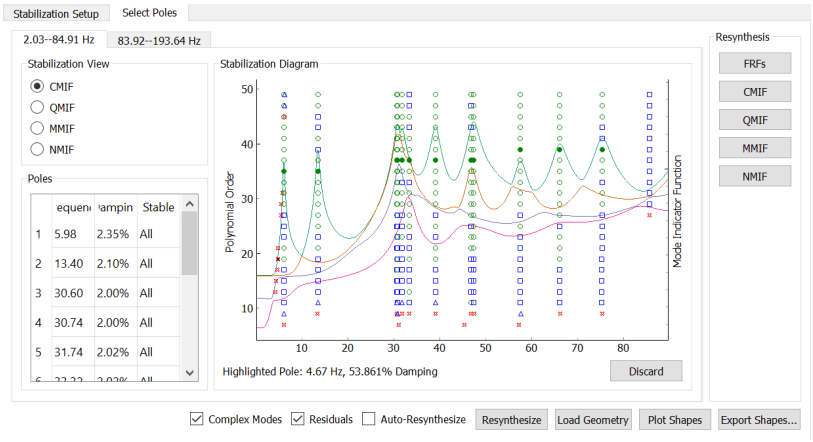


Figure 24: Selecting stable poles from the frequency ranges.

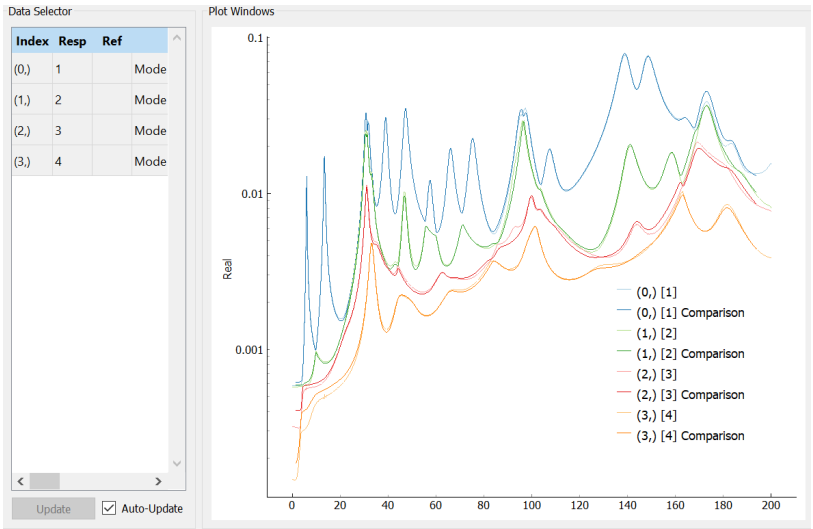


Figure 25: Complex Mode Indicator Function comparing experiment to reconstruction

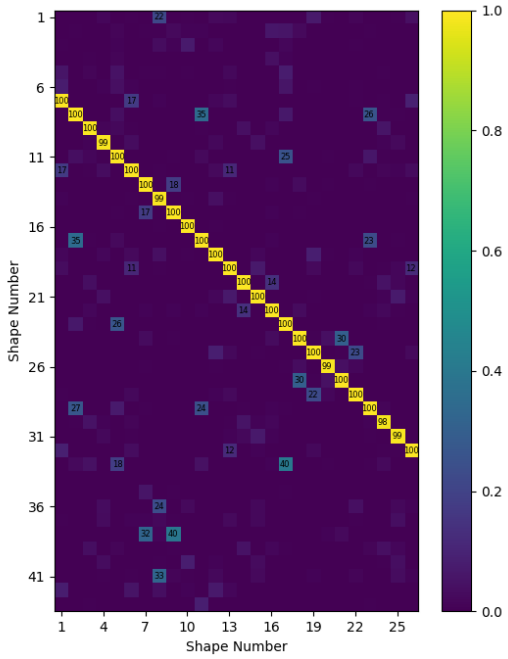
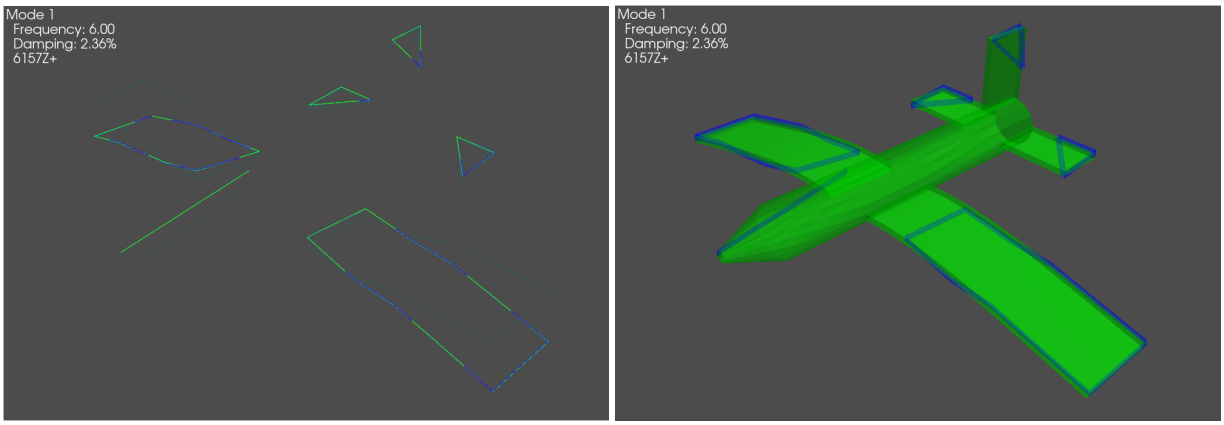


Figure 26: Modal Assurance Criterion comparison between test and finite element shapes

```
# Let's compare the shapes to the finite element model shapes
mac = sdpy.shape.mac(test_shapes,test_shapes_polypy)
sdpy.correlation.matrix_plot(
    mac,text_size=6)

# Find shape correspondences where the mac is high
shape_correspondences = np.where(mac > 0.9)
shape_1 = test_shapes_polypy[shape_correspondences[1]]
shape_2 = test_shapes[shape_correspondences[0]]
print(sdpy.shape.shape_comparison_table(shape_1, shape_2,
                                        percent_error_format='{:0.4f}%'))

# Compare shapes visually.  First we need to get the correct flipping in case
# the shapes are 180 out of phase
shape_phasing = np.sign(np.einsum('ij,ij->i',shape_1.shape_matrix,shape_2.shape_matrix))
shape_1 = shape_1*shape_phasing[:,np.newaxis]

# Plot on the test geometry
test_comparison_geometry,test_comparison_shapes = sdpy.shape.overlay_shapes(
    (test_geometry,test_geometry),(shape_1,shape_2),[1,7])
test_comparison_geometry.plot_shape(test_comparison_shapes,plot_options)
# Plot on the fem geometry
fem_comparison_geometry,fem_comparison_shapes = sdpy.shape.overlay_shapes(
    (test_geometry,geometry_global),(shape_1,shapes_global[shape_correspondences[0]]),[1,7])
fem_comparison_geometry.plot_shape(fem_comparison_shapes,plot_options,
                                   deformed_opacity=0.5,undeformed_opacity=0)
```

Table 1: Mode comparison table between test and finite element shapes

Mode	Freq 1 (Hz)	Freq 2 (Hz)	Freq Error	Damp 1	Damp 2	Damp Error	MAC
1	6.00	6.00	-0.0067%	2.36%	2.00%	17.9820%	100
2	13.40	13.40	0.0218%	2.06%	2.00%	2.9605%	100
3	30.59	30.60	-0.0458%	2.01%	2.00%	0.4512%	100
4	30.73	30.74	-0.0344%	2.01%	2.00%	0.4827%	99
5	31.73	31.73	0.0077%	1.99%	2.00%	-0.7148%	100
6	33.31	33.31	0.0025%	1.99%	2.00%	-0.3471%	100
7	39.02	39.01	0.0114%	1.98%	2.00%	-1.0855%	100
8	46.78	46.77	0.0156%	1.97%	2.00%	-1.4382%	99
9	47.28	47.27	0.0189%	2.00%	2.00%	-0.0342%	100
10	57.49	57.49	-0.0003%	2.00%	2.00%	0.1664%	100
11	66.02	66.02	0.0012%	2.00%	2.00%	-0.0377%	100
12	75.28	75.28	-0.0000%	2.00%	2.00%	0.0906%	100
13	92.58	92.58	-0.0026%	2.00%	2.00%	0.1076%	100
14	95.39	95.40	-0.0078%	2.00%	2.00%	0.1584%	100
15	97.19	97.20	-0.0061%	1.99%	2.00%	-0.3016%	100
16	99.94	99.94	-0.0013%	2.00%	2.00%	-0.1005%	100
17	107.30	107.30	-0.0018%	2.00%	2.00%	0.1139%	100
18	138.97	138.96	0.0065%	1.99%	2.00%	-0.5165%	100
19	140.82	140.81	0.0016%	2.01%	2.00%	0.3194%	100
20	142.06	142.04	0.0097%	2.00%	2.00%	0.2438%	99
21	148.12	148.13	-0.0045%	2.01%	2.00%	0.3527%	100
22	158.70	158.70	0.0005%	2.01%	2.00%	0.2717%	100
23	164.16	164.15	0.0063%	2.00%	2.00%	0.1195%	100
24	172.71	172.68	0.0183%	1.98%	2.00%	-0.8317%	98
25	172.99	172.96	0.0171%	1.98%	2.00%	-0.8663%	99
26	183.52	183.53	-0.0042%	1.98%	2.00%	-1.1180%	100



(a) **(b)**

Figure 27: Overlay of test and finite element shapes (a) reduced to the test geometry and (b) on the full finite element model

3.7.3 Finite Element Expansion

Many times after running a test, we would like to predict responses at positions that were not instrumented. For this, we will use SEREP. Because SDynPy tracks geometry and coordinates in its objects, it is able to handle all of the bookkeeping involved in this process, making it very simple to apply. Note that in the following commands, we are using the global shapes to expand to the global geometry, even though the test shapes and geometry contain local coordinate system. The `expand` function automatically performs the necessary coordinate transformations to make the expansion work correctly. An expanded mode shape is shown compared to the test geometry in Figure 28.

```
# Perform the expansion using the finite element shapes in the bandwidth
expansion_basis = shapes_global[shapes_global.frequency < shape_bandwidth]
expanded_shapes = test_shapes_polypy.expand(test_geometry,geometry_global,
                                            expansion_basis)
# We can then plot the expanded shapes on the original finite element geometry
geometry_global.plot_shape(expanded_shapes,plot_options)

# Or overlay the expanded geometry with the original test geometry
expansion_comparison_geometry,expansion_comparison_shapes = sdpy.shape.overlay_shapes(
    (test_geometry,geometry_global),(test_shapes_polypy,expanded_shapes),[1,7])
expansion_comparison_geometry.plot_shape(expansion_comparison_shapes,plot_options,
                                         deformed_opacity=0.5,undeformed_opacity=0)
```

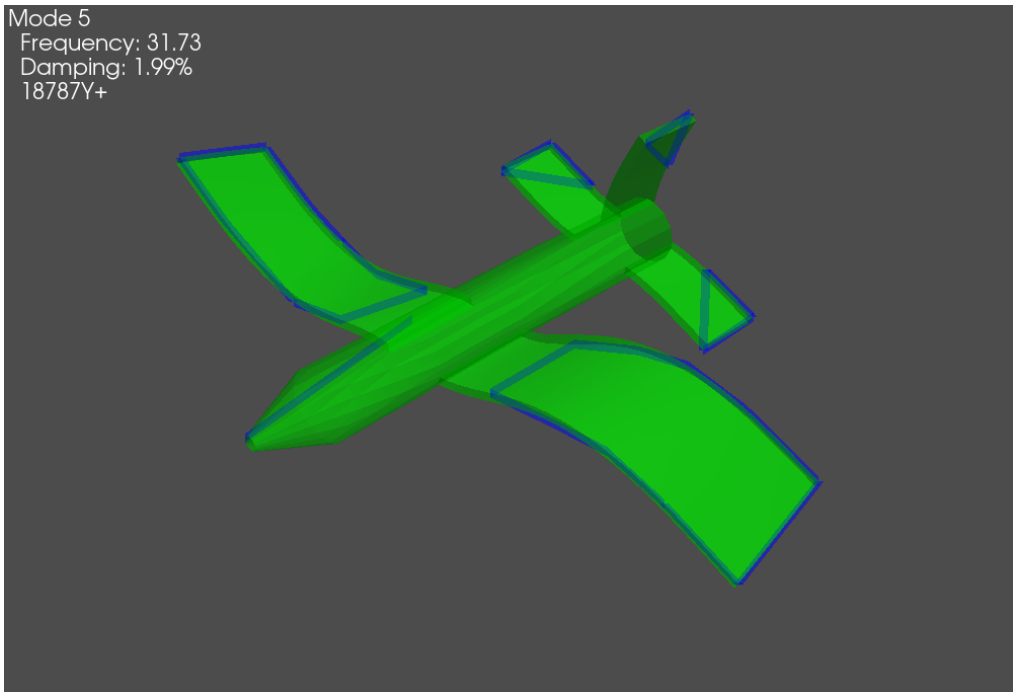


Figure 28: Comparison of test geometry and test geometry expanded to the full finite element geometry.

3.8 Documentation Generation

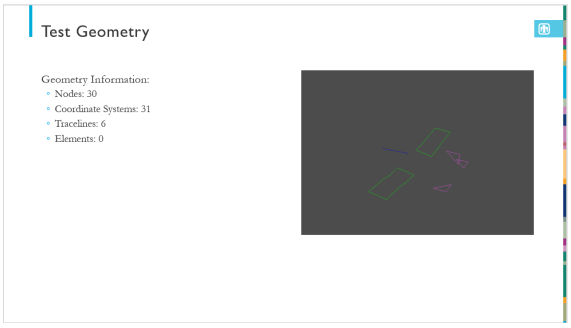
Documentation generation can be a tedious portion of any test, especially if a large number of mode shape figures are to be made. SDynPy has the ability to automatically generate portions of the documentation to aid in the creation of test reports. For this test, we will create a quick-look PowerPoint presentation that contains modal parameters and mode shape animations.

For best results, we should load in an existing PowerPoint template. While users could start from scratch, it might take more work to make the final presentation look nice after all information is added to it. SDynPy uses the `python-pptx` package to write to presentations, so we can use that package to load a presentation containing a template, and then add slides to it.

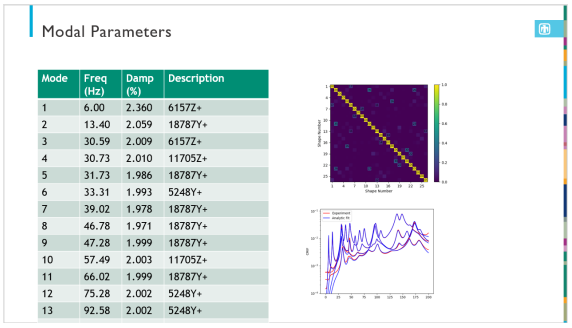
```
# First let's create a powerpoint presentation.  We will use the python-pptx
# module to work with presentations
import pptx

# Open up a presentation with the Sandia theme
prs = pptx.Presentation('Powerpoint_Sandia_Theme.pptx')
```

SDynPy can then use various functions in the `doc.ppt` subpackage to add information to this presentation. For example, a title slide, section headers, geometry overviews, data overviews, and mode shape animations. Figure 29 shows examples of slides generated by SDynPy.

```
# We can now use SDynPy functions to populate the powerpoint slides.  Note that
# the SDynPy functions require specifying the layout indices.  These will be
# different for each template, but they are basically used to tell which type
# of slide to use for each piece of data (Content, Title, Subtitle, Comparison)

# Title Slide - Add an initial slide with our test title and name
sdpy.doc.ppt.add_title_slide(prs, 'Airplane Modal Test Quicklook','Dan Rohe')

# Now we will start adding our test data in a new section, so we put a section
# header slide in
sdpy.doc.ppt.add_section_header_slide(
    prs, 'Test Information',subtitle='Airplane Modal Test',
    section_header_slide_layout_index=1)

# As part of our test data, we will show our test geometry
sdpy.doc.ppt.add_geometry_overview_slide(
    prs, test_geometry,
    title='Test Geometry',
    geometry_plot_kwargs=plot_options,
    content_slide_layout_index=2)

# Now we will show an overview of the modal parameters that we fit compared to
# the test data.  We will plot CMIFs as well as a table of shapes
sdpy.doc.ppt.add_shape_overview_slide(
    prs, test_shapes_polypy, 'Modal Parameters',
    exp_data = frf_sampled.compute_cmif(),
    fit_data = test_shapes_polypy.compute_frf(frf_sampled[0,0].abscissa[1:],
                                      np.unique(frf_sampled.coordinate[...,0]),
                                      np.unique(frf_sampled.coordinate[...,1])
                                      ).compute_cmif(),
    matrix_plot_kwargs={'text_size':8},
    axes_modifiers={'set_ylabel':'CMIF','set_yscale':'log'},
    empty_slide_layout_index=4)

# Now we will put the shape animations into the presentation.
sdpy.doc.ppt.add_shape_animation_slides(
    prs, test_geometry, test_shapes_polypy,title_format='Test Mode {number:}',
    content_slide_layout_index=2,
    geometry_plot_kwargs=plot_options)
```



(a) Geometry Slide



(b) Modal Parameters Slide

(c) Mode Shape Animation Slide

Figure 29: Example slides generated by SDynPy

4 Conclusions

SDynPy provides a consistent framework to perform structural dynamics analyses using the Python programming language. It provides core data classes that mirror common data types found in structural dynamics. These objects have the capability to simplify performing complex structural dynamics analyses by keeping track of geometry and degrees of freedom. SDynPy contains readers for common file types, state-of-the-art mode fitters to perform experimental modal analysis, and high-quality interactive data visualization tools. The author hopes the release of this tool will encourage the growth of open-source software in structural dynamics by providing a common framework in which users can work.

References

[1] T. Bregar, A. E. Mahmoudi, M. Kodric, D. Ocepek, F. Trainotti, M. Pogacar, M. Göldeli, G. Cepon, M. Boltezar, and D. J. Rixen, "pyFBS: A python package for frequency based substructuring," *Journal of Open Source Software*, vol. 7, no. 69, p. 3399, 2022.

[2] D. P. Rohe, R. Schultz, and N. Hunter, "Rattlesnake: An open-source multi-axis and combined environments vibration controller," in *Proceedings of the 40th International Modal Analysis Conference*, (Orlando, FL), Feb. 2022.

[3] D. P. Rohe, R. Schultz, and N. Hunter, "Rattlesnake user's manual," SAND Report SAND2021-14880, Sandia National Laboratories, Albuquerque, NM, 2021.

[4] Correlated Solutions, *Vic-3D Software Manual Version 8.4*. 121 Dutchman Blvd. Irmo, SC 29063, USA. `http://www.correlatedsolutions.com/supportcontent/VIC-3D-8-Manual.pdf`.

[5] L. A. Schoof and V. R. Yarberry, "EXODUS II: A finite element data model," Sandia Report SAND92-2137, Sandia National Laboratories, Sept. 1994.

[6] D. P. Hensley and R. L. Mayes, "Extending SMAC to multiple reference FRFs," in *Proceedings of the 24th International Modal Analysis Conference*, (St. Louis, Missouri), pp. 220–230, Jan. 2006.

[7] B. Peeters, H. Van der Auweraer, P. Guillaume, and J. Leuridan, "The polymax frequency-domain method: A new standard for modal parameter estimation?," *Shock and Vibration*, vol. 11, p. 523692, Jan. 2004.

[8] J. C. O'Callahan, P. Avitabile, and R. Riemer, "System equivalent reduction expansion process," in *Proceedings of the Seventh International Modal Analysis Conference*, (Las Vegas, NV), Feb 1989.

[9] D. C. Kammer and M. L. Tinker, "Optimal placement of triaxial accelerometers for modal vibration tests," *Mechanical Systems and Signal Processing*, vol. 18, no. 1, pp. 29 – 41, 2004.