LamAna Documentation

Release 0.4.11

P.R. II

June 09, 2016

The LamAna project is an extensible Python library for interactive laminate analysis and visualization.

What's New in LamAna

- appveyor builds (support on Windows)

LamAna enables users to **calculate/export/analyze** and **author** custom models based on laminate theory. *Feature modules* can be used to plot stress distributions, analyze thickessness effects and predict failure trends.

CHAPTER 1

User Benefits

The primary benefits to users is an scientific package that:

- **Simplicity**: given a model and parameters, analysis begins with three lines of code
- **Visualization**: plotting and physical representations
- **Analysis**: fast computational analysis using a Pandas backend
- **Extensibility**: anyone with a little Python knowledge can implement custom laminate models
- **Speed**: data computed, plotting and exported for dozens of configurations within seconds

CHAPTER 2

Community Benefits

Long-term goals for the laminate community are:

- **Standardization**: general abstractions for laminate theory analysis
- **Common Library**: R-like acceptance of model contributions for everyone to use

2.1 Quick View

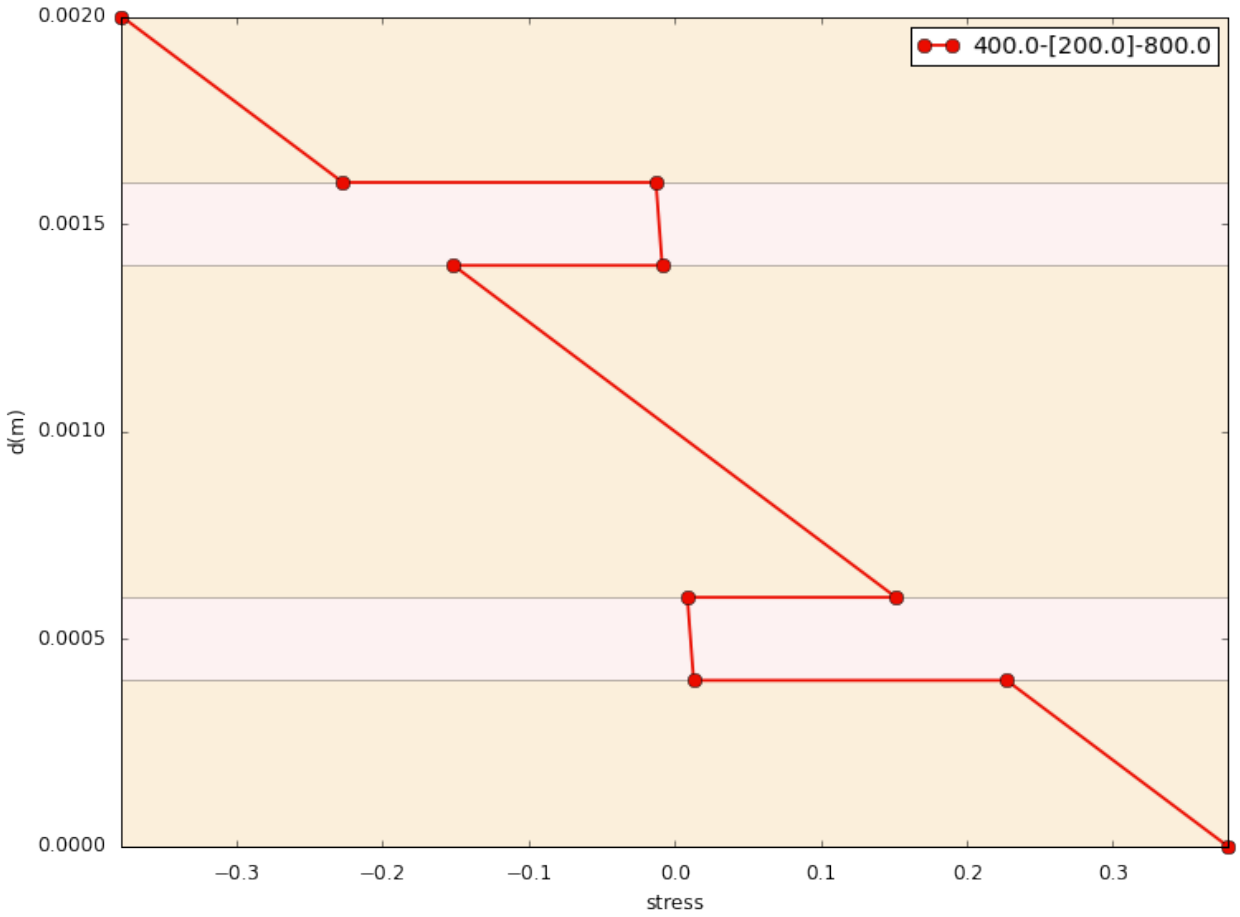
Here is a brief gallery of some stress distribution plots produced by LamAna.

2.1.1 Single Geometry Plots

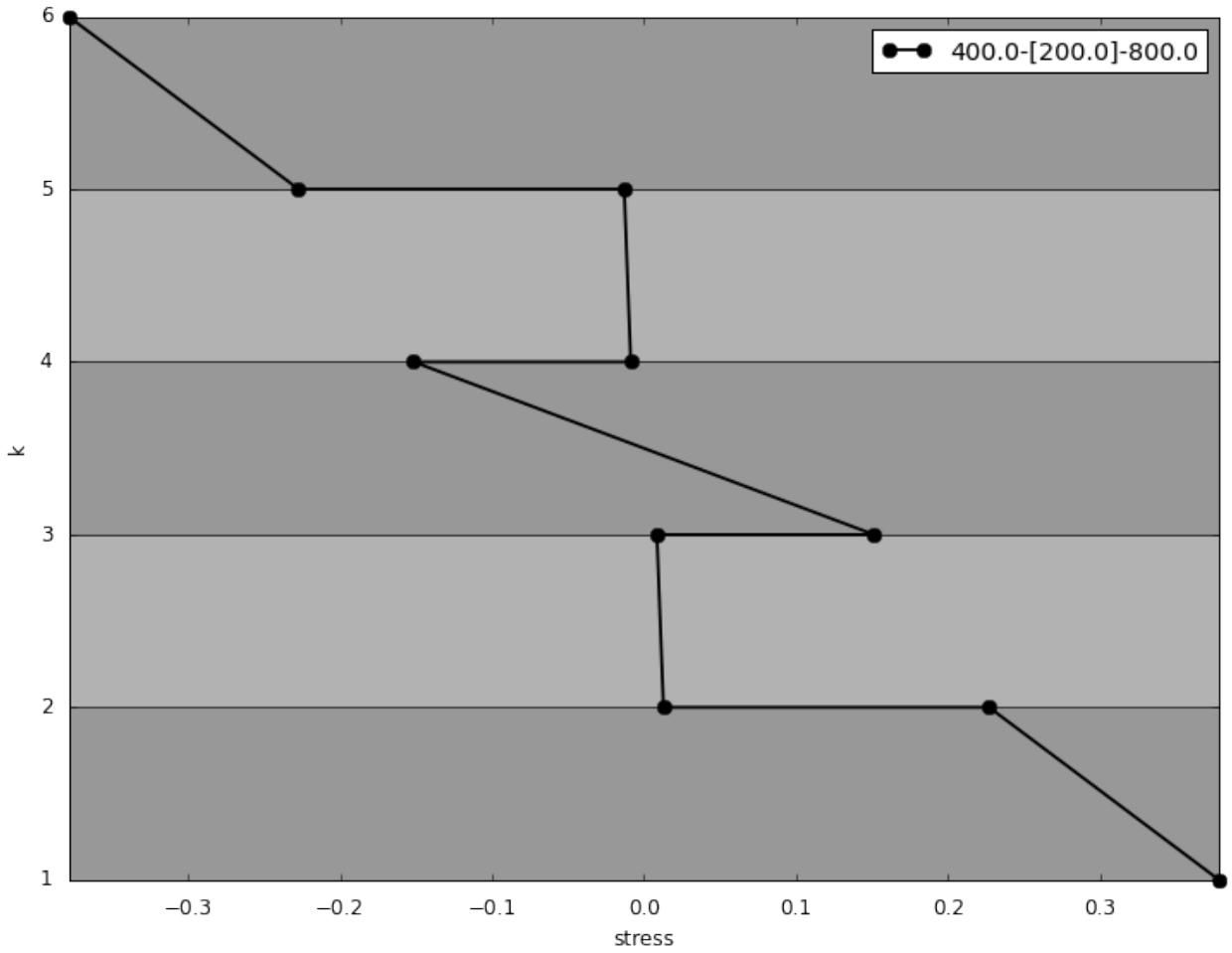
We can plot stress distributions for single laminates as a function of layer thicknesses or normalized thicknesses (default).

```
In [5]: case1 = la.distributions.Case(load_params, mat_props)      # instantiate a User Inp
        case1.apply(single_geo)
        case1.plot(normalized=False)
```

```
Converting mat_props to Standard Form.
User input geometries have been converted and set to Case.
```

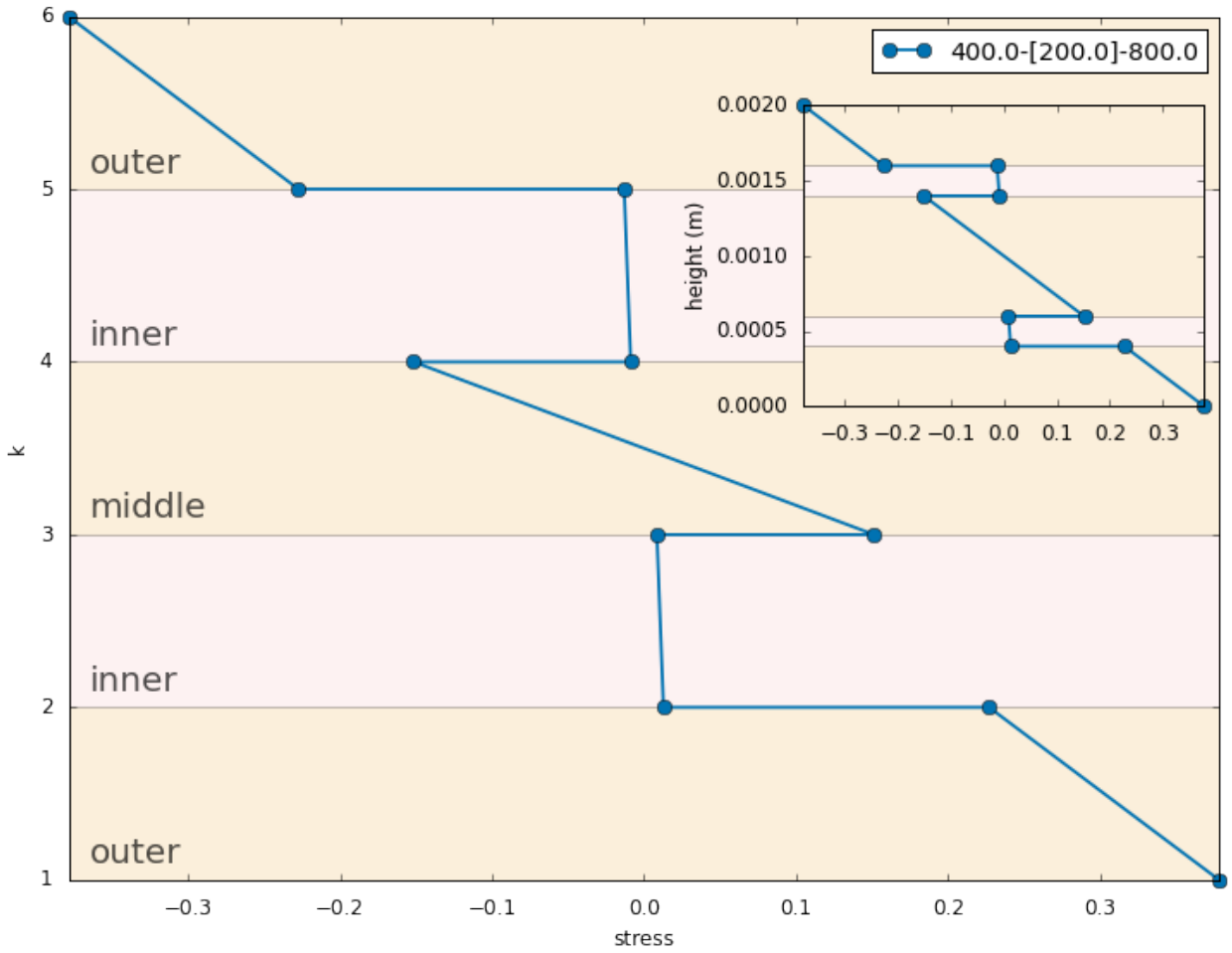



```
In [6]: case1.plot(normalized=True, grayscale=True)
```



We can superimpose both layer axes with insets, and adjust the colors for publications.

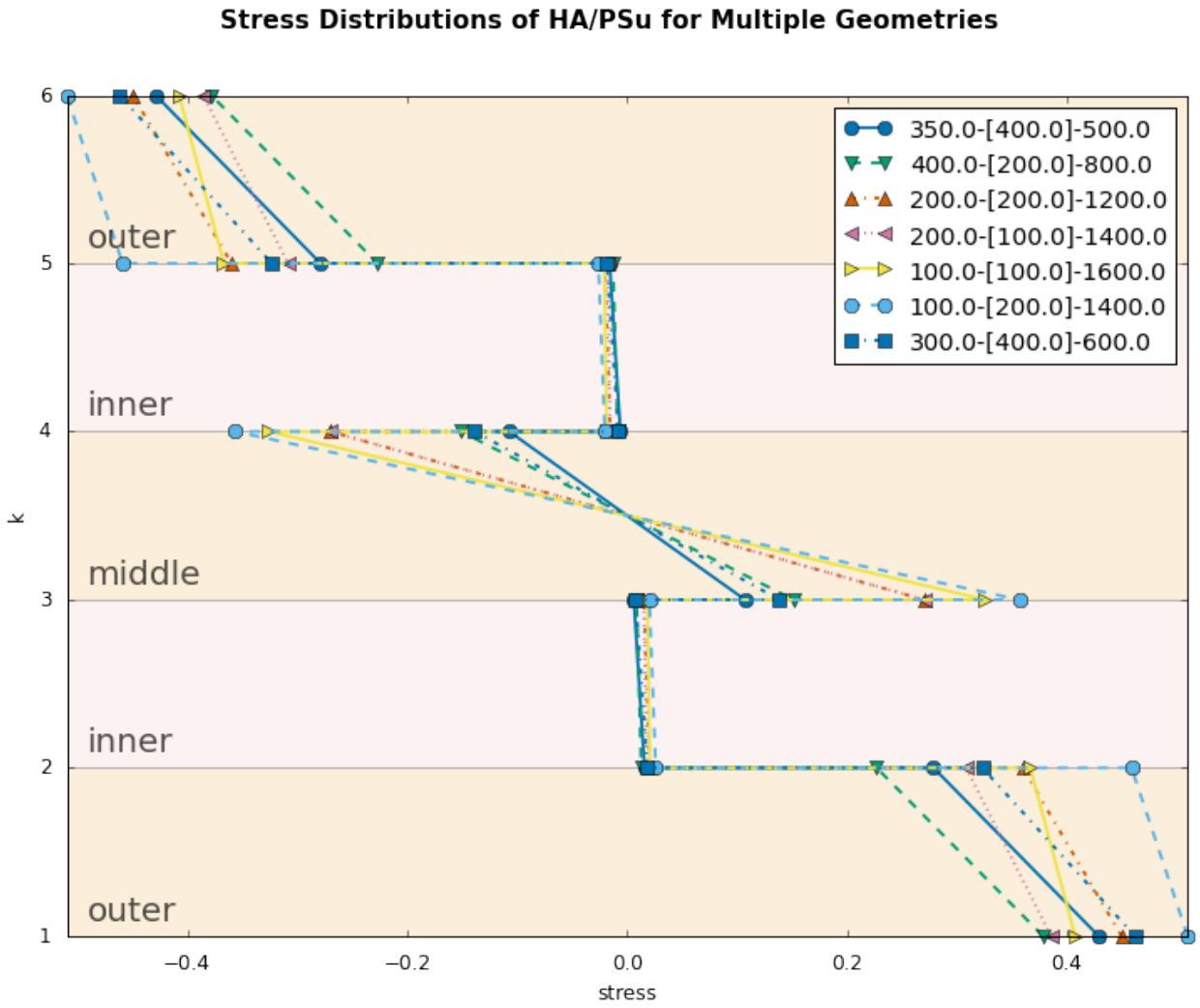
```
In [7]: case1.plot(annotate=True, colorblind=True, inset=True)
```



2.1.2 Multiple Geometry Plots

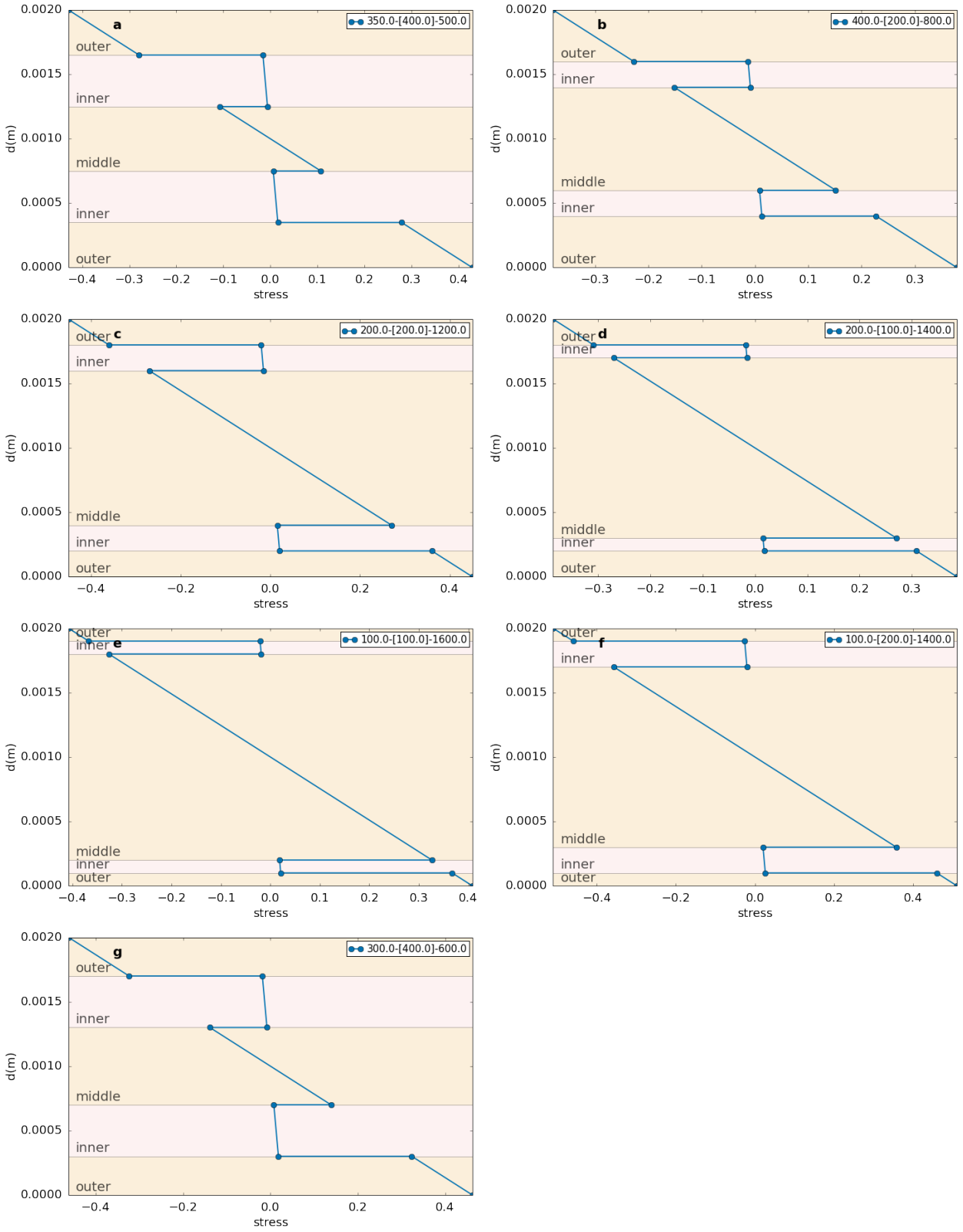
Normalized layers enables superimposed of stress distributions and concurrent laminate analysis. Data for multiple laminates are encapsulated in a `Case` object.

```
In [10]: case2.plot(title, colorblind=True, annotate=True)
```



These distributions can be separated as desired into a panel of multiple plots.

```
In [11]: case2.plot(title, colorblind=True, annotate=True, separate=True)
```


a
350.0-[400.0]-500.0
b
400.0-[200.0]-800.0
c
200.0-[200.0]-1200.0
d
200.0-[100.0]-1400.0
e
100.0-[100.0]-1600.0
f
100.0-[200.0]-1400.0
g
300.0-[400.0]-600.0
outer
inner
middle
d(m)
stress

Halfplots

The following has not been implemented yet, but demonstrates the idea of several multi-plots of tensile data. Each plot show cases some pattern of interest.

- 1. constant total thickness; varied layer thicknesses
- 2. constant outer layer
- 3. constant inner layer
- 4. constant middle layer

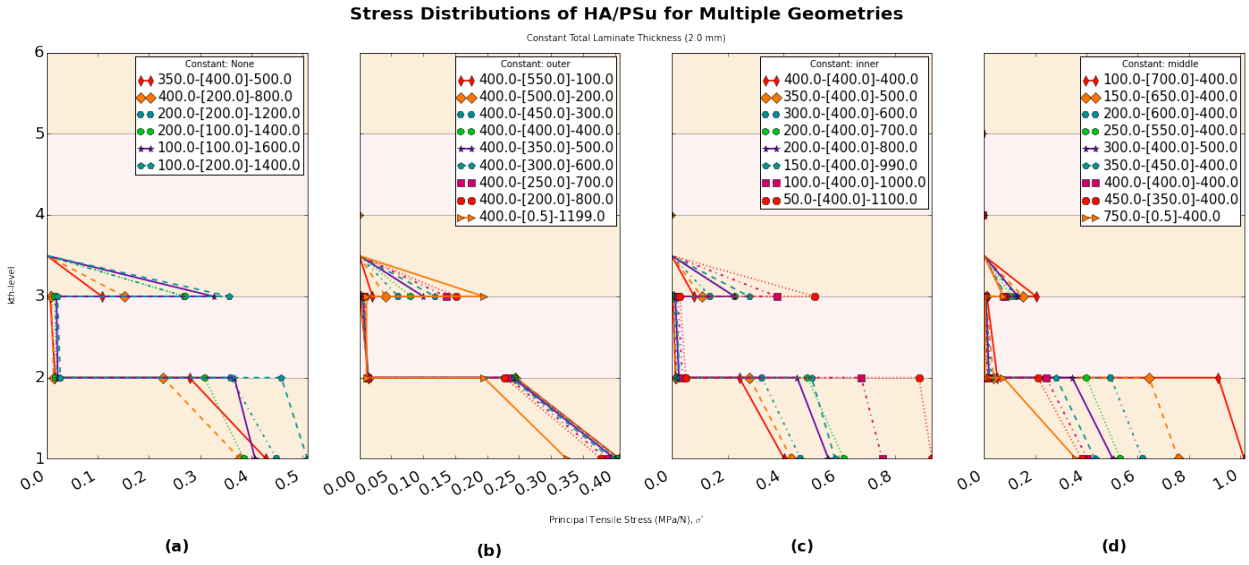
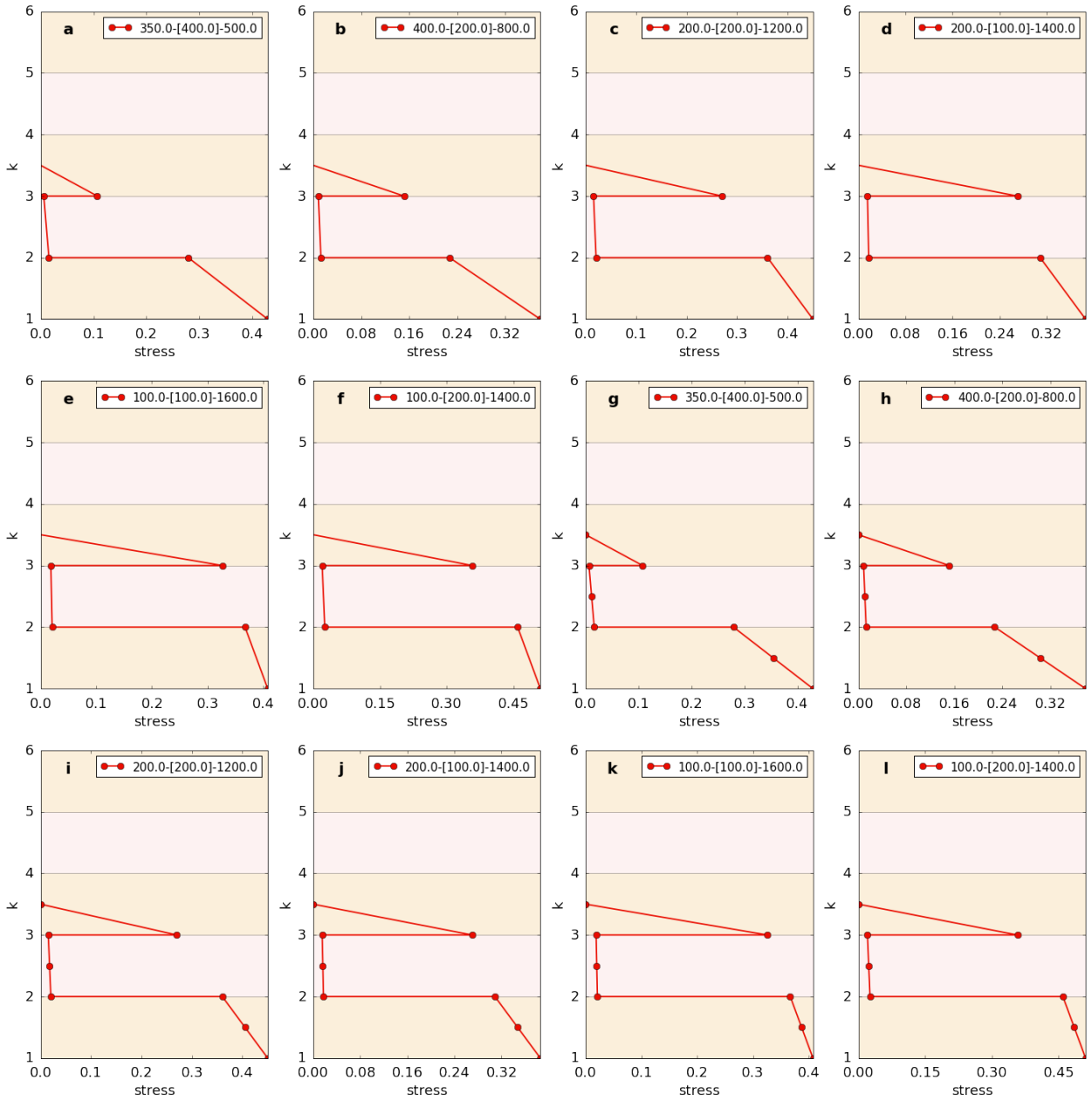


Fig. 2.1: halfplots

```
In [13]: cases1.plot(extrema=False)
```



2.1.3 Data Analysis

Using a prior case, we can analyze the data calculations based on a given theorical model.

Laminate data is contained in a `DataFrame`, a powerful data structure used in data analysis.

```
In [16]: #df = case1.frames
         df = case1.frames[0]
         df
         #df.style                                                    # pandas 0.17.1, css
         #df.style.bar(subset=['stress_f (MPa/N)', 'strain'], color='#d65f5f')
```

```
Accessing frames method.
```

Finally, we can export the latter data and parameters to Excel or .csv formats for archiving and further use. case1.to$_{x}lsx()$

2.2 Conclusion

With simple lines of code, we can quickly perform laminate analysis calculations, visulalize distributions and export data.

...and it's FREE.

2.3 Installation

These introductions explain how to install LamAna. We recommend installing Anaconda (Python 2.7.6+ or Python 3.3+) prior to installing LamAna. This will ensure dependencies are pre-installed.

2.3.1 Install LamAna

There are two simple options for installing LamAna. Open a terminal and run one of the following options:

```
$ pip install lamana                        # from source (default)

$ pip install lamana --use-wheel            # from binary (faster)
```

For more detailed installation instructions, see *Advanced Installation*.

Note: The first installation option is most succint and builds LamAna from source (slow, but canonical). The second option builds from a binary that uses pre-compiled libraries (faster). Both options install the most current dependencies.

2.4 Getting Started

2.4.1 Using LamAna with Jupyter (recommended)

LamAna was developed with visualization in mind.

LamAna works best with the Jupter Notebook 4.0+ (formerly IPyhon 3.2+). Jupyter is a powerful analytical tool that performs computations in separated cells and integrates well with Python. Plotting works best in Jupyter with some `matplotlib` backend initiated in a cell using idiomatic IPython magics e.g. `%matplotlib inline`.

The user starts by importing the desired Feature module, e.g. `distributions` for plots of stress or strain distributions.

2.4.2 Using LamAna from Commandline

If visualization is not important to you, you can still run calcuations and export data from the commandline.

Important: As of *lamana 0.4.9*, the Jupyter *notebook* is not an official dependency and does not install automatically. Rather *notebook* is only frozen in the requirements.txt file. For non-Anaconda users, it must be installed separately. However It is typically packaged with conda ([see documentation for installation](https://jupyter.readthedocs.org/en/latest/install.html))‘. See examples of notebooks in the github repository.

Note: User inputs are handled through feature modules that access the `input_` module using the apply method. Indirect access to `lamana.input_` was decided because importing `input_` first and then accessing a Feature module was cumbersome and awkward for the user. To reduce boilerpoint, the specific Feature module became the frontend while the input transactions where delegated as the backend.

We will now explore how the user can **input data** and **generate plots** using the `distributions` module.

2.4.3 User Setup

First we must input loading parameters and material pproperties. Secondly, we must invoke a selected laminate theory. The former requires knowlege of the specimen dimensions, the materials properties and loading configuration. For illustration, an example schematic of laminate loading parameters is provided below.

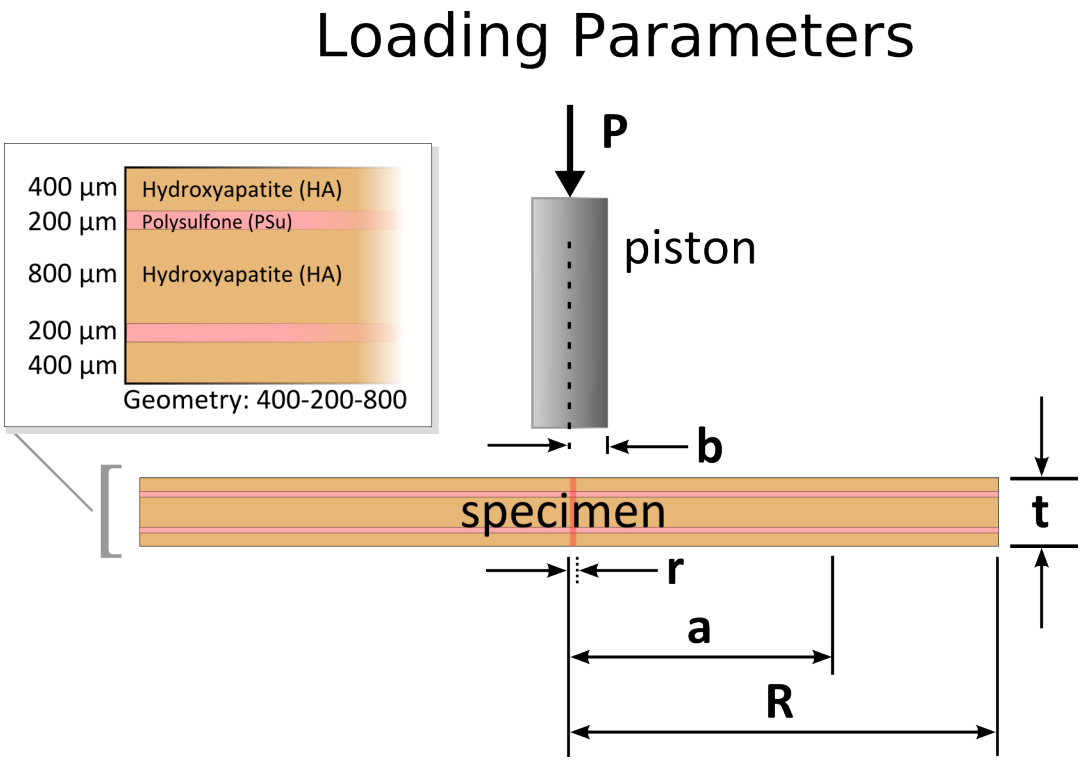


Fig. 2.3: Loading Parameters

A table is provided defining the illustrated parameters. These loading parameters are coded in a dictionary called `load_params`.

Parameter	Units (SI)	Definition
P	N	applied load
R	m	specimen radius
a	m	support radius
b	m	piston radius
r	m	radial distance from central loading
p	•	graphical points or DataFrame rows per layer

User Defined Parameters

Sample code is provided for setting up geometric dimensions, loading parameters and material properties.

```
In [14]: # SETUP ----------------------------------------------------------------------

         import lamana as la

         # For plotting in Jupyter
         %matplotlib inline

         # Build dicts for loading parameters and material properties
         load_params = {
             'P_a': 1,                                                  # applied load
             'R': 12e-3,                                                # specimen radius
             'a': 7.5e-3,                                               # support radius
             'p': 4,                                                    # points/layer
             'r': 2e-4,                                                 # radial distance from
         }

         # Using Quick Form (See Standard Form)
         mat_props = {
             'HA': [5.2e10, 0.25],                                      # modulus, Poissions
             'PSu': [2.7e9, 0.33],
         }

         # Build a list of geometry strings to test.  Accepted conventions shown below:
         # 'outer - [{inner...-...}_i] - middle'

         geos1 = ['400-400-400', '400-200-800', '400-350-500']      # = total thickness
         geos2 = ['400-[400]-400', '400-[200,100]-800']             # = outer thickness
         #-------------------------------------------------------------------------------
```

2.4.4 Generate Data in 3 Lines

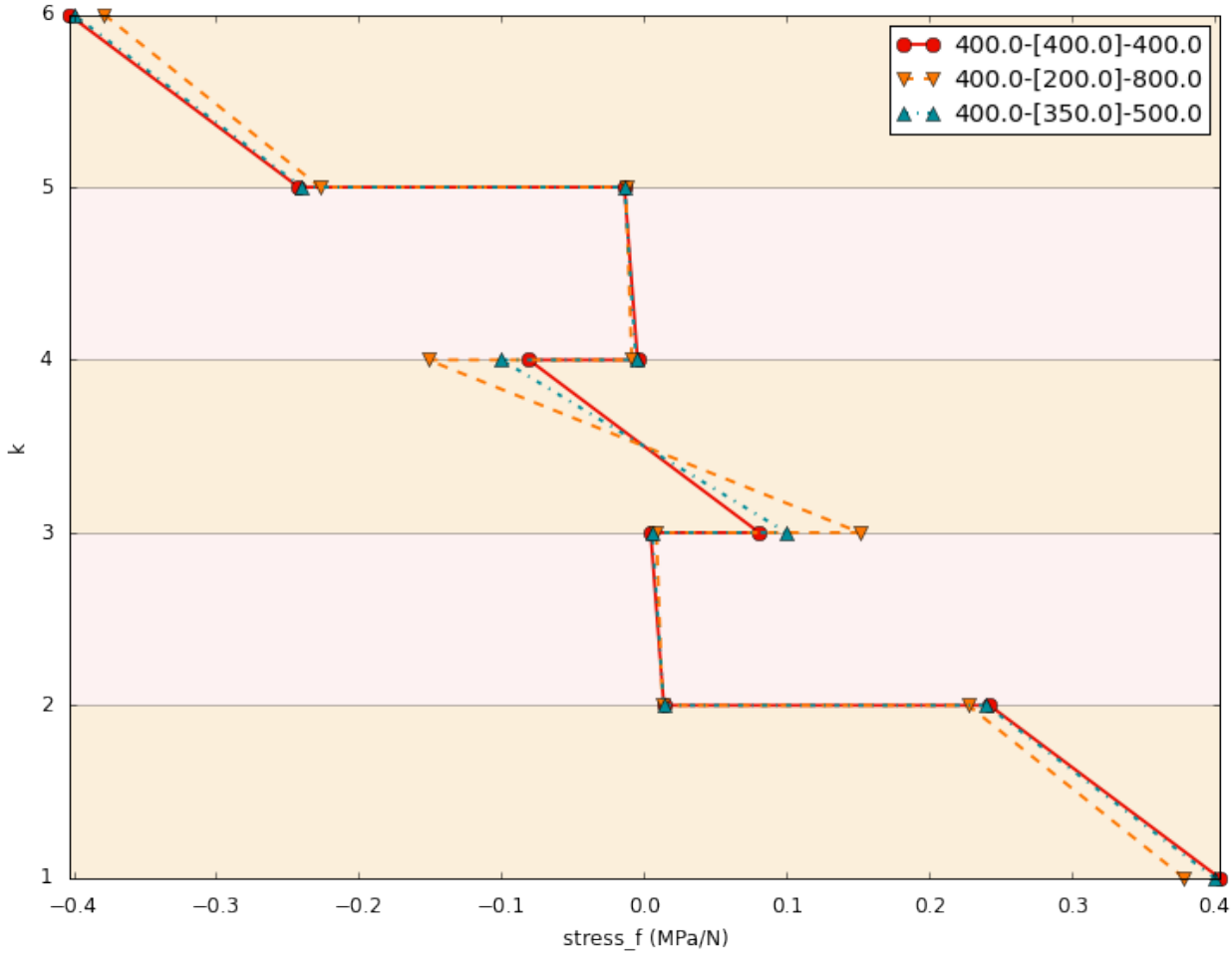
With the **loading and material** information, we can make stress distribution plots to define (reusable) test **cases** by implementing 3 simple steps.

1. *Instantiate* a Feature object with loading and material parameters (generates makes a user Case object)
2. `apply()` a model to a test with desired geometries (assumes mirrored at the neutral axis)
3. `plot()` the case object based on the chosen feature

Once the parameters geometries are set, in three lines of code, you can build a case and simultaneiously plot stress distributions for an indefinite number of laminates varying in compostion and dimension within seconds. Conveniently, the outputs are common Python data structures, specifically `pandas` DataFrames and `matplotlib` graphical plots ready for data munging and analysis.

```
In [13]: case1 = la.distributions.Case(load_params, mat_props)      # instantiate
         case1.apply(geos1, model='Wilson_LT')                     # apply
         case1.plot()                                              # plot
```

```
Converting mat_props to Standard Form.
User input geometries have been converted and set to Case.
```



2.4.5 Other Attributes

A case stores all of the laminate data for a particular set of parameters in two forms: a dict and a DataFrame (see tutorial for details). Once a case is built, there serveral covenient builtin attributes for accessing this data for further analysis.

```
# Case Attributes
case.geometries                                    # geometry object
case.total                                         # total laminate thickness (all)
case.inner                                         # layer thickness (all)
case.total_inner                                   # total layer type (all)
case.total_inner[0]                                # slicing
```

```
case.total_inner_i                                          # total inner layers
case1.snapshot                                              # list of all geometry stacks (unique laye
case1.frames                                                # list all DataFrames (all layers)
```

2.4.6 Extensibile

LamAna is extensible. Users can define custom or modified models based on laminate theory and apply these models to cases (see theories section for details).

```
# Classical Laminate Theory
case2 = la.distributions(load_params, mat_props)            # instantiate
case2.apply(geos2, model='Classic_LT')                      # apply model
case2.plot()

# Custom Biaxial Flexure Model
case3 = la.distributions(load_params, mat_props)            # instantiate
case3.apply(geos2, model='Wilson_LT')                       # custom model
case3.plot()
```

We can perform sepearate analyses by building different cases and apply different models (default model: "Wilson_LT" for circular disks in biaxial flexure).

2.5 API Reference

A quick reference to important LamAna objects.

2.5.1 Core Modules

`input_`	Classes and functions for handling user inputs.
`constructs`	A module that builds core objects, such as stacks and laminates.
`theories`	A interface between constructs and models modules.
`output_`	Classes and functions for handling visualizations, plots and exporting data.

2.5.2 Feature Modules

`distributions`	A Feature Module of classes and functions related to stress distributions.

2.5.3 Auxiliary Modules

`utils.tools`	Handy tools for global use.
`utils.references`	A module for storing weblinks and references for package development.

2.5.4 Models

`models.Wilson_LT`	A sample class-style, custom model using a modified laminate theory.

2.5.5 LamAna Objects

Classes

`input_.Geometry`(geo_input)	Parse input geometry string into floats.
`input_.BaseDefaults`()	Common geometry strings, objects and methods for building defaults.
`distributions.Case`([load_params, mat_props, ...])	Build a Case object that handles User Input parameters.
`distributions.Cases`(caselets[, load_params, ...])	Return a dict-like object of enumerated Case objects.
`constructs.Stack`(FeatureInput)	Build a StackTuple object containing stack-related methods.
`constructs.Laminate`(FeatureInput)	Create a *LaminateModel* object.
`theories.BaseModel`()	Provide attributes for sub-classing custom models.

Functions

`theories.handshake`(Laminate[, adjusted_z])	Return updated LaminateModel and FeatureInput objects.

2.6 Support

Contact the LamAna Team with questions or submit issues to Github.

Email: par2.get@gmail.com

Submit Issues: https://github.com/par2/lamana/issues

2.7 Writing Custom Models

Writing custom theoretical models is a powerful, extensible option of the `LamAna` package.

2.7.1 Authoring Custom Models

Custom models are simple `.py` files that can be locally placed by the user into the models directory. The API allows for calling these selected files in the `apply()` method of the `distributions` module. In order for these process to work smoothly, the following essentials are needed to talk to `theories` module.

1. Implement a `_use_model_()` hook that returns (at minimum) an updated DataFrame.
2. If using the class-style to make models, implement `_use_model_()` hook within a class named "Model" (must have this name) that inherits from `theories.BaseModel`.

Exceptions for specific models are maintained by the models author.

The following cell shows excerpts of the class-style model. Examples of function-style and class-style models are found in the "examples" folder found in the repository.

```
#------------------------------------------------------------------------------
# Class-style model

# ...

class Model(BaseModel):
    '''A custom CLT model.
```

```
    A modified laminate theory for circular biaxial flexure disks,
    loaded with a flat piston punch on 3-ball support having two distinct
    materials (polymer and ceramic).

    '''
    def __init__(self):
        self.Laminate = None
        self.FeatureInput = None
        self.LaminateModel = None

    def _use_model_(self, Laminate, adjusted_z=False):
        '''Return updated DataFrame and FeatureInput.

        ...

        Returns
        -------
        tuple
            The updated calculations and parameters stored in a tuple
            `(LaminateModel, FeatureInput)``.

            df : DataFrame
                LaminateModel with IDs and Dimensional Variables.
            FeatureInut : dict
                Geometry, laminate parameters and more.  Updates Globals dict for
                parameters in the dashboard output.


        '''
        self.Laminate = Laminate
        df = Laminate.LFrame.copy()
        FeatureInput = Laminate.FeatureInput

        # Author-defined Exception Handling
        if (FeatureInput['Parameters']['r'] == 0):
            raise ZeroDivisionError('r=0 is invalid for the log term in the moment eqn.')
        elif (FeatureInput['Parameters']['a'] == 0):
            raise ZeroDivisionError('a=0 is invalid for the log term in the moment eqn.')

        # ...

        # Calling functions to calculate Qs and Ds
        df.loc[:, 'Q_11'] = self.calc_stiffness(df, FeatureInput['Properties']).q_11
        df.loc[:, 'Q_12'] = self.calc_stiffness(df, FeatureInput['Properties']).q_12
        df.loc[:, 'D_11'] = self.calc_bending(df, adj_z=adjusted_z).d_11
        df.loc[:, 'D_12'] = self.calc_bending(df, adj_z=adjusted_z).d_12

        # Global Variable Update
        if (FeatureInput['Parameters']['p'] == 1) & (Laminate.nplies%2 == 0):
            D_11T = sum(df['D_11'])
            D_12T = sum(df['D_12'])
        else:
            D_11T = sum(df.loc[df['label'] == 'interface', 'D_11']) # total D11
            D_12T = sum(df.loc[df['label'] == 'interface', 'D_12'])
        #print(FeatureInput['Geometric']['p'])

        D_11p = (1./((D_11T**2 - D_12T**2)) * D_11T)         #
        D_12n = -(1./((D_11T**2 - D_12T**2))  *D_12T)        #
```

```
        v_eq = D_12T/D_11T                                      # equiv. Poisson's ratio
        M_r = self.calc_moment(df, FeatureInput['Parameters'], v_eq).m_r
        M_t = self.calc_moment(df, FeatureInput['Parameters'], v_eq).m_t
        K_r = (D_11p*M_r) + (D_12n*M_t)                         # curvatures
        K_t = (D_12n*M_r) + (D_11p*M_t)

        # Update FeatureInput
        global_params = {
            'D_11T': D_11T,
            'D_12T': D_12T,
            'D_11p': D_11p,
            'D_12n': D_12n,
            'v_eq ': v_eq,
            'M_r': M_r,
            'M_t': M_t,
            'K_r': K_r,
            'K_t:': K_t,
        }

        FeatureInput['Globals'] = global_params
        self.FeatureInput = FeatureInput                        # update with Globals
        #print(FeatureInput)

        # Calculate Strains and Stresses and Update DataFrame
        df.loc[:,'strain_r'] = K_r * df.loc[:, 'Z(m)']
        df.loc[:,'strain_t'] = K_t * df.loc[:, 'Z(m)']
        df.loc[:, 'stress_r (Pa/N)'] = (df.loc[:, 'strain_r'] * df.loc[:, 'Q_11']
                                    ) + (df.loc[:, 'strain_t'] * df.loc[:, 'Q_12'])
        df.loc[:,'stress_t (Pa/N)'] = (df.loc[:, 'strain_t'] * df.loc[:, 'Q_11']
                                ) + (df.loc[:, 'strain_r'] * df.loc[:, 'Q_12'])
        df.loc[:,'stress_f (MPa/N)'] = df.loc[:, 'stress_t (Pa/N)']/1e6

        del df['Modulus']
        del df['Poissons']

        self.LaminateModel = df

        return (df, FeatureInput)


    #  Add Defaults here
```

Note: DEV: If testing with both function- and class-styles, keep in mind any changes to the model model should be should be reflected in both styles.

What are `Defaults`?

Recall there are a set of geometric, loading and material parameters that are required to run LT calculations. For testing purposes, these parameters can become tedious to set up each time you wish to run a simple plot or test parallel case. Therefore, you can prepare variables that store default parameters with specific values. Calling these variables can reduce the redundancy of typing them over again.

The `BaseDefaults` class stores a number of common geometry strings, Geometry objects, arbitrary loading parameters and material properties. These values are intended to get you started, but can be altered easily to fit your common tolerance for your model. This customization is simple by subclassing `BaseDefaults`. This class also has methods

for easily building accepted, formated FeatureInput objects. Here you can build custom loading parameters, materials properties and FeatureInput objects.

```
class Defaults(BaseDefaults):
    '''Return parameters for building distributions cases.  Useful for consistent
    testing.

    Dimensional defaults are inherited from utils.BaseDefaults().
    Material-specific parameters are defined here by he user.

    - Default geometric parameters
    - Default material properties
    - Default FeatureInput

    Examples
    ========
    >>> dft = Defaults()
    >>> dft.load_params
    {'R' : 12e-3, 'a' : 7.5e-3, 'p' : 1, 'P_a' : 1, 'r' : 2e-4,}

    >>> dft.mat_props
    {'Modulus': {'HA': 5.2e10, 'PSu': 2.7e9},
    'Poissons': {'HA': 0.25, 'PSu': 0.33}}

    >>> dft.FeatureInput
     {'Geometry' : '400-[200]-800',
      'Geometric' : {'R' : 12e-3, 'a' : 7.5e-3, 'p' : 1, 'P_a' : 1, 'r' : 2e-4,},
      'Materials' : {'HA' : [5.2e10, 0.25], 'PSu' : [2.7e9, 0.33],},
      'Custom' : None,
      'Model' : Wilson_LT}

    '''
    def __init__(self):
        BaseDefaults.__init__(self)
        '''DEV: Add defaults first.  Then adjust attributes.'''
        # DEFAULTS ------------------------------------------------------------
        # Build dicts of geometric and material parameters
        self.load_params = {
            'R': 12e-3,                                     # specimen radius
            'a': 7.5e-3,                                    # support ring radius
            'p': 5,                                         # points/layer
            'P_a': 1,                                       # applied load
            'r': 2e-4,                                      # radial distance from center loading
        }

        self.mat_props = {
            'Modulus': {'HA': 5.2e10, 'PSu': 2.7e9},
            'Poissons': {'HA': 0.25, 'PSu': 0.33}
        }

        # ATTRIBUTES ----------------------------------------------------------
        # FeatureInput
        self.FeatureInput = self.get_FeatureInput(
            self.Geo_objects['standard'][0],
            load_params=self.load_params,
            mat_props=self.mat_props,
            model='Wilson_LT',
            global_vars=None
        )
```

Exceptions (0.4.3c6)

Since users can create their own models and use them in `LamAna`, it becomes important to handle erroroneous code. The oneous of exception handling is maintained by the model's author. However, basic handling is incorporated within `Laminate._update_calculations` to prevent erroroneous code from halting LamAna. In other words, provided the variables for Laminate construction are valid, a Laminate will be stored and accessed via `Laminate.LFrame`. This again is the a primitive DataFrame with IDs and Dimensional data prior to updating. When `_update_cacluations()` is called and any exception is raised, they are caught and LFrame is set to LMFrame, allowing other dependency code to work. A traceback will still print even though the exception was caught, allowing the author to improve their code and prevent breakage. LMFrame will not update unless the author model code lacks exceptions. Again, primary exception handling of models is the author's responsibility.

Modified Classical Laminate Theory - `Wilson_LT`

Here is a model that comes with LamAna that applies CLT to circular-disk laminates with alternating ceramic-polymer materials. Classical laminate theory (CLT) was modified for disks loaded in biaxial flexure.

Stiffness Matrix: E is elastic modulus, ν is Poisson's ratio.

$$|Q| = \begin{vmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{vmatrix}$$

$$Q_{11} = Q_{22} = E/(1-\nu^2)$$

$$Q_{12} = Q_{21} = \nu E/(1-\nu^2)$$

Bending : k is essentially the enumerated interface where $k = 0$ is tensile surface. h is the layer thickness relative to the neutral axis where $t_{middle} = h_{middle}/2$. z (lower case) f the relative distance betweeen the neuatral axis and a lamina centroid.

$$|D| = \begin{vmatrix} D_{11} & D_{12} \\ D_{21} & D_{22} \end{vmatrix}$$

$$D_{11} = D_{22} = \Sigma_{k=1}^{N} Q_{11(k)}((h_{(k)}^3/12) + h_{(k)} z_{(k)}^2)$$

$$D_{12} = D_{21} = \Sigma_{k=1}^{N} Q_{12(k)}((h_{(k)}^3/12) + h_{(k)} z_{(k)}^2)$$

Equivalent Poisson's Ratio

$$\nu_{eq} = D_{12}/D_{11}$$

Moments: radial and tangential bending moments. The tangential stress is used for the failure stress.

$$M_r = (P/4\pi)[(1+\nu_{eq})\log(a/r)]$$

$$M_t = (P/4\pi)[(1+\nu_{eq})\log(a/r) + (1-\nu_{eq})]$$

Curvature

$$\begin{Bmatrix} K_r \\ K_t \end{Bmatrix} = [D]^{-1} \begin{Bmatrix} M_r \\ M_t \end{Bmatrix}$$

Strain: Z (caplital) is the distance betwen the neutral axis and the lamina interface.

$$\begin{Bmatrix} \epsilon_r \\ \epsilon_t \end{Bmatrix} = Z_k \begin{Bmatrix} K_r \\ K_t \end{Bmatrix}$$

Stress

$$\begin{Bmatrix} \sigma_r \\ \sigma_t \end{Bmatrix} = [Q] \begin{Bmatrix} \epsilon_r \\ \epsilon_t \end{Bmatrix}$$

2.8 The Package Architecture

The current package stems from a simple legacy script (circa 2014). It has since been updated and currently abstracted with a focus to analyze more general problems related to laminates.

This repository is extensibily designed for various geometry constructs given a specific (or customized) model based on classical laminate theory. This package architecture is diagrammed below.

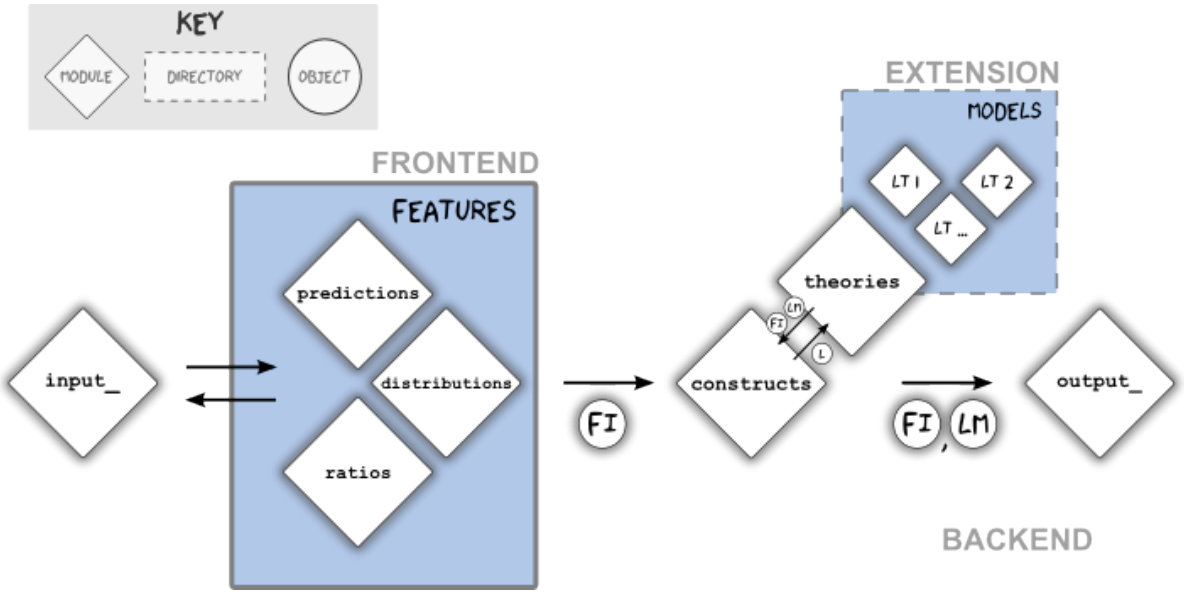


Fig. 2.4: API Diagram

As shown, each diamond represents a module. The diagram illustrates their relationships in passing important `LaminateModels` and `FeatureInput` objects between modules. The user-related areas are highlighted blue. The package is most extensible in these blue areas. The components of the lamana project can be distinguished as three types:

- Frontend: user-interacted, feature modules of particular interest that utilize laminate theory models
- Extension: directories/modules extending capabilities of the repository, e.g. `models` directory containing user defined laminate theories (`Classical_LT`, `Wilson_LT`).
- Backend: remaining Core modules, `input_`, `constructs_`, `theories_`, `output_`; workhorse facotries of LaminateModel objects.

2.8.1 Package Module Summary

This section details some important modules critical to LamAna's operation. The following table summarizes the core and feature modules in this package, what they intend to do and some important objects that result. Objects that get passed between modules are italicized. The Auxillary (or Utility) modules house support code that will not be discussed.

Module	Classifier	Purpose	Product
`input_`	Backend	Backend code for processing user inputs for all feature modules.	User *Input object* i.e. `Geometry`
`distrubtions`	Feature	Analyze stress distributions for different geometries.	*FeatureInput object*, `Case`, `Cases`
`ratios`	Feature	Thickness ratio analyses for optimizing stress-geomtry design.	*FeatureInput object*
`predictions`	Feature	Failure predictions using experimental and laminate theory data.	*FeatureInput object*
`constructs`	Backend	Code for building `Laminate` objects.	*LaminateModel object*
`theories`	Backend	Code for selecting `Model` objects	*Model* object
`<models>`	Extension	Directory of user-defined, custom LT models	*Model* objects
`output_`	Backend	Code for several plotting objects, exporting and saving	Output object e.g. plots, xls, figures

Note: This project is forked from legacy code: *Script - Laminate_Stress_Constant_Thickness_3a3.ipynb*.

Note: Only the `distributions` Feature module is implementated as of LamAna 0.4.10. ratios and predictions will be added in future releases.

Intermodular Products

The key inter-modular products will be mentioned briefly. These objects have information that is exchanged between package modules. These objects are illustrated as circles in the API Diagram.

FeatureInput

A FeatureInput is a Python dict that contains information from both a feature module and user-information processed by the `input_` module. Here is a sample dict and the associated items are tabulated:

Key	Value	Description
`'Geometry'`	Geometry object	a single tuple of Geometry thicknesses
`'Parameters'`	load_params	loading parameters
`'Properties'`	mat_props	material properties, e.g. modulus, Poisson's ratio
`'Materials'`	materials index	ordered list of materials from DataFrame index
`'Model'`	model str	selected string of model name
`'Globals'`	None	a placeholder for future ubiquitous model variables

```
FeatureInput = {
    'Geometry': Geometry,                          # defined in Case
    'Parameters': load_params,
    'Properties': mat_props,
    'Materials': materials,                        # set material order
    'Model': model,
    'Globals': None,                               # defined in models
}
```

Note: DEPRECATED *custom_matls* was depredated and replaced with the *materials* key. The materials order is saved

as a list and interacts with the material cycler in the *add_materials()* method. It can be overridden (see Demonstration for API use).

Note: PEP8 discusses use of trailing underscore to avoid conflicts with Python keywords.

LaminateModel

A `LaminateModel` is an `pandas` DataFrame object that combines data processed by the `constructs.Laminate` and `theories.<model>` classes. Details of this object will be discussed further in the *constructs* section.

2.9 Key Package Components

2.9.1 Core Module: `input_`

`Geometry` class

This class is designed for parsing a user input (assumed a geometry string) and converts it into a Geometry object.

```
LamAna.input_.Geometry(geo_input) --> <Geometry object>
```

A geometry string is formatted to a General Convention representing characteristic laminae types, i.e. outer-inner_i-middle. A `Geometry object` is created of mixed Pythonic types - specifically a namedtiple comprising floats, a list and a string (optional).

We distinguish the latter *string* and coverted *object* types with the following naming conventions:

- geometry string: raw string of the laminate geometry, e.g. `'400-200-800'`
- Geometry object: `Geometry` class instance e.g. `<Geometry object (400-[200]-800)>`

Names referencing geometry strings are lower-case:

- `g`, `geo_inputs`, `geos` or `geos_full`,
- `geos = ['400-[200]-800', '400-[100,100]-400S']`

Names referencing **"Geometry" objects** are capatlized:

- `G`, `Geo_objects`, `Geos` or `Geos_full`,
- `G = la.input_.Geometry(FeatureInput)`

`BaseDefaults` class

This class is essentially a storage for common geometry strings and Geometry objects. Placing them here enables simple inheritance of starter objects when using the API.

There are two main dicts which are stored as instance attributes: `geo_inputs` and `Geo_objects`

geo_inputs

This is a simple dict of common geometry strings with keys named by the number of plies. Again the number of plies is determined by

$$2(outer + inner) + middle$$

. Here is an example `geo_inputs` dict:

```
self.geo_inputs = {
    '1-ply': ['0-0-2000', '0-0-1000'],
    '2-ply': ['1000-0-0'],
    '3-ply': ['600-0-800', '600-0-400S'],
    '4-ply': ['500-500-0', '400-[200]-0'],
    '5-ply': ['400-200-800', '400-[200]-800', '400-200-400S'],
    '6-ply': ['400-[100,100]-0', '500-[250,250]-0'],
    '7-ply': ['400-[100,100]-800', '400-[100,100]-400S'],
    '9-ply': ['400-[100,100,100]-800'],
    '10-ply': ['500-[50,50,50,50]-0'],
    '11-ply': ['400-[100,100,100,100]-800'],
    '13-ply': ['400-[100,100,100,100,100]-800'],
}
```

Additional keys are added to this dict such as 'geos_even', 'geos_odd' and 'geos_all' which create new key-value pairs of groups for even, odd and all geometry strings. Notice the naming placement of 's': "geo_inputs" is the base dict while "geos_" is a grouping of existing dict values appended to the dict. Therefore an author or developer could extend either the base or appended dict items.

Geo_objects

This is a lazy dict. All entries of `geo_inputs` are automatically converted and stored as Geometry objects. The purpose here is to eliminate the added step of calling Geometry to convert strings. Both this dict and the `geo_inputs` dict are created using similar private methods, so there mechanisms are parallel.

Subclassing

The remaining defeaults such as `load_params`, `mat_props` and `FeatureInput` are specific to experimental setups and cannot be generalized effectively. However, this class can be subclassed into a custom `Defaults` class by the author. See the Authour Documentation for examples of subclassing.

Important: DEV: Only add geometry strings to *geo_inputs*. Removing or "trimming" these dicts may break tests.

Important: In future versions, `load_params`, `mat_props` and `FeatureInput` will be added to BaseDefaults() as attributes to partake in inheritance.

2.9.2 Feature Module: `distributions`

`Case` class

The `Case` class translates user information into managable, analytical units. A `Case` object is:

1. instantiated
2. user info is applied such as geoemtry strings, model name, etc.
3. method and proerties are accessed, such as `plot()` and `total`

Here is an idiomatic example of the latter characteristics:

```
case = la.distributions.Case(load_params, mat_props)
case.apply(geo_strings=None, model='Wilson_LT', **kwargs)
case.plot(**kwargs)
```

The `case` instance accepts loading and material information and sets up their associated dicts. Specific geometry strings and one model is applied to the case object. This `apply()` method generates `LaminateModel` objects (`FeatureInput` objects are also made). Information is parsed, calculated (such as layer thicknesses) and stored in attributes. These attributes and methods are then accessible for performing analysis, most importantly the `plot()` method.

Therefore, you can think of a case as an analytical unit comprising start up data converted to LaminateModel objects.

`Cases` class

The `Cases` class supplies options for manipulating multiple case objects. For example, set operations can be performed on multiple cases. In this context, each `case` is termed a `caselet` and typically correlated with a matplotlib subplot. Here is an idiomatic example:

```
import lamana as la

bdft = la.input_.BaseDefaults()
cases = Cases(bdft.geo_inputs['geos_all'], ps=[2,3,4])
```

The latter code builds cases for all geometry strings contained in the `BaseDefaults()` class, one for each `p` number of datapoints. Therefore in this example *dozens* of analytical units are built with only three lines of code. See LPEP 002 and LPEP 003 for the motivation and details on `Cases`.

2.9.3 Core Module: `constructs`

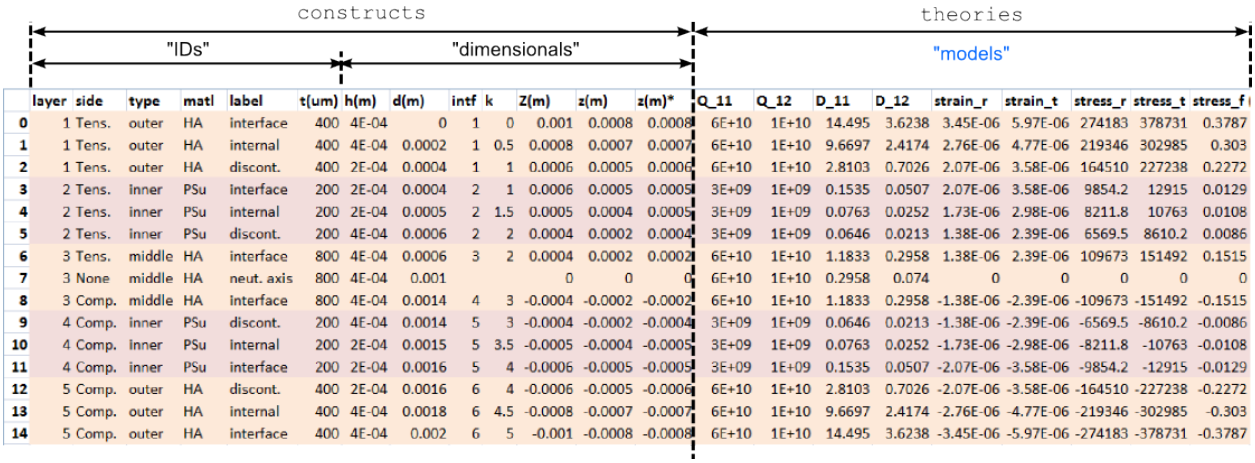
Principally, the `constructs` module builds a `LaminateModel` object. Technically a `LaminateModel` is a `pandas <http://pandas.pydata.org/>`__ DataFrames representing a physical laminate with a few helpful attributes. DataFrames were chosen as the backend object because they allow for powerful data manipulation analyses and database/spreadsheet-like visualizations with simple methods.

Additionally, the `constructs` module computes laminate dimensional columns and compiles theoretical calculations handled by the complementary `theories` module. Conventiently, all of this data is contained in tabular form within the DataFrame. The column names are closely related to computational variables defined in the next sub-section.

Variable Classifications

Before we discuss the Laminate structure, here we distinguish two ubiquitous variable categories used internally: "Laminate" and "model" variables. In in a full laminate DataFrame, these categories comprise variables that are represented as columns. The categories variables, columns and corresponding modules are illustrated in the image below and described in greater detail:

An image of the output for a DataFrame and their labeled categories of columns (IDs, dimensionals and models). The first two categories are computed by `constructs` classess; the models columns are computed by `theories`



	layer_	side_	type_	matl_	label_	t(um)	h(m)	d(m)	intf	k	Z(m)	z(m)	z(m)*	Q_11	Q_12	D_11	D_12	strain_r	strain_t	stress_r	stress_t	stress_f
0	1	Tens.	outer	HA	interface	400	4E-04	0	1	0	0.001	0.0008	0.0008	6E+10	1E+10	14.495	3.6238	3.45E-06	5.97E-06	274183	378731	0.3787
1	1	Tens.	outer	HA	internal	400	4E-04	0.0002	1	0.5	0.0008	0.0007	0.0007	6E+10	1E+10	9.6697	2.4174	2.76E-06	4.77E-06	219346	302985	0.303
2	1	Tens.	outer	HA	discont.	400	2E-04	0.0004	1	1	0.0006	0.0005	0.0006	6E+10	1E+10	2.8103	0.7026	2.07E-06	3.58E-06	164510	227238	0.2272
3	2	Tens.	inner	PSu	interface	200	2E-04	0.0004	2	1	0.0006	0.0005	0.0005	3E+09	1E+09	0.1535	0.0507	2.07E-06	3.58E-06	9854.2	12915	0.0129
4	2	Tens.	inner	PSu	internal	200	2E-04	0.0005	2	1.5	0.0005	0.0004	0.0005	3E+09	1E+09	0.0763	0.0252	1.73E-06	2.98E-06	8211.8	10763	0.0108
5	2	Tens.	inner	PSu	discont.	200	4E-04	0.0006	2	2	0.0004	0.0002	0.0004	3E+09	1E+09	0.0646	0.0213	1.38E-06	2.39E-06	6569.5	8610.2	0.0086
6	3	Tens.	middle	HA	interface	800	4E-04	0.0006	3	2	0.0004	0.0002	0.0002	6E+10	1E+10	1.1833	0.2958	1.38E-06	2.39E-06	109673	151492	0.1515
7	3	None	middle	HA	neut. axis	800	4E-04	0.001			0	0	0	6E+10	1E+10	0.2958	0.074	0	0	0	0	0
8	3	Comp.	middle	HA	interface	800	4E-04	0.0014	4	3	-0.0004	-0.0002	-0.0002	6E+10	1E+10	1.1833	0.2958	-1.38E-06	-2.39E-06	-109673	-151492	-0.1515
9	4	Comp.	inner	PSu	discont.	200	4E-04	0.0014	5	3	-0.0004	-0.0002	-0.0004	3E+09	1E+09	0.0646	0.0213	-1.38E-06	-2.39E-06	-6569.5	-8610.2	-0.0086
10	4	Comp.	inner	PSu	internal	200	2E-04	0.0015	5	3.5	-0.0005	-0.0004	-0.0005	3E+09	1E+09	0.0763	0.0252	-1.73E-06	-2.98E-06	-8211.8	-10763	-0.0108
11	4	Comp.	inner	PSu	interface	200	2E-04	0.0016	5	4	-0.0006	-0.0005	-0.0005	3E+09	1E+09	0.1535	0.0507	-2.07E-06	-3.58E-06	-9854.2	-12915	-0.0129
12	5	Comp.	outer	HA	discont.	400	2E-04	0.0016	6	4	-0.0006	-0.0005	-0.0006	6E+10	1E+10	2.8103	0.7026	-2.07E-06	-3.58E-06	-164510	-227238	-0.2272
13	5	Comp.	outer	HA	internal	400	4E-04	0.0018	6	4.5	-0.0008	-0.0007	-0.0007	6E+10	1E+10	9.6697	2.4174	-2.76E-06	-4.77E-06	-219346	-302985	-0.303
14	5	Comp.	outer	HA	interface	400	4E-04	0.002	6	5	-0.001	-0.0008	-0.0008	6E+10	1E+10	14.495	3.6238	-3.45E-06	-5.97E-06	-274183	-378731	-0.3787

Fig. 2.5: dataframe output

classes and models. The highlighted blue text indicates user interaction. Groups of rows are colored with alternating red and orange colors to distinguish separate layers.

What distinguishes "Laminate" variables from "Model" variables

- **Laminate** (or `constructs`) variables are responsible for building the laminate *stack* and defining dimensions of the laminate. Internally, these varibles will be semantically distinguished with one trailing underscore.
    1. **ID**: variables related to layer and row identifications
        (a) `layer_`, `side_`, `matl_`, `type_`, `t_`
    2. **Dimensional**: variables of heights relative to cross-sectional planes
        (a) `label_`, `h_`, `d_`, `intf_`, `k_`, `Z_`, `z_`
- **Model** (or `theories`) variables: all remaining variables are relevant for LT calculations and defined from a given model. Since these variables are model-specific, theres is no particular semantic or naming format.

The finer granularity seen with model variables is not essential for typcial API use, but may be helpful when authoring custom code that integrates with LamAna.

Further Details of Model Variables

For more detailed discussions, model variables can be further divided into sub-categories. There common subsets are as follows:

```
1. **User**: global variables delibrately set by the user at startup
2. **Inline**: variables used per lamina at a kth level (row)
3. **Global**: variables applied to the laminate, accessible by ks
```

Although model variables are often particular to a chosen model, e.g Wilson_LT, there are some general trends that may be adopted. Some model variables are provided at startup by the user (user_vars). Some variables are calculated for each row of the data within the table (inline_vars). Some variables are calculated by the designated laminate theory model, which provide constants for remaining calculations (global_vars). Global values would display as the same number for every row. These constants are thus removed from the DataFrame, but they are stored internally within a `dict`. The details of this storage are coded within each model module.

Global values are of particular importance to `FeatureInput` objects and when exporting meta data as dashboards in spreadsheets. In contrast, Inline values alter directly with the dimensional values thoroughout the lamainate thickness. Names of common variables used in `distributions` are organized below:

Model Variable Subsets

```
Model_vars = {user_vars, inline_vars, global_vars}
```

Examples of Subsets of Model Variables

- user_vars = [`mat_props`, `load_params`]
- global_vars = [`v_eq`, `D_11T`, `D_12T`, `M_r`, `M_t`, `D_11p`, `D_12n`, `K_r`, `K_t`]
- inline_vars = [`Q11`, `Q12`, `D11`, `D12`, `strain_r`, `strain_t`, `stress_r`, `stress_t`, `stress_f`]

TIP: Aside from user variables, all others are found as headers for columns in a DataFrame (or spreadsheet).

The `Laminate` Architecture

This section will describe in greater detail how `LaminateModels` are constructed.

When the user calls `case.apply()`, a number of objects are created. We begin with a primitive `Stack`, which comprises skeletal components for building a `Laminate` DataFrame (also interally called an LFrame). The phases for building a `LaminateModel` object are outlined below and outline the architecture of `constructs.Laminate` class.

- Phase 1: build a primitive laminate (Stack)
- Phase 2: calculate Laminate dimensional values (LFrame)
- Phase 3: calculate laminate theory Model values (LMFrame aka `LaminateModel`)

Phase 1: The `Stack` Class

The purpose of the `Stack` class is to build a skeletal, precusor of a primitive `Laminate` object. This class houses methods for parsing Geometry objects, ordering layers, adding materials labels for each layer and setting *expected* stress states for each tensile or compressive side. `Stack` returns a namedtuple containing stack-related information (described below).

For a given `Geometry` object instance (commonlly assigned to a capital "G") the `Stack().StackTuple` method creates a namedtuple of the stack information. This object contains attributes to access the:

- stack `order`
- the number of plies, `nplies`
- the technical `name` for the laminate, "4-ply", "5-ply"
- a convenient `alias` if any, e.g. "Bilayer", "Trilayer"

The `stack` attribute accesses a dict of the laminate layers ordered from bottom to top. Now although Python dicts are unsorted, this particular dict is sorted because each layer is enumerated and stored as keys to perserve the order, layer thickness and layer type (sometimes referred as "ltype").

```
Examples
--------
>>> import LamAna as la
>>> G = la.input_.Geometry(['400-200-800'])
>>> G
<Geometry object (400-[200]-800)>
```

```
Create a StackTuple and access its attributes
>>> st = constructs.Stack(G).StackTuple    # converts G to a namedtuple
>>> st.order                               # access namedtuple attributes
{1: [400.0, 'outer'],
 2: [200.0, 'inner']
 3: [800.0, 'middle']
 4: [200.0, 'inner']
 5: [400.0, 'outer']}
>>> st.nplies
5
>>> st.name
'5-ply'
>>> st.alias
'standard'
```

Phase 2: The `Laminate` class

The `Laminate` class simply builds a `LaminateModel` - an object containing all dimensional information of a physical `Laminate` **and** all theoretical calculations using a laminate theory `Model`, e.g. stress/strain.

The `Laminate` class builds an LFrame object based on the skeletal layout of a stack parsed by and returned from the `Stack` class. A `Geometry` object, material parameters and geometric parameters are all passed from the user in as a single `FeatureInput` object - a dict of useful information that is passed between modules. See **More on ''FeatureInput''** for details. Stack information is stored in an instance attribute called `Snapshot` and then converted to a set of DataFrames.

Therefore, the IDs and dimensional data are determined and computed by `Stack` and `Laminate`. Combined, this information builds an LFrame.

Phase 3: The `Laminate` class (continued)

`Laminate` then calls the `theories` module which "handshakes" between the `Laminate` module and the custom module containing code of a user-specified, theoretical LT model. It is common for a custom model to be named by the author, suffixed by the characters "`_LT`"). These computations update the Laminate DataFrame (`Laminate.LFrame`), creating a final `LaminateModel` (`Laminate.LMFrame`). The complete workflow is summarized below.

Summary of LaminateModel Workflow

```
constructs :: class Stack --> class Laminate

theories :: class BaseModel
```

Laminate object + "Model" object –> LaminateModel object

Detailed workflow of `constructs-theories` interaction:

```
class Stack --> StackTuple
 |
class Laminate --> Snapshot, LFrame, LMFrame
 |
 | # Phase 1 : Instantiate; Determine Laminate ID Values
```

```
 | Laminate._build_snapshot(stack) --> Snapshot
 |      |
 |   Stack.add_materials(stack)
 |   Stack.stack_to_df(stack)        # first creation of the Laminate df
 |   Laminate._set_stresses(stack)
 |
 | Laminate._build_laminate(snapshot) --> LFrame
 |
 | # Phase 2 : Calculate Laminate Dimensional Values
 | Laminate._update_columns._update_dimensions() --> LFrame (updated)
 |        label_, h_, d_, intf_, k_, z_, Z_
 |
 | # Phase 3 : Calculate Model Values
 | Laminate._update_columns._update_calculations() --> LMFrame
 |    theories.Model(Laminate)
 |    models.<selected model>
 |       _calc_stiffness()
 |       _calc_bending()
 |       _calc_moment()
 |       global_vars = [`v_eq`, `D_11T`, `D_12T`, ...]
 |       inline_vars = [`Q11`, `D11` `strain_r`, ...]
 |
LaminateModel : df
```

Additional Details

More on Material Stacking Order

The material order is initially defined by the user `mat_props` dict in `distributions` and automatically parsed in the `input_` module. Extracting order from a dict is not trivial, so the default sorting is alphabetical order. This order is handled by converting the dict to a pandas index. See `Stack.add_materials()` method for more details.

As of 0.4.3d4, the user can partially override the default ordering by setting the `materials` property in the Case instance. This allows simple control of the stacking order in the final laminate stack and `Laminate` objects. At the moment, a list of materials is cycled through; more customizations have not been implemented yet.

```
>>> case.material
['HA', 'PSu']                                  # alphabetical order
>>> case.material = ['PSu', 'HA']              # overriding order
>>> case.material
['PSu', 'HA']
>>> case.apply(...)
<materials DataFrame>                          # cycles the stacking order
```

More on `Laminate`

Using `Laminate._build_snapshot()`, the instance stack dict is converted to a DataFrame (`Snapshot`), giving a primitive view of the laminate geometry, idenfiers (IDs) and stacking order. This "snapshot" has the following ID columns of infornation, which are accessible to the user in a `Case` instance (see `distributions.Case.snapshot`):

```
Variables addressed: `layer_, matl_, type_, t_`
```

From this snapshot, the DataFrame can is updated with new information. For example, the sides on which to expected tensile and compressive stresses are located (`side_`) are assigned to a laminate through the `Laminate._set_stresses()` method. This function accounts for DataFrames with even and odd rows. For odd rows, 'None' is assigned to the neutral axis, implying "no stress".

```
Variables addressed: `side_`
```

Note: This stress assignment is a general designation, coarsely determined by which side of the netural axis a row is found. The rigorous or finite stress state must be calculated through other analytical tools means such as Finite Element Analysis.

Likewise, the DataFrame is further updated with columns of dimensional data (from Dimensional variables) and laminate theory data (from model variables). The current `LaminateModel` object is made by calling `Laminate._update_columns._build_laminates()` which updates the snapshot columns to build two DataFrame objects:

Here are similarities between the laminate data columns and the its objects:

- `Snapshot`: primiate DataFrame of the Stack (see materials, layer info order).
- `LFrame`: updated `Snapshot` of IDs and dimensionals.
- `LMFrame`: updated LFrame with models computed columns.

`LMFrame` is the paramount data structure of interest containing all IDs, Dimensional and Model variables and `p` number of rows pertaining to data points within a given lamina.

Dimensional variable columns are populated through the `Laminate._update_columns._update_dimensions()` method, which contains algorithms for calculating realative and absolute heights, thicknesses and midplane distances relative to the neutral axis. These columns contain dimensional data that are determined independent from the laminate theory model.

```
Variables addresed: `label_, h_, d_, intf_, k_, Z_, z_`
```

These variables are defined in the Laminate class docstring. See *More on label_* to understand the role of points, `p` and their relationship to DataFrame rows.

Finally Data variable columns are populated using `Laminate._update_columns._update_calculations()`. These columnns contain data based on calculations from laminate theory for a selected model. Here global_vars and inline_vars are calculated.

```
Variables addressed:
--------------------
global_vars = [`v_eq, D_11T, D_12T, M_r, M_t, D_11p, D_12n, K_r, K_t`] --> FeatureInput['Global'] (di

inline_vars = [`Q11, Q12, D11, D12, strain_r, strain_t, stress_r, stress_t, stress_f`] --> LaminateMo
```

More on `FeatureInput`

A Feature module defines a `FeatureInput` object.

For `distributions`, it is defined in `Case`. `FeatureInputs` contain information that is passed between objects. For instance, this object transfers user input data in `distributions` (converted in `input_`) to the `constructs` module to build the laminate stack and populate ID and dimensional columns. A FeatureInput from `distributions` looks like the following (as of 0.4.4b).

```
FeatureInput = {
    'Geometry': <Geometry object>,
    'Loading': <load_params dict>,
    'Materials': <mat_props dict>,
```

Snapshot

	layer	side	matl	type	t(um)
0	1	Tens.	HA	outer	400
1	2	Tens.	PSu	inner	200
2	3	INDET	HA	middle	800
3	4	Comp.	PSu	inner	200
4	5	Comp.	HA	outer	400

LFrame

	layer	side	type	matl	label	t(um)	h(m)	d(m)	intf	k	Z(m)	z(m)	z(m)*
0	1	Tens.	outer	HA	interface	400	0.0004	0.0000	1	0.0	0.0010	0.00080	0.00080
1	1	Tens.	outer	HA	internal	400	0.0004	0.0002	1	0.5	0.0008	0.00065	0.00070
2	1	Tens.	outer	HA	discont.	400	0.0002	0.0004	1	1.0	0.0006	0.00050	0.00060
3	2	Tens.	inner	PSu	interface	200	0.0002	0.0004	2	1.0	0.0006	0.00050	0.00050
4	2	Tens.	inner	PSu	internal	200	0.0002	0.0005	2	1.5	0.0005	0.00035	0.00045
5	2	Tens.	inner	PSu	discont.	200	0.0004	0.0006	2	2.0	0.0004	0.00020	0.00040
6	3	Tens.	middle	HA	interface	800	0.0004	0.0006	3	2.0	0.0004	0.00020	0.00020
7	3	None	middle	HA	neut. axis	800	0.0004	0.0010	NaN	NaN	0.0000	0.00000	0.00000
8	3	Comp.	middle	HA	interface	800	0.0004	0.0014	4	3.0	-0.0004	-0.00020	-0.00020
9	4	Comp.	inner	PSu	discont.	200	0.0004	0.0014	5	3.0	-0.0004	-0.00020	-0.00040
10	4	Comp.	inner	PSu	internal	200	0.0002	0.0015	5	3.5	-0.0005	-0.00035	-0.00045
11	4	Comp.	inner	PSu	interface	200	0.0002	0.0016	5	4.0	-0.0006	-0.00050	-0.00050
12	5	Comp.	outer	HA	discont.	400	0.0002	0.0016	6	4.0	-0.0006	-0.00050	-0.00060
13	5	Comp.	outer	HA	internal	400	0.0004	0.0018	6	4.5	-0.0008	-0.00065	-0.00070
14	5	Comp.	outer	HA	interface	400	0.0004	0.0020	6	5.0	-0.0010	-0.00080	-0.00080

LMFrame

	layer	side	type	matl	label	t(um)	h(m)	d(m)	intf	k	...	z(m)*	Q_11	Q_12	D_11	D_12	strain
0	1	Tens.	outer	HA	interface	400	0.0004	0.0000	1	0.0	...	0.00080	5.546667e+10	1.386667e+10	14.495289	3.623822	0.000
1	1	Tens.	outer	HA	internal	400	0.0004	0.0002	1	0.5	...	0.00070	5.546667e+10	1.386667e+10	9.669689	2.417422	0.000
2	1	Tens.	outer	HA	discont.	400	0.0002	0.0004	1	1.0	...	0.00060	5.546667e+10	1.386667e+10	2.810311	0.702578	0.000
3	2	Tens.	inner	PSu	interface	200	0.0002	0.0004	2	1.0	...	0.00050	3.029963e+09	9.998878e+08	0.153518	0.050661	0.000
4	2	Tens.	inner	PSu	internal	200	0.0002	0.0005	2	1.5	...	0.00045	3.029963e+09	9.998878e+08	0.076254	0.025164	0.000
5	2	Tens.	inner	PSu	discont.	200	0.0004	0.0006	2	2.0	...	0.00040	3.029963e+09	9.998878e+08	0.064639	0.021331	0.000
6	3	Tens.	middle	HA	interface	800	0.0004	0.0006	3	2.0	...	0.00020	5.546667e+10	1.386667e+10	1.183289	0.295822	0.000
7	3	None	middle	HA	neut. axis	800	0.0004	0.0010	NaN	NaN	...	0.00000	5.546667e+10	1.386667e+10	0.295822	0.073956	0.000
8	3	Comp.	middle	HA	interface	800	0.0004	0.0014	4	3.0	...	-0.00020	5.546667e+10	1.386667e+10	1.183289	0.295822	-0.000
9	4	Comp.	inner	PSu	discont.	200	0.0004	0.0014	5	3.0	...	-0.00040	3.029963e+09	9.998878e+08	0.064639	0.021331	-0.000
10	4	Comp.	inner	PSu	internal	200	0.0002	0.0015	5	3.5	...	-0.00045	3.029963e+09	9.998878e+08	0.076254	0.025164	-0.000
11	4	Comp.	inner	PSu	interface	200	0.0002	0.0016	5	4.0	...	-0.00050	3.029963e+09	9.998878e+08	0.153518	0.050661	-0.000
12	5	Comp.	outer	HA	discont.	400	0.0002	0.0016	6	4.0	...	-0.00060	5.546667e+10	1.386667e+10	2.810311	0.702578	-0.000
13	5	Comp.	outer	HA	internal	400	0.0004	0.0018	6	4.5	...	-0.00070	5.546667e+10	1.386667e+10	9.669689	2.417422	-0.000
14	5	Comp.	outer	HA	interface	400	0.0004	0.0020	6	5.0	...	-0.00080	5.546667e+10	1.386667e+10	14.495289	3.623822	-0.000

Fig. 2.6: laminate objects

```
    'Custom': <undefined>,
    'Model': <string>,
    'Globals': <dict>,
}
```

After calculating model data, the "Globals" key is updated containing all necessary `globabl_vars`. These variables are constant and are necessary for further calculations of `inline_vars`. Here is an example of Global variables key-value pair in FeatureInput.

```
FeatureInput['Globals'] = [v_eq, D_11T, D_12T, M_r, M_t, D_11p, D_12n, K_r, K_t]
```

More on `label_`

See LPEP 001.02 for standards of API units.

For this explanation, imagine we transverse the absolute height of the laminate at different cross-sectional planes. The values of inline stress points are calculated along different planes throughout the laminate thickness. What happens at interfaces where two materials meet with different stresses? How are these two stress points differentiated in a DataFrame or in a plot? For plotting purposes, we need to define diferent types of points. Here we define some rulse and four types of points found within a (i.e. DataFrame rows):

1. interfacial - point on unbound outer surfaces and bound internal surfaces.
2. internal - point with the lamina thickness between interfaces
3. discontinuity - point on bounded interfaces pertaining to an adjacent lamina
4. neutralaxis - the middle, symmetric axial plane

How these points are distributed depends on their locations within each lamina and whether they are located on the tensile or compressive `side_`. The neutral axis exists in physical laminates, but they are only represented as a row in DataFrames of odd ply, odd p laminates; they are not displayed in even laminates. The image below illustrates the different points from above with respect to `k_` (the fractional height for a given layer).

Notice various layers have different point types.

- Middle layers have two interfacial points, no discontinuities and a neutral axis.
- All other layers have one interfacial point with a discontinuity if p >= 2.
- All layers may (or may not) have internal points.
- Monoliths do not have discontinuities

Note: Only the interfacial points can be theoreticlly verified, representing the maximum principal strains and stresses. The internal and discontinuity points are merely used by matplotlib to connect the points, assuming a linear stress distribution.

Note: The midplane z height ($z\_$) for discontinuities assumes a non-zero, lower limit value equal to the **Z_** height of the bounding layer. This value should be verified.

More on `IndeterminateError`

An `IndeterminateError` is thrown in cases where values cannot be calculated. An `INDET` keyword is given as values in DataFrame cells. An example for such an error is determining the stress state `side_` for a monolith with one data point (nplies=1, p=1). From a design perspective, the location of the point is ambiguous, either one one interface, but more intuitively at the neutral access. At such a position, the value of stress would report zero, which is misleading for the true stress state of the monolith. Therefore, the `InderminateError` is thrown, recommending at least p = 2 for disambiguated stress calculations.

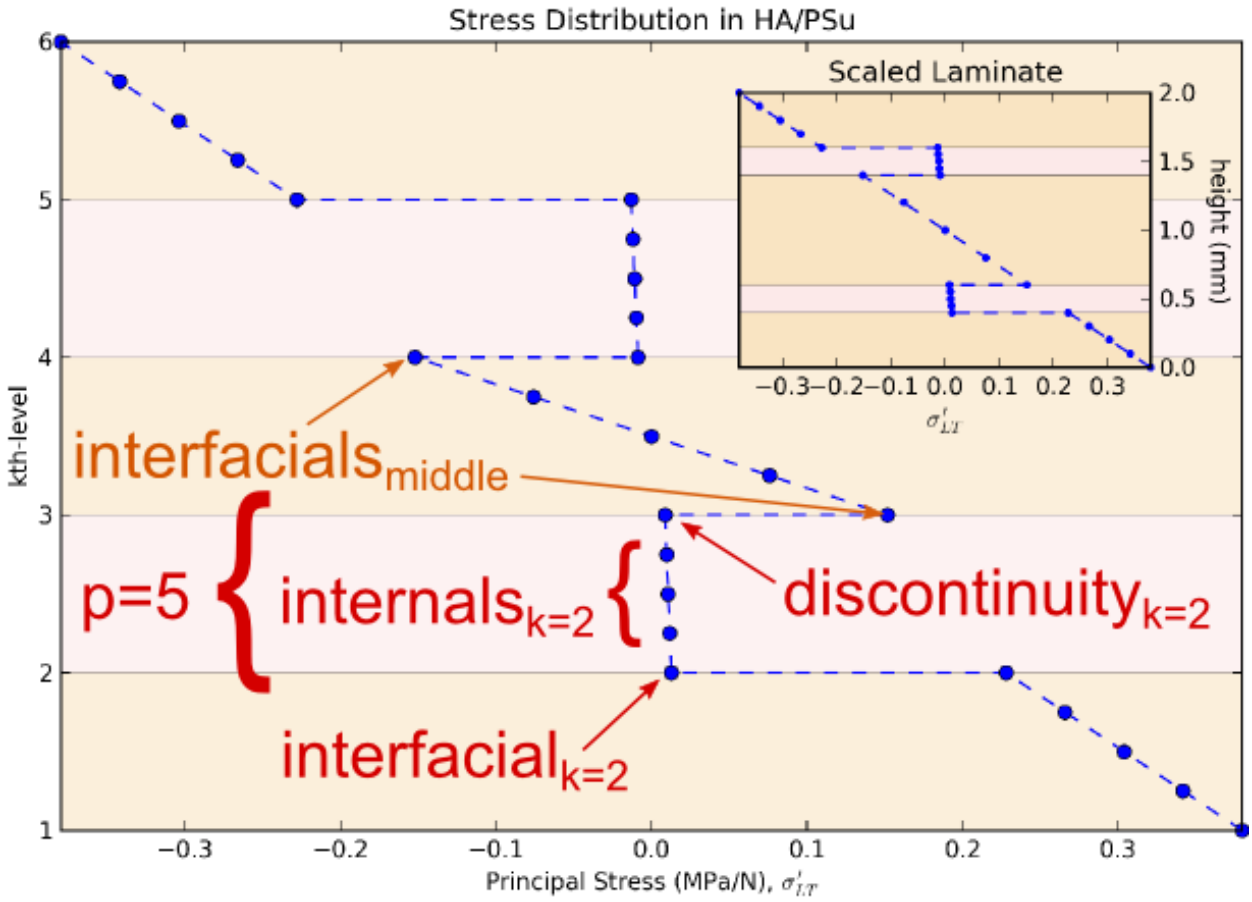


Fig. 2.7: points

2.9.4 Core Module: `theories`

Laminate theory is merged with dimensional data to create a `LaminateModel`.

`LaminateModel` Handling

For clarify, an illustration of LaminateModel handling is shown below.

The `Laminate` DataFrame (LFrame) is passed from `constructs` to `theories`. If successful the `LaminateModel` is returned to `constructs`; otherwise an exception is thrown, consumed and the `Laminate` is returned unchanged (LFrame).

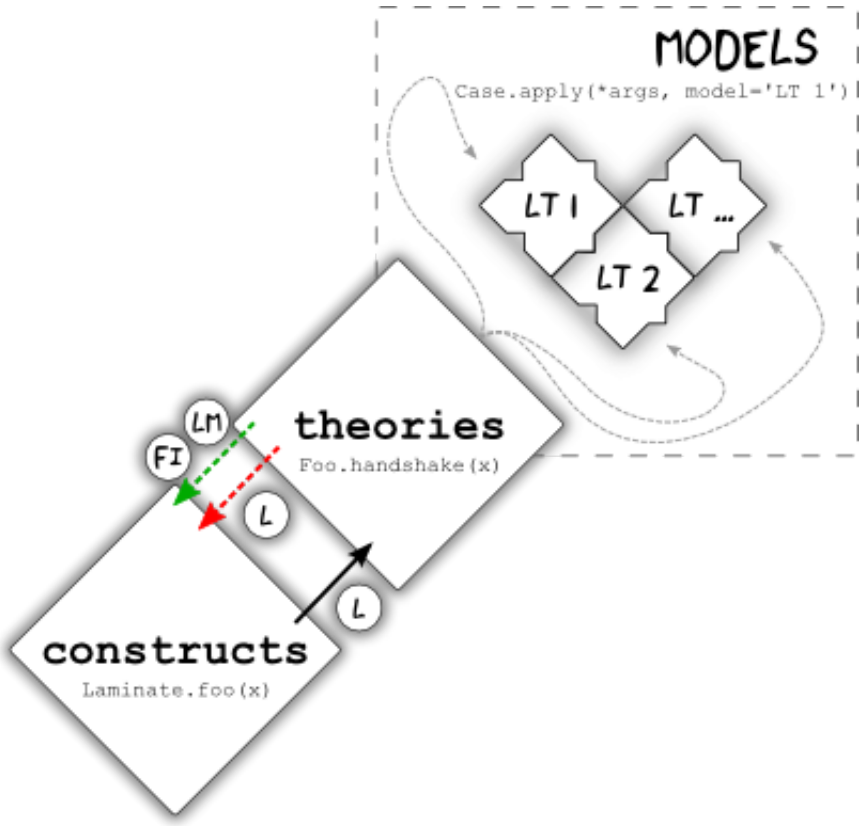


Fig. 2.8: theories flowchart

Note: The term repr for *<LaminateModel object>* remains constant refering to a post-theories operation, whether LMFrame is updated with Model columns or not.

When `Laminate._update_columns._update_calculations()` (represented as `Laminate.foo()`) is called, an instance of the `Laminate self` (shown as "x") is passed to `theories.handshake()` (black arrow). This function handles all updates to the primitive `Laminate` DataFrame (LFrame) which comprise IDs and Dimensional columns only. The `Laminate` gives the models author full access to its attributes. From here, `theories.handshakes()` searches within the models directory for a model (grey, dashed arrows) specified by the user at the time of instantiation, i.e. `Case.apply(*args, model=<model_name>)`.

A model is simply a module containing code that handles laminate theory calculations. The purpose of the model is to update the primitive LFame with LT calculations. `handshake()` automatically distinguishes whether the author

implemented a class-style or function-style model. The **most important hook method/function is ‘‘_use_model_()‘‘**, which must be present somewhere inside the model module and must return a tuple containing:

```
- the updated Laminate DataFrame with model data columns (a.k.a. `LaminateModel`)
- the `FeatureInput` with updated ``'Globals'`` key.  ``'Globals'`` is a dict of calculated constants, us
```

Finally, the `Laminate.LMFrame` attribute is updated with the new `LaminateModel` and `FeatureInput` (green arrow). However, if exceptions are raised, `Laminate._update_calculations()` handles them by reverting the LMFrame to a copy of LFrame, printing a warning and printing a minor traceback informing the author to refactor the code. This is commom for Laminates with `p`=1, which detects an INDET in middle layers and must revert to LFrame. The `handshake()` method for more details on Exceptions.

Custom Models

Sometimes Classical Laminate Theory needs to be modified to fit a specific set of constraints or boundary conditions. The LamAna package has powerful, extensible options for integrating user user-defined (authored) implementations of their own custom laminate theory models.

A library of these custom models, tests and pre-defined defaults are stored in the `models` directory (sub-package). Code for calculations, related exceptions, FeatureInput variables and defaults are stored in a Models module. `theories` then merges the model calculations with the passed in `Laminate` to calculate data columns in the `LaminateModel` object. This exchange is possbile since the `theories` module “handshakes” with the `constructs` module, and the selected model from the `models` sub-package.

2.9.5 Core Module: `output_`

A summary of `output` objects

Object	Purpose
`SinglePlot`	Stress distribution for a single geometry
`MultiPlot`	Stress distributions for a multiple geometries
`HalfPlot`	Partial plot of either compression or tension side
`QuarterPlot`	Partial halfplot excluding side without data
`PanelPlot`	A series of subplots side-by-side
`RatioPlot`	Ratio thickness plot; prinicipal stress vs. ratio
`PredictPlot`	Plot of experimental failure load or stress vs. middle layer princ. stress

Note: Development is beta for this module, therefore these objects are not yet implemented. The majority of plotting options are handled by temporary private functions called _distribplot() and _multiplot().

The `utils.tools.export()` function is used to save regular or temporary file versions of .xslx or .csv files. Files are automatically stored in the default export folder. More details are shown in the Demonstrations file.

2.10 Advanced Installation

This document details installation philosophies and approaches for maintaining reproducible packages.

2.10.1 Philosphophy on Reproducibility

Problem: The same package can behave differently based on:

1. the dependency environment and versions
2. third-party updates to dependencies
3. operating system (OS)

The first is and second items are prevented with pinned dependencies, one method for repeatable packages. The third item is prevented by through continuous integration, specifically with Travis and Appveyor (for Linux and Windows systems respectively). We will discuss proposals for the first two items only.

Proposal: We endorse freezing of dependencies at the start and end of the delevopment release cycle.

- *Start*: freeze the conda enviroment in an `environment.yaml` file
- *End*: freeze all dependencies in `dev_requirement.txt` and critical dependencies in `requirements.txt` files.

```
In [ ]: # TODO: Add diagram of trends in typical release cycle; show start and end freezing
```

2.10.2 How Reprodicibility Is Approached in LamAna

If packages work flawlessly, reproducible environments are generally not necessary for successful package use. Reproducible enviroments do become important when dependencies conflict with the package due to bugged patches or API changes in sub-dependencies. LamAna support either a "hands-off" or "hands-off" approach to versioning dependencies.

Hands-off Approach

By default, LamAna (and most Pythonic packages) assume that dependencies are coded with minimal API changes that intentionally break code. For example, sub-dependencies may require non-pythonic extensions to build correctly such as C/C++ compiliers. If so, warnings are commonly issued to the users. With this in mind, users can simply:

```
$ pip install lamana
```

This command searches for dependencies in the `install_requires` key of the `setup.py` file. Dependencies intentionally unpinned here, which means a user will download the latest version of every dependency listed.

Hands-on Approach

In the case where a dependency change breaks the package, the user is empowered to recreate a the dependency environment in which the release was oringially developed and known to work. The recreated environment installs pinned dependencies from a frozen `requirements.txt` file. This file represents the last list of known dependencies to a work with package correctly.

```
$ pip install -r </path/to/requirements.txt>

$ pip install lamana                          # source
```

Locating this file is not hard. Each release is shipped with this a requirements.txt file. The file simply needs to be download from the archives of the correct version of lamana hosted at GitHub releases or search on PyPI. Extract the file to your computer and run the commands.

It should be noted that installing pinned dependencies will change the current environement by upgrading or more likely downgrading existing packages to versions assigned in the requirements file. A developement environment is recommended for testing installations.

Installing from wheels (optional)

Sometimes installing from source is slow. You can force the latter installation method to install with faster binaries.

```
$ pip install lamana --use-wheel            # binary
```

2.10.3 Creating a developer environment with `conda`

The latter methods can be very slow, especially when intalling dependencies that rely on C extensions (numpy, pandas). Anaconda serves as the most consistent option for building dependencies and sub-dependencies. Here is a supporting rationale for using `conda` in travis. The following creates a fresh conda environment with critical dependencies that trigger installation of sub-dependencies required for LamAna.

```
$ git clone -b <branch name> https://github.com/par2/lamana
$ conda create -n <testenv name> pip nose numpy matplotlib pandas
$ source activate <testenv name>       # exclude source for Windows
$ pip install -r dev_requirements.txt
$ pip install .                        # within lamana directory
```

The first command downloads the repo from a spefic branch using git. The second command creates a reproducbile virtual environment using `conda` where therein, isolated versions of pip and nose are installed. Specific dependencies of the latest versions are downloaded within this environment which contain a necessary backend of sub-dependencies that are difficult to install manually. The environment is activated in the next command. Once the conda build is setup, `pip` will downgrade the existing versions to the pinned versions found in the requirments.txt file. Afterwards, the package is finally installed mimicking the original release environment.

The latter installation method should work fine. To check, the following command should be able to run without errors:

```
$ nosetests
```

Now, you should be able to run include jupyter notebook Demos.

Installing dependencies from source

In the absence of Anaconda, installing the three major dependendencies from source can be tedious and arduous, specifically `numpy`, `pandas` and `matplotlib`. Here are some reasons and tips 1, 2 for installing dependencies if they are not setup on your system.

On Debian-based systems, install the following pre-requisites.

```
$ apt-get install build-essential python3-dev
```

On Windows systems, be certain to install the appropriate Visual Studio C-compilers.

Note: Installing dependencies on windows can be troublesomes. See the installation guide for matplotlib. Try this or this for issues installing matplotlib. Future developments will work towards OS agnostiscism with continuous Integration on Linux, OS and Windows using Travis and Appveyor.

Important: If issues still arise, ensure the following requisites are satisfied:

- the conda environment is properly set up with dependencies and compiled sub-dependencies e.g. C-extensions (see above)

- the appropriate compiler libraries are installed on your specific OS, i.e. gcc for Linux, Visual Studio for Windows. With conda, this should not be necessary.
- sufficient memory is available to compile C-extensions, e.g. 0.5-1 GB minimum
- the appropriate LamAna version, compatible python version and dependency versions are installed according to requirements.txt (see the *Dependencies chart*)

Dependencies

- Mandatory Dependencies
  - numpy
  - matplotlib
  - pandas
- Recommended Dependencies
  - notebook

The following table shows a chart of tested build build compatible with LamAna:

lamana	python	dependency	OS
0.4.8	2.7.6, 2.7.10, 3.3, 3.4, 3.5, 3.5.1	numpy==1.10.1, pandas==0.16.2, matplotlib==1.5.0	linux, local win(?)
0.4.9	2.7, 3.3, 3.4, 3.5, 3.5.1	conda==3.19.0, numpy==1.10.1, pandas==0.16.2, matplotlib==1.4.3	linux, win(?)
0.4.10	2.7, 2.7.11, 3.3, 3.4, 3.5, 3.5.1	conda==3.19.0, numpy==1.10.2, pandas==0.17.1, matplotlib==1.5.1	linux
0.4.10	2.7 (x32, x64), 3.4 (x32), 3.5 (x32, x64)	conda==3.19.0, numpy==1.10.2, pandas==0.17.1, matplotlib==1.5.1	win

2.11 Contributing Code

First, thank you for your interests in improving LamAna. At the moment, you can contribute to the LamAna community as an Author or Developer in several ways.

2.11.1 As an Author

to the Models Library

So you have worked on your custom model and you would like to share it with others...

You can sumbit your custom model as an extension to the current models library as a pull request on GitHub. As an extension, other users can access your model with ease. With further review and community acceptance, favorable models will be accepted into the core LamAna models library. Please do the following:

1. create Python files subclassed from `theories.Model`
2. write tests
3. document your code
4. cite academic references

5. include supporting validation or FEA data (preferred)

The LamAna Project welcomes open contributions to the offical `models` library. It is envisioned that a stable source of reliable laminate anaylsis models will be useful to other users, similar to the way R package libraries are maintained.

2.11.2 As a Developer

to LPEPs

If you are not interested in writing code, but would like to propose an idea for enchancing LamAna, you can submit an LPEP for review.

1. Please draft your proposals in a similar format to existing LPEPs as jupyter notebooks
2. Submit a pull request on GitHub.

The LPEP submission and content guidelines closely follow PEP 0001.

to Project Code

If you would like to improve the latest version, please do the following:

1. `git clone` the `develop` branch
2. use gitflow to make feature branches
3. since GitHub squashes local commits, **add detailed commit history** to the feature branch merge commits
4. write tests for your enhancement
5. modify the code with appropriate comments
6. successfully run tests
7. write documentation in a jupyter notebook
8. submit your test, code and documentation as a pull requests on GitHub.

The latter steps pertain to adding or enchancing Feature modules and improving core modules. For starters, you can look to the demonstrations or gallery as motivation for improving LamAna's *functionality*, *clarity* or *optimization*.

Thanks for your contributions. You are helping to improve the LamAna community!

```
In [ ]: # TODO: add gitflow, workflow diagram
```

2.11.3 Resources

If you are new to developing, here are some resources to get started on this project:

- git: default version control
- GitHub: git host opensource projects
- gitflow-avh: a git extension used to ease development workflow
- atom: an great, open source text editor [using linter, flake8, minimap, pygments, colorpicker]

The following packages are available in `conda` or `pip`:

- virtualenv: create reproducible, isolated virtual enviroments
- vituralenvwrapper: simplify virtualenv creation; see also virtualenvwrapper-win for Windows

- nose: test code; see also `pytest` alternative.

The following are Sphinx extensions helpful in documenting code:

- autodoc: auto generate api docs
- autosummary: make a custom API reference, summary table
- nbsphinx: auto generate rst files from jupyter notebook files
- numpydoc: auto format docs using numpy-style docstrings
- napoleon: numpy and google-style doctrings
- viewcode: link api references and see implementation of the code directly in source.

The following are useful web-based tools:

- Travis: test builds on linux/Mac systems and dependency versions
- readthedocs: automate doc builds (in a travis-like way)
- TestPyPI: Test code before officially hosting to PyPI
- PyPI: host Python projects

2.12 Testing Code

A guide for testing code prior to submitting pull request.

Testing LamAna occurs in two flavors:

1. Unit-testing with nose
2. Regression testing of API with Jupyter or runipy

2.12.1 Testing code with `nose`

The current testing utility is `nose`. From the root directory, you can test all files prepended with "test_" by running:

```
$ nosetests
```

There are three types of tests contained in the source `lamana` directory:

1. module tests: normal test files located in the "./tests" directory
2. model tests: test files specific for custom models, located in "./models/tests"
3. controls: .csv files located "./tests/controls_LT"

Models tests are separated to support an extensibile design for author contributions. This design enables authors to create models and tests together with a single pull request to the standard module directory.

Tests for the `utils` module writes and removes temporary files in a root directory called "export". If this directory does not exist, one will be created. These test check that writing and reading of files are consistent. Temporary files are prefixed with "temp", but should be removed by these test functions.

Note: The locations for tests may change in future releases.

Control files

LamAna maintains .csv files with expected data for different lamanate configurations. These files are tested with the `test_controls` module. This module reads each control file and parses information such as layer numbers, number of points per layer and geometry. Control files are named by these variables.

Controls files can be created manually, but it may be simpler to make and then edit a starter file. This process can be expedited for multiple files by passing LaminateModels into the `utils.tools.write_csv()` function. This function will create a csv file for every LaminateModel, which can be altered as desired and tested by copying into the "lamana/tests/controls_LT" folder.

2.12.2 Coverage

We use the following tools and commands to assess test coverage. `nose-cov` helps to combine coverage reports for sub-packages automatically. The remaining flags will report missing lines for the source directory.

```
$ pip install coverage, nose-cov
$ nosetests --with-cov --cov lamana
```

or

```
$ nosetests --with-cov --cov-report term-missing --cov lamana
```

LamAna aims for the highest "reasonable" coverage for core modules. A separate ideology must be developed for testing `output_` as plots are tricky to test fully.

2.12.3 Regression Testing

Prior to a release, it is fitting to test API regression tests on any demonstration notebooks in a development virtual environment and release branch (see docs/demo.ipynb). A simple way to validate API is to run demo.ipynb notebook in Python 3 and Python 2 kernels.

Another simple way to validate notebook cells without errors is to use the `'runipy` tool <https://github.com/paulgb/runipy>'__. This tool will run notebook cells in the command prompt and halt if errors are found. The command is simple to apply to a notebook:

```
$ pip install runipy
$ runipy <demo>.ipynb
```

If no errors were found, your tested API works, and you can advance in the release workflow.

Note: There are occassions where *runipy* throws an error for cells that run normally in the Jupyter notebook. It is then reasonable to simply run all cells in a notebook as usual.

2.13 Documenting Code

A general guide for authors and developers.

2.13.1 As a Developer

Setup

This section is related to maintainers of the project repository. Documentation for the LamAna package uses Sphinx, readthedocs and the nbsphinx extension. These three tools give a simple documentating experience by directly rendering jupyter notebooks. Some details are written here for reference.

The critical docs file structure is present below:

```
docs
|
|+ _archive
|+ _images
|- conf.py
|- Makefile
|- make.bat
|- index.rst
|- *.ipynb
|- ...
```

Jupyter notebooks dwell in the "docs" folder. By adding the `nbsphinx` extension to `conf.py`, notebooks extant in this folder are automatically converted to html from ipynb files by running the `make html` build command. This setup has several benefits:

1. Edit notebooks directly; no copies or moves required from twin files.
2. Notebooks are rendered as-is.
3. Timestamps and command line info can be "hidden" prior to rendering (edit the metadata).
4. Images can pull from a single directory

Only the index.rst file uses the native reST format.

Images

Images for the entire packages are currently reserved in the `./docs/_images` folder. This placement eases root access to images for any notebook. There is an "_archive" folder within used to store older versions of image files. The README in the docs folder reminds not to enumerate updated image files, otherwise notebook links will break. Rather, copy and enumerate the old file and archive for reference.

Important: Do not add spaces filenames of images. They do not render with nbsphinx.

Note: A specical "Doc builder" file is retained in the "_notebook" folder to assist in building docs.

API Docs

The sphinx.ext.autosummary documentation is followed closely to build the API Reference page. `cd` into the root package and run this code to update the API reference.

```
$ sphinx-api -f -o ./docs/generated .
```

You can optionally clean the build and make a new one afterwards.

```
$ make clean

$ make html
```

This will create api docs or "stubs" for all modules found in your package and store them in the "generated" folder. This folder appears to be important for linking object rendered by autosummary to their appropriate api doc. (Projects like seaborn and pandas seem to gitignore this folder and its contents). It was obeserved here that without this folder versioned, the api docs will break links.

Note: The alternative option is to figure out how to invoke the latter command on readthedocs, but at this time, that option has not been successfully executed.

The sphinx extension `viewcode` links to the exact code where each module is documented in the API reference. The alternative is to use the "View in GitHub" link on each page in readthedocs (not observed locally).

Note: The API Reference currently generates long list of WARNINGS when run, related to the location of the files needed to link to the reference documentation. The links seem to work despite these warnings. Alternatives are welcome.

2.14 Demonstration

The following demonstration includes basic and intermediate uses of the LamAna Project library. It is intended to exhaustively reference all API features, therefore some advandced demonstrations will favor technical detail.

2.15 Tutorial: Basic

2.15.1 User Input Startup

```
In [4]: #-----------------------------------------------------------------------------
        import pandas as pd

        import lamana as la
        #import LamAna as la

        %matplotlib inline
        #%matplotlib nbagg
        # PARAMETERS ------------------------------------------------------------------
        # Build dicts of geometric and material parameters
        load_params = {'R' : 12e-3,                                    # specimen radius
                       'a' : 7.5e-3,                                   # support ring radius
                       'r' : 2e-4,                                     # radial distance from c
                       'P_a' : 1,                                      # applied load
                       'p' : 5,                                        # points/layer
                       }

        # Quick Form: a dict of lists
        mat_props = {'HA' : [5.2e10, 0.25],
                     'PSu' : [2.7e9, 0.33],
```

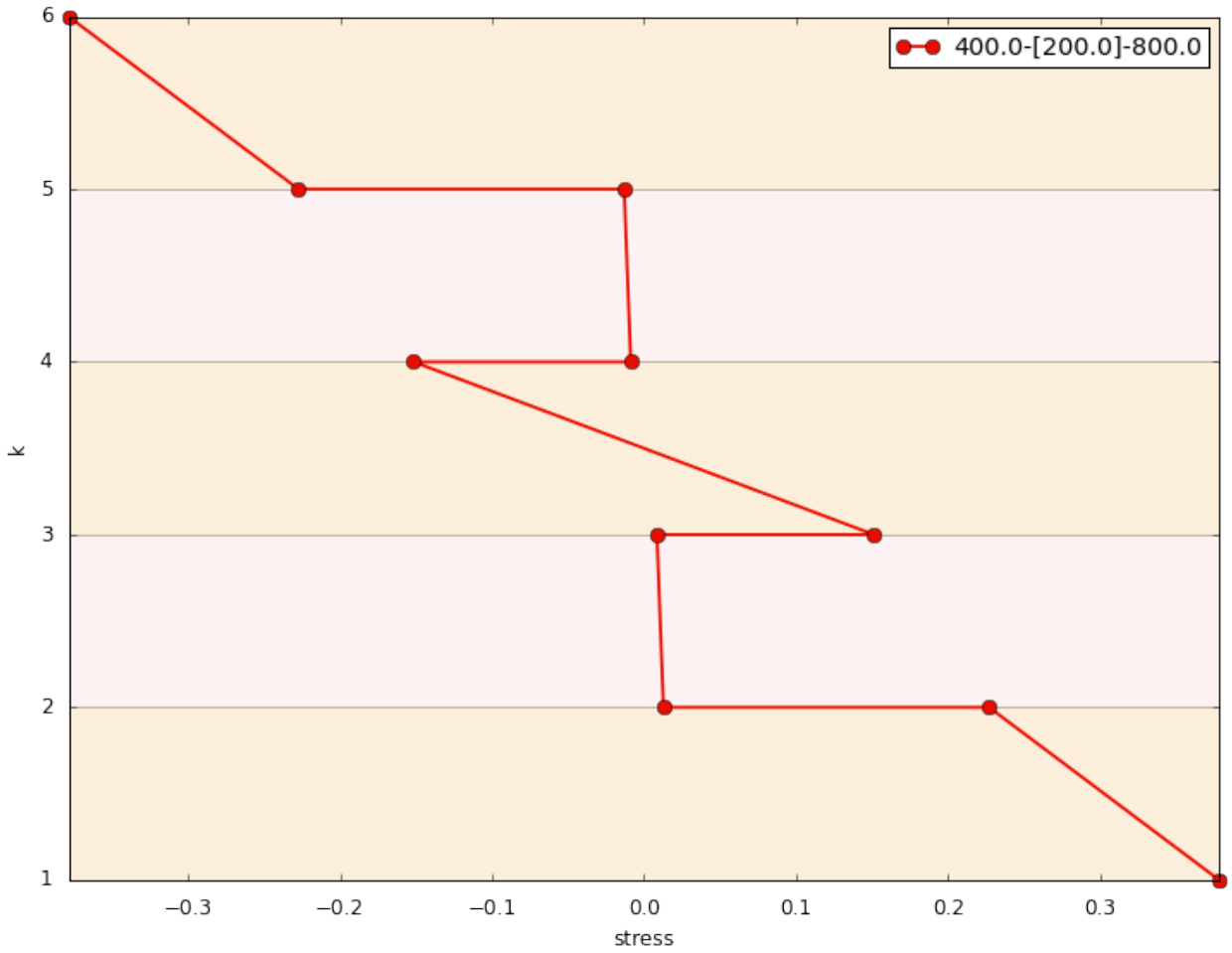
```
                }

        # Standard Form: a dict of dicts
        # mat_props = {'Modulus': {'HA': 5.2e10, 'PSu': 2.7e9},
        #              'Poissons': {'HA': 0.25, 'PSu': 0.33}}


        # What geometries to test?
        # Make tuples of desired geometeries to analyze: outer - {inner...-....}_i - middle

        # Current Style
        g1 = ('0-0-2000')                                                    # Monolith
        g2 = ('1000-0-0')                                                    # Bilayer
        g3 = ('600-0-800')                                                   # Trilayer
        g4 = ('500-500-0')                                                   # 4-ply
        g5 = ('400-200-800')                                                 # Short-hand; <= 5-ply
        g6 = ('400-200-400S')                                                # Symmetric
        g7 = ('400-[200]-800')                                               # General convention; 5
        g8 = ('400-[100,100]-800')                                           # General convention; 7
        g9 = ('400-[100,100]-400S')                                          # General and Symmetric

        '''Add to test set'''
        g13 = ('400-[150,50]-800')                                           # Dissimilar inner_is
        g14 = ('400-[25,125,50]-800')


        geos_most = [g1, g2, g3, g4, g5]
        geos_special = [g6, g7, g8, g9]
        geos_full = [g1, g2, g3, g4, g5, g6, g7, g8, g9]
        geos_dissimilar = [g13, g14]

        # Future Style
        #geos1 = ((400-400-400),(400-200-800),(400-350-500))         # same total thickness
        #geos2 = ((400-400-400), (400-500-1600), (400-200-800))      # same outer thickness
```

In [5]:
```
        #import pandas as pd
        pd.set_option('display.max_columns', 10)
        pd.set_option('precision', 4)
```

2.15.2 Goal: Generate a Plot in 3 Lines of Code

In [6]:
```
        case1 = la.distributions.Case(load_params, mat_props)       # instantiate a User Inp
        case1.apply(['400-200-800'])
        case1.plot()
```

```
Converting mat_props to Standard Form.
User input geometries have been converted and set to Case.
```



That's it! The rest of this demonstration showcases API functionality of the LamAna project.

2.15.3 Calling Case attributes

Passed in arguments are acessible, but can be displayed as pandas Series and DataFrames.

```
In [7]: # Original
        case1.load_params

Out[7]: {'P_a': 1, 'R': 0.012, 'a': 0.0075, 'p': 5, 'r': 0.0002}

In [8]: # Series View
        case1.parameters

Out[8]: P_a    1.000e+00
        R      1.200e-02
        a      7.500e-03
        p      5.000e+00
        r      2.000e-04
        dtype: float64

In [9]: # Original
        case1.mat_props

Out[9]: defaultdict(<class 'dict'>, {'Poissons': {'PSu': 0.33, 'HA': 0.25}, 'Modulus': {'PS
```

```
In [10]: # DataFrame View
         case1.properties
```

```
In [11]: # Equivalent Standard Form
         case1.properties.to_dict()
```

```
Out[11]: {'Modulus': {'HA': 52000000000.0, 'PSu': 2700000000.0},
          'Poissons': {'HA': 0.25, 'PSu': 0.33000000000000002}}
```

Reset material order. Changes are relfected in the properties view and stacking order.

```
In [12]: case1.materials = ['PSu', 'HA']
         case1.properties
```

```
Overriding materials order...
```

Serial resets

```
In [13]: case1.materials = ['PSu', 'HA', 'HA']
         case1.properties
```

```
Overriding materials order...
```

```
In [14]: case1.materials                                          # get reorderd list of ma
```

```
Getting materials...
```

```
Out[14]: ['PSu', 'HA', 'HA']
```

```
In [15]: case1._materials
```

```
Out[15]: ['PSu', 'HA', 'HA']
```

```
In [16]: case1.apply(geos_full)
```

```
User input geometries have been converted and set to Case.
```

```
In [17]: case1.snapshots[-1]
```

```
Accessing snapshot method.
```

```
In [18]: '''Need to bypass pandas abc ordering of indicies.'''
```

```
Out[18]: 'Need to bypass pandas abc ordering of indicies.'
```

Reset the parameters

```
In [19]: mat_props2 = {'HA' : [5.3e10, 0.25],
                       'PSu' : [2.8e9, 0.33],
                      }
```

```
In [20]: case1 = la.distributions.Case(load_params, mat_props2)
         case1.properties
```

```
Converting mat_props to Standard Form.
```

2.15.4 `apply()` Geometries and LaminateModels

Construct a laminate using geometric, matrial paramaters and geometries.

```
In [21]: case2 = la.distributions.Case(load_params, mat_props)
         case2.apply(geos_full)                                  # default model Wilson_
```

```
Converting mat_props to Standard Form.
User input geometries have been converted and set to Case.
```

Access the user input geometries

```
In [22]: case2.Geometries                                                    # using an attribute, _
```

```
Out[22]: [Geometry object (0.0-[0.0]-2000.0),
          Geometry object (1000.0-[0.0]-0.0),
          Geometry object (600.0-[0.0]-800.0),
          Geometry object (500.0-[500.0]-0.0),
          Geometry object (400.0-[200.0]-800.0),
          Geometry object (400.0-[200.0]-400.0S),
          Geometry object (400.0-[200.0]-800.0),
          Geometry object (400.0-[100.0,100.0]-800.0),
          Geometry object (400.0-[100.0,100.0]-400.0S)]
```

```
In [23]: print(case2.Geometries)                                             # uses __str__
```

```
[Geometry object (0.0-[0.0]-2000.0), Geometry object (1000.0-[0.0]-0.0), Geometry object (
```

```
In [24]: case2.Geometries[0]                                                 # indexing
```

```
Out[24]: Geometry object (0.0-[0.0]-2000.0)
```

We can compare Geometry objects with builtin Python operators. This process directly compares GeometryTuples in the `Geometry` class.

```
In [25]: bilayer = case2.Geometries[1]                                       # (1000.0-[0.0]-0.0)
         trilayer = case2.Geometries[2]                                      # (600.0-[0.0]-800.0)
         #bilayer == trilayer
         bilayer != trilayer
```

```
Out[25]: True
```

Get all thicknesses for selected layers.

```
In [26]: case2.middle
```

```
Out[26]: [2000.0, 0.0, 800.0, 0.0, 800.0, 400.0, 800.0, 800.0, 400.0]
```

```
In [27]: case2.inner
```

```
Out[27]: [[0.0],
          [0.0],
          [0.0],
          [500.0],
          [200.0],
          [200.0],
          [200.0],
          [100.0, 100.0],
          [100.0, 100.0]]
```

```
In [28]: case2.inner[-1]
```

```
Out[28]: [100.0, 100.0]
```

```
In [29]: case2.inner[-1][0]                                                  # List indexing allowe
```

```
Out[29]: 100.0
```

```
In [30]: [first[0] for first in case2.inner]                                 # iterate
```

```
Out[30]: [0.0, 0.0, 0.0, 500.0, 200.0, 200.0, 200.0, 100.0, 100.0]
```

```
In [31]: case2.outer
```

```
Out[31]: [0.0, 1000.0, 600.0, 500.0, 400.0, 400.0, 400.0, 400.0, 400.0]
```

A general and very important object is the LaminateModel.

```
In [32]: case2.LMs
```

```
Out[32]: [<lamana LaminateModel object (0.0-[0.0]-2000.0), p=5>,
          <lamana LaminateModel object (1000.0-[0.0]-0.0), p=5>,
          <lamana LaminateModel object (600.0-[0.0]-800.0), p=5>,
          <lamana LaminateModel object (500.0-[500.0]-0.0), p=5>,
          <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>,
          <lamana LaminateModel object (400.0-[200.0]-400.0S), p=5>,
          <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>,
          <lamana LaminateModel object (400.0-[100.0,100.0]-800.0), p=5>,
          <lamana LaminateModel object (400.0-[100.0,100.0]-400.0S), p=5>]
```

Sometimes might you want to throw in a bunch of geometry strings from different groups. If there are repeated strings in different groups (set intersections), you can tell `Case` to only give a unique result.

For instane, here we combine two groups of geometry strings, 5-plys and odd-plys. Clearly these two groups overlap, and there are some repeated geometries (one with different conventions). Using the `unique` keyword, Case only operates on a unique set of `Geometry` objects (independent of convention), resulting in a unique set of LaminateModels.

```
In [33]: fiveplys = ['400-[200]-800', '350-400-500', '200-100-1400']
         oddplys = ['400-200-800', '350-400-500', '400.0-[100.0,100.0]-800.0']
         mix = fiveplys + oddplys
         mix
```

```
Out[33]: ['400-[200]-800',
          '350-400-500',
          '200-100-1400',
          '400-200-800',
          '350-400-500',
          '400.0-[100.0,100.0]-800.0']
```

```
In [34]: # Non-unique, repeated 5-plys
         case_ = la.distributions.Case(load_params, mat_props)
         case_.apply(mix)
         case_.LMs
```

```
Converting mat_props to Standard Form.
User input geometries have been converted and set to Case.
```

```
Out[34]: [<lamana LaminateModel object (400.0-[200.0]-800.0), p=5>,
          <lamana LaminateModel object (350.0-[400.0]-500.0), p=5>,
          <lamana LaminateModel object (200.0-[100.0]-1400.0), p=5>,
          <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>,
          <lamana LaminateModel object (350.0-[400.0]-500.0), p=5>,
          <lamana LaminateModel object (400.0-[100.0,100.0]-800.0), p=5>]
```

```
In [35]: # Unique
         case_ = la.distributions.Case(load_params, mat_props)
         case_.apply(mix, unique=True)
         case_.LMs
```

```
Converting mat_props to Standard Form.
User input geometries have been converted and set to Case.
```

```
Out[35]: [<lamana LaminateModel object (400.0-[200.0]-800.0), p=5>,
          <lamana LaminateModel object (350.0-[400.0]-500.0), p=5>,
```

```
<lamana LaminateModel object (200.0-[100.0]-1400.0), p=5>,
<lamana LaminateModel object (400.0-[100.0,100.0]-800.0), p=5>]
```

2.15.5 DataFrame Access

You can get a quick view of the stack using the `snapshot` method. This gives access to a `Construct` - a DataFrame converted stack.

```
In [36]: case2.snapshots[-1]
```

```
Accessing snapshot method.
```

We can easily view entire laminate DataFrames using the `frames` attribute. This gives access to `LaminateModels` (DataFrame) objects, which extends the stack view so that laminate theory is applied to each row.

```
In [37]: '''Consider head command for frames list'''
```

```
Out[37]: 'Consider head command for frames list'
```

```
In [38]: #case2.frames
```

```
In [39]: ##with pd.set_option('display.max_columns', None):           # display all columns
         ##    case2.frames[5]
```

```
In [40]: case2.frames[5].head()
```

```
Accessing frames method.
```

```
In [41]: '''Extend laminate attributes'''
```

```
Out[41]: 'Extend laminate attributes'
```

```
In [42]: case3 = la.distributions.Case(load_params, mat_props)
         case3.apply(geos_dissimilar)
         #case3.frames
```

```
Converting mat_props to Standard Form.
User input geometries have been converted and set to Case.
```

NOTE, for even plies, the material is set alternate for each layer. Thus outers layers may be different materials.

```
In [43]: case4 = la.distributions.Case(load_params, mat_props)
         case4.apply(['400-[100,100,100]-0'])
         case4.frames[0][['layer', 'matl', 'type']]
         ;
```

```
Converting mat_props to Standard Form.
User input geometries have been converted and set to Case.
Accessing frames method.
```

```
Out[43]: ''
```

```
In [44]: '''Add functionality to customize material type.'''
```

```
Out[44]: 'Add functionality to customize material type.'
```

Totaling

The `distributions.Case` class has useful properties available for totaling specific layers for a group of laminates as lists. As these properties return lists, these results can be **sliced** and **iterated**.

```
In [45]: '''Show Geometry first then case use.'''
```

```
Out[45]: 'Show Geometry first then case use.'
```

`.total` property

```
In [46]: case2.total
Out[46]: [2000.0, 2000.0, 2000.0, 2000.0, 2000.0, 2000.0, 2000.0, 2000.0, 2000.0]
In [47]: case2.total_middle
Out[47]: [2000.0, 0.0, 800.0, 0.0, 800.0, 800.0, 800.0, 800.0, 800.0]
In [48]: case2.total_middle
Out[48]: [2000.0, 0.0, 800.0, 0.0, 800.0, 800.0, 800.0, 800.0, 800.0]
In [49]: case2.total_inner_i
Out[49]: [[0.0],
          [0.0],
          [0.0],
          [1000.0],
          [400.0],
          [400.0],
          [400.0],
          [200.0, 200.0],
          [200.0, 200.0]]
In [50]: case2.total_outer
Out[50]: [0.0, 2000.0, 1200.0, 1000.0, 800.0, 800.0, 800.0, 800.0, 800.0]
In [51]: case2.total_outer[4:-1]                                    # slicing
Out[51]: [800.0, 800.0, 800.0, 800.0]
In [52]: [inner_i[-1]/2.0 for inner_i in case2.total_inner_i]       # iterate
Out[52]: [0.0, 0.0, 0.0, 500.0, 200.0, 200.0, 200.0, 100.0, 100.0]
```

`Geometry` Totals

The total attribute used in Case actually dervive from attributes for Geometry objects individually. On Geometry objects, they return specific thicknesses instead of lists of thicknesses.

```
In [53]: G1 = case2.Geometries[-1]
         G1
Out[53]: Geometry object (400.0-[100.0,100.0]-400.0S)
In [54]: G1.total                                                   # laminate thickness (t
Out[54]: 2000.0
In [55]: G1.total_inner_i                                           # inner_i laminae
Out[55]: [200.0, 200.0]
In [56]: G1.total_inner_i[0]                                        # inner_i lamina pair
Out[56]: 200.0
In [57]: sum(G1.total_inner_i)                                      # inner total
Out[57]: 400.0
In [58]: G1.total_inner                                             # inner total
```

```
Out[58]: 400.0
```

2.15.6 `LaminateModel` Attributes

Access the LaminateModel object directly using the `LMs` attribute.

```
In [59]: case2.LMs[5].Middle
```

```
In [60]: case2.LMs[5].Inner_i
```

Laminates are assumed mirrored at the neutral axis, but dissimilar inner_i thicknesses are allowed.

```
In [61]: case2.LMs[5].tensile
```

Separate from the case attributes, Laminates have useful attributes also, such as `nplies`, `p` and its own `total`.

```
In [62]: LM = case2.LMs[4]
         LM.LMFrame.tail(7)
```

Often the extreme stress values (those at the interfaces) are most important. This is equivalent to p=2.

```
In [63]: LM.extrema
```

```
In [64]: LM.p                                                    # number of rows per gr
```

```
Out[64]: 5
```

```
In [65]: LM.nplies                                               # number of plies
```

```
Out[65]: 5
```

```
In [66]: LM.total                                                # total laminate thickn
```

```
Out[66]: 0.002
```

```
In [67]: LM.Geometry
```

```
Out[67]: Geometry object (400.0-[200.0]-800.0)
```

```
In [68]: '''Overload the min and max special methods.'''
```

```
Out[68]: 'Overload the min and max special methods.'
```

```
In [69]: LM.max_stress                                          # max interfacial failur
```

```
Out[69]: 0     0.379
         5     0.013
         10    0.151
         14   -0.151
         19   -0.013
         24   -0.379
         Name: stress_f (MPa/N), dtype: float64
```

NOTE: this feature gives a different result for p=1 since a single middle cannot report two interfacial values; INDET.

```
In [70]: LM.min_stress
```

```
Out[70]: 4     0.227
         9     0.009
         15   -0.009
         20   -0.227
         Name: stress_f (MPa/N), dtype: float64
```

```
In [71]: '''Redo tp return series of bool an index for has_attrs'''
```

```
Out[71]: 'Redo tp return series of bool an index for has_attrs'
```

```
In [72]: LM.has_neutaxis
```

```
Out[72]: 0     False
         1     False
         2     False
         3     False
         4     False
         5     False
         6     False
         7     False
         8     False
         9     False
         10    False
         11    False
         12     True
         13    False
         14    False
         15    False
         16    False
         17    False
         18    False
         19    False
         20    False
         21    False
         22    False
         23    False
         24    False
         Name: label, dtype: bool
```

```
In [73]: LM.has_discont
```

```
Out[73]: 0     False
         1     False
         2     False
         3     False
         4      True
         5     False
         6     False
         7     False
         8     False
         9      True
         10    False
         11    False
         12    False
         13    False
         14    False
         15     True
         16    False
         17    False
         18    False
         19    False
         20     True
         21    False
         22    False
         23    False
```

```
         24    False
         Name: label, dtype: bool
```

```
In [74]: LM.is_special
```

```
Out[74]: False
```

```
In [75]: LM.FeatureInput
```

```
Out[75]: {'Geometry': Geometry object (400.0-[200.0]-800.0),
          'Globals': {'D_11T': 31.664191802890315,
           'D_11p': 0.033700807714524279,
           'D_12T': 7.9406108505093584,
           'D_12n': -0.0084513446948124519,
           'K_r': 0.0034519261262397653,
           'K_t:': 0.0059650953251038216,
           'M_r': 0.15666895161350616,
           'M_t': 0.216290324549788,
           'v_eq ': 0.25077573114575868},
          'Materials': ['HA', 'PSu'],
          'Model': 'Wilson_LT',
          'Parameters': {'P_a': 1, 'R': 0.012, 'a': 0.0075, 'p': 5, 'r': 0.0002},
          'Properties': defaultdict(<class 'dict'>, {'Poissons': {'PSu': 0.33, 'HA': 0.25},
```

```
In [76]: '''Need to fix FeatureInput and Geometry inside LaminateModel'''
```

```
Out[76]: 'Need to fix FeatureInput and Geometry inside LaminateModel'
```

As with Geometry objects, we can compare LaminateModel objects also. [STRIKEOUT:This process directly compares two defining components of a LaminateModel object: the LM DataFrame (`LMFrame`) and FeatureInput. If either is False, the equality returns `False`.]

```
In [77]: case2 = la.distributions.Case(load_params, mat_props)
         case2.apply(geos_full)
```

```
Converting mat_props to Standard Form.
User input geometries have been converted and set to Case.
```

```
In [78]: bilayer_LM = case2.LMs[1]
         trilayer_LM = case2.LMs[2]
         trilayer_LM == trilayer_LM
         #bilayer_LM == trilayer_LM
```

```
Out[78]: True
```

```
In [79]: bilayer_LM != trilayer_LM
```

```
Out[79]: True
```

Use python and pandas native comparison tracebacks that to understand the errors directly by comparing FeatureInput dict and LaminateModel DataFrame.

```
In [80]: #bilayer_LM.FeatureInput == trilayer_LM.FeatureInput    # gives detailed traceback
```

```
In [81]: '''Fix FI DataFrame with dict.'''
```

```
Out[81]: 'Fix FI DataFrame with dict.'
```

```
In [82]: bilayer_LM.FeatureInput
```

```
Out[82]: {'Geometry': Geometry object (1000.0-[0.0]-0.0),
          'Globals': {'D_11T': 19.498876544595319,
           'D_11p': 0.054826177209184083,
```

```
 'D_12T': 4.9555181486053437,
 'D_12n': -0.013933731800259629,
 'K_r': 0.0055968142719747937,
 'K_t:': 0.009677945375294943,
 'M_r': 0.15709082448075087,
 'M_t': 0.21644417677735781,
 'v_eq ': 0.25414377783621128},
'Materials': ['HA', 'PSu'],
'Model': 'Wilson_LT',
'Parameters': {'P_a': 1, 'R': 0.012, 'a': 0.0075, 'p': 5, 'r': 0.0002},
'Properties': defaultdict(<class 'dict'>, {'Poissons': {'PSu': 0.33, 'HA': 0.25},
```

```
In [83]: #bilayer_LM.LMFrame == trilayer_LM.LMFrame                # gives detailed traceba
```

2.15.7 `plot()` LT Geometries

CAVEAT: it is recommended to use at least p=2 for calculating stress. Less than two points for odd plies is indeterminant in middle rows, which can raise exceptions.

```
In [84]: '''Find a way to remove all but interfacial points.'''
```

```
Out[84]: 'Find a way to remove all but interfacial points.'
```

We try to quickly plot simple stress distriubtions with native pandas methods. We have two variants for displaying distributions:

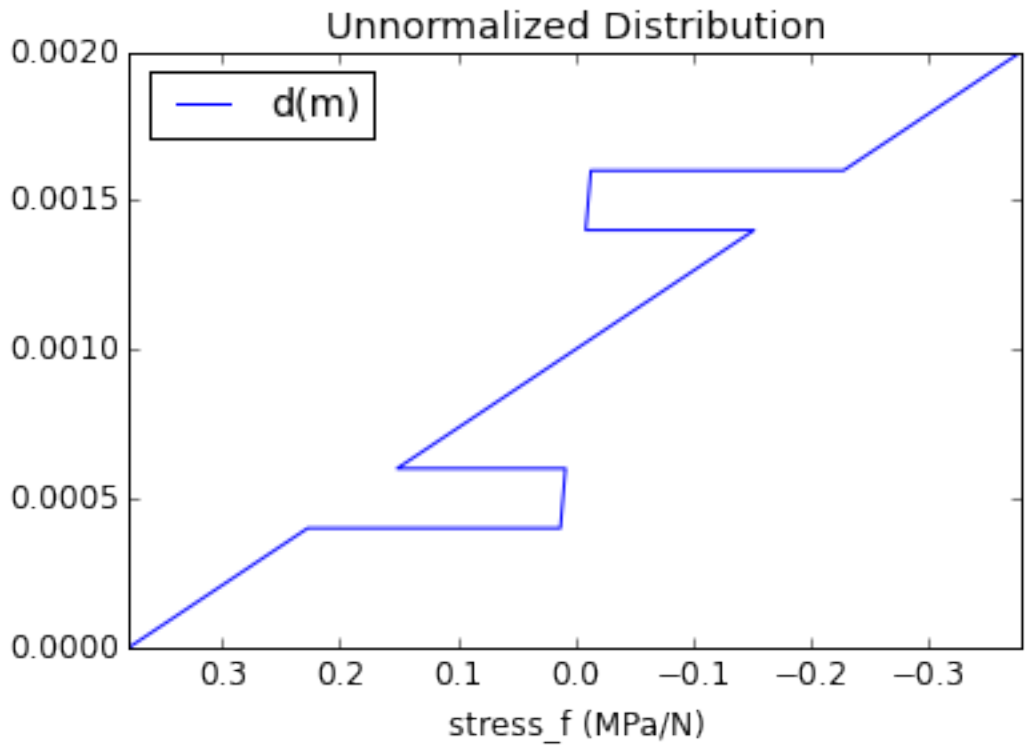
```
- Unnoormalized: plotted by the height (`d_`). Visaully: thicknesses vary, material slopes are consta
- Normalized: plotted by the relative fraction level (`k_`). Visually: thicknesses are constant, mate
```

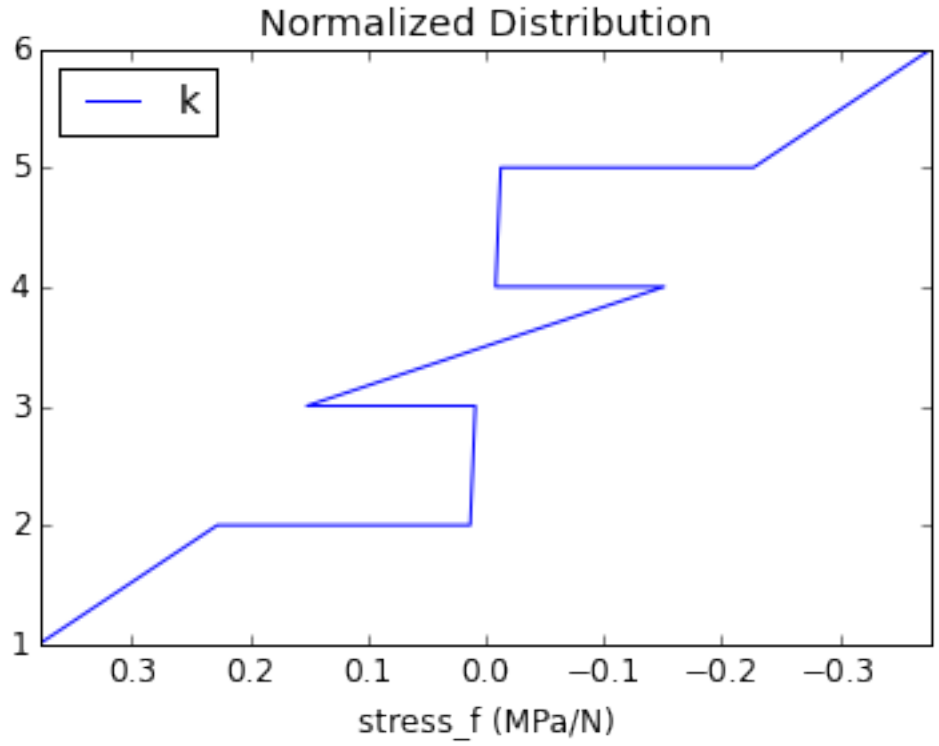
Here we plot with the nbagg matplotlib backend to generatre interactive figures. NOTE: for Normalized plots, slope can vary for a given material.

```
In [85]: from lamana.utils import tools as ut
         from lamana.models import Wilson_LT as wlt

         dft = wlt.Defaults()
         #%matplotlib nbagg

         # Quick plotting
         case4 = ut.laminator(dft.geos_standard)
         for case in case4.values():
             for LM in case.LMs:
                 df = LM.LMFrame

         df.plot(x='stress_f (MPa/N)', y='d(m)', title='Unnormalized Distribution')
         df.plot(x='stress_f (MPa/N)', y='k', title='Normalized Distribution')
```

```
Converting mat_props to Standard Form.
User input geometries have been converted and set to Case.
```

```
Out[85]: <matplotlib.axes._subplots.AxesSubplot at 0x9e43390>
```





While we get reasonable stress distribution plots rather simply, LamAna offers some plotting methods pertinent to laminates than assisting with visualization.

Demo - An example illustration of desired plotting of multiple geometries from `distributions`.

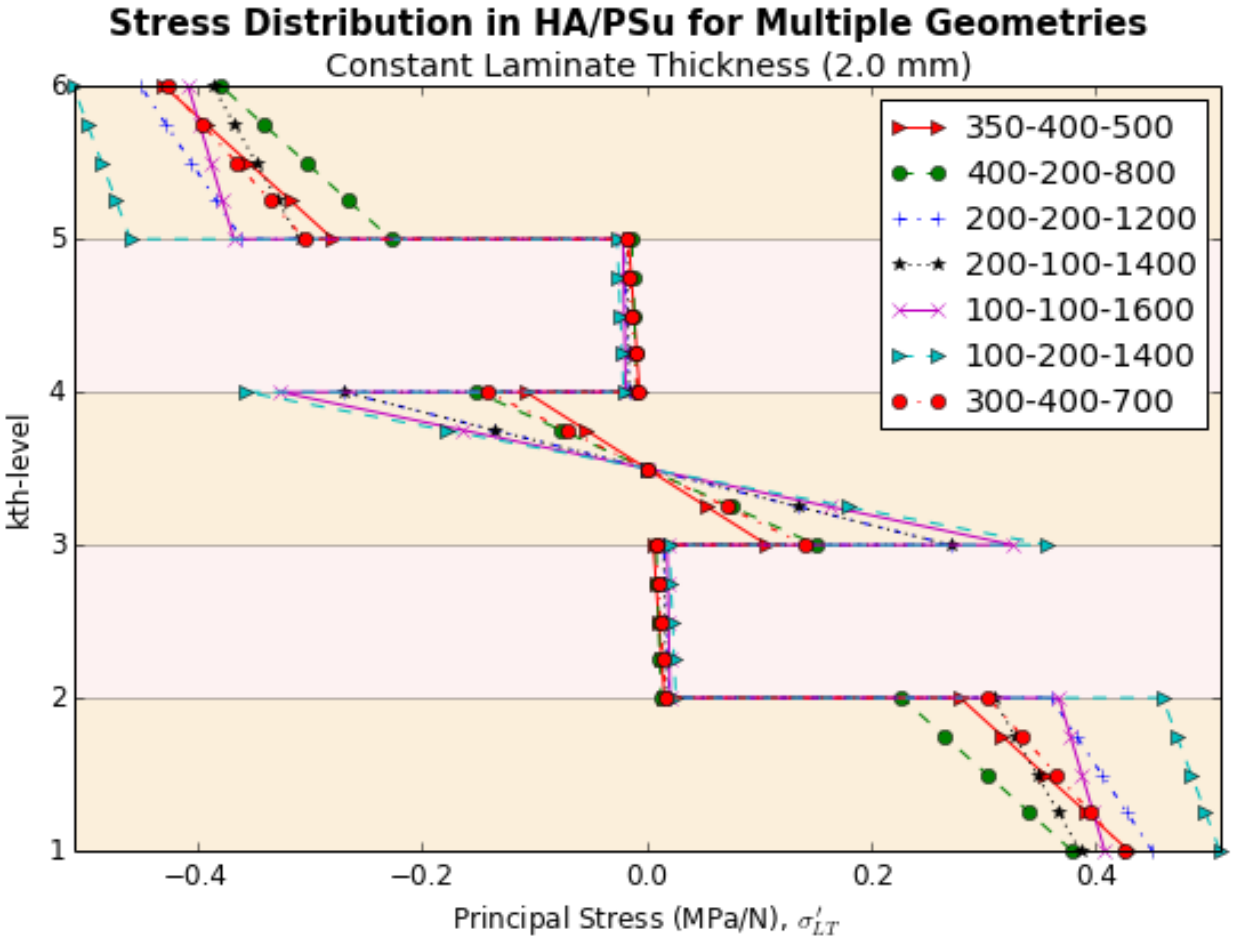

Stress Distribution in HA/PSu for Multiple Geometries
Constant Laminate Thickness (2.0 mm)
350-400-500
400-200-800
200-200-1200
200-100-1400
100-100-1600
100-200-1400
300-400-700
kth-level
1
2
3
4
5
6
−0.4
−0.2
0.0
0.2
0.4
Principal Stress (MPa/N), σ'_{LT}

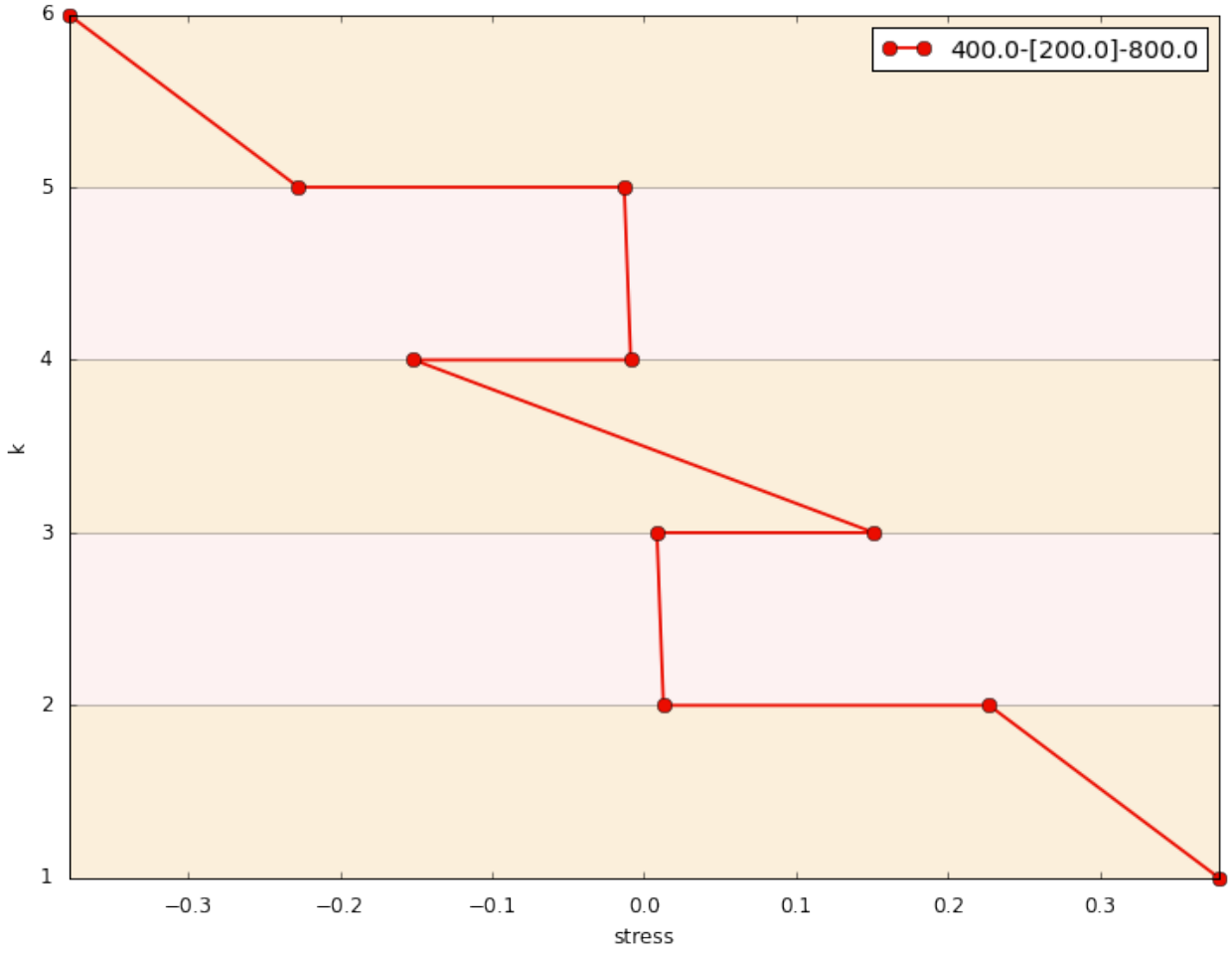
Fig. 2.9: demo

This is image of results from legacy code used for comparison.

We can plot the stress distribution for a case of a single geometry.

```
In [86]: case3 = la.distributions.Case(load_params, mat_props)
         case3.apply(['400-200-800'], model='Wilson_LT')
         case3.plot()
```

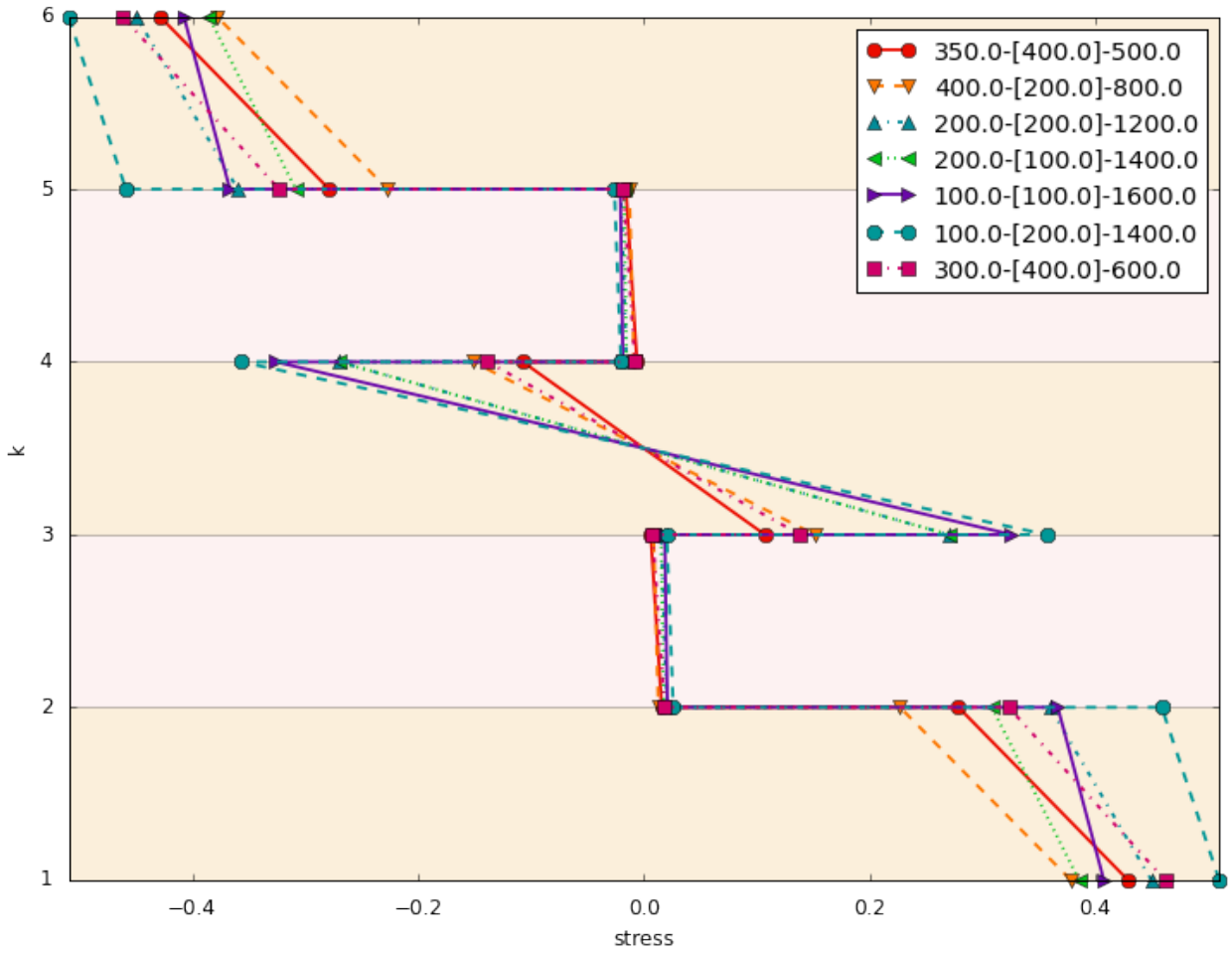
```
Converting mat_props to Standard Form.
User input geometries have been converted and set to Case.
```



We can also plot multiple geometries of similar total thickness.

```
In [87]: five_plies = ['350-400-500',  '400-200-800', '200-200-1200', '200-100-1400',
                       '100-100-1600', '100-200-1400', '300-400-600']

         case4 = la.distributions.Case(load_params, mat_props)
         case4.apply(five_plies, model='Wilson_LT')
         case4.plot()
```

```
Converting mat_props to Standard Form.
User input geometries have been converted and set to Case.
```




```
In [88]: '''If different plies or patterns, make new caselet (subplot)'''
         '''[400-200-800, '300-[400,200]-600']           # non-congruent? equi-ply'''
         '''[400-200-800, '400-200-0']                   # odd/even ply'''
         # currently superimposes plots.  Just needs to separate.

Out[88]: "[400-200-800, '400-200-0']                    # odd/even ply"
```

2.15.8 Exporting

Saving data is critical for future analysis. LamAna offers two formas for exporting your data and parameters. Parameters used to make calculations such as the FeatureInput information are saved as "dashboards" in different forms. - '.xlsx': (default); convient for storing multiple calculationa amd dashboards as se[arate worksheets in a Excel workbook. - '.csv': universal format; separate files for data and dashboard.

The lowest level to export data is for a LaminateModel object.

```
In [89]: LM = case4.LMs[0]
         LM.to_xlsx(temp=True, delete=True)                    # or `to_csv()`
         ;

Out[89]: ''
```

NOTE For demonstration purposes, the `temp` and `delete` are activated. This will create temporary files in the OS temp directory and automatically delete them. For practical use, ignore setting these flags.

The latter LaminateModel data was saved to an .xlsx file in the default export folder. The filepath is returned (currently suppressed with the `;` line).

The next level to export data is for a case. This will save all files comprise in a case. If exported to csv format, files are saved seperately. In xlsx format, a single file is made where each LaminateModel data and dashboard are saved as seperate worksheets.

```
In [90]: case4.to_xlsx(temp=True, delete=True)                    # or `to_csv()`
         ;
Out[90]: ''
```

2.16 Tutorial: Intermediate

So far, the barebones objects have been discussed and a lot can be accomplished with the basics. For users who have some experience with Python and Pandas, here are some intermediate techniques to reduce repetitious actions.

This section dicusses the use of abstract base classes intended for reducing redundant tasks such as **multiple case creation** and **default parameter definitions**. Custom model subclassing is also discussed.

```
In [91]: #-----------------------------------------------------------------------------
         import pandas as pd

         import lamana as la

         %matplotlib inline
         #%matplotlib nbagg
         # PARAMETERS ----------------------------------------------------------------
         # Build dicts of loading parameters and and material properties
         load_params = {'R' : 12e-3,                                  # specimen radius
                        'a' : 7.5e-3,                                 # support ring radius
                        'r' : 2e-4,                                   # radial distance from
                        'P_a' : 1,                                    # applied load
                        'p' : 5,                                      # points/layer
                        }

         # # Quick Form: a dict of lists
         # mat_props = {'HA' : [5.2e10, 0.25],
         #              'PSu' : [2.7e9, 0.33],}

         # Standard Form: a dict of dicts
         mat_props = {'Modulus': {'HA': 5.2e10, 'PSu': 2.7e9},
                      'Poissons': {'HA': 0.25, 'PSu': 0.33}}


         # What geometries to test?
         # Make tuples of desired geometeries to analyze: outer - {inner...-....}_i - midd

         # Current Style
         g1 = ('0-0-2000')                                            # Monolith
         g2 = ('1000-0-0')                                            # Bilayer
         g3 = ('600-0-800')                                           # Trilayer
         g4 = ('500-500-0')                                           # 4-ply
```

```
g5 = ('400-200-800')                                    # Short-hand; <= 5-ply
g6 = ('400-200-400S')                                   # Symmetric
g7 = ('400-[200]-800')                                  # General convention;
g8 = ('400-[100,100]-800')                              # General convention;
g9 = ('400-[100,100]-400S')                             # General and Symmetri

'''Add to test set'''
g13 = ('400-[150,50]-800')                              # Dissimilar inner_is
g14 = ('400-[25,125,50]-800')


geos_most = [g1, g2, g3, g4, g5]
geos_special = [g6, g7, g8, g9]
geos_full = [g1, g2, g3, g4, g5, g6, g7, g8, g9]
geos_dissimilar = [g13, g14]
```

2.16.1 Generating Multiple Cases

We've already seen we can *generate a case object and plots with three lines of code*. However, sometimes it is necessary to generate different cases. These invocations can be tedious with three lines of code per case. Have no fear. A simple way to produce more cases is to instantiate a `Cases` object.

Below we will create a `Cases` which houses multiples cases that: - share similiar loading parameters/material properties and laminate theory model with - different numbers of datapoints, p

```
In [92]: cases1 = la.distributions.Cases(['400-200-800', '350-400-500',
                                          '400-200-0', '1000-0-0'],
                                         load_params=load_params,
                                         mat_props=mat_props, model= 'Wilson_LT',
                                         ps=[3,4,5])
         cases1
```

```
Caselets not using `combine`.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
```

```
Out[92]: <lamana.distributions.Cases object at 0x0000000009B15710>, {0: <<class 'lamana.dis
```

`Cases()` accepts a list of geometry strings. Given appropriate default keywords, this lone argument will return a dict-like object of cases with indicies as keys. The `model` and `ps` keywords have default values.

A `Cases()` object has some interesting characteristics (this is not a dict):

- if user-defined, tries to import `Defaults()` to simplify instantiations
- dict-like storage and access of cases
- list-like ordering of cases

- gettable: list-like, get items by index (including negative indicies)
- sliceable: slices the dict keys of the Cases object
- viewable: contained LaminateModels
- iterable: by values (unlike normal dicts, not by keys)
- writable: write DataFrames to csv files
- selectable: perform set operations and return unique subsets

```
In [93]: # Gettable

         cases1[0]                                                    # normal dict key selec
         cases1[-1]                                                   # negative indices
         cases1[-2]                                                   # negative indicies
```

```
Out[93]: <<class 'lamana.distributions.Case'> p=5, size=1>
```

```
In [94]: # Sliceable

         cases1[0:2]                                                  # range of dict keys
         cases1[0:3]                                                  # full range of dict ke
         cases1[:]                                                    # full range
         cases1[1:]                                                   # start:None
         cases1[:2]                                                   # None:stop
         cases1[:-1]                                                  # None:negative index
         cases1[:-2]                                                  # None:negative index
         #cases1[0:-1:-2]                                             # start:stop:step; Not
         #cases1[::-1]                                                # reverse; NotImplement
```

```
Out[94]: [<<class 'lamana.distributions.Case'> p=3, size=1>,
          <<class 'lamana.distributions.Case'> p=3, size=1>,
          <<class 'lamana.distributions.Case'> p=3, size=1>,
          <<class 'lamana.distributions.Case'> p=3, size=1>,
          <<class 'lamana.distributions.Case'> p=4, size=1>,
          <<class 'lamana.distributions.Case'> p=4, size=1>,
          <<class 'lamana.distributions.Case'> p=4, size=1>,
          <<class 'lamana.distributions.Case'> p=4, size=1>,
          <<class 'lamana.distributions.Case'> p=5, size=1>,
          <<class 'lamana.distributions.Case'> p=5, size=1>]
```

```
In [95]: # Viewable
         cases1
         cases1.LMs
```

```
Out[95]: [<lamana LaminateModel object (400.0-[200.0]-800.0), p=3>,
          <lamana LaminateModel object (350.0-[400.0]-500.0), p=3>,
          <lamana LaminateModel object (400.0-[200.0]-0.0), p=3>,
          <lamana LaminateModel object (1000.0-[0.0]-0.0), p=3>,
          <lamana LaminateModel object (400.0-[200.0]-800.0), p=4>,
          <lamana LaminateModel object (350.0-[400.0]-500.0), p=4>,
          <lamana LaminateModel object (400.0-[200.0]-0.0), p=4>,
          <lamana LaminateModel object (1000.0-[0.0]-0.0), p=4>,
          <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>,
          <lamana LaminateModel object (350.0-[400.0]-500.0), p=5>,
          <lamana LaminateModel object (400.0-[200.0]-0.0), p=5>,
          <lamana LaminateModel object (1000.0-[0.0]-0.0), p=5>]
```

```
In [96]: # Iterable
         for i, case in enumerate(cases1):                                    # __iter__ values
             print(case)
             #print(case.LMs)                                                 # access LaminateModels
```

```
<<class 'lamana.distributions.Case'> p=3>
<<class 'lamana.distributions.Case'> p=3>
<<class 'lamana.distributions.Case'> p=3>
<<class 'lamana.distributions.Case'> p=3>
<<class 'lamana.distributions.Case'> p=4>
<<class 'lamana.distributions.Case'> p=4>
<<class 'lamana.distributions.Case'> p=4>
<<class 'lamana.distributions.Case'> p=4>
<<class 'lamana.distributions.Case'> p=5>
<<class 'lamana.distributions.Case'> p=5>
<<class 'lamana.distributions.Case'> p=5>
<<class 'lamana.distributions.Case'> p=5>
```

```
In [97]: # Writable
         #cases1.to_csv()                                                    # write to file
```

```
In [98]: # Selectable
         cases1.select(nplies=[2,4])                                         # by # plies
         cases1.select(ps=[3,4])                                             # by points/DataFrame r
         cases1.select(nplies=[2,4], ps=[3,4], how='intersection')           # by set operations
```

```
Out[98]: {<lamana LaminateModel object (400.0-[200.0]-0.0), p=3>,
          <lamana LaminateModel object (1000.0-[0.0]-0.0), p=4>,
          <lamana LaminateModel object (1000.0-[0.0]-0.0), p=3>,
          <lamana LaminateModel object (400.0-[200.0]-0.0), p=4>}
```

LamainateModels can be compared using set theory. Unique subsets of LaminateModels can be returned from a mix of repeated geometry strings. We will use the default `model` and `ps` values.

```
In [99]: set(geos_most).issubset(geos_full)                                  # confirm repeated items
```

```
Out[99]: True
```

```
In [100]: mix = geos_full + geos_most                                        # contains repeated ite
```

```
In [101]: # Repeated Subset
          cases2 = la.distributions.Cases(mix, load_params=load_params, mat_props=mat_props
          cases2.LMs
```

```
Caselets not using `combine`.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
```

```
Out[101]: [<lamana LaminateModel object (0.0-[0.0]-2000.0), p=5>,
           <lamana LaminateModel object (1000.0-[0.0]-0.0), p=5>,
           <lamana LaminateModel object (600.0-[0.0]-800.0), p=5>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=5>,
           <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>,
           <lamana LaminateModel object (400.0-[200.0]-400.0S), p=5>,
           <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>,
           <lamana LaminateModel object (400.0-[100.0,100.0]-800.0), p=5>,
           <lamana LaminateModel object (400.0-[100.0,100.0]-400.0S), p=5>,
           <lamana LaminateModel object (0.0-[0.0]-2000.0), p=5>,
           <lamana LaminateModel object (1000.0-[0.0]-0.0), p=5>,
           <lamana LaminateModel object (600.0-[0.0]-800.0), p=5>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=5>,
           <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>]
```

```
In [102]: # Unique Subset
          cases2 = la.distributions.Cases(mix, load_params=load_params, mat_props=mat_props
                                          unique=True)
          cases2.LMs
```

```
Single geometry string detected. unique not applied. See combine=True keyword.
Single geometry string detected. unique not applied. See combine=True keyword.
Single geometry string detected. unique not applied. See combine=True keyword.
Single geometry string detected. unique not applied. See combine=True keyword.
Single geometry string detected. unique not applied. See combine=True keyword.
Single geometry string detected. unique not applied. See combine=True keyword.
Single geometry string detected. unique not applied. See combine=True keyword.
Single geometry string detected. unique not applied. See combine=True keyword.
Single geometry string detected. unique not applied. See combine=True keyword.
Single geometry string detected. unique not applied. See combine=True keyword.
Single geometry string detected. unique not applied. See combine=True keyword.
Single geometry string detected. unique not applied. See combine=True keyword.
Single geometry string detected. unique not applied. See combine=True keyword.
Single geometry string detected. unique not applied. See combine=True keyword.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
```

```
Out[102]: [<lamana LaminateModel object (0.0-[0.0]-2000.0), p=5>,
           <lamana LaminateModel object (1000.0-[0.0]-0.0), p=5>,
           <lamana LaminateModel object (600.0-[0.0]-800.0), p=5>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=5>,
           <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>,
           <lamana LaminateModel object (400.0-[200.0]-400.0S), p=5>,
```

```
 <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>,
 <lamana LaminateModel object (400.0-[100.0,100.0]-800.0), p=5>,
 <lamana LaminateModel object (400.0-[100.0,100.0]-400.0S), p=5>,
 <lamana LaminateModel object (0.0-[0.0]-2000.0), p=5>,
 <lamana LaminateModel object (1000.0-[0.0]-0.0), p=5>,
 <lamana LaminateModel object (600.0-[0.0]-800.0), p=5>,
 <lamana LaminateModel object (500.0-[500.0]-0.0), p=5>,
 <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>]
```

2.16.2 Subclassing Custom Default Parameters

We observed the benefits of using *implicit*, default keywords (`models`, `ps`) in simplifying the writing of `Cases()` instantiations. In general, the user can code *explicit* defaults for `load_params` and `mat_props` by subclassing `BaseDefaults()` from `inputs_`. While subclassing requires some extra Python knowledge, this is a relatively simple process that reduces a significant amount of redundant code, leading to a more effiencient anaytical setting.

The `BaseDefaults` contains a dict various geometry strings and Geometry objects. Rather than defining examples for various geometry plies, the user can call from all or a groupings of geometries.

```
In [103]: from lamana.input_ import BaseDefaults

          bdft = BaseDefaults()

          # geometry String Attributes
          bdft.geo_inputs                                          # all dict key-values
          bdft.geos_all                                            # all geo strings
          bdft.geos_standard                                       # static
          bdft.geos_sample                                         # active; grows

          # Geometry Object Attributes; mimics latter
          bdft.Geo_objects                                         # all dict key-values
          bdft.Geos_all                                            # all Geo objects
          # more ...

          # Custom FeatureInputs
          #bdft.get_FeatureInput()                                  # quick builds
          #bdft.get_materials()                                     # convert to std. for

Out[103]: [Geometry object (0.0-[0.0]-2000.0),
           Geometry object (0.0-[0.0]-1000.0),
           Geometry object (1000.0-[0.0]-0.0),
           Geometry object (600.0-[0.0]-800.0),
           Geometry object (600.0-[0.0]-400.0S),
           Geometry object (500.0-[500.0]-0.0),
           Geometry object (400.0-[200.0]-0.0),
           Geometry object (400.0-[200.0]-800.0),
           Geometry object (400.0-[200.0]-800.0),
           Geometry object (400.0-[200.0]-400.0S),
           Geometry object (400.0-[100.0,100.0]-0.0),
           Geometry object (500.0-[250.0,250.0]-0.0),
           Geometry object (400.0-[100.0,100.0]-800.0),
           Geometry object (400.0-[100.0,100.0]-400.0S),
           Geometry object (400.0-[100.0,100.0,100.0]-800.0),
           Geometry object (500.0-[50.0,50.0,50.0,50.0]-0.0),
```

```
 Geometry object (400.0-[100.0,100.0,100.0,100.0]-800.0),
 Geometry object (400.0-[100.0,100.0,100.0,100.0,100.0]-800.0)]
```

The latter geometric defaults come out of the box when subclassed from `BaseDefaults`. If custom geometries are desired, the user can override the `geo_inputs` dict, which automatically builds the `Geo_objects` dict.

Users can override three categories of defaults parameters:

1. geometric variables
2. loading parameters
3. material properties

As mentioned, some geometric variables are provided for general laminate dimensions. The other parameters cannot be predicted and need to be defined by the user. Below is an example of a Defaults() subclass. If a custom model has been implemented (see next section), it is convention to place `Defaults()` and all other custom code within this module. If a custom model is implemented an located in the models directory, Cases will automatically search will the designated model modules, locate the `load_params` and `mat_props` attributes and load them automatically for all `Cases` instantiations.

```
In [104]: # Example Defaults from LamAna.models.Wilson_LT
          class Defaults(BaseDefaults):
              '''Return parameters for building distributions cases.  Useful for consisten
              testing.

              Dimensional defaults are inheirited from utils.BaseDefaults().
              Material-specific parameters are defined here by he user.

              - Default geometric and materials parameters
              - Default FeatureInputs

              Examples
              ========
              >>>dft = Defaults()
              >>>dft.load_params
              {'R' : 12e-3, 'a' : 7.5e-3, 'p' : 1, 'P_a' : 1, 'r' : 2e-4,}

              >>>dft.mat_props
              {'Modulus': {'HA': 5.2e10, 'PSu': 2.7e9},
              'Poissons': {'HA': 0.25, 'PSu': 0.33}}

              >>>dft.FeatureInput
               {'Geometry' : '400-[200]-800',
                'Geometric' : {'R' : 12e-3, 'a' : 7.5e-3, 'p' : 1, 'P_a' : 1, 'r' : 2e-4,},
                'Materials' : {'HA' : [5.2e10, 0.25], 'PSu' : [2.7e9, 0.33],},
                'Custom' : None,
                'Model' : Wilson_LT,
               }

              '''
              def __init__(self):
                  BaseDefaults.__init__(self)
                  '''DEV: Add defaults first.  Then adjust attributes.'''
                  # DEFAULTS ---------------------------------------------------------
                  # Build dicts of geometric and material parameters
                  self.load_params = {'R' : 12e-3,                    # specimen radius
```

```
                                    'a' : 7.5e-3,                   # support ring radius
                                    'p' : 5,                        # points/layer
                                    'P_a' : 1,                      # applied load
                                    'r' : 2e-4,                     # radial distance fro
                                    }

                    self.mat_props = {'Modulus': {'HA': 5.2e10, 'PSu': 2.7e9},
                                      'Poissons': {'HA': 0.25, 'PSu': 0.33}}

                    # ATTRIBUTES ------------------------------------------------------------
                    # FeatureInput
                    self.FeatureInput = self.get_FeatureInput(self.Geo_objects['standard'][0
                                                              load_params=self.load_params,
                                                              mat_props=self.mat_props,
                                                              ##custom_matls=None,
                                                              model='Wilson_LT',
                                                              global_vars=None)
```

```
In [105]: '''Use Classic_LT here'''
```

```
Out[105]: 'Use Classic_LT here'
```

```
In [106]: from lamana.distributions import Cases
          # Auto load_params and mat_params

          dft = Defaults()
          cases3 = Cases(dft.geos_full, model='Wilson_LT')
          #cases3 = la.distributions.Cases(dft.geos_full, model='Wilson_LT')
          cases3
```

```
Caselets not using `combine`.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
```

```
Out[106]: <lamana.distributions.Cases object at 0x000000000AF85EB8>, {0: <<class 'lamana.di
```

```
In [107]: '''Refine idiom for importing Cases '''
```

```
Out[107]: 'Refine idiom for importing Cases '
```

2.16.3 Subclassing Custom Models

One of the most powerful feauteres of LamAna is the ability to define customized modifications to the Laminate Theory models.

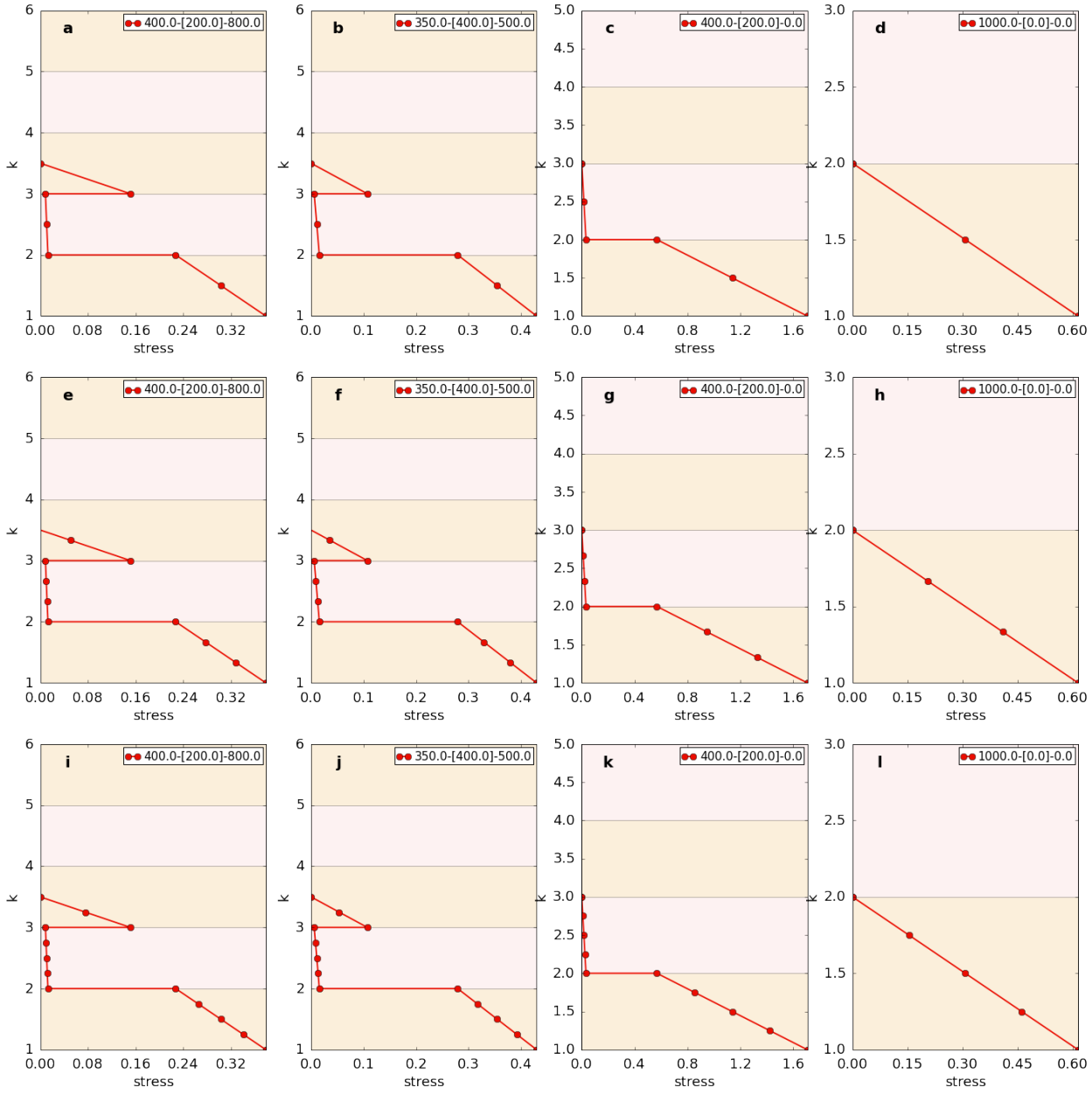
Code for laminate theories (i.e. Classic_LT, Wilson_LT) are are located in the models directory. These models can be simple functions or sublclass from `BaseModels` in the `theories` module. Either approach is acceptable (see narrative docs for more details on creating custom models.

This ability to add custom code make this library extensibile to use a larger variety of models.

2.16.4 Plotting Cases

An example of multiple subplots is show below. Using a former case, notice each subplot is indepent, woth separate geometries for each. LamAna treats each subplot as a subset or "caselet":

```
In [108]: cases1.plot(extrema=False)
```



Each caselet can also be a separate case, plotting multiple geometries for each as accomplished with `Case`.

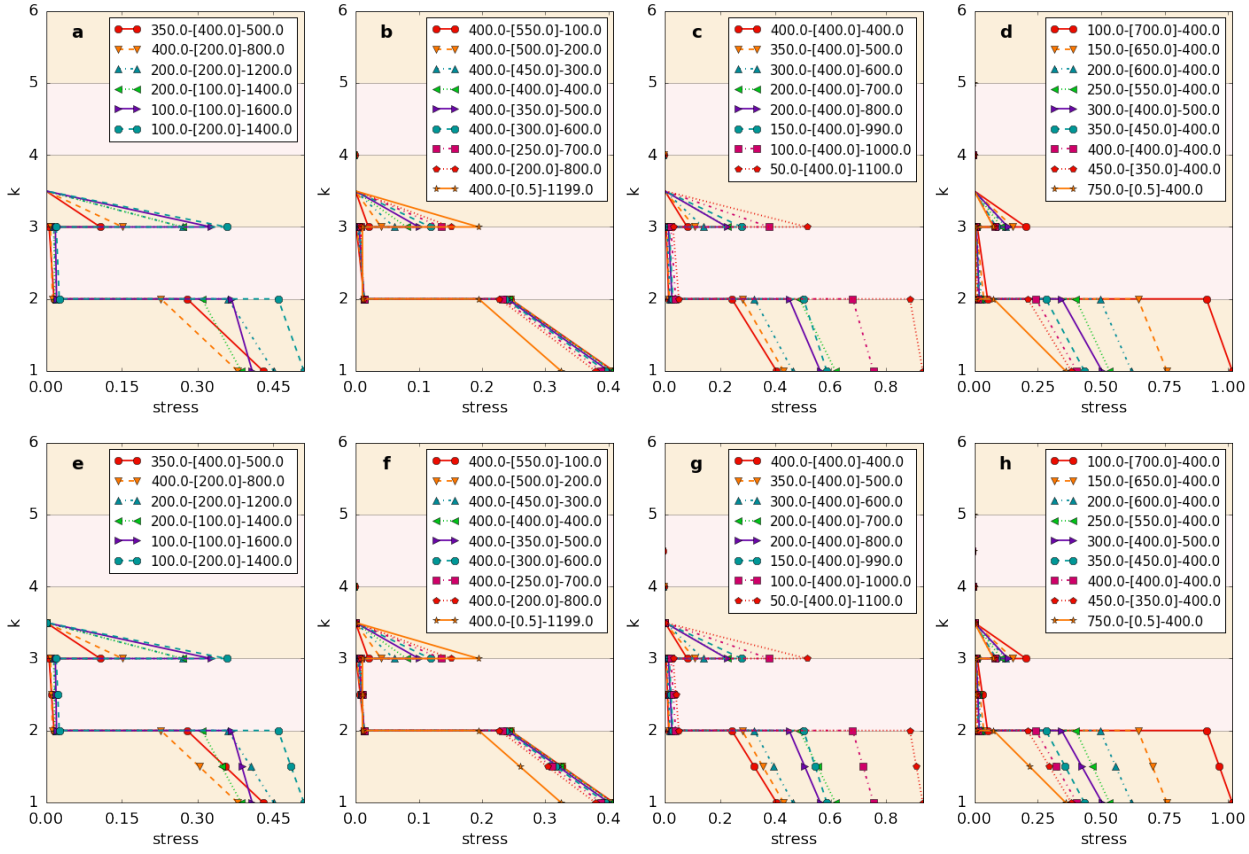
```
In [109]: const_total = ['350-400-500',  '400-200-800', '200-200-1200',
                         '200-100-1400', '100-100-1600', '100-200-1400',]
          const_outer = ['400-550-100', '400-500-200', '400-450-300',
                         '400-400-400', '400-350-500', '400-300-600',
                         '400-250-700', '400-200-800', '400-0.5-1199']
          const_inner = ['400-400-400', '350-400-500', '300-400-600',
                         '200-400-700', '200-400-800', '150-400-990',
```

```
                    '100-400-1000', '50-400-1100',]
        const_middle = ['100-700-400', '150-650-400', '200-600-400',
                        '250-550-400', '300-400-500', '350-450-400',
                        '400-400-400', '450-350-400', '750-0.5-400']

        case1_ = const_total
        case2_ = const_outer
        case3_ = const_inner
        case4_ = const_middle

        cases_ = [case1_, case2_, case3_, case4_]
```

```
In [110]: cases3 = la.distributions.Cases(cases_, load_params=load_params,
                                           mat_props=mat_props, model= 'Wilson_LT',
                                           ps=[2,3])
```

```
Caselets not using `combine`.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
```

```
In [111]: cases3.plot(extrema=False)
```



See Demo notebooks for more examples of plotting.

2.16.5 More on Cases

```
In [112]: '''Fix importing cases'''
```

```
Out[112]: 'Fix importing cases'
```

```
In [113]: from lamana.distributions import Cases
```

Applying caselets

The term "caselet" is defined in LPEP 003. Most importantly, the various types a caselet represents is handled by `Cases` and discussed here. In 0.4.4b3+, caselets are contained in lists. LPEP entertains the idea of containing caselets in dicts.

```
In [114]: from lamana.models import Wilson_LT as wlt
          dft = wlt.Defaults()

          %matplotlib inline


          str_caselets = ['350-400-500',  '400-200-800', '400-[200]-800']
          list_caselets = [['400-400-400', '400-[400]-400'],
                           ['200-100-1400', '100-200-1400',],
                           ['400-400-400', '400-200-800','350-400-500',],
                           ['350-400-500']]
          case1 = la.distributions.Case(dft.load_params, dft.mat_props)
          case2 = la.distributions.Case(dft.load_params, dft.mat_props)
          case3 = la.distributions.Case(dft.load_params, dft.mat_props)
          case1.apply(['400-200-800', '400-[200]-800'])
          case2.apply(['350-400-500', '400-200-800'])
          case3.apply(['350-400-500', '400-200-800', '400-400-400'])
          case_caselets = [case1, case2, case3]
          mixed_caselets = [['350-400-500', '400-200-800',],
                            [['400-400-400', '400-[400]-400'],
                             ['200-100-1400', '100-200-1400',]],
                            [case1, case2,]
                           ]
          dict_caselets = {0: ['350-400-500',  '400-200-800', '200-200-1200',
                               '200-100-1400', '100-100-1600', '100-200-1400'],
                           1: ['400-550-100', '400-500-200', '400-450-300',
                               '400-400-400', '400-350-500', '400-300-600'],
                           2: ['400-400-400', '350-400-500', '300-400-600',
                               '200-400-700', '200-400-800', '150-400-990'],
                           3: ['100-700-400', '150-650-400', '200-600-400',
                               '250-550-400', '300-400-500', '350-450-400'],
                           }
```

```
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
```

```
In [115]: cases = Cases(str_caselets)
          #cases = Cases(str_caselets, combine=True)
          #cases = Cases(list_caselets)
          #cases = Cases(list_caselets, combine=True)
          #cases = Cases(case_caselets)
          #cases = Cases(case_caselets, combine=True)                # collapse to one pl
```

```
#cases = Cases(str_caselets, ps=[2,5])
#cases = Cases(list_caselets, ps=[2,3,5,7])
#cases = Cases(case_caselets, ps=[2,5])
#cases = Cases([], combine=True)                                    # test raises


# For next versions
#cases = Cases(dict_caselets)
#cases = Cases(mixed_caselets)
#cases = Cases(mixed_caselets, combine=True)
cases
```

```
Caselets not using `combine`.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
```

```
Out[115]: <lamana.distributions.Cases object at 0x000000000C46E748>, {0: <<class 'lamana.di
```

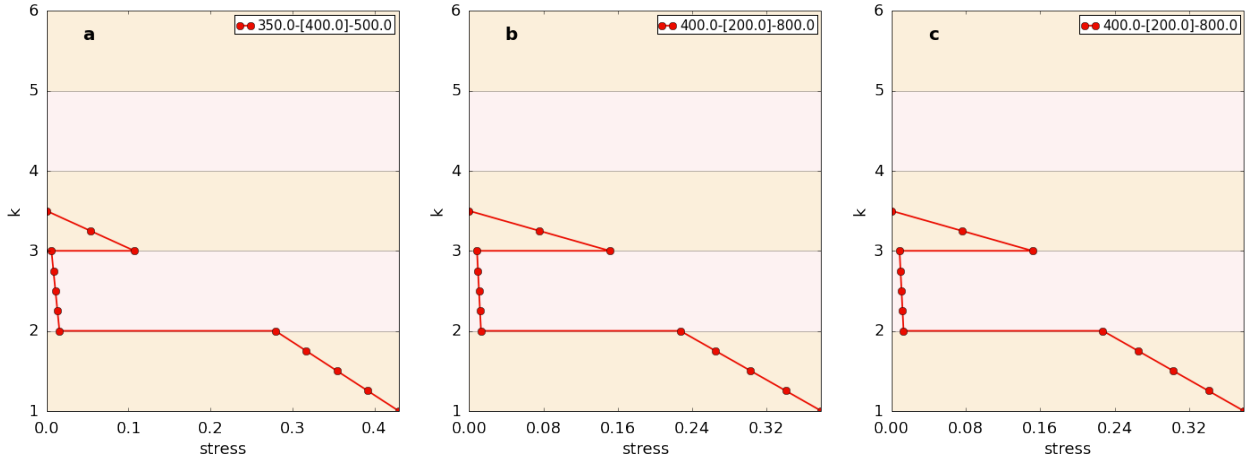
```
In [116]: cases.LMs
```

```
Out[116]: [<lamana LaminateModel object (350.0-[400.0]-500.0), p=5>,
           <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>,
           <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>]
```

```
In [117]: '''BUG: Following cell raises an Exception in Python 2'''
```

```
Out[117]: 'BUG: Following cell raises an Exception in Python 2'
```

```
In [118]: #cases.plot()
          #cases.plot(normalized=False)
          #cases.plot(colorblind=True, grayscale=True)
          cases.plot(extrema=False)                                # needed to see ps
```




```
In [119]: cases.caselets
```

```
Out[119]: ['350.0-[400.0]-500.0', '400.0-[200.0]-800.0', '400.0-[200.0]-800.0']
```

```
In [120]: '''get out tests from code'''
          '''run tests'''
          '''test set seletions'''
```

```
Out[120]: 'test set seletions'
```

Characteristics

```
In [121]: from lamana.models import Wilson_LT as wlt

          dft = wlt.Defaults()
          cases = Cases(dft.geo_inputs['5-ply'], ps=[2,3,4])

Caselets not using `combine`.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.

In [122]: len(cases)                                                    # test __len__

Out[122]: 9

In [123]: cases.get(1)                                                  # __getitem__

Out[123]: <<class 'lamana.distributions.Case'> p=2, size=1>

In [124]: #cases[2] = 'test'                                            # __setitem__; not imp

In [125]: cases[0]                                                      # select

Out[125]: <<class 'lamana.distributions.Case'> p=2, size=1>

In [126]: cases[0:2]                                                    # slice (__getitem__)

Out[126]: [<<class 'lamana.distributions.Case'> p=2, size=1>,
           <<class 'lamana.distributions.Case'> p=2, size=1>]

In [127]: del cases[1]                                                  # __delitem__

In [128]: cases                                                         # test __repr__

Out[128]: <lamana.distributions.Cases object at 0x000000000B6292E8>, {0: <<class 'lamana.di

In [129]: print(cases)                                                  # test __str__

{0: <<class 'lamana.distributions.Case'> p=2, size=1>, 2: <<class 'lamana.distributions.Cas

In [130]: cases == cases                                                # test __eq__

Out[130]: True

In [131]: not cases != cases                                            # test __ne__

Out[131]: True

In [132]: for i, case in enumerate(cases):                              # __iter__ values
              print(case)
              #print(case.LMs)

<<class 'lamana.distributions.Case'> p=2>
<<class 'lamana.distributions.Case'> p=2>
<<class 'lamana.distributions.Case'> p=3>
<<class 'lamana.distributions.Case'> p=3>
<<class 'lamana.distributions.Case'> p=3>
<<class 'lamana.distributions.Case'> p=4>
```

```
<<class 'lamana.distributions.Case'> p=4>
<<class 'lamana.distributions.Case'> p=4>
```

```
In [133]: cases.LMs                                                    # peek inside cases
```

```
Out[133]: [<lamana LaminateModel object (400.0-[200.0]-800.0), p=2>,
           <lamana LaminateModel object (400.0-[200.0]-400.0S), p=2>,
           <lamana LaminateModel object (400.0-[200.0]-800.0), p=3>,
           <lamana LaminateModel object (400.0-[200.0]-800.0), p=3>,
           <lamana LaminateModel object (400.0-[200.0]-400.0S), p=3>,
           <lamana LaminateModel object (400.0-[200.0]-800.0), p=4>,
           <lamana LaminateModel object (400.0-[200.0]-800.0), p=4>,
           <lamana LaminateModel object (400.0-[200.0]-400.0S), p=4>]
```

```
In [134]: cases.frames                                                 # get a list of DataFr
```

```
Accessing frames method.
```

```
Out[134]: [   layer   side    type matl      label        ...          strain_r  \
           0      1  Tens.   outer   HA  interface        ...         3.452e-06
           1      1  Tens.   outer   HA   discont.        ...         2.071e-06
           2      2  Tens.   inner  PSu  interface        ...         2.071e-06
           3      2  Tens.   inner  PSu   discont.        ...         1.381e-06
           4      3  Tens.  middle   HA  interface        ...         1.381e-06
           5      3  Comp.  middle   HA  interface        ...        -1.381e-06
           6      4  Comp.   inner  PSu   discont.        ...        -1.381e-06
           7      4  Comp.   inner  PSu  interface        ...        -2.071e-06
           8      5  Comp.   outer   HA   discont.        ...        -2.071e-06
           9      5  Comp.   outer   HA  interface        ...        -3.452e-06

               strain_t  stress_r (Pa/N)  stress_t (Pa/N)  stress_f (MPa/N)
           0  5.965e-06       274182.824       378730.663             0.379
           1  3.579e-06       164509.695       227238.398             0.227
           2  3.579e-06         9854.181        12915.334             0.013
           3  2.386e-06         6569.454         8610.223             0.009
           4  2.386e-06       109673.130       151492.265             0.151
           5 -2.386e-06      -109673.130      -151492.265            -0.151
           6 -2.386e-06        -6569.454        -8610.223            -0.009
           7 -3.579e-06        -9854.181       -12915.334            -0.013
           8 -3.579e-06      -164509.695      -227238.398            -0.227
           9 -5.965e-06      -274182.824      -378730.663            -0.379

           [10 rows x 22 columns],
              layer   side    type matl      label        ...          strain_r  \
           0      1  Tens.   outer   HA  interface        ...         3.452e-06
           1      1  Tens.   outer   HA   discont.        ...         2.071e-06
           2      2  Tens.   inner  PSu  interface        ...         2.071e-06
           3      2  Tens.   inner  PSu   discont.        ...         1.381e-06
           4      3  Tens.  middle   HA  interface        ...         1.381e-06
           5      3  Comp.  middle   HA  interface        ...        -1.381e-06
           6      4  Comp.   inner  PSu   discont.        ...        -1.381e-06
           7      4  Comp.   inner  PSu  interface        ...        -2.071e-06
           8      5  Comp.   outer   HA   discont.        ...        -2.071e-06
           9      5  Comp.   outer   HA  interface        ...        -3.452e-06

               strain_t  stress_r (Pa/N)  stress_t (Pa/N)  stress_f (MPa/N)
```

```
0  5.965e-06       274182.824       378730.663            0.379
1  3.579e-06       164509.695       227238.398            0.227
2  3.579e-06         9854.181        12915.334            0.013
3  2.386e-06         6569.454         8610.223            0.009
4  2.386e-06       109673.130       151492.265            0.151
5 -2.386e-06      -109673.130      -151492.265           -0.151
6 -2.386e-06        -6569.454        -8610.223           -0.009
7 -3.579e-06        -9854.181       -12915.334           -0.013
8 -3.579e-06      -164509.695      -227238.398           -0.227
9 -5.965e-06      -274182.824      -378730.663           -0.379

[10 rows x 22 columns],
    layer   side    type matl       label        ...          strain_r  \
0       1  Tens.   outer   HA   interface        ...         3.452e-06
1       1  Tens.   outer   HA    internal        ...         2.762e-06
2       1  Tens.   outer   HA    discont.        ...         2.071e-06
3       2  Tens.   inner  PSu   interface        ...         2.071e-06
4       2  Tens.   inner  PSu    internal        ...         1.726e-06
5       2  Tens.   inner  PSu    discont.        ...         1.381e-06
6       3  Tens.  middle   HA   interface        ...         1.381e-06
7       3   None  middle   HA  neut. axis        ...         0.000e+00
8       3  Comp.  middle   HA   interface        ...        -1.381e-06
9       4  Comp.   inner  PSu    discont.        ...        -1.381e-06
10      4  Comp.   inner  PSu    internal        ...        -1.726e-06
11      4  Comp.   inner  PSu   interface        ...        -2.071e-06
12      5  Comp.   outer   HA    discont.        ...        -2.071e-06
13      5  Comp.   outer   HA    internal        ...        -2.762e-06
14      5  Comp.   outer   HA   interface        ...        -3.452e-06

     strain_t  stress_r (Pa/N)  stress_t (Pa/N)  stress_f (MPa/N)
0   5.965e-06       274182.824       378730.663            0.379
1   4.772e-06       219346.259       302984.530            0.303
2   3.579e-06       164509.695       227238.398            0.227
3   3.579e-06         9854.181        12915.334            0.013
4   2.983e-06         8211.817        10762.778            0.011
5   2.386e-06         6569.454         8610.223            0.009
6   2.386e-06       109673.130       151492.265            0.151
7   0.000e+00            0.000            0.000            0.000
8  -2.386e-06      -109673.130      -151492.265           -0.151
9  -2.386e-06        -6569.454        -8610.223           -0.009
10 -2.983e-06        -8211.817       -10762.778           -0.011
11 -3.579e-06        -9854.181       -12915.334           -0.013
12 -3.579e-06      -164509.695      -227238.398           -0.227
13 -4.772e-06      -219346.259      -302984.530           -0.303
14 -5.965e-06      -274182.824      -378730.663           -0.379

[15 rows x 22 columns],
    layer   side    type matl       label        ...          strain_r  \
0       1  Tens.   outer   HA   interface        ...         3.452e-06
1       1  Tens.   outer   HA    internal        ...         2.762e-06
2       1  Tens.   outer   HA    discont.        ...         2.071e-06
3       2  Tens.   inner  PSu   interface        ...         2.071e-06
4       2  Tens.   inner  PSu    internal        ...         1.726e-06
5       2  Tens.   inner  PSu    discont.        ...         1.381e-06
```

```
6       3  Tens.  middle   HA   interface        ...         1.381e-06
7       3   None  middle   HA  neut. axis        ...         0.000e+00
8       3  Comp.  middle   HA   interface        ...        -1.381e-06
9       4  Comp.   inner  PSu    discont.        ...        -1.381e-06
10      4  Comp.   inner  PSu    internal        ...        -1.726e-06
11      4  Comp.   inner  PSu   interface        ...        -2.071e-06
12      5  Comp.   outer   HA    discont.        ...        -2.071e-06
13      5  Comp.   outer   HA    internal        ...        -2.762e-06
14      5  Comp.   outer   HA   interface        ...        -3.452e-06

     strain_t  stress_r (Pa/N)  stress_t (Pa/N)  stress_f (MPa/N)
0   5.965e-06       274182.824       378730.663             0.379
1   4.772e-06       219346.259       302984.530             0.303
2   3.579e-06       164509.695       227238.398             0.227
3   3.579e-06         9854.181        12915.334             0.013
4   2.983e-06         8211.817        10762.778             0.011
5   2.386e-06         6569.454         8610.223             0.009
6   2.386e-06       109673.130       151492.265             0.151
7   0.000e+00            0.000            0.000             0.000
8  -2.386e-06      -109673.130      -151492.265            -0.151
9  -2.386e-06        -6569.454        -8610.223            -0.009
10 -2.983e-06        -8211.817       -10762.778            -0.011
11 -3.579e-06        -9854.181       -12915.334            -0.013
12 -3.579e-06      -164509.695      -227238.398            -0.227
13 -4.772e-06      -219346.259      -302984.530            -0.303
14 -5.965e-06      -274182.824      -378730.663            -0.379

[15 rows x 22 columns],
    layer   side    type matl       label        ...          strain_r  \
0       1  Tens.   outer   HA   interface        ...         3.452e-06
1       1  Tens.   outer   HA    internal        ...         2.762e-06
2       1  Tens.   outer   HA    discont.        ...         2.071e-06
3       2  Tens.   inner  PSu   interface        ...         2.071e-06
4       2  Tens.   inner  PSu    internal        ...         1.726e-06
5       2  Tens.   inner  PSu    discont.        ...         1.381e-06
6       3  Tens.  middle   HA   interface        ...         1.381e-06
7       3   None  middle   HA  neut. axis        ...         0.000e+00
8       3  Comp.  middle   HA   interface        ...        -1.381e-06
9       4  Comp.   inner  PSu    discont.        ...        -1.381e-06
10      4  Comp.   inner  PSu    internal        ...        -1.726e-06
11      4  Comp.   inner  PSu   interface        ...        -2.071e-06
12      5  Comp.   outer   HA    discont.        ...        -2.071e-06
13      5  Comp.   outer   HA    internal        ...        -2.762e-06
14      5  Comp.   outer   HA   interface        ...        -3.452e-06

     strain_t  stress_r (Pa/N)  stress_t (Pa/N)  stress_f (MPa/N)
0   5.965e-06       274182.824       378730.663             0.379
1   4.772e-06       219346.259       302984.530             0.303
2   3.579e-06       164509.695       227238.398             0.227
3   3.579e-06         9854.181        12915.334             0.013
4   2.983e-06         8211.817        10762.778             0.011
5   2.386e-06         6569.454         8610.223             0.009
6   2.386e-06       109673.130       151492.265             0.151
7   0.000e+00            0.000            0.000             0.000
```

```
8  -2.386e-06      -109673.130      -151492.265           -0.151
9  -2.386e-06        -6569.454        -8610.223           -0.009
10 -2.983e-06        -8211.817       -10762.778           -0.011
11 -3.579e-06        -9854.181       -12915.334           -0.013
12 -3.579e-06      -164509.695      -227238.398           -0.227
13 -4.772e-06      -219346.259      -302984.530           -0.303
14 -5.965e-06      -274182.824      -378730.663           -0.379

[15 rows x 22 columns],
    layer   side    type matl      label        ...          strain_r  \
0       1  Tens.   outer   HA  interface        ...         3.452e-06
1       1  Tens.   outer   HA   internal        ...         2.992e-06
2       1  Tens.   outer   HA   internal        ...         2.531e-06
3       1  Tens.   outer   HA   discont.        ...         2.071e-06
4       2  Tens.   inner  PSu  interface        ...         2.071e-06
5       2  Tens.   inner  PSu   internal        ...         1.841e-06
6       2  Tens.   inner  PSu   internal        ...         1.611e-06
7       2  Tens.   inner  PSu   discont.        ...         1.381e-06
8       3  Tens.  middle   HA  interface        ...         1.381e-06
9       3  Tens.  middle   HA   internal        ...         4.603e-07
10      3  Comp.  middle   HA   internal        ...        -4.603e-07
11      3  Comp.  middle   HA  interface        ...        -1.381e-06
12      4  Comp.   inner  PSu   discont.        ...        -1.381e-06
13      4  Comp.   inner  PSu   internal        ...        -1.611e-06
14      4  Comp.   inner  PSu   internal        ...        -1.841e-06
15      4  Comp.   inner  PSu  interface        ...        -2.071e-06
16      5  Comp.   outer   HA   discont.        ...        -2.071e-06
17      5  Comp.   outer   HA   internal        ...        -2.531e-06
18      5  Comp.   outer   HA   internal        ...        -2.992e-06
19      5  Comp.   outer   HA  interface        ...        -3.452e-06

     strain_t  stress_r (Pa/N)  stress_t (Pa/N)  stress_f (MPa/N)
0   5.965e-06       274182.824       378730.663             0.379
1   5.170e-06       237625.114       328233.241             0.328
2   4.374e-06       201067.404       277735.820             0.278
3   3.579e-06       164509.695       227238.398             0.227
4   3.579e-06         9854.181        12915.334             0.013
5   3.181e-06         8759.272        11480.297             0.011
6   2.784e-06         7664.363        10045.260             0.010
7   2.386e-06         6569.454         8610.223             0.009
8   2.386e-06       109673.130       151492.265             0.151
9   7.953e-07        36557.710        50497.422             0.050
10 -7.953e-07       -36557.710       -50497.422            -0.050
11 -2.386e-06      -109673.130      -151492.265            -0.151
12 -2.386e-06        -6569.454        -8610.223            -0.009
13 -2.784e-06        -7664.363       -10045.260            -0.010
14 -3.181e-06        -8759.272       -11480.297            -0.011
15 -3.579e-06        -9854.181       -12915.334            -0.013
16 -3.579e-06      -164509.695      -227238.398            -0.227
17 -4.374e-06      -201067.404      -277735.820            -0.278
18 -5.170e-06      -237625.114      -328233.241            -0.328
19 -5.965e-06      -274182.824      -378730.663            -0.379

[20 rows x 22 columns],
```

```
    layer   side    type matl      label        ...          strain_r  \
0       1  Tens.   outer   HA  interface        ...         3.452e-06
1       1  Tens.   outer   HA   internal        ...         2.992e-06
2       1  Tens.   outer   HA   internal        ...         2.531e-06
3       1  Tens.   outer   HA   discont.        ...         2.071e-06
4       2  Tens.   inner  PSu  interface        ...         2.071e-06
5       2  Tens.   inner  PSu   internal        ...         1.841e-06
6       2  Tens.   inner  PSu   internal        ...         1.611e-06
7       2  Tens.   inner  PSu   discont.        ...         1.381e-06
8       3  Tens.  middle   HA  interface        ...         1.381e-06
9       3  Tens.  middle   HA   internal        ...         4.603e-07
10      3  Comp.  middle   HA   internal        ...        -4.603e-07
11      3  Comp.  middle   HA  interface        ...        -1.381e-06
12      4  Comp.   inner  PSu   discont.        ...        -1.381e-06
13      4  Comp.   inner  PSu   internal        ...        -1.611e-06
14      4  Comp.   inner  PSu   internal        ...        -1.841e-06
15      4  Comp.   inner  PSu  interface        ...        -2.071e-06
16      5  Comp.   outer   HA   discont.        ...        -2.071e-06
17      5  Comp.   outer   HA   internal        ...        -2.531e-06
18      5  Comp.   outer   HA   internal        ...        -2.992e-06
19      5  Comp.   outer   HA  interface        ...        -3.452e-06

     strain_t  stress_r (Pa/N)  stress_t (Pa/N)  stress_f (MPa/N)
0   5.965e-06       274182.824       378730.663             0.379
1   5.170e-06       237625.114       328233.241             0.328
2   4.374e-06       201067.404       277735.820             0.278
3   3.579e-06       164509.695       227238.398             0.227
4   3.579e-06         9854.181        12915.334             0.013
5   3.181e-06         8759.272        11480.297             0.011
6   2.784e-06         7664.363        10045.260             0.010
7   2.386e-06         6569.454         8610.223             0.009
8   2.386e-06       109673.130       151492.265             0.151
9   7.953e-07        36557.710        50497.422             0.050
10 -7.953e-07       -36557.710       -50497.422            -0.050
11 -2.386e-06      -109673.130      -151492.265            -0.151
12 -2.386e-06        -6569.454        -8610.223            -0.009
13 -2.784e-06        -7664.363       -10045.260            -0.010
14 -3.181e-06        -8759.272       -11480.297            -0.011
15 -3.579e-06        -9854.181       -12915.334            -0.013
16 -3.579e-06      -164509.695      -227238.398            -0.227
17 -4.374e-06      -201067.404      -277735.820            -0.278
18 -5.170e-06      -237625.114      -328233.241            -0.328
19 -5.965e-06      -274182.824      -378730.663            -0.379

[20 rows x 22 columns],
    layer   side    type matl      label        ...          strain_r  \
0       1  Tens.   outer   HA  interface        ...         3.452e-06
1       1  Tens.   outer   HA   internal        ...         2.992e-06
2       1  Tens.   outer   HA   internal        ...         2.531e-06
3       1  Tens.   outer   HA   discont.        ...         2.071e-06
4       2  Tens.   inner  PSu  interface        ...         2.071e-06
5       2  Tens.   inner  PSu   internal        ...         1.841e-06
6       2  Tens.   inner  PSu   internal        ...         1.611e-06
7       2  Tens.   inner  PSu   discont.        ...         1.381e-06
```

```
8       3  Tens.  middle   HA  interface        ...         1.381e-06
9       3  Tens.  middle   HA   internal        ...         4.603e-07
10      3  Comp.  middle   HA   internal        ...        -4.603e-07
11      3  Comp.  middle   HA  interface        ...        -1.381e-06
12      4  Comp.   inner  PSu   discont.        ...        -1.381e-06
13      4  Comp.   inner  PSu   internal        ...        -1.611e-06
14      4  Comp.   inner  PSu   internal        ...        -1.841e-06
15      4  Comp.   inner  PSu  interface        ...        -2.071e-06
16      5  Comp.   outer   HA   discont.        ...        -2.071e-06
17      5  Comp.   outer   HA   internal        ...        -2.531e-06
18      5  Comp.   outer   HA   internal        ...        -2.992e-06
19      5  Comp.   outer   HA  interface        ...        -3.452e-06

     strain_t  stress_r (Pa/N)  stress_t (Pa/N)  stress_f (MPa/N)
0   5.965e-06       274182.824       378730.663             0.379
1   5.170e-06       237625.114       328233.241             0.328
2   4.374e-06       201067.404       277735.820             0.278
3   3.579e-06       164509.695       227238.398             0.227
4   3.579e-06         9854.181        12915.334             0.013
5   3.181e-06         8759.272        11480.297             0.011
6   2.784e-06         7664.363        10045.260             0.010
7   2.386e-06         6569.454         8610.223             0.009
8   2.386e-06       109673.130       151492.265             0.151
9   7.953e-07        36557.710        50497.422             0.050
10 -7.953e-07       -36557.710       -50497.422            -0.050
11 -2.386e-06      -109673.130      -151492.265            -0.151
12 -2.386e-06        -6569.454        -8610.223            -0.009
13 -2.784e-06        -7664.363       -10045.260            -0.010
14 -3.181e-06        -8759.272       -11480.297            -0.011
15 -3.579e-06        -9854.181       -12915.334            -0.013
16 -3.579e-06      -164509.695      -227238.398            -0.227
17 -4.374e-06      -201067.404      -277735.820            -0.278
18 -5.170e-06      -237625.114      -328233.241            -0.328
19 -5.965e-06      -274182.824      -378730.663            -0.379

[20 rows x 22 columns]]
```

```
In [135]: cases
```

```
Out[135]: <lamana.distributions.Cases object at 0x000000000B6292E8>, {0: <<class 'lamana.di
```

```
In [136]: #cases.to_csv()                                                    # write to file
```

Unique Cases from Intersecting Caselets

`Cases` can check if caselet is unique by comparing the underlying geometry strings. Here we have a non-unique caselets of different types. We get unique results *within each caselet* using the `unique` keyword. Notice, different caselets could have similar LaminateModels.

```
In [137]: str_caselets = ['350-400-500',  '400-200-800', '400-[200]-800']
          str_caselets2 = [['350-400-500', '350-[400]-500'],
                           ['400-200-800', '400-[200]-800']]
          list_caselets = [['400-400-400', '400-[400]-400'],
                           ['200-100-1400', '100-200-1400',],
                           ['400-400-400', '400-200-800','350-400-500',],
                           ['350-400-500']]
          case1 = la.distributions.Case(dft.load_params, dft.mat_props)
```

```
          case2 = la.distributions.Case(dft.load_params, dft.mat_props)
          case3 = la.distributions.Case(dft.load_params, dft.mat_props)
          case1.apply(['400-200-800', '400-[200]-800'])
          case2.apply(['350-400-500', '400-200-800'])
          case3.apply(['350-400-500', '400-200-800', '400-400-400'])
          case_caselets = [case1, case2, case3]
```

```
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
```

```
In [138]: def process_cases(cases_):
              for i, case in enumerate(cases_):
                  print('Case #: {}'.format(i))
                  for LM in case.LMs:
                      ##print(' {0}: {1:>4}'.format('LaminateModel', LM))   # Python 3.3
                      print(' {0}: {1!r:>4}'.format('LaminateModel', LM)) # Python 3.4+
```

```
In [139]: cases3 = Cases(str_caselets2, unique=True)
          process_cases(cases3)
```

```
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
Case #: 0
 LaminateModel: <lamana LaminateModel object (350.0-[400.0]-500.0), p=5>
Case #: 1
 LaminateModel: <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>
```

```
In [140]: cases3 = Cases(list_caselets, unique=True)
          process_cases(cases3)
```

```
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
Case #: 0
 LaminateModel: <lamana LaminateModel object (400.0-[400.0]-400.0), p=5>
Case #: 1
 LaminateModel: <lamana LaminateModel object (200.0-[100.0]-1400.0), p=5>
 LaminateModel: <lamana LaminateModel object (100.0-[200.0]-1400.0), p=5>
Case #: 2
 LaminateModel: <lamana LaminateModel object (350.0-[400.0]-500.0), p=5>
 LaminateModel: <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>
 LaminateModel: <lamana LaminateModel object (400.0-[400.0]-400.0), p=5>
Case #: 3
 LaminateModel: <lamana LaminateModel object (350.0-[400.0]-500.0), p=5>
```

```
In [141]: cases3 = Cases(case_caselets, unique=True)
          process_cases(cases3)
```

```
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
Case #: 0
 LaminateModel: <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>
Case #: 1
 LaminateModel: <lamana LaminateModel object (350.0-[400.0]-500.0), p=5>
 LaminateModel: <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>
```

```
Case #: 2
 LaminateModel: <lamana LaminateModel object (350.0-[400.0]-500.0), p=5>
 LaminateModel: <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>
 LaminateModel: <lamana LaminateModel object (400.0-[400.0]-400.0), p=5>
```

Gotcha: A single list of individual geometry strings is actually only one case.

```
str_caselets = ['350-400-500', '400-200-800', '400-[200]-800']
```

Since `Cases` is designed to produce multiple cases, it will create a separate subplot for each string by default. When `unique=True`, since individual strings are already unique sets, `Cases` returns that same geometry. No operation is performed, so a warning is prompted.

```
In [142]: cases3 = Cases(str_caselets, unique=True)
          process_cases(cases3)
```

```
Single geometry string detected. unique not applied. See combine=True keyword.
Single geometry string detected. unique not applied. See combine=True keyword.
Single geometry string detected. unique not applied. See combine=True keyword.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
Case #: 0
 LaminateModel: <lamana LaminateModel object (350.0-[400.0]-500.0), p=5>
Case #: 1
 LaminateModel: <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>
Case #: 2
 LaminateModel: <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>
```

If it is desired to combine such a list into one plot, a convenient `combine` keyword is available if `Cases` is already loaded with parameters, but it is best to use `Case`. The `unique` option is available and internally uses the native `Case(unique=True)` keyword.

```
In [143]: cases3 = Cases(str_caselets, combine=True)
          process_cases(cases3)
```

```
User input geometries have been converted and set to Case.
Case #: 0
 LaminateModel: <lamana LaminateModel object (350.0-[400.0]-500.0), p=5>
 LaminateModel: <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>
 LaminateModel: <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>
```

```
In [144]: cases3 = Cases(str_caselets, combine=True, unique=True)
          process_cases(cases3)
```

```
User input geometries have been converted and set to Case.
Case #: 0
 LaminateModel: <lamana LaminateModel object (350.0-[400.0]-500.0), p=5>
 LaminateModel: <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>
```

It is important to note that once set operations are performed, order is no longer a preserved. This is related to how Python handles hashes. This applies to `Cases()` in two areas:

- The `unique` keyword optionally invoked during instantiation.
- Any use of set operation via the `how` keyword within the `Cases.select()` method.

Revamped Idioms

Gotcha: Although a `Cases` instance is a dict, as if 0.4.4b3, it's `__iter__` method has been overriden to iterate the values by default (not the keys as in Python). This choice was decided since keys are uninformative integers, while

the values (curently cases)are of interest, which saves from typing .items() when interating a `Cases` instance.

```
>>> cases = Cases()
>>> for i, case in cases.items()                    # python
>>> ... print(case)
>>> for case in cases:                              # modified
>>> ... print(case)
```

This behavior may change in future versions.

```
In [145]: #------------------------------------------------------------+

In [146]: # Iterating Over Cases
          from lamana.models import Wilson_LT as wlt
          dft = wlt.Defaults()

In [147]: # Multiple cases, Multiple LMs
          cases = Cases(dft.geos_full, ps=[2,5])                    # two cases (p=2,5)
          for i, case in enumerate(cases):                          # iter case values()
              print('Case #:', i)
              for LM in case.LMs:
                  print(LM)

          print("\nYou iterated several cases (ps=[2,5]) comprising many LaminateModels.")
```

```
Caselets not using `combine`.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
Case #: 0
<lamana LaminateModel object (0.0-[0.0]-2000.0), p=2>
Case #: 1
<lamana LaminateModel object (1000.0-[0.0]-0.0), p=2>
Case #: 2
<lamana LaminateModel object (600.0-[0.0]-800.0), p=2>
Case #: 3
<lamana LaminateModel object (500.0-[500.0]-0.0), p=2>
Case #: 4
<lamana LaminateModel object (400.0-[200.0]-800.0), p=2>
Case #: 5
<lamana LaminateModel object (400.0-[100.0,100.0]-0.0), p=2>
Case #: 6
<lamana LaminateModel object (400.0-[100.0,100.0]-800.0), p=2>
Case #: 7
```

```
<lamana LaminateModel object (400.0-[100.0,100.0,100.0]-800.0), p=2>
Case #: 8
<lamana LaminateModel object (0.0-[0.0]-2000.0), p=5>
Case #: 9
<lamana LaminateModel object (1000.0-[0.0]-0.0), p=5>
Case #: 10
<lamana LaminateModel object (600.0-[0.0]-800.0), p=5>
Case #: 11
<lamana LaminateModel object (500.0-[500.0]-0.0), p=5>
Case #: 12
<lamana LaminateModel object (400.0-[200.0]-800.0), p=5>
Case #: 13
<lamana LaminateModel object (400.0-[100.0,100.0]-0.0), p=5>
Case #: 14
<lamana LaminateModel object (400.0-[100.0,100.0]-800.0), p=5>
Case #: 15
<lamana LaminateModel object (400.0-[100.0,100.0,100.0]-800.0), p=5>

You iterated several cases (ps=[2,5]) comprising many LaminateModels.
```

```
In [148]: # A single case, single LM
          cases = Cases(['400-[200]-800'])                    # a single case and LM
          for i, case_ in enumerate(cases):                   # iter i and case
              for LM in case_.LMs:
                  print(LM)

          print("\nYou processed a case and LaminateModel w/iteration.  (Recommended)\n")
```

```
Caselets not using `combine`.
User input geometries have been converted and set to Case.
<lamana LaminateModel object (400.0-[200.0]-800.0), p=5>

You processed a case and LaminateModel w/iteration.  (Recommended)
```

```
In [149]: # Single case, multiple LMs
          cases = Cases(dft.geos_full)                        # auto, default p=5
          for case in cases:                                  # iter case values()
              for LM in case.LMs:
                  print(LM)

          print("\nYou iterated a single case of many LaminateModels.")
```

```
Caselets not using `combine`.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
<lamana LaminateModel object (0.0-[0.0]-2000.0), p=5>
<lamana LaminateModel object (1000.0-[0.0]-0.0), p=5>
<lamana LaminateModel object (600.0-[0.0]-800.0), p=5>
<lamana LaminateModel object (500.0-[500.0]-0.0), p=5>
```

```
<lamana LaminateModel object (400.0-[200.0]-800.0), p=5>
<lamana LaminateModel object (400.0-[100.0,100.0]-0.0), p=5>
<lamana LaminateModel object (400.0-[100.0,100.0]-800.0), p=5>
<lamana LaminateModel object (400.0-[100.0,100.0,100.0]-800.0), p=5>

You iterated a single case of many LaminateModels.
```

Selecting

From cases, subsets of LaminateModels can be chosen. `select` is a method that performs on and returns sets of LaminateModels. Plotting functions are not implement for this method directly, however the reulsts can be used to make new cases instances from which `.plot()` is accessible. Example access techniques using `Cases`.

- Access all cases : `cases`
- Access specific cases : `cases[0:2]`
- Access all LaminateModels : `cases.LMs`
- Access LaminateModels (within a case) : `cases.LMs[0:2]`
- Select a subset of LaminateModels from all cases : `cases.select(ps=[3,4])`

```
In [150]: # Iterating Over Cases
          from lamana.models import Wilson_LT as wlt
          dft = wlt.Defaults()

In [151]: #geometries = set(dft.geos_symmetric).union(dft.geos_special + dft.geos_standard
          #cases = Cases(geometries, ps=[2,3,4])
          cases = Cases(dft.geos_special, ps=[2,3,4])

          # Reveal the full listdft.geos_specia
          # for case in cases:                                          # iter case values()
          #     for LM in case.LMs:
          #         print(LM)

Caselets not using `combine`.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.

In [152]: # Test union of lists
          #geometries

In [153]: cases

Out[153]: <lamana.distributions.Cases object at 0x000000000A846BA8>, {0: <<class 'lamana.di

In [154]: '''Right now a case shares p, size.  cases share geometries and size.'''

Out[154]: 'Right now a case shares p, size.  cases share geometries and size.'

In [155]: cases[0:2]
```

```
Out[155]: [<<class 'lamana.distributions.Case'> p=2, size=1>,
           <<class 'lamana.distributions.Case'> p=2, size=1>]

In [156]: '''Hard to see where these comem from.  Use dict?'''

Out[156]: 'Hard to see where these comem from.  Use dict?'

In [157]: cases.LMs

Out[157]: [<lamana LaminateModel object (0.0-[0.0]-2000.0), p=2>,
           <lamana LaminateModel object (1000.0-[0.0]-0.0), p=2>,
           <lamana LaminateModel object (600.0-[0.0]-800.0), p=2>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=2>,
           <lamana LaminateModel object (0.0-[0.0]-2000.0), p=3>,
           <lamana LaminateModel object (1000.0-[0.0]-0.0), p=3>,
           <lamana LaminateModel object (600.0-[0.0]-800.0), p=3>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=3>,
           <lamana LaminateModel object (0.0-[0.0]-2000.0), p=4>,
           <lamana LaminateModel object (1000.0-[0.0]-0.0), p=4>,
           <lamana LaminateModel object (600.0-[0.0]-800.0), p=4>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=4>]

In [158]: cases.LMs[0:6:2]
          cases.LMs[0:4]

Out[158]: [<lamana LaminateModel object (0.0-[0.0]-2000.0), p=2>,
           <lamana LaminateModel object (1000.0-[0.0]-0.0), p=2>,
           <lamana LaminateModel object (600.0-[0.0]-800.0), p=2>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=2>]
```

Selections from latter cases.

```
In [159]: cases.select(nplies=[2,4])

Out[159]: {<lamana LaminateModel object (500.0-[500.0]-0.0), p=4>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=3>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=2>,
           <lamana LaminateModel object (1000.0-[0.0]-0.0), p=4>,
           <lamana LaminateModel object (1000.0-[0.0]-0.0), p=3>,
           <lamana LaminateModel object (1000.0-[0.0]-0.0), p=2>}

In [160]: cases.select(ps=[2,4])

Out[160]: {<lamana LaminateModel object (600.0-[0.0]-800.0), p=4>,
           <lamana LaminateModel object (600.0-[0.0]-800.0), p=2>,
           <lamana LaminateModel object (0.0-[0.0]-2000.0), p=4>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=2>,
           <lamana LaminateModel object (1000.0-[0.0]-0.0), p=4>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=4>,
           <lamana LaminateModel object (1000.0-[0.0]-0.0), p=2>,
           <lamana LaminateModel object (0.0-[0.0]-2000.0), p=2>}

In [161]: cases.select(nplies=4)

Out[161]: {<lamana LaminateModel object (500.0-[500.0]-0.0), p=3>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=2>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=4>}

In [162]: cases.select(ps=3)
```

```
Out[162]: {<lamana LaminateModel object (1000.0-[0.0]-0.0), p=3>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=3>,
           <lamana LaminateModel object (600.0-[0.0]-800.0), p=3>,
           <lamana LaminateModel object (0.0-[0.0]-2000.0), p=3>}
```

Advanced techniques: multiple selections.

Set operations have been implemented in the selection method of `Cases` which enables filtering of unique LaminateModels that meet given conditions for `nplies` and `ps`.

- union: all LMs that meet either conditions (or)
- intersection: LMs that meet both conditions (and)
- difference: LMs
- symmetric difference:

```
In [163]: cases.select(nplies=4, ps=3)                                  # union; default

Out[163]: {<lamana LaminateModel object (600.0-[0.0]-800.0), p=3>,
           <lamana LaminateModel object (0.0-[0.0]-2000.0), p=3>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=4>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=3>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=2>,
           <lamana LaminateModel object (1000.0-[0.0]-0.0), p=3>}

In [164]: cases.select(nplies=4, ps=3, how='intersection')          # intersection

Out[164]: {<lamana LaminateModel object (500.0-[500.0]-0.0), p=3>}
```

By default, difference is subtracted as `set(ps) - set(nplies)`. Currently there is no implementation for the converse difference, but set operations still work.

```
In [165]: cases.select(nplies=4, ps=3, how='difference')           # difference

Out[165]: {<lamana LaminateModel object (1000.0-[0.0]-0.0), p=3>,
           <lamana LaminateModel object (600.0-[0.0]-800.0), p=3>,
           <lamana LaminateModel object (0.0-[0.0]-2000.0), p=3>}

In [166]: cases.select(nplies=4) - cases.select(ps=3)               # set difference

Out[166]: {<lamana LaminateModel object (500.0-[500.0]-0.0), p=2>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=4>}

In [167]: '''How does this work?'''

Out[167]: 'How does this work?'

In [168]: cases.select(nplies=4, ps=3, how='symm diff')           # symm difference

Out[168]: {<lamana LaminateModel object (1000.0-[0.0]-0.0), p=3>,
           <lamana LaminateModel object (0.0-[0.0]-2000.0), p=3>,
           <lamana LaminateModel object (600.0-[0.0]-800.0), p=3>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=2>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=4>}

In [169]: cases.select(nplies=[2,4], ps=[3,4], how='union')

Out[169]: {<lamana LaminateModel object (600.0-[0.0]-800.0), p=4>,
           <lamana LaminateModel object (600.0-[0.0]-800.0), p=3>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=3>,
           <lamana LaminateModel object (0.0-[0.0]-2000.0), p=4>,
           <lamana LaminateModel object (0.0-[0.0]-2000.0), p=3>,
```

```
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=2>,
           <lamana LaminateModel object (1000.0-[0.0]-0.0), p=4>,
           <lamana LaminateModel object (1000.0-[0.0]-0.0), p=3>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=4>,
           <lamana LaminateModel object (1000.0-[0.0]-0.0), p=2>}

In [170]: cases.select(nplies=[2,4], ps=[3,4], how='intersection')

Out[170]: {<lamana LaminateModel object (1000.0-[0.0]-0.0), p=3>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=4>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=3>,
           <lamana LaminateModel object (1000.0-[0.0]-0.0), p=4>}

In [171]: cases.select(nplies=[2,4], ps=3, how='difference')

Out[171]: {<lamana LaminateModel object (0.0-[0.0]-2000.0), p=3>,
           <lamana LaminateModel object (600.0-[0.0]-800.0), p=3>}

In [172]: cases.select(nplies=4, ps=[3,4], how='symmeric difference')

Out[172]: {<lamana LaminateModel object (600.0-[0.0]-800.0), p=4>,
           <lamana LaminateModel object (600.0-[0.0]-800.0), p=3>,
           <lamana LaminateModel object (0.0-[0.0]-2000.0), p=3>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=2>,
           <lamana LaminateModel object (1000.0-[0.0]-0.0), p=4>,
           <lamana LaminateModel object (1000.0-[0.0]-0.0), p=3>,
           <lamana LaminateModel object (0.0-[0.0]-2000.0), p=4>}
```

Current logic seems to return a union.

Enhancing selection algorithms with set operations

Need logic to append LM for the following:

- all, either, neither (and, or, not or)
  - a, b are int
  - a, b are list
  - a, b are mixed
  - b, a are mixed

```
In [173]: import numpy as np
          a = []
          b = 1
          c = np.int64(1)
          d = [1,2]
          e = [1,2,3]
          f = [3,4]

          test = 1

          test in a
          #test in b
          #test is a
          test is c
          # if test is a or test is c:
          #     True

Out[173]: False
```

```
In [174]: from lamana.utils import tools as ut
          ut.compare_set(d, e)
          ut.compare_set(b, d, how='intersection')
          ut.compare_set(d, b, how='difference')
          ut.compare_set(e, f, how='symmertric difference')
          ut.compare_set(d, e, test='issubset')
          ut.compare_set(e, d, test='issuperset')
          ut.compare_set(d, f, test='isdisjoint')

Out[174]: True

In [175]: set(d) ^ set(e)
          ut.compare_set(d,e, how='symm')

Out[175]: {3}

In [176]: g1 = dft.Geo_objects['5-ply'][0]
          g2 = dft.Geo_objects['5-ply'][1]

In [ ]:


In [177]: cases = Cases(dft.geos_full, ps=[2,5])                    # two cases (p=2,5)
          for i, case in enumerate(cases):                          # iter case values()
              for LM in case.LMs:
                  print(LM)

Caselets not using `combine`.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
User input geometries have been converted and set to Case.
<lamana LaminateModel object (0.0-[0.0]-2000.0), p=2>
<lamana LaminateModel object (1000.0-[0.0]-0.0), p=2>
<lamana LaminateModel object (600.0-[0.0]-800.0), p=2>
<lamana LaminateModel object (500.0-[500.0]-0.0), p=2>
<lamana LaminateModel object (400.0-[200.0]-800.0), p=2>
<lamana LaminateModel object (400.0-[100.0,100.0]-0.0), p=2>
<lamana LaminateModel object (400.0-[100.0,100.0]-800.0), p=2>
<lamana LaminateModel object (400.0-[100.0,100.0,100.0]-800.0), p=2>
<lamana LaminateModel object (0.0-[0.0]-2000.0), p=5>
<lamana LaminateModel object (1000.0-[0.0]-0.0), p=5>
<lamana LaminateModel object (600.0-[0.0]-800.0), p=5>
<lamana LaminateModel object (500.0-[500.0]-0.0), p=5>
<lamana LaminateModel object (400.0-[200.0]-800.0), p=5>
```

```
<lamana LaminateModel object (400.0-[100.0,100.0]-0.0), p=5>
<lamana LaminateModel object (400.0-[100.0,100.0]-800.0), p=5>
<lamana LaminateModel object (400.0-[100.0,100.0,100.0]-800.0), p=5>
```

In order to compare objects in sets, they must be hashable. The simple requirement equality is include whatever makes the hash of a equal to the hash of b. Ideally, we should hash the Geometry object, but the inner values is a list which is unhashable due to its mutability. Conventiently however, strings are not hashable. We can try to hash the geometry input string once they have been converted to General Convention as unique identifiers for the geometry object. This requires some reorganization in `Geometry`.

- [STRIKEOUT:isolate a converter function `_to_gen_convention()`]
- privative all functions invisible to the API
- [STRIKEOUT:hash the converted `geo_strings`]
- [STRIKEOUT:privatize `_geo_strings`. This cannot be altered by the user.]

Here we see the advantage to using geo_strings as hashables. They are inheirently hashable.

UPDATE: decided to make a hashalbe version of the `GeometryTuple`

```
In [178]: hash('400-200-800')
```

```
Out[178]: 4937145087982841834
```

```
In [179]: hash('400-[200]-800')
```

```
Out[179]: -8985357057136576258
```

Need to make `Laminate` class hashable. Try to use unique identifiers such as Geometry and p.

```
In [180]: hash((case.LMs[0].Geometry, case.LMs[0].p))
```

```
Out[180]: 1570369202565922880
```

```
In [181]: case.LMs[0]
```

```
Out[181]: <lamana LaminateModel object (400.0-[100.0,100.0,100.0]-800.0), p=5>
```

```
In [182]: L = [LM for case in cases for LM in case.LMs]
```

```
In [183]: L[0]
```

```
Out[183]: <lamana LaminateModel object (0.0-[0.0]-2000.0), p=2>
```

```
In [184]: L[8]
```

```
Out[184]: <lamana LaminateModel object (0.0-[0.0]-2000.0), p=5>
```

```
In [185]: hash((L[0].Geometry, L[0].p))
```

```
Out[185]: 4636833212297578389
```

```
In [186]: hash((L[1].Geometry, L[1].p))
```

```
Out[186]: 5861696211961991069
```

```
In [187]: set([L[0]]) != set([L[8]])
```

```
Out[187]: True
```

Use sets to filter unique geometry objects from `Defaults()`.

```
In [188]: from lamana.models import Wilson_LT as wlt

          dft = wlt.Defaults()
```

```
          mix = dft.Geos_full + dft.Geos_all

In [189]: mix

Out[189]: [Geometry object (0.0-[0.0]-2000.0),
           Geometry object (1000.0-[0.0]-0.0),
           Geometry object (600.0-[0.0]-800.0),
           Geometry object (500.0-[500.0]-0.0),
           Geometry object (400.0-[200.0]-800.0),
           Geometry object (400.0-[100.0,100.0]-0.0),
           Geometry object (400.0-[100.0,100.0]-800.0),
           Geometry object (400.0-[100.0,100.0,100.0]-800.0),
           Geometry object (0.0-[0.0]-2000.0),
           Geometry object (0.0-[0.0]-1000.0),
           Geometry object (1000.0-[0.0]-0.0),
           Geometry object (600.0-[0.0]-800.0),
           Geometry object (600.0-[0.0]-400.0S),
           Geometry object (500.0-[500.0]-0.0),
           Geometry object (400.0-[200.0]-0.0),
           Geometry object (400.0-[200.0]-800.0),
           Geometry object (400.0-[200.0]-800.0),
           Geometry object (400.0-[200.0]-400.0S),
           Geometry object (400.0-[100.0,100.0]-0.0),
           Geometry object (500.0-[250.0,250.0]-0.0),
           Geometry object (400.0-[100.0,100.0]-800.0),
           Geometry object (400.0-[100.0,100.0]-400.0S),
           Geometry object (400.0-[100.0,100.0,100.0]-800.0),
           Geometry object (500.0-[50.0,50.0,50.0,50.0]-0.0),
           Geometry object (400.0-[100.0,100.0,100.0,100.0]-800.0),
           Geometry object (400.0-[100.0,100.0,100.0,100.0,100.0]-800.0)]

In [190]: set(mix)

Out[190]: {Geometry object (1000.0-[0.0]-0.0),
           Geometry object (400.0-[100.0,100.0]-0.0),
           Geometry object (400.0-[200.0]-800.0),
           Geometry object (400.0-[100.0,100.0,100.0,100.0]-800.0),
           Geometry object (500.0-[250.0,250.0]-0.0),
           Geometry object (500.0-[50.0,50.0,50.0,50.0]-0.0),
           Geometry object (400.0-[100.0,100.0]-800.0),
           Geometry object (600.0-[0.0]-400.0S),
           Geometry object (0.0-[0.0]-1000.0),
           Geometry object (400.0-[100.0,100.0]-400.0S),
           Geometry object (600.0-[0.0]-800.0),
           Geometry object (500.0-[500.0]-0.0),
           Geometry object (400.0-[200.0]-0.0),
           Geometry object (0.0-[0.0]-2000.0),
           Geometry object (400.0-[100.0,100.0,100.0,100.0,100.0]-800.0),
           Geometry object (400.0-[100.0,100.0,100.0]-800.0),
           Geometry object (400.0-[200.0]-400.0S)}
```

2.16.6 Mixing Geometries

See above. Looks like comparing the order of these lists give different results. This test has been quarantine from the repo until a solution is found.

```
In [191]: mix = dft.geos_most + dft.geos_standard                    # 400-[200]-800 common
          cases3a = Cases(mix, combine=True, unique=True)
          cases3a.LMs
```

```
User input geometries have been converted and set to Case.
```

```
Out[191]: [<lamana LaminateModel object (0.0-[0.0]-2000.0), p=5>,
           <lamana LaminateModel object (1000.0-[0.0]-0.0), p=5>,
           <lamana LaminateModel object (600.0-[0.0]-800.0), p=5>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=5>,
           <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>]
```

```
In [192]: load_params['p'] = 5
          cases3b5 = la.distributions.Case(load_params, dft.mat_props)
          cases3b5.apply(mix)
```

```
User input geometries have been converted and set to Case.
```

```
In [193]: cases3b5.LMs[:-1]
```

```
Out[193]: [<lamana LaminateModel object (0.0-[0.0]-2000.0), p=5>,
           <lamana LaminateModel object (1000.0-[0.0]-0.0), p=5>,
           <lamana LaminateModel object (600.0-[0.0]-800.0), p=5>,
           <lamana LaminateModel object (500.0-[500.0]-0.0), p=5>,
           <lamana LaminateModel object (400.0-[200.0]-800.0), p=5>]
```

2.16.7 Idiomatic Case Making

As we transition to more automated techniques, tf parameters are to be reused multiple times, it can be helpful to store them as default values.

```
In [194]: '''Add how to build Defaults()'''
```

```
Out[194]: 'Add how to build Defaults()'
```

```
In [195]: # Case Building from Defaults
          import lamana as la
          from lamana.utils import tools as ut
          from lamana.models import Wilson_LT as wlt

          dft = wlt.Defaults()
          ##dft = ut.Defaults()                                        # user-definable
          case2 = la.distributions.Case(dft.load_params, dft.mat_props)
          case2.apply(dft.geos_full)                                  # multi plies
          #LM = case2.LMs[0]
          #LM.LMFrame
          print("\nYou have built a case using user-defined defaults to set geometric \
          loading and material parameters.")
          case2
```

```
User input geometries have been converted and set to Case.

You have built a case using user-defined defaults to set geometric loading and material pa
```

```
Out[195]: <<class 'lamana.distributions.Case'> p=5, size=8>
```

Finally, if building several cases is required for the same parameters, we can use higher-level API tools to help automate the process.

Note, for every case that is created, a seperate ‘‘Case()‘‘ instantiation and ‘‘Case.apply()‘‘ call is required. These techniques obviate such redundancies.

```
In [196]: # Automatic Case Building
          import lamana as la
          from lamana.utils import tools as ut

          #Single Case
          dft = wlt.Defaults()
          ##dft = ut.Defaults()
          case3 = ut.laminator(dft.geos_full)                          # auto, default p=5
          case3 = ut.laminator(dft.geos_full, ps=[5])                  # declared
          #case3 = ut.laminator(dft.geos_full, ps=[1])                  # LFrame rollbacks
          print("\nYou have built a case using higher-level API functions.")
          case3
```

```
Converting mat_props to Standard Form.
User input geometries have been converted and set to Case.
Converting mat_props to Standard Form.
User input geometries have been converted and set to Case.

You have built a case using higher-level API functions.
```

```
Out[196]: {0: <<class 'lamana.distributions.Case'> p=5, size=8>}
```

```
In [197]: # How to get values from a single case (Python 3 compatible)
          list(case3.values())
```

```
Out[197]: [<<class 'lamana.distributions.Case'> p=5, size=8>]
```

Cases are differentiated by different ps.

```
In [198]: # Multiple Cases
          cases1 = ut.laminator(dft.geos_full, ps=[2,3,4,5])          # multi ply, multi p
          print("\nYou have built many cases using higher-level API functions.")
          cases1
```

```
Converting mat_props to Standard Form.
User input geometries have been converted and set to Case.
Converting mat_props to Standard Form.
User input geometries have been converted and set to Case.
Converting mat_props to Standard Form.
User input geometries have been converted and set to Case.
Converting mat_props to Standard Form.
User input geometries have been converted and set to Case.

You have built many cases using higher-level API functions.
```

```
Out[198]: {0: <<class 'lamana.distributions.Case'> p=2, size=8>,
           1: <<class 'lamana.distributions.Case'> p=3, size=8>,
           2: <<class 'lamana.distributions.Case'> p=4, size=8>,
           3: <<class 'lamana.distributions.Case'> p=5, size=8>}
```

```
In [199]: # How to get values from multiple cases (Python 3 compatible)
          list(cases1.values())
```

```
Out[199]: [<<class 'lamana.distributions.Case'> p=2, size=8>,
           <<class 'lamana.distributions.Case'> p=3, size=8>,
           <<class 'lamana.distributions.Case'> p=4, size=8>,
           <<class 'lamana.distributions.Case'> p=5, size=8>]
```

Python 3 no longer returns a list for `.values()` method, so list used to evalate a the dictionary view. While consuming a case's, dict value view with `list()` works in Python 2 and 3, iteration with loops and comprehensions is a preferred technique for both single and mutiple case processing. After cases are accessed, iteration can access the contetnts of all cases. Iteration is the preferred technique for processing cases. It is most general, cleaner, Py2/3 compatible out of the box and agrees with The Zen of Python:

> There should be one– and preferably only one –obvious way to do it.

```
In [200]: # Iterating Over Cases
          # Latest style
          case4 = ut.laminator(['400-[200]-800'])                    # a sinle case and LM
          for i, case_ in case4.items():                             # iter p and case
              for LM in case_.LMs:
                  print(LM)

          print("\nYou processed a case and LaminateModel w/iteration.  (Recommended)\n")

          case5 = ut.laminator(dft.geos_full)                        # auto, default p=5
          for i, case in case5.items():                              # iter p and case with
              for LM in case.LMs:
                  print(LM)

          for case in case5.values():                                # iter case only with
              for LM in case.LMs:
                  print(LM)

          print("\nYou processed many cases using Case object methods.")
```

```
Converting mat_props to Standard Form.
User input geometries have been converted and set to Case.
<lamana LaminateModel object (400.0-[200.0]-800.0), p=5>

You processed a case and LaminateModel w/iteration.  (Recommended)

Converting mat_props to Standard Form.
User input geometries have been converted and set to Case.
<lamana LaminateModel object (0.0-[0.0]-2000.0), p=5>
<lamana LaminateModel object (1000.0-[0.0]-0.0), p=5>
<lamana LaminateModel object (600.0-[0.0]-800.0), p=5>
<lamana LaminateModel object (500.0-[500.0]-0.0), p=5>
<lamana LaminateModel object (400.0-[200.0]-800.0), p=5>
<lamana LaminateModel object (400.0-[100.0,100.0]-0.0), p=5>
<lamana LaminateModel object (400.0-[100.0,100.0]-800.0), p=5>
<lamana LaminateModel object (400.0-[100.0,100.0,100.0]-800.0), p=5>
<lamana LaminateModel object (0.0-[0.0]-2000.0), p=5>
<lamana LaminateModel object (1000.0-[0.0]-0.0), p=5>
<lamana LaminateModel object (600.0-[0.0]-800.0), p=5>
<lamana LaminateModel object (500.0-[500.0]-0.0), p=5>
<lamana LaminateModel object (400.0-[200.0]-800.0), p=5>
<lamana LaminateModel object (400.0-[100.0,100.0]-0.0), p=5>
<lamana LaminateModel object (400.0-[100.0,100.0]-800.0), p=5>
```

```
<lamana LaminateModel object (400.0-[100.0,100.0,100.0]-800.0), p=5>

You processed many cases using Case object methods.
```

```
In [201]: # Convert case dict to generator
          case_gen1 = (LM for p, case in case4.items() for LM in case.LMs)

          # Generator without keys
          case_gen2 = (LM for case in case4.values() for LM in case.LMs)

          print("\nYou have captured a case in a generator for later, one-time use.")
```

```
You have captured a case in a generator for later, one-time use.
```

We will demonstrate comparing two techniques for generating equivalent cases.

```
In [202]: # Style Comparisons
          dft = wlt.Defaults()
          ##dft = ut.Defaults()

          case1 = la.distributions.Case(load_params, mat_props)
          case1.apply(dft.geos_all)

          cases = ut.laminator(geos=dft.geos_all)
          case2 = cases

          # Equivalent calls
          print(case1)
          print(case2)

          print("\nYou have used classic and modern styles to build equivalent cases.")
```

```
User input geometries have been converted and set to Case.
Converting mat_props to Standard Form.
User input geometries have been converted and set to Case.
<<class 'lamana.distributions.Case'> p=5>
{0: <<class 'lamana.distributions.Case'> p=5, size=18>}

You have used classic and modern styles to build equivalent cases.
```

2.17 LPEP

LamAna Python Enhancement Proposals (LPEP) and Micro PEPs.

See also:

The LPEP types, submission and content guidelines closely follows PEP 0001.

Note: Most Active LPEPs include a Next Action Section.

2.17.1 LPEP 001: Implementing Coding and Package Standards

- **Status: Active**
- **Type: Standards Track**
- **Date: Epoch**
- **Current Version: 0.1**

Standards

This LPEP preserves best practices, standards or customs for develpopers that maintain code consistency. Tne following micro-PEPs are numerically assigned. New micro-PEPs will be added over time or modified with caution.

1. A *General Convention* will be standardized for internal code, such that the inner layer(s) is/are consistently returned as a list of floats i.e. `400.0-[200.0]-800.0` and `400.0-[100.0-100.0]-800.0`. This format is used to maintain type checking consistency within the code. External use by the user input is not bound by this restriction however; shorthand notation is fine too, e.g. `400-200-800`. Such notation will be internally converted to the General Convention.
2. Except for user values such as layer thicknesses and total calculations (microns, um), all other internal, dimensional variables will assume SI units (i.e. meters, m). These values will be converted for convenience for the user in the DataFrames, (e.g. millimeters, mm). This PEP is adopted to limit excessive unit conversions within code.
3. Per PEP 8, semi-private variables are marked with a single preceding underscore, i.e. `_check_layer_order()`. This style is used to visually indicate internal methods/attributes, not particularly important for the user. Double underscores will only be used (sparingly) to prevent name collisions. Internal hook methods with use both trailing and leading underscores, e.g. `_use_model_`.
4. The true lamina thickness value (`t_`) will remain constant in the DataFrame and not vary with height (`d_`).
5. In general, use convenient naming conventions that indicate modules where the objects originates, e.g. `FeatureInput` object. However, whenever possible, aim to use descriptive names that reduce confusion over convienient names, e.g. `LaminateModel` object instead of `ConstructsTheories` object.
6. For compatibilty checks, run nose 2.x and nose 3.x before commits to target Py3to2 errors in tests, (e.g. `dict.values()`).
7. Materials parameters are handled internally as a dict formatted in *Standard Form* (compatible with pandas DataFrames) , but it is displayed as a DataFrame when the materials attribute is called by the user. The Standard form comprises a dict of materials property dicts. By contrast, a *Quick Form* is allowed as input by the user, but interally converted to the Standard Form.
   - Quick Form: `{Material: [Modulus value, Poissons value], ...}`
   - Standard Form: `{'Modulus': {'Mat1': value,...},'Poissons': {'Mat1': value, ...}`
8. Internally, middle layers from `Geometry` return the full thickness, not the symmetric thickness.
9. Thicknesses will be handled this way.
   - t is the total laminate thickness
   - t_k is the thickess at lamina `k`
   - `t_` is the internal variable that refers to true lamina thicknesses.
   - The DataFrame column label $t(um)$ will refer to lamina thicknesses.

- h_ is also a lamina thickness, relative to the neutral axis; therefore middle layers (and h_) are symmeric about the neutral axis $t_{middle} = 2h_{middle}$

10. p=2 give the most critical points to calculate - interfacial minima and maxima per layer. Maxima correlate with the 'interface' label_ and minima correspond to the 'discont.' label_. However, at minimun it is importannt to test with p>=5 to calculate all point types (interfacial, internals and neutural axes) perferably for odd plies.
11. in geometry strings, the dash character – separates layer types outer-inner-middle. The comma , separates other things, such as similar layer types, such as inner_i -[200,100,300]-. The following is an invalid geomtry string '400-[200-100-300]-800'.
12. Two main branches will be maintained: "master" and "stable". "master" will reflect development versions, always ahead of stable releases. "stable" will remain relatively unchanged except for minor point releases to fix bugs.
13. This package will adopt semantic versioning format (MAJOR.MINOR.PATCH). >- MAJOR version when you make incompatible API changes, >- MINOR version when you add functionality in a backwards-compatible manner, and >- PATCH version when you make backwards-compatible bug fixes.
14. Package releases pin dependencies to prevent breakage due to dependency patches/updates. This approach assumes the development versions will actively address patches to latest denpendency updates prior to release. User must be aware that installing older versions may downgradetheir current installs.
15. Use incremented, informative names for tests, e.g. the following says "testing a Case method called "plot" with x feature:

    - test_<class>_mtd_<method name>_<optional feature>#
    - test_<class>_prop_<property name>_<optional feature>#.

    Class tests are ordered as below: - Args: args - Keywords: kw - Attribtutes: attr - Special Methods: spmthd - Methods: mthd - Properties: prop

    Function tests apply similarly, where appropriate. Features are appended and purpose: - test_<func>_<feature 1>_<feature ...>_<purpose>#

Copyright

This document has been placed in the public domain.

2.17.2 LPEP 002: Extending Cases with Patterns

- **Status: Deferred**
- **Type: Process**
- **Date: October 01, 2015**
- **Current Version: 0.4.4b**

Motivation

As of 0.4.4b, a Cases object supports a group of cases distinguished by different ps where each case is a set of LaminateModels with some pattern that relates them. For example, an interesting plot might show multiple geometries of:

- Pattern A: constant total thickness
- Pattern B: constant midddle thickness

In this example, two cases are represented, each comprising LaminateModels with geometries satisfying a specific pattern. Currently `Cases` does not support groups of cases distinguished by pattern, but refactoring it thusly should be simple and will be discussed here. Our goal is to extend the `Cases` class to generate cases that differ by parameters other than `p`.

Desired Ouptut

To plot both patterns together, we need to feed each case seperately to plotting functons. We need to think of what may differ between cases:

- p
- loading parameters
- material properties
- different geometries, similar plies
- number plies (complex to plot simulataneously)
- orientation (not implemented yet)
- ...

Given the present conditions, the most simple pattern is determined by geometry. Here are examples of cases to plot with particular patterns of interest.

```
# Pattern A: Constant Total Thickness
case1.LMs = [<LamAna LaminateModel object (400-200-800) p=5>,
             <LamAna LaminateModel object (350-400-500) p=5>,
             <LamAna LaminateModel object (200-100-1400) p=5>,
            ]

# Pattern B: Constant Middle and Total Thickness
case2.LMs = [<LamAna LaminateModel object (400-200-800) p=5>,
             <LamAna LaminateModel object (300-300-800) p=5>,
             <LamAna LaminateModel object (200-400-800) p=5>,
            ]
```

Specification

To encapsulate these patterns, we can manually create a dict of keys and case values. Here the keys label each case by the pattern name, which aids in tracking what the cases do. The `Cases` dict should emulate this modification to support labeling.

```
cases = {'t_total': case1,
         'mid&t_total': case2,}
```

`Cases` would first have to support building different cases given groups of different geometry strings. Perhaps given a dict of geometry strings, the latter object gets automatically created. For example,

```
patterns = {
    't_total': ['400-200-800', '350-400-500', '200-100-1400'],
    'mid&t_total': ['400-200-800', '300-300-800', '200-400-800'],
}
```

The question then would be, how to label different ps or combine patterns i.e., t_total and ps. Advanced `Cases` creation is a project for another time. Meanwhile, this idea of plotting by dicts of this manner will be beta tested.

Next Actions

- Objective: organize patterns of interest and plot them easily with `Case` and `Cases` plot methods.
  - Refactor Case and Cases to handle dicts in for the first arg.
  - Parse keys to serve as label names (priority).
  - Iterate the dict items to detect groups by the comma and generate a caselets for cases, which get plotted as subplots using an instanace of 'output_.PanelPlot'

See Also

- LPEP 003

Copyright

2.17.3 LPEP 003: A humble case for caselets

- **Status: Replaced**
- **Type: Process**
- **Date: October 05, 2015, March 15, 2016**
- **Current Version: 0.4.4b, 0.4.11**

Motivation

By the final implementation of 0.4.4b, each case will generate a plot based on laminate data given loading, material and geometric information. Single plots are created, but subplots are desired also, where data can be compared from different cases in a single figure. This proposal suggests methods for organizing such plotting data by defining a new case-related term, a `caselet` object and its application to a figure object comprising subplots, based on a [STRIKEOUT:`PanelPlot`] `FigurePlot` subclass.

Definitions

- **LaminateModel** (LM): an object that combines physical laminate dimensions and laminate theory data, currently in the form of DataFrames.
- **case**: a group of LMs; an analytical unit typically sharing similar loading, material and geometric parameters. The final outcome is commonly represented by a matplotlib axes.
- **cases**: a group of cases each differentiated by some "pattern" of interest, e.g. p, geometries, etc. (see LPEP 002).
- **caselet**: (new) [STRIKEOUT:a sub-unit of a case or cases object. Forms are either a single geometry string, list of geometry strings or list of cases.] The final outcome is strongly associated with data pertaining to a matplotlib axes, or subplot component (not an instance or class). (See LPEP 006 for revised definitions)
- **input**: (new) The user arg passed to `Case()` or `Cases()`.

Types of Inputs

The generation of caselet plots as matplotlib subplots requires us to pass objects into `Case(*input*)` or `Cases(*input*)`. To pass in caselet data, the *input* must be a container (e.g. list, tuple, dict, etc.) to encapsulate the objects. The container of any type contain caselets or various types including a string, list or case.

For example, if a list is used, there are at least three options for containing caselets:

1. A list of geometry strings: `type(caselet) == str`
2. A nested list of geometry strings: `type(caselet) == list`
3. A list of cases: `type(caselet) == <LamAna.distributions.Case object>`

If a dict is used to contain caselets, the latter options can substitute as dict values. The keys can be either integers or explict labels.

NOTE: as of 0.4.5, the List will be the default input type of caselets . The dict may or may not be implemented in future versions.

[STRIKEOUT:The following is completed implementation as of v0.4.5.]

Forms of Caselet Inputs

- Container : list or dict Contains the various types that represent cases
- Contained : str, list or str, cases (0.4.11.dev0) Input types that represent, user-defined separate cases.

List of Caselets

Here we assume the input container type is a homogenous list of caselets. The caselets can be either geometry strings, lists of geometry strings or cases.

Caselets as geometry strings (Implemented) The idea behind caselets derives from situations where a user desires to produce a figure of subplots. Each subplot might show a subset of the data involved. The simplest situation is a figure of subplots where each subplot (a caselet) plots a different geometry.

```
>>> import LamAna as la
>>> from LamAna.models import Wilson_LT as wlt
>>> dft = wlt.Defaults()
>>> input = ['400-200-800', '350-400-500', '200-100-1400']
>>> case = la.distributions.Case(dft.load_params, dft.mat_props)
>>> case.apply(input)

Figure of three subplots with different geoemetries.

.. plot::
        :context: close-figs

        >>> case.plot(separate=True)
```

Here the `Case.plot()` method plots each geometry independently in a grid of subplots using a special`separate` keyword. *NOTE: Currently this feature uses ‘‘_multiplot()‘‘ to plot multiple subplots. Future implentation should include ‘‘Panelplot‘‘* The `Cases` class is a more generic way to plot multiple subplots, which does not require a `separate` keyword and handles other caselet types.

```
>>> cases = la.distributions.Cases(input)

Figure of three subplots with different geoemetries.

.. plot::
        :context: close-figs

        >>> cases.plot()
```

Caselets as lists

(Implemented) Another example, if we deisre to build a figure of subplots where each subplot is a subset of a case showing constant total thickness, constant middle thickness, constant outer thickness. We define each subset as a `caselet` and could plot them each scenario as follows:

```
>>> import LamAna as la
>>> list_patterns = [
        ['400-200-800', '350-400-500', '200-100-1400'],
        ['400-200-800', '300-300-800', '200-400-800'],
        ['400-200-800', '400-100-1000', '400-300-600']
    ]
>>> cases = la.distributions.Cases(list_patterns)

Figure of three subplots with constant total thickness, middle and outer.

.. plot::
        :context: close-figs

        >>> cases.plot()
```

Caselets as cases

(Implemented) What if we already have cases? Here is a means of comparing different cases on the same figure.

```
>>> import LamAna as la
>>> list_caselets = [
        ['400-200-800'],
        ['400-200-800', '400-400-400'],
        ['400-200-800', '400-400-400', '350-400-500']
    ]
>>> case1 = la.distributions.Case(dft.load_params, dft.mat_props)
>>> case2 = la.distributions.Case(dft.load_params, dft.mat_props)
>>> case3 = la.distributions.Case(dft.load_params, dft.mat_props)
>>> case1.apply(list_caselets[0])
>>> case2.apply(list_caselets[1])
>>> case3.apply(list_caselets[2])

>>> list_cases = [case1, case2, case3]
>>> cases = la.distributions.Cases(list_patterns)

Figure of three subplots with constant total thickness and different geometries.

.. plot::
        :context: close-figs

        >>> cases.plot()
```

The following will not be implemented in v0.4.5.

Dict of Caselets

Key-value pairs as labeled cases.

(NotImplemented) What if we want to compare different cases in a single figure? We can arrange data for each case per subplot. We can abstract the code of such plots into a new class `PanelPlot`, which handles displaying subplots. Let's extend `Cases` to make a `PanelPlot` by supplying a dict of cases.

```
>>> dict_patterns = {'HA/PSu': case1,
...                  'mat_X/Y': case2,}
>>> cases = la.distributions.Cases(dict_patterns)

Figure of two subplots with three differnt patterns for two laminates with different materials.

.. plot::
        :context: close-figs

        >>> cases.plot()
```

Key-value pairs as labeled lists

(NotImplemented) We could explicitly try applying a dict of patterns instead of a list. This inital labeling by keys can help order patterns as well as feed matplotlib for rough plotting titles. Let's say we have a new case of different materials.

```
>>> dict_patterns = {
...     't_tot': ['400-200-800', '350-400-500', '200-100-1400'],
...     't&mid': ['400-200-800', '300-300-800', '200-400-800'],
...     't&out': ['400-200-800', '400-100-1000', '400-300-600']
... }
>>> new_matls = {'mat_X': [6e9, 0.30],
...              'mat_Y': [20e9, 0.45]}
>>> cases = la.distributions.Cases(
...     dict_patterns, dft.load_params, new_matls
... )

Figure of three subplots with constant total thickness, middle and outer for different materials.

.. plot::
        :context: close-figs

        >>> cases.plot()
```

Key-value pairs as numbered lists

(NotImplemented) We can make a caselets in dict form where each key enumerates a list of geometry strings. This idiom is probably the most generic. [STRIKEOUT:This idiom is currently accepted in `Cases.plot()`.] Other idioms may be developed and implemented in future versions.

```
>>> dict_caselets = {0: ['350-400-500',  '400-200-800', '200-200-1200',
...                      '200-100-1400', '100-100-1600', '100-200-1400',]
...                  1: ['400-550-100', '400-500-200', '400-450-300',
...                      '400-400-400', '400-350-500', '400-300-600'],
...                  2: ['400-400-400', '350-400-500', '300-400-600',
...                      '200-400-700', '200-400-800', '150-400-990'],
...                  3: ['100-700-400', '150-650-400', '200-600-400',
...                      '250-550-400', '300-400-500', '350-450-400'],
...                 }
>>> #dict_patterns == dict_caselets
```

```
>>> cases = la.distributions.Cases(dict_caselets)

Figure of four subplots with different caselets.  Here each caselet represents a different case (not

.. plot::
        :context: close-figs

        >>> cases.plot()
```

Specification

Currently, the specification outlined here is to convert a caselet input into a caselet using a conversion function. Implementation of a formal caselet object are subject to future consideration.

The current application is to feed a `Cases.plot()` method with input which is converted to one of the latter types of caselets. At the moment, type handling for caselets occurs in `Cases()`. This section proposes that type handling for caselets be implemented in the `input_` module instead for general use.

This function will handle processing of various input container types.

```
def to_caselet(input):
    '''Return a Case obect given an input.

    This function accepts each item of a container and processes them into a Case.

    Parameters
    ----------
    input : str, list (of str), case
        This user input becomes a Case object, representing a caselet - a subcomponent
        of other related cases.

    Notes
    -----
    Uses error handling to convert an input into one of the defined caselet types
    str, list of str or case (see LPEP 003).  These caselets derive from homogenous types.

    Heterogenous caselets are not handled, but may be implemented in the future.


    Raises
    ------
    FormatError
        Only a geometry string, homogenous list of geometry strings or case is accepted.

    Returns
    -------
    Case object
        Integer-case, key-value pairs.

    '''
    try:
        # Assuming a list of geometry strings
        case_ = la.distributions.Case(self.load_params, self.mat_props)
        if unique:
            case_.apply(input, unique=True)
        else:
            case_.apply(input)
```

```
        self.caselets = [case_]
        # TODO: Brittle; need more robust try-except
    except(AttributeError, TypeError):             # raised from Geometry._to_gen_convention()
        try:
            # If a list of lists
            flattened_list = list(it.chain(*caselets))
            # lists are needed for Cases to recognize separate caselets
            # automatically makes a unique set
            #print(caselets)
            # TODO: what else is _get_unique doing?
            ##self.caselets = [self._get_unique(flattened_list)]
            #print(self.caselets)
        except(TypeError):
            # if a list of cases, extract LMs, else raise
            flattened_list = [LM.Geometry.string for caselet in caselets
                              for LM in caselet.LMs]
            # list is needed for Cases to recognize as one caselet
            # automatically makes a unique set
            ##self.caselets = [self._get_unique(flattened_list)]
            #print(self.caselets)
     raise FormatError('Caselet type is not accepted.  Must be str, list of strings or case') #?
```

```
'''
Need to iterate caselets (lists of objects) to package the order of the data.
Then pass that data into the plot functions.  Plot functions should simply
make an axes for each data unit, then return an ax (for singleplot) or figure
(for multiplot).

1. Case only need a single list of input because it only handles one case/time.
2. Cases takes multiple lists or case objects
   - may require separating a caselet into cases bases on what's given.

A Caselets object should accept either number or inputs.  Should rearrange caselets.
Should return a rearrange caselet input.  If this self is passed in, the order
of cases should be preserved

'''
```

Next Actions

- Objective: Make abstract `PanelPlot` class that accepts dicts of LMs for cases to output figures of caselets or cases.
  - build `PanelPlot` which wraps matplotlib subplots method.
  - inherit from `PanelPlot` in `Case.plot()` or `Cases.plot()`
  - implement in `output_`
  - make plots comparing different conditions in the same `Case` (caselets)
  - [STRIKEOUT:make plots comparing different cases using `Cases`]
- Abstract idiom for building caselets accepted in `Cases.plot()`.
- Implement general caselet converter, error-handler in `input_`
- Make a caselets class.

- Revise LPEP to accept LM or LMs as caselet types; refactor `to_caselet` to handle these types. See `output_._multiplot`, which defines caselet differently.

See Also

- LPEP 002

Copyright

This document has been placed in the public domain.

2.17.4 LPEP 004: Refactoring class `Stack`

- **Status: Draft**
- **Type: Process**
- **Date: October 20, 2015, March 17, 2016 (revised)**
- **Current Version: 0.4.4b1, 0.4.11**

Motivation

Inspired to adhere to classic data structures, we attempt to refactor some classes. The present `la.constructs.Stack` class is not a true stack. Athough built in a LIFO style, there are no methods for reversing the stack. It may be beneficial to the user to add or delete layers on the fly. Stacks, queues and other data structures have methods for such manipulations. Here are some ideas that entertain this train of thought.

Desired Output

- Insert and remove any layers
- Access geometry positions in an index way

Specification

- Make stacks from deques
- Extend Stack to interpret from geometry strings also

Examples

```
>>> LM = la.distributions.Cases('400-200-800').LMs
>>> LM.insert('[:,100]')          # eqv. ':-[:,100]-:'
>>> print(LM.geometry, LM.nplies)
<Geometry object (400-[200,100]-800)>, 7

>>> LM.remove('middle')
>>> print(LM.geometry, LM.nplies)
<Geometry object (400-[200,100]-0)>, 6
```

```
>>> LM.remove(['outer', 'inner'])
StackError 'If inner layers are removed, outer layers must exist.'
```

Next Actions

- Write specification for using deques
- Write specification for implementing geo_string interpretation.
- Develop the idea of duples in tandem

See Also

- `analyze_geostrings()`: interpret strings nplies, thickness, order.

Copyright

This document has been placed in the public domain.

2.17.5 LPEP 005: Making Concurrent `LaminateModels` with the new `asyncio`

- **Status: Draft**
- **Type: Process**
- **Date: February 23, 2016**
- **Current Version: 0.4.10**

Motivation

The idea of concurrency offers a potential option for improving creation of LamAna objects. For instance, if 10 `LaminateModels` are called to be made, rather then waiting for each object to instantiate serially, it may be better to create them in parallel. This proposal is entertains current object creation using concurrency, and it is adapted from this simple, well written set of examples of coroutines and chained coroutines.

Definitions

- G : Geometry object
- FI : FeatureInput object
- St : Stack
- Sp : Snapshot
- L : Laminate
- LM : LaminateModel

When `la.distributions.Case.apply()` is called, the `get_LaminateModel()` function creates a generated list of LaminateModels. A series of objects a created accessing 3 core modules.

$$\left[G_{input\_} \rightarrow FI_{distributions-input\_}\right] \longrightarrow \left[St \rightarrow Sp \rightarrow L \rightarrow LM\right]_{constructs}$$

When `apply()` is called, it has to **wait** for other serial processes to finish in a certain **order** before completing. These characteristics of waiting on ordered processes **may** qualify the LamAna architecture as a candidate for concurrency features in the new Python 3.5 `asyncio` module.

Implemented Chained Coroutines

We attempt to apply these concepts to LamAna. A summary of the main coroutine is outlined below.

```
import asyncio

async def get_LaminateModel(geo_string):
    '''Run set of processes in order to give finally create a LaminateModel from a geo_string.'''
    # conv_geometry converts a geo_string to general convention
    # TODO: include geo_string caching

    # TODO: comvert these objects to coroutines (or keep as generators?)
    G = await = la.input_.Geometry(conv_geomtry)
    FI = await la.input_.BaseDefaults.get_FeatureInput(G, **kwargs)      # rewrite FeatureInput
    St = await la.constructs.Stack(FI)
    Sp = await la.constructs.Snapshot(St)                                # snapshot
    L = await la.constructs.Laminate(Sp)                                 # LFrame
    LM = await la.constructs.Laminate(L)                                 # LMFrame

# The main event loop
event_loop = asyncio.get_event_loop()
for geo_string in geo_strings:                                           # unsure if this would work
    try:
        # Consider alternatives to this default loop
        laminate_model = event_loop.run_until_complete(get_LaminateModel(geo_string))
    finally:
        event_loop.close()
```

NOTE: It is unclear how to advance the geo_strings iterable object in the default asyncio loops

Vetting

Pros:

- Possbible concurrency, multitasking of LaminateModel creation
- Clear, explicit illustration of order

Cons:

- Limited to Python 3.5

Next Actions

- Look into advancing iterables in an asynio default loop
- Use mock objects to test this LPEP as proof of concept

Copyright

This document has been placed in the public domain.

2.17.6 LPEP 006: Defining LamAna Objects

- **Status: Draft**
- **Type: Informational**
- **Date: March 17, 2016**
- **Current Version: 0.4.11**

Motivation

This LPEP is written to clarify certain types used within LamAna documentation and codebase.

Definitions

Laminate classifications

- **Symmetric**: a laminate with symmetry across the neutral axis.
- **Asymmetric**: non-symmetry across the netural axis.

Representations

- **geo_string** (g): a geometry string typically formatted to general convention (see LPEP 001)
- **Geo_object** (G): a `Geometry` object, an instance of the Geometry class
- **Geo_orient** (GO) (NotImplemented): a `GeoOrient` object, containing in-plane, directional, ply-angle information

Layer types (lytpe)

- **outer**: the top and bottom-most layers
- **inner_i**: a list (or string representation) of all inner layer thicknesses; inners refers to a subset of inner_i
- **inner**: an internal, non-middle, non-outer layer
- **middle**: for odd plies, the center layer; for symmetric laminates, this layers passing through the neutral axis

Geometry string containers

Pythonic objects used to signify groups of layer thickness:

- **list**: a pythonic list of inner layers, e.g. [100, 100, 50]. Each entry represents equivalent layer thicknesses for both tensile and compressive sides.
- **token**: pertaining to one of the layer types
- **duple** (NotImplemented): a tuple of dual layer thicknesses for corresponding (tensile, compressive) layers, e.g. (100,300). Each entry represents a significant thickness of a tensile/compressive side for a given layer type. Zero is also not allowed (0,400). A duple replaces one of the thickness positions in a geometry string. The sum of a duple contributes to the total laminate thickness. By definition, duples are only used to specify asymmetric geometries, therefore repeated values are disallowed e.g. (400,400). Also, since middles are singular constructs, duples are disallowed for middle layers.

Geometry strings

Regular geometry strings: a simple, *symmetric* stacking sequence of outer, inner_i and middle layers. e.g.

```
- '400-[200]-800'                                    # simple
- '400-[150,50]-800'                                 # inner_i
```

These strings follow a simple algorithms for calculating layer thicknesses:

$$t_{total,outer} = 2t_{outer}$$

$$t_{total,inner} = 2t_{inner_i}$$

$$t_{total,inner_i} = 2\sum_{i}^{m} t_{inner}$$

$$t_{total} = 2(t_{outer} + t_{inner_i}) + t_{middle}$$

$$n_{plies} = 2(n_{outer} + n_{inner_i}) + n_{middle}$$

Irregular geometry strings: includes *assymmetric* laminates; involves

```
- '(300,100)-[150,50]-800'                           # outer duple
- '400-[150,(75,50),25]-800'                         # inner duple
- '(300,100)-[150,(75,50),25]-800'                   # outer and inner duple
```

These strings can follow more complex algorithms for calculating layer thickness. For every ith item in the list of inner_i and jth index within an i (duple or non), where m is the end of the squence and $C = 1$ for duples and $C = 2$ for non-duples:

$$t_{total,outer} = C\sum_{i}^{m}\sum_{j}^{m=2} t_{outer}$$

$$t_{total,inner} = C\sum_{j}^{m_j} t_{inner}$$

$$t_{total,inner_i} = C\sum_{i}^{m}\sum_{j}^{m_j} t_{inner}$$

$$t_{total} = t_{outer} + t_{inner_i} + t_{middle}$$

$$n_{plies} = C_1\sum_{i}^{m_i}\sum_{j}^{m_j} n_{outer} + C_2\sum_{i}^{m_i}\sum_{j}^{m_j} n_{inner} + n_{middle}$$

Data structures

Conceptual structures used to represent groups of data:

- **packet**: a user-defined input data, e.g. a list of geometry strings. This group are structured according to some pattern of interest (see LPEP 002). A packet may become processed datasets (LMs) encased within a `Case` object.
- **packets**: a group of packets, units that represent separated cases, e.g. a list of lists comprising geometry strings. These groups are ordered according to desired output.
- **stack**: the bottom-to-top (tensile-to-compressive) stacking sequence of laminated layers. Represented as lists, deques or other pertinent data structures. Regular stacks reverse inner_i order post middle layer. Irregular stacks must parse duple indices, tensile and compressive.
  - Regular stack: '400-[150,50]-800' –> [400.0, 150.0, 50.0, 800.0, 50.0, 150.0, 400.0]
  - Irregular stack: '400-[(150,50)]-800' –> [400.0, 150.0, 800.0, 50.0, 400.0]
- **LaminateModel** (LM): an object that combines physical laminate dimensions and laminate theory data, currently in the form of DataFrames.
- **case**: an analytical unit typically sharing similar loading, material and geometric parameters. This object contains a group of LMs; the final outcome is commonly represented by a matplotlib axes.
- **cases**: a group of cases each differentiated by some "pattern" of interest, e.g. p, geometries, etc. (see LPEP 002). The final product is commonly represented as a matplotlib Figure. Each group is a smaller case "caselet"
- **caselet**: a samller case that is related to a larger group of cases. This conceptual unit is finally manifested as a matplotlib subplot.

Plotting objects

- figureplot: a matplotlib Figure with attributes for quick access to plot objects. A base class for setting figure options and appearance, akin to a seaborn FacetGrid
- singleplot: a single matplotlib axes.
- multiplot: a figure of singleplots represented in subplots.
- feature plot: a LamAna plotting object based on a specific feature module e.g. `DistribPlot`

Examples

```
Analyzed string Information

Number of plies, total laminate thickness and stacking order
(nplies, t_total, order)

General Convention
'400.0-[100.0,100.0]-800.0'
# (7, 2.0, [400.0,100.0,100.0,800.0,100.0,100.0,400.0])

Duple
'(300.0,100.0)-[(50.0, 150.0),100.0]-800.0
# (7, 1.6, [300.0,50.0,100.0,800.0,100.0,150.0,100.0])
```

Next Actions

- Swap definitions of "inner_i" and "inner".

See Also

- LPEP 001: General Convention

Copyright

This document has been placed in the public domain.

2.17.7 LPEP 007: Redesigning the `distributions` Plotting Architecture

- **Status: Draft**
- **Type: Process**
- **Date: April 05, 2016**
- **Current Version: 0.4.11**

Motivation

The plotting functions were quickly put together prior to LamAna's offical release. This original plotting architecture lacks robustness and scalability for future feature modules. The current version of `Case.plot()` and `Cases.plot()` methods use non-public functions located the `output_` module for plotting single axes figures ("single plots") and multi-axes figures ("multi plots"). The purpose of this proposal is to lay out a robust, lucid architecture for plotting `distributions` and future feature module outputs.

Desired Ouptut

...

Definitions

See LPEP 007 for formal definitions.

Basic Plot Options

The following objects associate with lower level matplotlib objects:

- singleplot, multiplot, figureplot

The following definition pertains to a unique LamAna objects that inherits the latter objects:

- DistribPlot: a class that handles the output of a `distributions` plot.

Specification

A `DistribPlot` should be given `LamainateModels`. While iterating over `LaminateModels`, information is extracted (e.g. nplies, p) and axes are generated both combining plotting lines and separating unique laminates under various conditions. THis class should inherit from a base that controls how a figure appears. Through iterating the given argument, this class should determine whether the resulting figure should be a singleplot or multiplot. Here is a sample signature for the `Distriplot`.

```
import lamana as la

class _FigurePlot(object):
    '''Return a matplotlib Figure with base control.'''
    def __init__(self):
        self.nrows = 1
        self.ncols = 1

        fig, ax = plt.subplots(self.nrows, self.ncols)
        self = fig
        self.axes = ax
        self.naxes = len(ax)
        self.x_data = fig.axes.Axes[0]
        self.y_data = fig.axes.Axes[1]
        #self.patches = extract_patches()

    def update_figure():
        '''Update figure dimensions.'''
        pass
    pass


class DistribPlot(_FigurePlot):
    '''Return a distributions FigurePlot.

    This class needs to process LaminateModels and honor the user-defined packages.

    Parameters
    ----------
    cases_ : Case or Cases object
        The self object of the Case and Cases classes.  Relies on the pre-ordered
        arrangement of the user-defined, package input.
    kwargs : dict
        Various plotting keywords.

    See Also
    --------
    Entry points
    - lamana.distributions.Case.apply: primarily singleplots unless separated
    - lamana.distributions.Cases: primarily multiplots unless combined

    '''
    def __init__(self, cases_, **kwargs):
        super(DistribPlot, self).__init__(cases_, **kwargs)
        self = self.make_fig(cases_)
        self.packages = cases_.packages                    # NotImplemented


    # Temporary
    # TODO: these plotters need to be abstracted from distributions code.
```

```
    def _singleplot(self, case):
        singleplot = la.output_._distribplot(case.LMs)
        return singleplot

    def _multiplot(self, cases):
        multiplot = la.output_._multiplot(cases)
        return multiplot

    def make_fig(self, cases_ordered):
        '''Return a figure given cases data.

        Parameters
        ----------
        cases_ordered : Case- or Cases-like
            Contains data required to generate the plots.  Assumes the cases
            preserve the the user-defined order at onset in the caselet_input.

        '''
        if isinstance(cases_ordered, la.distributions.Case):
            # Give a single Case object
            case = cases_ordered
            fig = plt.figure()
            ax = self._singleplot(case)
            fig.axes.append(ax)
        elif isinstance(cases_ordered, la.distributions.Cases:
            # Give a Cases object
            fig = self._multiplot(cases_ordered)
            #plt.suptitle()
            #plt.legend()
        else:
            raise TypeError(
                'Unknown distributions type was pass into {}.'.format(self.__class__)
            )

        return fig


# Mock Implementations ----------------------

# Handles singleplots from Case
def Case.plot(self, **kwargs):
    return la.output_.DistribPlot(*args, **kwargs)


# Handles multiplots from Cases
def Cases.plot(self, **kwargs):
    return la.output_.DistribPlot(*args, **kwargs)
```

Examples

```
Singleplots

>>> case = Case(['400-200-800', '400-400-400'])
>>> singleplot = cases.plot()
<matplotlib Figure>
>>> multiplot.naxes
1
```

```
Multiplots

>>> cases = Cases([['400-200-800', '400-400-400'], ['100-100-1600']])
>>> multiplot = cases.plot()
<matplotlib Figure>
>>> multiplot.naxes
2

>>> multiplot.axes
[<maplotlib AxesSupbplot>, <maplotlib AxesSupbplot>]
>>> multiplot.packages                                    # orig. input
[['400-200-800', '400-400-400'], ['100-100-1600']]
```

Vetting

Next Actions

- Add more attributes as ideas come about
- Implement and test in future versions; revise `Case`, `Cases` and add `Packages`
-

See Also

- LPEP 002: on patterns
- LPEP 003: packets; a revised form of caselets
- LPEP 006: unoffical glossary
- LPEP: 008 Formalizing Packets

Copyright

This document has been placed in the public domain.

2.17.8 LPEP 008: Formalizing `Packets`: input data for caselets

- **Status: Draft**
- **Type: Process**
- **Date: April 07, 2016, (Rev. 04/10/16)**
- **Current Version: 0.4.11**

Motivation

Multiplots require *structured data* to display plots correctly. Ulitmately this product requires data that has be validated and organized in a simple and discernable format. The `Packets` class is in `input_` datastructure that attempts to funnel various input type into a simple that that object after processing inputs as follows:

- *validate* the geometry strings and input formats

- *reformat* geometry data to according to internally-accepted, Generation Conventions.
- *reorder* or rearrange data if exceptions are raised
- *analyze* geometry data

The user can now focus on arranging the data into analytical sub-groups (caselets). This information new, restructured data is supplied to feature module objects such as `Case` or `Cases` that package the data according to the user-defined order. Most importantly, plotting functions can simply iterate over the structured data an output plots that reflect this order.

NOTE: the ''Packets'' object was alpha coded in 0.4.11.dev0

Desired Ouptut

An enumerated dict of packet inputs.

This class should focus on cleaning and organizing the data for a feature module function. *Let Case and Cases handle the data.*

Definitions/Keywords

Terms such as *caselet* and *packet* have been developed during the planning phases of defining and refactoring `output_` objects into a logical, scaleable framework. See *LPEP 006* for formal definitions.

- packet, packets, LaminateModel, caselet, case, cases

Specification

The simplest approach to ordering data is to handling all incoming inputs upfront. The packets can them be funneled into a clean, restructured form. As of 0.4.11, we introduced the `Packet` class, intended to convert packet inputs in said object.

Error handling is important for certain scenarios. For example, given a list of geometry strings, a separate caselet must be generated when:

```
1. The current nplies does not match the nplies in the current axes
2. Another set of ps is discovered
```

As `Packets` must handle such an event by analyzing the raw geometry strings upfront. Packet requirement may vary for different feature modules.

```
class Packets(object):
    '''Return a Packets object by processing user-defined packet inputs.

    This class is an interface for converting various inputs to a formal datastructure.
    It serves to precess inputs as follows:
    - validate: geo_strings are a valid and interpretible
    - reformat: convert a geo_string to General Convention
    - reorder: split unequal-plied geo_strings in separate caselets (if needed)
    - analyze: assessed for duples (NotImplemented)

    This class also handles unique exceptions to form new, separate packets based on various conditio

    1. The current nplies does not match the nplies in the current axes
    2. Another set of ps is discovered
    3. ...
```

```
    Parameters
    ----------
    packet : list
        User input geometry strings; dictates how caselets are organized.

    Returns
    -------
    dict
        Fast, callable, ordered.  Contains int-caselet input, key-value pairs.

    See Also
    --------
    - LPEP 003: original ideas on caselets

    Notes
    -----
    Due to many container types, this class will be gradually extended:
    - 0.4.12.dev0: supports the list container of str, lists of strs and cases.

    Examples
    --------
    See below.

    '''
    def __init__(self, packets):
        self.packets = self.clean(packets)
        self = to_dict(self.packets)

        self.nplies = None
        self.t_total = None
        self.size = len(self.packets)

    def clean(packets):
        '''Return an analyzed reformatted, validated, orderd list of packets.

        Exploits the fine-grain iteration to extract geo_string data
        via analyze_string().

        '''
        # Handle analyses and converting geo_strings to General Convention

        if self.nplies is None:
            self.nplies = {}
        if self.t_total is None:
            self.t_total = {}

        nplies_last = None
        caselet_conv = []

        for packet in packets:
            caselet_new = []
            for geo_string in packet:
                # Validate
                if is_valid(geo_string):
                    # Reformat: Should raise error if invalid geo_string
                    geo_string_conv = la.input_.to_gen_convention(geo_string)
                    # Analyze: extract geo_string data while in the loop
```

```
                    nplies, t_total, _ = la.input_.analyze_geostring(geo_string)

                # Store analyzed data in attributes
                self.nplies.add(nplies)
                self.t_total.add(t_total)

                # Reorder: make new list for unequal nplies
                if nplies != nplies_last:
                    geo_string_conv = list(geo_string_conv)

                caselet_new.append(geo_string_conv)
                nplies_last = nplies

            # Ship for final handling and formatting
            if len(caselet_new) == 1:
                return self._handle_types(caselet_new)
            else:
                packets_conv.append(caselet_new)
            return self._handle_types(packets_conv)

    def _handle_types(self):
        '''Return the accepted packets format given several types.

        As of 0.4.11, the list is the only accpeted objecct container. At this
        entry point, users should not be aware of Case or LaminateModels,
        but they included anyway.

        '''
        # Forward Compatibility ----------------------------------------
        # List of Case objects
        # [case_a, case_b, ...] --> [['geo_str1', 'geo_str2'], ['geo_str1'], ...]


        # A Single Case
        # [case] or case --> ['geo_str1', 'geo_str2', 'geo_str3']


        # List of LaminateModels (LMs)
        # [<LM1>, <LM2>, ...] --> [['geo_str1', 'geo_str2'], ['geo_str1'], ...]


        # A Single LaminateModel (LM)
        # [LM] or LM --> [['geo_str1', 'geo_str2', 'geo_str3'], ...]


        # --------------------------------------------------------------
        # List of lists or geo_strings
        # [['geo_str1', 'geo_str2'], ['geo_str1'], ...] --> _


        # List of geo_strings
        # ['geo_str1', ...] --> _


        # Single geo_string
        # ['geo_str1'] or 'geo_str1' --> ['geo_str1']
```

```
        except(AttributeError) as e:
            raise FormatError(
            'Caselet input () is an unrecognized format.'
            ' Use a list of geo_strings'.format(e)
            )

        pass

    def to_dict(self)
        '''Return an enumerated dict of packets.'''
        dict_ = ct.defaultdict(list)
        for i, caselet in enumerate(self.packets):
            dict_[i] = caselet

        return dict_

    def to_list(self):
        '''Return lists of packets.'''
        pass

    @property
    def info(self):
        '''Return DataFrame of information per caselet.'''
        pass
```

Examples

```
Boilerplate

>>> import lamana as la
>>> from lamana.input_ import Packets
>>> from lamana.models import Wilson_LT as wlt
>>> dft = wlt.Defaults()

Usage: A packet --> a future case and a singleplot (one axes)

>>> packet = Packets(['400-200-800', '400-400-400'])
>>> packet
<lamana Packets object, `distribution`, size=1>
>>> packet()
{0:  ['400-200-800', '400-400-400']}
>>> case = la.distributions.Case(dft.load_params, dft.mat_props)
>>> case.apply(packet)
>>> singleplot = case.plot
>>> singleplot.naxes
1

Usage: Packets --> a group of cases (caselets) --> multiplot (n>1 axes)

>>> packets = Packets([['400-200-800', '400-400-400'], ['400-0-1200']])
>>> packets()
{0:  ['400-200-800', '400-400-400'],
 1: ['400-0-1200']}
>>> cases = la.distributions.Cases(packets)            # assumes default parameters and properties
>>> singleplot = case.plot
>>> singleplot.naxes
2
```

```
Handling: if unequal plies are found, a new packet is generated automatically

>>> str_packets = [                                    # should be one caselet
... '400-200-800', '400-400-400',                      # but nplies=5
... '400-0-1200'                                       # and nplies=3; cannot plot together
]
>>> packets = Packets(str_packets)
Using default distributions objects.
Unequal nplies found.  Separating...
>>> packets()
{0: ['400-200-800', '400-400-400'],
 1: ['400-0-1200']}
>>> packets.size
2
>>> packets.nplies
[5, 3]                                                  # ordered by input position
>>> packets.info                                        # pandas DataFrame
   nplies  p  contained
0  5       5  '400-200-800', '400-400-400'
1  3       5  '400-0-1200'

Feature: For a distributions `Case` or `Cases` object --> stress distribution

>>> packets = Packets(['400-200-800', '400-400-400'], feature='distributions')
>>> packets
<lamana Packets object `distributions`, size=1>

Feature: For a predictions module object (NotImplemented) --> regression plot

>>> packets = Packets(['400-200-800', '400-400-400'], feature='predictions')
>>> packets
<lamana Packets object `predictions`, size=1>

Feature: For a ratios module (NotImplemented) --> layer group in a ratio plot

>>> packets = Packets(['400-200-800', '400-400-400'], feature='ratios')
>>> packets
<lamana Packets object `ratios`, size=1>
```

Vetting

Benefits: - This approach handles all analyses, conversions,

validations, and reorderings (e.g. nply separation) of user input data. | - It feeds a consistent form to `Case` and `Cases` - Off loads the need to figure out what kind of caselet should be made. - Preprocesses with light, strings and lists. - Can later use in conjunction with a some startup functions e.g. `Start` to simplify user API. - Handle future input types e.g. `GeoOrient` object.

Next Actions

- Develop post 0.4.11.
- Implement the General Convention strings.
- Implement the ordering algorithms.

- Implement the isvalid method.
- Implement into `input_` module; refactor distributions `Case` and `Cases` to accept `Packets`. Remove redundant code.

See Also

- LPEP 003: original ideas on caselets
- LPEP 007: plotting redesign

Copyright

This document has been placed in the public domain.

2.17.9 LPEP 009: Revisiting Entry Points

- **Status: Draft**
- **Type: Process**
- **Date: April 07, 2016**
- **Current Version: 0.4.11**

Motivation

The user input can be complex and difficult to predict. Additionaly, the user should not be bothered with the following:

1. Worrying about which type to use as an entry point e.g. `Case` or `Cases`
2. Remembering to `apply` as in `Case.apply`
3. Worrying about particular signatures for each feature module.

As feature modules are added, the entry points to LamAna increase while also broading the signature for caselets. This broadening may become confusing over time. The purpose of this proposal is to mitigate the user responsibility in setting up boilerpoint and focus on analysis.

Desired Ouptut

After supplying caselet information, prompt the user with information it requires per feature module, e.g. load_params or mat_props.

Definitions

Specification

Examples

```
>>> # Geometries to analyze
>>> caselet = ['400-200-800', '400-400-400']
>>> # What kind of analysis?
>>> la.input_.Start(caselet, feature='distributions')
... Please supply loading paramaters.  Press Enter to use defaults.
... Please supply material properties.  Press Enter to use defaults.
... Please supply laminate theory model.  Press Enter to use defaults.
Using default load_params and mat_props...
Using Wilson_LT model...
Done.
[<lamana Case object size=1, p=5>]
```

Vetting

Next Actions

- Design an object that routes user to specific feature module objects and prompts for necessary data.

See Also

Copyright

This document has been placed in the public domain.

2.17.10 LPEP 010: Decoupling `LaminateModels` from `Laminate`

- **Status: Draft**
- **Type: Standards Track**
- **Date: May 30, 2016**
- **Current Version: 0.4.11**

Motivation

The `LaminateModel` object is not a class, but it is rather a DataFrame object assigned to an instance attribute of the `Laminate` class. The implementation was originally intended to reduce class objects creation (reducing memory), encourage simplicity and likely reduce the number of looping operations for populating DataFrame rows and columns. However, this implicit architecture of the clandestine `LaminateModels` can lead to misunderstanding when trying to track the flow of objects. In addition, during refactoring the `theories` objects, passing a pure `Laminate` object into the `theories.handshake()` has proven is impossible at the moment.

In effort to access separate objects and for clarity, this proposal maps out a plan to decouple `LaminateModel` from `Laminate` as a seprate object through subclassing.

Desired Ouptut

A `LaminateModel` object that inherits from `Laminate` and `Stack`.

Specification

1. Given a `FeatureInput`, create `LaminateModel`.
2. If Exception raised, return a `Laminate` object.
3. Update tests expecting `Laminate` to return `LaminateModel`
4. Duplicate `_build_laminate` to `_build_primitive`; merge former with Phase 2.

The latter objects should be achieved by extracting Phase 3 `_update_calucations` into `LaminateModel`. For cleanup, we can separate object parsing attributes into their associated objects. We can then serially call lower level objects to make the final product.

`LaminateModel(Laminate(Stack))`

Examples

```
# Create Separate Objects
>>> S = la.constructs.Stack(FI)
>>> S
<Stack object>
>>> L = la.contstructs.Laminate(FI)
>>> L
<Laminate object 400-[200]-800>
>>>LM = la.constructs.LaminateModel(FI)
>>> LM
<LaminateModel object 400-[200]-800>

>>> # Attribute Inheritance
>>> S_attrs = ['stack_order', 'nplies', 'name', 'alias']
>>> all([hasattr(S, attr) for attr in S_attrs])
True
>>> L_attrs = ['FeatureInput', 'Stack', 'Snapshot', 'LFrame']
>>> all([hasattr(L, attr) for attr in ''.join([L_attrs, S_attrs])
True
>>> LM_attrs = ['LMFrame']
>>> all([hasattr(LM, attr) for attr in ''.join([LM_attrs, L_attrs, S_attrs])
True
```

Vetting

Next Actions

- Reduce object recreation; notice a FI is passed to `Stack` and `Laminate`.
- Get image of how objects are passed prior to refactoring.

See Also

- LPEP 004: Refactoring Stack to optimize object creation
- LPEP 006: Defining objects

Copyright

In [ ]:

2.18 Version Log

```
In [1]: '''
        =======
        Scripts
        =======

        Fast Plots, Initial Conversion from Excel to Python

        0.1         Legacy Script - Laminate_Stress_Constant_Thickness_1g.ipynb
                    - Calculates stress, plot, export data for single geometry.
                    Supports insets.
        0.2         - Legacy Script - Laminate_Stress_Constant_Thickness_2d.ipynb
                    - Handles multiple geometries, heiarchical indexing, more...
        0.3         Legacy Script - Laminate_Stress_Constant_Thickness_3a3.ipynb
                    - Abstracted layers to classes.
                    - Forked to become the LamAna project.

        =============
        Local Program
        =============

        Standalone Program, Abstracted for General Application, Single Notebook

        0.4.0       Organized module and package layout
        0.4.1       Most user inpurt API conventions worked out
                    - `input_.Geometry`
                    - `distributions.Case(*args)`, `distributions.Case().apply()`
                    - `Geometry().itotal()`
        0.4.2       General Convention standardized types for Geometry object
        0.4.2a      Developed total method; float and list wrappers covering attributes
        0.4.2b      Develop total methods and tests; cleaned docs
        0.4.3       Developed `contructs` module to return DataFrames
        0.4.3a      Built Stack class giving skeletal layout for each Geometry
        0.4.3b      Implemented namedtuples called `StackTuple` in the `Stack` class
                    - Deprecated ModdedList and ModdedFloat wrappers
                    - Implemented namedtuples called GeometryTuple in the `Geometry` class
                    - Made unittest for p >= 1 (odd and even)
        0.4.3c      Added stress states to laminates.
        0.4.3c1     Refactored the ConstuctTheory workflow
                    - Refocused on building separate Construct and Theory objects
                    - Removed double call of snapshot and laminate model in Case
        0.4.3c2     Laminate object made the official LaminateModel object w/ __repr__
                    - laminate_tuple deprecated. self attrs used instead
                    - LaminatModel object is now Laminae self called directly in Case
                    - `LM` attribute returns the LaminateModel object
```

```
            - Refactored frames and snapshots to generate lists only when called.
0.4.3c3     Developed Laminate Object to calculate Planar variables
0.4.3c4     Implemented update_columns() into Laminate as `_update_dimensions()`.
            First trial with version 4a.  Testing needs resolution.
            Refactored `_update_dim..()` w/copies and returns; cleared warnings.
            1.5x longer to run 62 tests with now; from 33s to 86s.
            Tested Planar values for different plies, ps.
0.4.3c4b    Verify values and algorightms with CLT.
            - Added drag and drop spreadsheet testing of laminates.
            - Removed a labels test with improved test speed.
            - Built the `utils.tools` modulue and implemented in tests.
            - Fixed critical BUG in inner_i ordering.
0.4.3c4c    Cleaned deprecations, and made use of `laminator()`
            - Applied `shift` to discontinuity values.
            - Added nplies, p, total attrs to Laminate
0.4.3c4d    Vectorize _update_dimensions().  Deprecated for tech.
            - Documented alogrithms.
            - Made test_controls for various case controls.
            - Deprecated Laminae classes; maybe someday.  Only dfs for now.
            - Rename to LFrame. Changed planar term to dimensionals.
0.4.3c5     Deprecation code cleared.
            Learned about iterators, generators, gen expressions and  more.
0.4.3c5a    Add iterator functionality; replaced empty list/append code.
            - Add lt_exceptions for relevent error handling.
0.4.3c5b    The Big Generator Switch
            - Improved architexture using generators in Geometry and Stack.
            - The StackTuple has added alias and unifed name to 'n-ply'.
              decode_geometry() and indentify_geometry() were refactored.
            - Running 75 tests at 109s.  No big speed up.  The tests are slow.
0.4.3c6     Marked for theories model and Data variable implementation.
            - LFrame returns IDs and Dimensional columns only
            - LMFrame returns full DataFrame.
            - Many tests developed for theories and models.
            - Models accepted as functions or classes
            - Minor exception handling to prevent backdoor API breaking.
0.4.3d      Deprecations, cleanup, reorganization, documentation.
            - Deprecated .itotal()
            - Renamed Model to BaseModel and subclassed from it.
            - Refactored laminator to yield case, not LMs
            - Reorganized docs; added Delta Code and new case build idioms (2/3)
            - Refactored tests for new case build ideology (Py2/3 compliant)
            - Started Cases object outline to replace laminator.
0.4.3d1     continued...
            - Defaults() abstracted to BaseDefaults()
            - Defaults() subclassed in Wilson_LT
            - Refactored BaseDefaults() geo_inputs dict keys with names
            - Automated BaseDefaults() groupings
            - Built smarter tests to allow extension of BaseDefaults()
            - Moved BaseDefaults() from utils.tools to input_ module
            - Geometry and LaminateModel objects support == and != comparisions
0.4.3d2     continued...
            - Rename geo_params to load_params
0.4.3d3     continued...
            - Changed mat_params default type from Dataframe to (nested) dict
```

```
                - mat_params now DataFrame and assert_equally-friendly
                - Cleaner exception handing of apply(mat_params)
                - Deprecated materials_to_df in favor of Standard Form conversions
                - Replaced @staticmethod with @classmethod
                - Cleaned Case helpers; moved to BaseDefaults
    0.4.3d4     continued...
                - Refactored FeatureInput; Parameters, Properties, Materials...
                - Support of ordered materials lists
                - Added get_materials()
                - Extended case params and props to give Series and DataFrame views
                - Tested models columns by written new .csv files
                - Cleanup, privatizing, Geo_object comparison tests, user-defined
                  material setting, material stack order tests, change to mat_props.
    0.4.4       Marked `distributions` plotting.
                - Defined Cases and select method.
                - Made Geometry and Laminate hashable for set comparison w/tests.
                - Extended Cases.select() with set operations.
                - Implemented Cases.to_csv
                - Implemented Cases in distributions
                - Wrote Cases tests.
    0.4.4a      Cases documentation in intermediate tutorial; upgrade from beta
    0.4.4b      Cases.__eq__ handles DataFrame/Series attrs
                - Laminate.__eq__ handles DataFrame/Series attrs
                - Refactored LPEP numbers.  LPEP 001 contains micro-PEPs.
                - Plotting
                - Reverted to k=1 to correlate w/LT definitions in the paper.
    0.4.4b1     continued...
                - Initial implementation of case.plot()
                - Using library from external notebook for 1st demo to prof.
                - Worked through labeling keywords for plots
                - _distribplot: halfplots, annotations, insets, extrema
                - _multiplot: nrows, ncols, deletes empty axes, labels
                - Added LAMANA_PALETTE
                - Implemeentated Laminate.extrema
                - Rewrote controls with new k algorithm
                - Reorganized ToDo list: headings (functionalize, clarify, optimize).
    0.4.4b2     continued...
                - Final implementation of case.plot(), _distribplot() & _multiplot()
                - Cleanup docstrings, move abstractions to beta, clean beta
                - Implement Case.plot() multiplots
                - Revise Demo notebook.
    0.4.4b3     continued...
                - Make Cases process cases
                - Implement Cases.plot() multiplots
                - LPEP 003 Pattern development and cases input standards.
    0.4.5       Marked for release candidate 1
                - Cleanup
                - Refined exception handing of rollback; improved dormant tests
                - Implemented lt_exceptions and IndeterminateError
                - Register GitHub, pypitest and pypi
                - prepare revised repo structure
    0.4.5a1     BRCH: Refactor iteration in Geometry, distributions and Laminate.
                - Refactored *.apply iter strings, not objects, cache ... 100s faster
                - 198 tests ~300s, 200s faster since output_
```

```
            - Coded is_valid() with regex
            - Write References to utils
            - Finalize stable release before pip and GitHub upload; old-sytle
0.4.5b1     First Fork from stable package.
            - move is_valid to Beta-functionalize; cleanup Quick tests
            - Use text editor, Atom, to rename lamana, sp check and remove tabs
            - Make Main template for pull and push
            - Repo redesign
            - flake8
0.4.5b2     "Pull" from flake8'd core modules.
            - Pulled updated modules
            - Figure out __version__
            - Suspended repo changes until official GitHub push.

===============
The Large Split
===============

Version Control, GitHub Upload, Open Source Tools, PyPI Releases, Separated Noteboo

0.4.6       Official GitHub release (usable; needs repo maintenace)
            - Reflects suspended repo state as of 0.4.5b2
            - Nearly equals 0.4.5b2 but cleaner directory, new repo tree
            - Uses new templates to run code; clonable
            - Minimal point releases allowed.
            - Updated version number.
0.4.7       Initial PyPI release
            - Rolled back due to pandas breaking
0.4.8       Pre-conda, gitflow, travis release: dev reliable
            - Init gitflow
            - Add Travis CI to install pinned dependencies and check builds
            - Pass builds 2.7, 3.3, 3.4, 3.5, 3.5.1 on Linux
            - Main deps matplotlib==1.4.3, numpy==1.9.2, pandas==0.16.2
            - Reproducible builds with `pip freeze`, pinned deps (LPEP 001.14)
            - Suggest "Hands-on/off" pip install options
            - Update license info
            - Update release workflow in tutorial
0.4.9       Flagship PyPI release
            - Install with conda through pip
            - Works with IPython/Jupyter
            - Add `find_version()` to `setup.py`; confirm tarball url works
            - Update source to use latest `pandas>=0.17.1` and `numpy`
            - Fixed pandas issue (backwards compatible):
              - replace `np.cumsum` with pandas `aggregate` method
              - replace pandas `sort` with `sort_index`
0.4.10      Documentation Release
            - flake8
            - docs
            - numpy docstrings
            - Register readthedocs
            - Initial codecov and coverage
            - nb templates (main, doc-builder)
            - Fix df.sort to address pandas API changes in 0.17.1
```

```
    0.4.11      Housekeeping - cross-platform stability, coverage and more
                - Add environment.yaml files for 2.7 and 3.3
                - Continuous Integration on Windows (using Appveyor); no 64x Py3.4 supp
                - Add negative index handling in Cases (not negative step)
                - Deprecate __getslice__ in Cases(); affect Py < 2.6
                - Improve coverage (+20%)
                - Try logging
                - Add dashboards; revamp export
                - Speed up readthedocs
                - Refactor BaseModel with abstract hook method
                - Add warnings
                - Deprecate warn util.write_csv, Cases.to_csv
                - Add config file
                - Add py35 environment yaml

    '''
    pass
```

CHAPTER 3

Indices and Tables

- genindex
- modindex
- search