pyNastran Documentation

Release 0.5.0-locr

Steven Doyle **Al Danial**

Jun 22, 2017

Contents

The pyNastran software interfaces to Nastran's complicated input and output files and provides a simplified interface to read/edit/write the various files. The software is compatible currently being used on Windows, Linux, and Mac.

The **BDF reader/editor/writer** supports about 210 cards including coordinate systems. Card objects have methods to access data such as Mass, Area, etc. The BDF writer writes a small field formatted file, but makes full use of the 8-character Nastran field. The OpenMDAO BDF parametrization syntax is also supported.

The **OP2 reader** supports static/transient results, which unless you analyzing frequency response data should be good enough. It also supports **F06 Writing** for most of the objects. Results include: displacement, velocity, acceleration, temperature, eigenvectors, eigenvalues, SPC forces, MPC forces, grid point forces, load vectors, applied loads, strain energy, as well as stress and strain.

The **F06 reader/writer** works for simple problems, but it's still preliminary. At this point, you should just use the OP2 reader. It's faster, more robust, and supports more results. The F06 reader is more used as a verification tool for the OP2 reader.

The **Cython OP4** reader supports ASCII and binary dense matrices. The **Python OP4** reader supports ASCII and binary sparse and dense matrices.

A simple GUI has been developed that can view BDF models and display static stress results from the OP2.

CHAPTER 1

pyNastran Manual

Brief Project Overview

Since the 1960's NASTRAN (NASA Structural ANalysis) has been used to solve structural/thermal/aerodynamic/dynamics/etc. problems. The file formats were originally developed by MSC for a product now called MSC Nastran. There have been many spinoff version of NASTRAN that have been created based on the 2001 source code release of MSC Nastran in 2002 after settlement with the FTC (Federal Trade Commisson). There is now NX Nastran and NEi Nastran, which are developed independently.

`pyNastran` is at it's core an API (Application Programming Interface) to the legacy formats used by Nastran. These files include the BDF, F06, OP2, OP4, and PCH files. Other code has been added to `pyNastran` in order to drive development of the software in order to be able to solve other engineering problems. For example, Code_Aster, an open-source finite element code developed by the EDF (Electricity of France), has a Nastran to Code_Aster converter that is in development. The development has helped to define the API for the loads in order to be able to extract them in a way that makes sense. However, this is not the focus of the software.

Target Audience

`pyNastran` target audience are users of Nastran and therefore are expected to be familiar with the software. This has greatly reduced the necessity of documenting every variable exhaustively as users can easily reference existing Nastran documentation. The BDF file has roughly 700 cards availble to a user with 200 being currently supported by pyNastran. The majority of the cards, defined as separate Python classes, are not documented. However, the Quick Reference Guide (QRG) defines each input to the card. A user with the QRG should have little effort in understanding what the various objects do. However, for convenience, it's still good to document variables.

pyNastran target audience largely uses MATLAB, a matrix based programming language, and typically has little experience with general programming. There are also users that know Python, but have never used a class or a dictionary, which makes an API seems complicated. Additionally, while not difficult to build, the Cython OP4 reader, which requires a C compiler, has been a source of frustration for many users. That said, most operations should be relatively easy to accomplish.

Nonmenclature

Symbol	Description
$[B]$	Displacement-to-Strain Matrix
$[D]$	Material Matrix
E	Young's Modulus (F/L^2)
$\{F\}$	Applied force vector
$\{F_g\}$	Global applied force vector
$\{F_a\}$	Reduced applied force vector
$\{f\}$	Generalized (Modal) force vector
G	Shear Modulus (F/L^2)
$[J]$	Jacobian Matrix
$[K]$	Stiffness matrix
$[K_e]$	Element stiffness matrix (F/L^2)
$[K_{gg}]$	Global Stiffness Matrix
$[K_{aa}]$	Reduced Stiffness Matrix
k	Spring stiffness (F/L^2)
L	Length (L)
L, M, N	X,Y,Z Moment (FL)
M_x, M_y, M_z	X,Y,Z Moment (FL)
$[M]$	Matrix
$[M_L]$	Mass Matrix
$[N]$	Function Matrix
m	mass (M)
nsm	non-structural mass (M)
$[T]$	Transformation matrix
t	thickness (L)
T	translation (L)
R	rotation (L/radians)
u	generalized load (translation, rotation)
X, Y, Z	X,Y,Z Force (F)
F_x, F_y, F_z	X,Y,Z Force (F)
x, y, z	displacement (L)
$\{x\}$	generalized load (translation, rotation)
V	Volume (L^3)
V_i	i^{th} Volume (L^3)
ϵ	strain, shear strain ($\Delta L/L$)
γ	shear strain (F/L^2)

Continued on next page

Table 1.1 – continued from previous page

Symbol	Description
ν	Poisson's ratio
Ω	Natural frequencies (Hz)
ω	natural frequency (Hz)
ρ	density (M/L^3)
σ_{ij}	normal stress, shear stress (F/L^2)
τ_{ij}	shear stress (F/L^2)
θ	rotation (radians)

User: BDF

This document is intended to be a reference guide for users.

Nodes

GRID

In NASTRAN, nodes can be in the global or a local coordinate frame (rectangular, cylindrical, spherical). Each node may reference a different coordinate frame cp for the reference coordinate frame (what the value is in the x_1, x_2, and x_3 fields) and for the analysis coordinate frame aid.

Node Specific Variable Names:

Name	Description
`xyz`	Raw location <x_1, x_2, x_3> in the BDF
`cp`	reference coordinate system
`cd`	analysis coordinate system
`Position`	location in an arbitrary coordinate system
`UpdatePosition`	change the value of xyz and cp
`resolveGrids`	update all the positions of the grids to a common coordinate system in an more efficient way

Using the node object:

```
bdf = BDF()
bdf.readBDF(bdfName)
node = bdf.Node(nid)
node.Position()               % gets the position of the node in the global␣
↪frame
node.Position(cid=0)          % same
node.Position(cid=1)          % gets the position of the node in a local frame

node.UpdatePosition(bdf,array([1.,2.,3.]),cid=3) % change the location of the␣
↪node

bdf.resolveGrids(cid=0)       % change the xyz of all nodes to the same␣
↪coordinate system as cid
```

```
bdf2 = BDF()
bdf2.readBDF(bdfNameAlt)
bdf.unresolveGrids(bdf2)      % change the coordinte system back to the␣
↪coordinate system in bdf2
```

CORDx

A coordinate system may be defined by 3 points in any non-circular coordinate system or by 3 vectors.

Once cross-referenced a node may use it's `node.Position(cid)` method to determine the position in any frame defined by a coordinate system.

Developer: Getting Started

Code Formatting - PEP-8

`pyNastran` is attempting to implement Python's PEP-8 formatting standard after not following it for multiple versions. It's a work in progress to upgrade the code, so try to follow it. The biggest challenge has been to come up with a clear way to reference methods and very similar variable names. For example, the element.Area() method uses the element.area variable in the case of the CQUAD4, but references element.areas in the case of a multi-segmented CBAR. It's recommended that developers use Spyder with pylint integration to help with identifying PEP-8 deficiencies.

Most important points of Python PEP-8 (http://www.python.org/dev/peps/pep-0008/):

1. Use spaces (not tabs) when indenting code
2. Try to limit code to **80**-characters width
3. Files are named **as_follows.py**
4. Classes are named **AsFollows**
5. Functions are are named **as_follows()** and private functions are named **_as_follows()**
6. Variable names**_have_embedded_underscores** or without any (e.g. self.elements). A private variable would look **_similar_to_a_function**
7. Don't use **import ***
8. **Break the rules if it's clearer**
9. Document functions/classes/variables

Additional Guidelines

1. `pyNastran` liberally uses classes (it has too many!), so try not to write more classes than necessary. Functions are often better to avoid having extra functions in objects the user has immediate access to (imagine having 100 functions to sift through) in an IDE. Watch this video for a better explanation. Again, it's a work in progress.

2. When writing classes, group similar classes in the same file to reduce having to search for a class across multiple files (very common in Java).
3. When deprecating functions, throw a warning using the Python warnings module. It's easier to make sure the rest of the code isn't using it this way. This is not nearly as important if it's not a function the user is likely to interact with.

Documentation

`pyNastran` uses Doxygen for docstrings. Doxygen is nice because it allows LaTeX to be embedded in the documentation and HTML docs are easily generated. Sphinx is another documentation format, It is known that Sphinx also allows LaTeX and that it is much more commonly used.

Some guidelines for Doxygen:

@param in_variable_name a description goes here

@retval out_variable_name another description

@see look at AsFollows class or as_follows function

@note a message

@warning a warning

@todo something that's not done

Developer: BDF Reading

This document is a reference for developers of pyNastran, but are not necessarily people that are familiar with Nastran or even Finite Element Analysis (FEA).

`bdf`: Introduction

`bdf` module controls the model object that is instantiated with `model = BDF()` the `BDF.__init__` method is called when `model = BDF()` is run. The `pyNastran.bdf.bdf.BDF` used to be very large and was split (in a non-standard way) into multiple files that allowed for simpler development. The classes:

- *pyNastran.bdf.bdfInterface.bdf_Reader.BDFReader*,
- pyNastran.bdf.bdfInterface.getCard.GetMethods,
- *pyNastran.bdf.bdfInterface.addCard.AddMethods*,
- pyNastran.bdf.bdfInterface.bdf_writeMesh.WriteMesh,
- *pyNastran.bdf.bdfInterface.bdf_cardMethods.CardMethods*,
- *pyNastran.bdf.bdfInterface.crossReference.XrefMesh*

are basically bags of functions for the "model" object.

The `pyNastran.bdf.bdf.BDF.cardsToRead` attribute limits the cards that `pyNastran` processes and can be modified by the user in order to fix bugs or make their code run faster.

Moving onto `pyNastran.bdf.bdf.BDF._init_solution` sets a series of alternate names for Nastran solution types. For example, a solution 101 is a static solution (no acceleration) and will calculate the displacements of the system $[K]\{x\} = \{F\}$. You can then extract stresses and strains. Other solution numbers solve different equations.

In methods `pyNastran.bdf.bdf.BDF._init_structural_defaults()`, `pyNastran.bdf.bdf.BDF._init_aero_defaults()`, `pyNastran.bdf.bdf.BDF._init_thermal_defaults()` the structural, aerodyanmic, and thermal card holders (e.g. model.materials) are defined as dictionaries.

Finally, the `pyNastran.bdf.bdf.BDF.readBDF()` method is defined. There are three sections to a BDF/DAT/NAS file. BDF (Bulk Data File) is the file format used by MSC Nastran and the DAT (data?) file is used by NX Nastran. NAS (Nastran) is also used.

The first section is the "Executive Control Deck" and contains a "SOL 101" or "SOL 103" or "SOL STATIC" depending on the solution type. It ends when the "CEND" marker is found. Then the "Case Control Deck" is read. Here, general solution information is listed, such as what outputs do you want and what loads are applied (not all loads in the file are necessarily applied). Finally this section defines one or more subcases, which are different load combinations. The last section is the "Bulk Data Deck". This section stores 99% of the file and this section introduces very specific formatting restrictions for "cards".

A basic model will be made of nodes, elements, properties, and materials. For example, for a square plate made of steel model GRID cards are used for the nodes, CQUAD4 or CTRIA3 cards are used for the elements (the C generally indicates the card is an element so quadrilateral element and triangular element). The element has a property (PSHELL) that defines the thickness. Similarly, properties generally start with P. Finally, the property references a material (MAT1) that defines the material as steel. INCLUDE cards may also be used to add additional files into the BDF.

`bdf`: Card Formatting

A "card" is at most 72-characters wide. Comments may follow the card if a $ sign is used.

The standard card is called small field format (single precision) and has 9 fields defined per line, each with 8-characters and are fixed width. Multiline cards are implied by leaving 8 spaces at the beginning of the following line. Alternatively, a + sign may be used in the first 8 spaces.

The large field format (double precision) card uses a $1 * 8 + 4 * 16$ to reach the 72-character width instead of $1 * 8 + 8 * 8$ characters. If the first line of a card is double precision, a * follows the card name, so all card names are 7-characters or less. If the second line of a card is double precision, a * begins the line. A single line of a small field formatted takes exactly two lines to write if large field format is used.

The CSV (comma separated value) format is similar to small field format. It's less picky then the 8-character format, but much harder to read. It is still subject to the 9 fields per line restriction. If a CSV card has a * on it, the card becomes a large field CSV formatted card and may have only 5 fields on the line (including the blank field).

Although introduced as separate types of cards, small field format and large field format may be mixed and matched. However, this only occurs for hand-edited BDFs. There's also a difficult to understand format known as a "continuation card". This uses values from previous cards and is basically a *for* loop. Hundreds of cards may be defined in just a few lines.

`bdf` : Parsing

A basic card is a GRID card. Once parsed, a standard grid card will have fields of `['GRID', nodeID, coord, x, y, z]`. This section will discuss how a card is parsed.

The `pyNastran.bdf.bdf.BDF.readBDF()` method must generalize the way files are opened because INCLUDE files may be used. Once the Executive and Case Control Decks are read (using simple while loops), the `pyNastran.bdf.bdf.BDF._read_bulk_data_deck()` method is called.

This method (`BDF._read_bulk_data_deck()`) keeps looping over the file as long as there are files opened (an INCLUDE file side effect) and calls: `(rawCard, card, cardName) = self._get_card(debug=False)`. `cardName` is just the card's name, while `rawCard` is the full, unparsed card. `card` is the card parsed into fields as a `card` object, which is basically a list of fields `['GRID', nodeID, coord, x, y, z]`.

The basic idea of the `self._get_card()` method(see *pyNastran.bdf.bdfInterface.bdf_cardMethods.CardMethods._get_card()*) is to make a `self.linesPack`, (`pyNastran.bdf.bdf.BDF.linesPack`)which is a list of 1500 lines that are stored in memory. When a new line is added to `self.linesPack`, it is first stripped of comments in order to simplify parsing. If the linesPack is empty or 50 blank lines are found, the code assumes an infinite loop has been entered and closes the file. If additional files are open, the `linesPack` from the previous file will be used (INCLUDE file case).

Now that we have 1500 lines in linesPack, we must call: `(i, tempcard) = self._get_multi_line_card(i, tempcard)` to get the card. `tempcard` starts out as the first line in the card and afterwards contains all lines of the card. `tempcard` will eventually become `rawCard`. It's important to note the `tempcard` passed into *pyNastran.bdf.bdfInterface.bdf_cardMethods.CardMethods._get_multi_line_card()* is a 72-character string (generally) and the `tempcard` output is a list of 8-character (or 16) width fields. Why: the original data isn't needed, so the variable is reused.

pyNastran.bdf.bdfInterface.bdf_cardMethods.CardMethods._get_multi_line_card() will search through the `linesPack` and try to end the card by looking for a non-whitespace character in the first character position (all cards must have `field[0]` justified). If a * is found, it's double precision, if not it's small field. Additionally, if a ',' is found it's CSV format. So the formats are:

1. small field,
2. small field CSV,
3. large field,
4. large field CSV.

Once the format of the line is known, it's an easy process to split the card (see *pyNastran.bdf.bdfInterface.bdf_cardMethods.CardMethods.processCard()*) and turn it into a

pyNastran.bdf.bdfInterface.BDF_Card.BDFCard object. Note that the first field in any line beyond the first one must be blank and is ignored. This prevents cards defined in small field and large field to have the same value defined in different positions of the list.

Finally, as Nastran is very specific in putting a decimal on float values, it's easy to parse field values into their proper type dynamically. This is especially important when a field may be defined as an integer, float, a string, or be left blank and the variable is different depending on variable type. Strings, must being with alphabetical characters (A, B, C) and are case insensitive, which is why a "GRID" card is called a "GRID" card and not a "grid" card.

`bdf` : Card Object

A *pyNastran.bdf.bdfInterface.BDF_Card.BDFCard* object is basically a list of fields of `['GRID', nodeID, coord, x, y, z]` with methods to get the 1st entry (`nodeID`) as `card.field(1)` instead of `fields[1]` for a list. A card object is useful for setting defaults. The `x, y`, and `z` values on the GRID card have defaults of 0.0, so `card.field(3,0.0)` may be used to get the `x` coordinate. Finally, `card.fields(3,5,[0.,0.,0.])` may be used to get `xyz` and set the defaults in a single step. Additionally, the `card` object is useful when parsing "continuation cards", but is typically disabled.

After an excessively long branch of `cardNames` in `pyNastran.bdf.BDF.readBDF()`, the card object is turned into a GRID, CTRIA3, CQUAD4, PSHELL, MAT1 or any of 200 other card types. There are roughly as many nodes as there are elements, which make up roughly 95% of the cards in large models. The difference in a large model and a small model, is the discretization and will change nodes, elements, loads, and constraints. Loads and constraints are applied to only small portions of the model and (generally) only the boundary of a model. The number of propertie and materials is very likely the same.

Most cards are stored in a dictionary based on their integer ID. IDs may be used only once, but if a card is exactly duplicated, it is still valid.

`shell`: CQUAD4 Object

In `bdf/cards/elements/shell.py`, the `pyNastran.bdf.cards.elements.shell.CQUAD4` is defined.

The `pyNastran.bdf.cards.elements.shell.CQUAD4` is a shell-type element and must reference a PSHELL (isotropic property) or a PCOMP (composite property) card. An example of an isotropic material is steel or aluminum and a composite material would be fiberglass or layers of carbon fiber/epoxy resin at layed up at different angles.

The PSHELL may reference MAT1 (isotropic material) cards, while the PCOMP card may reference MAT1 or MAT8 (orthotropic material) cards. An orthotropic material is stronger in longitudinally than laterally (e.g. fibers are oriented unidirectionally in a carbon fiber composite).

The `pyNastran.bdf.cards.elements.shell.CQUAD4` class inherits from the `pyNastran.bdf.cards.elements.shell.QuadShell` class which defines common methods to the various QUAD-type cards. There are additional QUAD element with different numbers of nodes (8-CQUAD8, 9-CQUAD) and the CQUADR and CQUADX are axi-symmetric versions of the CQUAD4, and CQUAD8 respectively. However, the `Area()`, `Mass()`, `Density()`, etc. methods are calculated in the the same

way for each card (although the axi-symmetric cards return mass per unit theta). The last thing to note is `rawFields` and `reprFields` are very important to how the code integrates.

`rawFields` is used to check if a duplicated card is the same as another card and is also used for testing. After reading and writing, reading back in, writing back out, reading back in, if the fields are the same, then there's likely no error in reading a card (fields can still be missed while reading, so it's not perfect). `rawFields` returns a list of the fields (similar to the list-esque card object from before).

`reprFields` is analogous to the `__repr__()` method, and is an abbreviated way to write the card. For example, the `T1, T2, T3`, and `T4` values (thickness at nodes 1, 2, 3, 4) are generally 0.0 and instead are set at an elemental level using the PSHELL card. If these fields were printed, the CQUAD4 card would be a two line card instead of a one line card. `reprFields` is used instead of `__repr__()` in order to be able to write the card in large field or small field format. Defaults are generally not written by the `__repr__()` method, but are written for certain fields (e.g. the `xyz` fields on the GRID card).

To get the CQUAD4, with an element ID of 1000, you would type:

```
elem = model.elements[1000]
```

or:

```
elem = model.Element(1000)
```

to use the function.

Then to print the card, type:

```
print(elem)
```

to see the Nastran formatted card. The `__repr__()` method is defined in `bdf/cards/baseCard.py` the `pyNastran.bdf.cards.baseCard` class (which is used by the `pyNastran.bdf.cards.baseCard.Element` class also defined in `baseCard.py`).

`shell`: Cross-Referencing the CQUAD4 Object

Previously, it was mentioned that the square plate model built with quads and triangles had a thickness and a material. The nodes of the elements also have positions. The nodes further be defined in a rectangular, cylindrical, or spherical coordinate system, so to get the mass of an element is actually quite involved. Creating a function to access the mass becomes possible without passing the entire model object around to every function through the use of cross-referencing.

Cross Referencing takes a CQUAD4 card and replaces the GRID references with actual GRID cards. The GRID cards in turn reference two COORDx (CORD1R, CORD2R, CORD1C, COR2DC, CORD1S, CORD2S) cards, which also may reference two CORDx cards. The CQUAD4 references a PSHELL or PCOMP card. The PSHELL references a single MAT1 card, and as mentioned before the PCOMP card may reference one or more MAT1/MAT8 cards. In order to calculate something simple like the mass of the CQUAD4 requires the formula:

$$m = A\left(t\rho + \frac{nsm}{A}\right)$$

for a PSHELL or:

$$m = A\left(\sum_{i=0}^{i=1} t\rho + \frac{nsm}{A}\right)$$

for a PCOMP.

By using classes and functions, it's easy to just call the `element.MassPerArea()` method and get the proper data to apply the formula. Similarly, the `element.Area()` method calls the `node.Position()` method to get the node in the global XYZ coordinate frame and can then find the area using vectors in a 3D space:

$$A = \frac{1}{2}|(n_1 - n_3) \times (n_2 - n_4)|$$

(see http://en.wikipedia.org/wiki/Quadrilateral).

`crossReference`: Cross-Referencing Process

Cross referencing must first be done on the coordinate cards. Then, once they're done, the nodes are cross referenced. Once this is done, the coordinate systems may be resolved (CORD1x cards reference GRID cards). Then elements, properties, materials, loads, boundary conditions, aerodynamic cards, thermal, dynamic, etc. cards are mapped. The order doesn't matter, but CORD1x cards and GRID cards must be mapped first.

Cross Referencing is performed by looping over the card objects and calling the `card.crossReference()` method. This will setup all cross-referencing and a full list of the status of various cards is listed in `bdf_crossReferencing.txt`.

`writeMesh.py`: Writing the BDF

The BDF is written by looping through all the objects and calling the `card.__repr__()` method by typing `str(card)`.

Currently, only small field format is supported for writing. The list from `reprFields()` is passed into `fieldWriter.py` function `printCard(listObj)` and it dynamically figures out how to write the card based on the data type. For float values, the highest precision 8-character width field will be used even if it uses Nastran's strange syntax of "1.2345+8" to represent a more standard "1.2345e+08".

CHAPTER 2

pyNastran Package

bdf Package

bdf Module

bdf_Methods Module

caseControlDeck Module

class pyNastran.bdf.caseControlDeck.**CaseControlDeck**(*lines*, *log=None*)

Bases: object

Parameters

- **self** – the case control deck object
- **lines** – list of lines that represent the case control deck ending with BEGIN BULK
- **log** – a logger object

_add_parameter_to_subcase(*key*, *value*, *options*, *param_type*, *isubcase*)
internal method

_clean_lines(*lines*)
removes comment characters $

_parse_data_from_user(*param*)

_parse_entry(*lines*)

@brief internal method for parsing a card of the case control deck

parses a single case control deck card into 4 sections 1. paramName - obvious 2. Value - still kind of obvious 3. options - rarely used data 4. paramType - STRESS-type, SUBCASE-type, PARAM-type, SET-type, BEGIN_BULK-type

It's easier with examples:

paramType = SUBCASE-type SUBCASE 1 -> paramName=SUBCASE value=1 options=[]

paramType = STRESS-type STRESS = ALL -> paramName=STRESS value=ALL options=[] STRAIN(PLOT) = 5 -> paramName=STRAIN value=5 options=[PLOT] TITLE = stuff -> paramName=TITLE value=stuff options=[]

paramType = SET-type SET 1 = 10,20,30 -> paramName=SET value=[10,20,30] options = 1

paramType = BEGIN_BULK-type BEGIN BULK -> paramName=BEGIN value=BULK options = []

paramType = CSV-type PARAM,FIXEDB,-1 -> paramName=PARAM value=FIXEDB options = [-1]

The paramType is the "macro" form of the data (similar to integer, float, string). The value is generally whats on the RHS of the equals sign (assuming it's there). Options are modifiers on the data. Form things like the PARAM card or the SET card they arent as clear, but the paramType lets the program know how to format it when writing it out.

Parameters

- **self** – the case control deck object
- **lines** – list of lines

Returns paramName see brief

Returns value see brief

Returns options see brief

Returns paramType see brief

_read(*lines*)
reads the case control deck

Note: supports comment lines

Warning: doesnt check for 72 character width lines, but will follow that when it's written out

add_parameter_to_global_subcase(*param*)
takes in a single-lined string

Note: dont worry about overbounding the line

add_parameter_to_local_subcase(*isubcase*, *param*)

convert_to_sol_200(*model*)
Takes a case control deck and changes it from a

Todo

not done...

copy_subcase(*i_from_subcase*, *i_to_subcase*, *overwrite_subcase=True*)
Overwrites the parameters from one subcase to another.

Parameters

- **self** – the case control deck object
- **i_from_subcase** – the subcase to pull the data from
- **i_to_subcase** – the subcase to map the data to
- **overwrite_subcase** – NULLs i_to_subcase before copying i_from_subcase

create_new_subcase(*isubcase*)
Method create_new_subcase:

Warning: be careful you dont add data to the global subcase after running this...is this True???

cross_reference(*model*)

delete_subcase(*isubcase*)

finish_subcases()
removes any unwanted data in the subcase...specifically the SUBCASE data member. Otherwise it will print out after a key like stress.

get_local_subcase_list()

get_op2_data()
returns the relevant op2 parameters required for a given subcase

get_subcase_list()

get_subcase_parameter(*isubcase*, *param_name*)

has_parameter(*isubcase*, *param_name*)
Checks to see if a parameter (e.g. STRESS) is defined in a certain subcase ID.

Parameters

- **self** – the CaseControl object
- **isubcase** – the subcase ID to check
- **param_name** – the parameter name to look for

has_subcase(*isubcase*)
Checks to see if a subcase exists.

Parameters

- **self** – the case control deck object
- **isubcase** – the subcase ID

Returns does_subcase_exist (type = bool)

nlines_max = 10000

update_solution(*isubcase*, *sol*)
sol = STATICS, FLUTTER, MODAL, etc.

fieldWriter Module

fieldWriter16 Module

pyNastran.bdf.fieldWriter16.**printCard_16**(*fields*, *tol=0.0*)
Prints a nastran-style card with 16-character width fields.

Parameters

- **fields** – all the fields in the BDF card (no blanks)
- **tol** – the abs(tol) to consider value=0 (default=0.)

Note: A large field format follows the 8-16-16-16-16-8 = 80 format where the first 8 is the card name or blank (continuation). The last 8-character field indicates an optional continuation, but because it's a left-justified unneccessary field, printCard doesnt use it.

---

pyNastran.bdf.fieldWriter16.**printField16**(*value*, *tol=0.0*)
prints a single 16-character width field

Parameters

- **value** – the value to print
- **tol** – the abs(tol) to consider value=0 (default=0.)

Returns field an 16-character (tested) string

pyNastran.bdf.fieldWriter16.**printFloat16**(*value*, *tol=1e-08*)
Prints a float in nastran 16-character width syntax using the highest precision possbile.

See also:

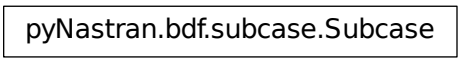
see `printFloat8()`

> **Warning:** completely unimplemented & untested

pyNastran.bdf.fieldWriter16.**printScientific16**(*value*)
Prints a value in 16-character scientific notation. This is a sub-method and shouldnt typically be called

> **Warning:** not tested...

subcase Module



class pyNastran.bdf.subcase.**Subcase**(*id=0*)
Bases: `object`

_add_data(*key*, *value*, *options*, *param_type*)

_simplify_data(*key*, *value*, *options*, *param_type*)

crossReference(*mesh*)
Method crossReference:

> **Note:** this is not integrated and probably never will be as it's not really that necessary. it's only really useful when running an analysis

finish_subcase()
Removes the subcase parameter from the subcase to avoid printing it in a funny spot

get_analysis_code(*sol*)
8 - post-buckling (maybe 7 depending on NLPARM???)

not important 3/4 - differential stiffness (obsolete) 11 - old geometric nonlinear statics 12 - contran (???)

Todo

verify

---

get_device_code(*options*, *value*)

get_format_code(*options*, *value*)
returns the format code that will be used by the op2 based on the options

---

Todo

not done...only supports REAL, IMAG, PHASE

---

get_op2_data(*sol*, *solmap_toValue*)

get_parameter(*paramName*)

get_sort_code(*options*, *value*)

get_stress_code(*key*, *options*, *value*)
Method get_stress_code:

Note: the individual element must take the stress_code and reduce it to what the element can return. For example, for an isotropic CQUAD4 the fiber field doesnt mean anything.

BAR - no von mises/fiber ISOTROPIC - no fiber

Todo

how does the MATERIAL bit get turned on? I'm assuming it's element dependent...

---

get_table_code(*sol*, *tableName*, *options*)

has_parameter(*paramName*)

print_param(*key*, *param*, *printBeginBulk=True*)
Prints a single entry of the a subcase from the global or local subcase list.

solCodeMap = {64: 106, 1: 101, 66: 106, 68: 106, 76: 101, 144: 101, 21: 101, 24: 101, 26: 101, 99: 129, 187: 10

subcase_sorted(*listA*)
does a "smart" sort on the keys such that SET cards increment in numerical order.

Parameters

- **self** – the subcase object
- **listA** – the list of subcase list objects

Returns listB the sorted version of listA

update_param_name(*param_name*)
takes an abbreviated name and expands it so the user can type DISP or DISPLACEMENT and get the same answer

Parameters

- **self** – the subcase object
- **param_name** – the parameter name to be standardized (e.g. DISP vs. DIPLACEMENT)

Todo

not a complete list

---

write_subcase(*subcase0*)
internal method to print a subcase

Parameters

- **self** – the subcase object
- **subcase0** – the global subcase

Subpackages

bdfInterface Package

BDF_Card Module

pyNastran.bdf.bdfInterface.BDF_Card.BDFCard

class pyNastran.bdf.bdfInterface.BDF_Card.**BDFCard**(*card=None*, *oldCardObj=None*, *debug=False*)

Bases: object

Is(*cardName*)
Returns True if the card is of type cardName

Parameters

- **self** – the object pointer
- **cardName** – the cardName to compare against

Returns IsACardName True/False

_wipeEmptyFields(*card*)
Removes an trailing Nones from the card. Also converts empty strings to None.

Parameters

- **self** – the object pointer
- **card** – the fields on the card as a list

Returns shortCard the card with no trailing blank fields

applyOldFields(*cardCount=0*)
used for nastran = format

field(*i*, *default=None*)
gets the ith field on the card

Parameters

- **self** – the object pointer
- **i** – the ith field on the card (following list notation)
- **default** – the default value for the field

Returns the value on the ith field

fields(*i=0*, *j=None*, *defaults=None*, *debug=False*)
gets multiple fields on the card

Parameters

- **self** – the object pointer
- **i** – the ith field on the card (following list notation)
- **j** – the jth field on the card (None means till the end of the card)
- **defaults** – the default value for the field (as a list) len(defaults)=i-j-1
- **debug** – prints out the values at intermediate steps

Returns the values on the ith-jth fields

getOldField(*i*)
used for nastran = format

isSameName()
used for nastran = format

nFields()
gets how many fields are on the card

Parameters **self** – the object pointer

Returns nFields the number of fields on the card

replaceExpression(*fieldNew*, *fieldOld*, *replaceChar=u'='*, *replaceChar2=u''*)
used for nastran = format

pyNastran.bdf.bdfInterface.BDF_Card.**wipeEmptyFields**(*card*)
Removes an trailing Nones from the card. Also converts empty strings to None.

Parameters

- **self** – the object pointer
- **card** – the fields on the card as a list

Returns shortCard the card with no trailing blank fields

addCard Module

class pyNastran.bdf.bdfInterface.addCard.**AddMethods**
Bases: object

addAEFact(*aefact*, *allowOverwrites=False*)

addAELink(*aelink*)

addAEList(*aelist*)

addAEParam(*aeparam*)

addAEStat(*aestat*)

addAESurf(*aesurf*)

addASet(*setObj*)

addAero(*aero*)

addAeros(*aero*)

addBSet(*setObj*)

addCAero(*caero*)

addCMethod(*cMethod*, *allowOverwrites=False*)

addCSet(*setObj*)

addConstraint(*constraint*)

addConstraint_MPC(*constraint*)

addConstraint_MPCADD(*constraint*)

addConstraint_SPC(*constraint*)

addConstraint_SPCADD(*constraint*)

addConvectionProperty(*prop*)

addCoord(*coord*, *allowOverwrites=False*)

addCreepMaterial(*material*, *allowOverwrites=False*)
Method addCreepMaterial:

Note: May be removed in the future. Are CREEP cards materials? They have an MID, but reference structural materials.

---

addDArea(*darea*, *allowOverwrites=False*)

addDConstr(*dconstr*)

addDDVal(*ddval*)

addDEQATN(*deqatn*, *allowOverwrites=False*)

addDLink(*dlink*)

addDMI(*dmi*, *allowOverwrites=False*)

addDMIG(*dmig*, *allowOverwrites=False*)

addDMIJ(*dmij*, *allowOverwrites=False*)

addDMIJI(*dmiji*, *allowOverwrites=False*)

addDMIK(*dmik*, *allowOverwrites=False*)

addDResp(*dresp*)

addDamperElement(*elem*, *allowOverwrites=False*)

Warning: can dampers have the same ID as a standard element?

addDesvar(*desvar*)

addDvmrel(*dvmrel*)

addDvprel(*dvprel*)

addElement(*elem*, *allowOverwrites=False*)

addFLFACT(*flfact*)

addFREQ(*freq*)

addFlutter(*flutter*)

addGUST(*gust*)

addLSeq(*load*)

addLoad(*load*)

addMKAero(*mkaero*)

addMassElement(*elem*, *allowOverwrites=False*)

addMaterial(*material*, *allowOverwrites=False*)
only for adding structural materials

Deprecated since version this: method will be renamed in v0.3 to addStructuralMaterial.

addMaterialDependence(*material*, *allowOverwrites=False*)

addMethod(*method*, *allowOverwrites=False*)

addNLParm(*nlparm*)

addNode(*node*, *allowOverwrites=False*)

addPAero(*paero*)

addPHBDY(*prop*)

addParam(*param*, *allowOverwrites=False*)

addProperty(*prop*, *allowOverwrites=False*)

addQSet(*setObj*)

addRandomTable(*table*)

addRigidElement(*elem*, *allowOverwrites=False*)

addSPoint(*spoint*)

addSet(*setObj*)

addSetSuper(*setObj*)

addSpline(*spline*)

addStructuralMaterial(*material*, *allowOverwrites=False*)

addSuport(*suport*)

addTSTEP(*tstep*, *allowOverwrites=False*)

addTSTEPNL(*tstepnl*, *allowOverwrites=False*)

addTable(*table*)

addThermalBC(*bc*, *key*)

addThermalElement(*elem*)
same as addElement at the moment...

addThermalLoad(*load*)

addThermalMaterial(*material*, *allowOverwrites=False*)

addTrim(*trim*, *allowOverwrites=False*)

bdf_Reader Module

pyNastran.bdf.bdfInterface.bdf_Reader.BDFReader

class pyNastran.bdf.bdfInterface.bdf_Reader.**BDFReader**(*debug*, *log*)
Bases: object

_set_infile(*bdf_filename*, *includeDir=None*)
Sets up the basic file/lines/cardCounting operations

Parameters

- **self** – the BDF object
- **bdf_filename** – the input BDF filename
- **includeDir** – the location of include files if an absolute/relative path is not used (not supported in Nastran)

close_file(*debug=False*)
Closes the active file object. If no files are open, the function is skipped. This method is used in order to support INCLUDE files.

Parameters

- **self** – the object pointer
- **debug** – developer debug

get_file_stats()
gets information about the active BDF file being read

Parameters **self** – the object pointer

Returns lineNumber the active file's line number

get_line_number()
Gets the line number of the active BDF (used for debugging).

Parameters **self** – the object pointer

Returns returns the line number of the active BDF filename

get_next_line(*debug=False*)
Gets the next line in the BDF

Parameters

- **self** – the BDF object
- **debug** – developer debug

Returns line the next line in the BDF or None if it's the end of the current file

open_file(*infileName*)
Takes a filename and opens the file. This method is used in order to support INCLUDE files.

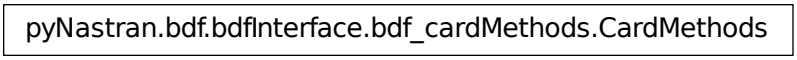
print_filename(*filename*)
Takes a path such as C:/work/fem.bdf and locates the file using relative paths. If it's on another drive, the path is not modified.

Parameters

- **self** – the object pointer
- **filename** – a filename string

Returns filenameString a shortened representation of the filename

bdf_cardMethods Module



class pyNastran.bdf.bdfInterface.bdf_cardMethods.**CardMethods**(*nCardLinesMax=1000*)
Bases: object

_get_card(*debug=False*)
gets a single unparsed card

_get_multi_line_card(*i*, *tempcard*, *isCSV=False*, *debug=False*)

_increaseCardCount(*cardName*)
Used for testing to check that the number of cards going in is the same as each time the model is read verifies proper writing of cards

> **Warning:** this wont guarantee proper reading of cards, but will help

_make_lines_pack(*debug=False*)

expandTabCommas(*line*)
The only valid tab/commas format in nastran is having the first field be a tab and the rest of the fields be separated by commas.

Parameters

- **self** – the object pointer
- **line** – a BDF line

getValue(*valueRaw*, *card*, *debug=False*)
converts a value from nastran format into python format.

isLargeField(*card*)
returns True if the card is in 16-character width fields

parseDynamicSyntax(*key*)
Applies the dynamic syntax for %varName

Parameters

- **self** – the object pointer
- **key** – the uppercased key

Returns value the dynamic value defined by dictOfVars

Note: %varName is actually %VARNAME b/c of auto-uppercasing the string, so the setDynamicSyntax method uppercases the key prior to this step.

See also:

see `setDynamicSyntax()`

processCard(*tempcard*, *debug=False*)
takes a list of strings and returns a list with the proper value in the fields of the list

update_card_lines(*lines*)
expands a card with tabs in it

pyNastran.bdf.bdfInterface.bdf_cardMethods.**interpretValue**(*valueRaw*, *card=u''*, *debug=False*)
converts a value from nastran format into python format.

pyNastran.bdf.bdfInterface.bdf_cardMethods.**make_single_streamed_card**(*log*, *card*, *debug=False*)
takes a card that has been split b/c it's a multiline card and gets rid of the required blanks in it.

pyNastran.bdf.bdfInterface.bdf_cardMethods.**nastran_split**(*log*, *line*, *isLargeField*, *debug=False*)
Splits a single BDF line into large field or small field format

Parameters

- **self** – the object pointer

- **line** – the BDF line
- **a** – flag indicating small/large field format (True/False)
- **debug** – extra developer debug

Returns fields the 9 (small) or 5 (large) fields for the line

---

Note: CSV Format is handled by parse_csv

Note: tabs are handled prior to running this

---

pyNastran.bdf.bdfInterface.bdf_cardMethods.**parse_csv**(*sline*)

pyNastran.bdf.bdfInterface.bdf_cardMethods.**stringParser**(*stringIn*)
not used

bdf_writeMesh Module

crossReference Module

class pyNastran.bdf.bdfInterface.crossReference.**XrefMesh**
Bases: object

Links up the various cards in the BDF.

The main BDF class defines all the parameters that are used.

_cross_reference_aero()
Links up all the aero cards

_cross_reference_constraints()
Links the SPCADD, SPC, SPCAX, SPCD, MPCADD, MPC cards.

_cross_reference_coordinates()
Links up all the coordinate cards to other coordinate cards and nodes

_cross_reference_elements()
Links the elements to nodes, properties (and materials depending on the card).

_cross_reference_loads()
Links the loads to nodes, coordinate systems, and other loads.

_cross_reference_materials()
Links the materials to materials (e.g. MAT1, CREEP) often this is a pass statement

_cross_reference_nodes()
Links the nodes to coordinate systems

_cross_reference_properties()
Links the properties to materials

crossReference(*xref=True*)

See also:

see *cross_reference()*

cross_reference(*xref=True*)
Links up all the cards to the cards they reference

getCard Module

cards Package

aero Module

baseCard Module

constraints Module

coordinateSystems Module

dmig Module

dynamic Module

materials Module

methods Module

nodes Module

optimization Module

params Module

sets Module

tables Module

Subpackages

elements Package

bars Module

bush Module

damper Module

elements Module

mass Module

rigid Module

shell Module

solid Module

springs Module

loads Package

loads Module

staticLoads Module

properties Package

bars Module

bush Module

damper Module

mass Module

properties Module

shell Module

springs Module

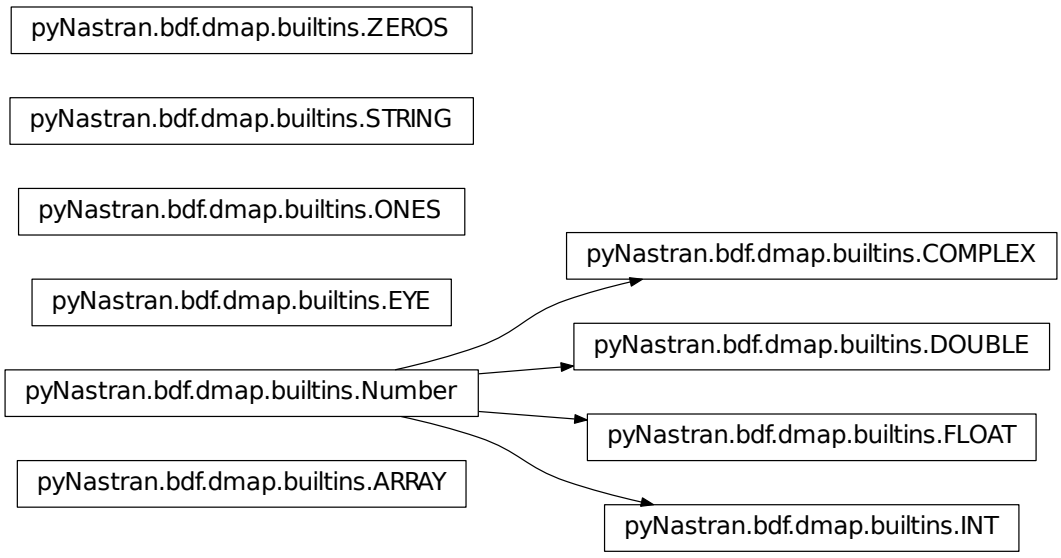
thermal Package

loads Module

thermal Module

dmap Package

builtins Module



class pyNastran.bdf.dmap.builtins.**ARRAY**(*name*, *nrows*, *ncols*)

Bases: object

shape()

class pyNastran.bdf.dmap.builtins.**COMPLEX**(*name*, *real*, *imag*)
Bases: *pyNastran.bdf.dmap.builtins.Number*

class pyNastran.bdf.dmap.builtins.**DOUBLE**(*real*)
Bases: *pyNastran.bdf.dmap.builtins.Number*

class pyNastran.bdf.dmap.builtins.**EYE**(*name*, *shape*, *dtype='i'*)
Bases: object

shape()

class pyNastran.bdf.dmap.builtins.**FLOAT**(*name*, *real*)
Bases: *pyNastran.bdf.dmap.builtins.Number*

class pyNastran.bdf.dmap.builtins.**INT**(*name*, *real*)
Bases: *pyNastran.bdf.dmap.builtins.Number*

class pyNastran.bdf.dmap.builtins.**Number**(*name*)
Bases: object

class pyNastran.bdf.dmap.builtins.**ONES**(*name*, *shape*, *dtype='i'*)
Bases: object

shape()

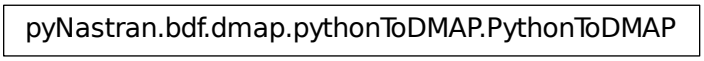
class pyNastran.bdf.dmap.builtins.**STRING**(*name*, *word*)
Bases: object

length(*globalsDict*)

class pyNastran.bdf.dmap.builtins.**ZEROS**(*shape*, *dtype='i'*)
Bases: object

shape()

pythonToDMAP Module



class pyNastran.bdf.dmap.pythonToDMAP.**PythonToDMAP**(*pyName*)
Bases: object

parse_assign(*variable*, *expr*)

parse_line(*line*)

parse_operation(*variable*, *operator*, *expr*)

parse_value(*variable*, *expr*)

run()

write_dmap(*bdf*)

write_operation(*operation*, *Type*, *var*, *operator*, *expr*, *comment*)

write_sub_operation(*operation*, *var*, *operator*, *expr*)

pyNastran.bdf.dmap.pythonToDMAP.**get_comment**(*line*)

converters Package

toCodeAster Module

Subpackages

LaWGS Package

wgsIO Module

wgsReader Module

cart3d Package

cart3dIO Module

cart3d_reader Module

toPanair Module

panair Package

panairGrid Module

panairGridPatch Module

panairIO Module

panairWrite Module

pyNastran.converters.panair.panairWrite.PanairWrite

class pyNastran.converters.panair.panairWrite.**PanairWrite**

Bases: object

printAbutments()

printGridSummary()

printOptions()

printOutHeader()

writeDataCheck()

writePrintout()

f06 Package

f06 Module

f06Writer Module

pyNastran.f06.f06Writer.F06Writer

class pyNastran.f06.f06Writer.**F06Writer**(*model='tria3'*)

Bases: object

loadOp2(*isTesting=False*)

makeF06Header()

If this class is inherited, the F06 Header may be overwritten

makeStamp(*Title*)

If this class is inherited, the PAGE stamp may be overwritten

setF06Name(*model*)

writeF06(*f06OutName*, *isMagPhase=False*, *makeFile=True*, *deleteObjects=True*)
Writes an F06 file based on the data we have stored in the object

Parameters

- **self** – the object pointer
- **f06OutName** – the name of the F06 file to write
- **isMagPhase** – should complex data be written using Magnitude/Phase instead of Real/Imaginary (default=False; Real/Imag) Real objects don't use this parameter.
- **makeFile** – True -> makes a file, False -> makes a StringIO object for testing (default=True)

pyNastran.f06.f06Writer.**makeEnd**()

pyNastran.f06.f06Writer.**makeF06Header**()

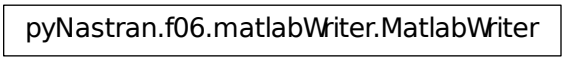
pyNastran.f06.f06Writer.**makeStamp**(*Title*)

f06_classes Module

class pyNastran.f06.f06_classes.**MaxDisplacement**(*data*)
Bases: object

writeF06(*pageStamp=''*, *pageNum=1*)

matlabWriter Module



class pyNastran.f06.matlabWriter.**MatlabWriter**(*model='tria3'*)
Bases: object

loadOp2(*isTesting=False*)

makeF06Header()
If this class is inherited, the F06 Header may be overwritten

makeStamp(*Title*)
If this class is inherited, the PAGE stamp may be overwritten

setF06Name(*model*)

writeMatlab(*mFileOutName*, *isMagPhase=False*, *deleteObjects=True*)
Writes an F06 file based on the data we have stored in the object

Parameters

- **self** – the object pointer
- **mFileOutName** – the name of the M (Matlab) file to write
- **isMagPhase** – should complex data be written using Magnitude/Phase instead of Real/Imaginary (default=False; Real/Imag) Real objects don't use this parameter.

pyNastran.f06.matlabWriter.**makeEnd**()

pyNastran.f06.matlabWriter.**makeF06Header**()

pyNastran.f06.matlabWriter.**makeStamp**(*Title*)

Subpackages

tables Package

oes Module

oqg Module

oug Module

general Package

general Module

mathematics Module

logger Module

pyNastran.general.logger.**buildDummyLogger2**(*level='debug'*)

class pyNastran.general.logger.**debugLogger**
Bases: object

critical(*msg*)

debug(*msg*)

error(*msg*)

fixMessage(*msg*, *n=54*)

fname()
Returns the current file in our program.

frame()

funcName()
Returns the current function name in our program.

info(*msg*)

lineno()
Returns the current line number in our program.

properties()

warning(*msg*)

class pyNastran.general.logger.**dummyLogger**
Bases: object

startLog(*level*)

pyNastran.general.logger.**fileNamE**()

pyNastran.general.logger.**fileNameLineNum**(*n=0*)

pyNastran.general.logger.**frame**(*n*)

pyNastran.general.logger.**getLogger**(*log=None*, *level='debug'*)
This function is useful as it will instantiate a dummy logger object if log=None log may be a logger object or None pyFile is the python file that the code was called from (unused)

class pyNastran.general.logger.**infoLogger**
Bases: *pyNastran.general.logger.debugLogger*

debug(*msg*)

pyNastran.general.logger.**lineNum**()

makeLog Module

pyNastran.general.makeLog.**get_graphic_card_properties**()

pyNastran.general.makeLog.**makeLog**()

utils Module

pyNastran.general.utils.**__object_attr**(*obj*, *mode*, *attr_type*)

pyNastran.general.utils.**object_attributes**(*obj*, *mode='public'*)
List the names of attributes of a class as strings. Returns public methos as default.

Parameters

- **obj** – the object for checking
- **mode** – defines what kind of attributes will be listed
  - "public" - names that do not begin with underscore
  - "private" - names that begin with single underscore
  - "both" - private and public
  - "all" - all attributes that are defined for the object

Returns sorted list of the names of attributes of a given type or None if the mode is wrong

pyNastran.general.utils.**object_methods**(*obj*, *mode='public'*)
List the names of methods of a class as strings. Returns public methos as default.

Parameters

- **obj** – the object for checking
- **mode** – defines what kind of methods will be listed
  - "public" - names that do not begin with underscore
  - "private" - names that begin with single underscore
  - "both" - private and public
  - "all" - all methods that are defined for the object

Returns sorted list of the names of methods of a given type or None if the mode is wrong

mesh Package

nastranMesh Module

op2 Package

fortranFile Module

class pyNastran.op2.fortranFile.**FortranFile**

Bases: object

getBlockIntEntry(*data*, *n*)
given a data set, grabs the nth word and casts it as an integer

getData(*n*)
gets a data set of length N

getDoubles(*data*)
unpacks a data set into a series of doubles

getFloats(*data*)
unpacks a data set into a series of floats

getFloats2(*data*, *endian*)
unpacks a data set into a series of floats

getInts(*data*, *debug=True*)
unpacks a data set into a series of ints

getInts2(*data*, *endian*, *debug=True*)
unpacks a data set into a series of ints

getLongs(*data*)
unpacks a data set into a series of longs

getMarker(*expected=None*, *debug=True*)

getNMarkers(*nMarkers*, *rewind=False*)
gets the next N markers, verifies they're correct

getStrings(*data*)
unpacks a data set into a series of characters

getStrings2(*data*, *endian*)
unpacks a data set into a series of characters

getTableCode(*expected=None*, *debug=True*)

goto(*n*)
jumps to position n in the file

Parameters

- **self** – the object pointer
- **n** – the position to goto

Note: n>0

---

hasMoreTables()

isTableDone(*expectedMarkers*)

printBlock(*data*, *nMax=200*)
prints a data set in int/float/double/string format to determine table info. doesn't move cursor.

Note: this is a great function for debugging

printBlock2(*data*, *endian*)
prints a data set in int/float/double/string format to determine table info. doesn't move cursor.

Note: this is a great function for debugging

printSection(*nBytes*)
prints data, but doesn't move the cursor

Parameters

- **self** – the object pointer
- **nBytes** – the number of bytes to print the data specs on

Returns msg ints/floats/strings of the next nBytes (handles poorly sized nBytes; uncrashable :))

Note: this the BEST function when adding new cards/tables/debugging

printSection2(*nBytes*, *endian*)
prints data, but doesn't move the cursor

Parameters

- **self** – the object pointer
- **nBytes** – the number of bytes to print the data specs on

Returns msg ints/floats/strings of the next nBytes (handles poorly sized nBytes; uncrashable :))

Note: this the BEST function when adding new cards/tables/debugging

readBlock()
reads a fortran formatted data block nWords data1 data2 data3 nWords

readData(*n*)

readDoubleBlock()
reads a fortran formatted block assumes that the data is made up of doubles only

readDoubles(*nData*, *debug=True*)
reads a list of nDoubles

Parameters

- **self** – the object pointer
- **nData** – the number of doubles to read
- **debug** – for developer: debug combined with makeOp2Debug

readFloatBlock()
reads a fortran formatted block assumes that the data is made up of floats only

readFloats(*nData*, *debug=True*)
reads nFloats

readFullBlock()
reads a fortran formatted data block nWords data1 data2 data3 nWords includes nWords in the output

readFullIntBlock()
reads a fortran formatted block assumes that the data is made up of integers only

readHeader(*expected=None*, *debug=True*)
a header is defined as (4,i,4), where i is an integer

readHollerith()
doesnt really read a hollerith, it's an integer of value=528 which corresponds to the length of iTable=3

readIntBlock()
reads a fortran formatted block assumes that the data is made up of integers only

readInts(*nInts*, *debug=True*)
reads a list of nIntegers

Parameters

- **self** – the object pointer
- **nInts** – the number of ints to read
- **debug** – for developer: debug combined with makeOp2Debug

readMarker(*expected=None*)

readMarkers(*markers*, *tableName=None*, *debug=False*, *printErrorOnFailure=True*)
Reads a set of predefined markers e.g. [-3,1,0] and makes sure it is correct.

A marker (e.g. a -3) is a series of 3 integers [4,-3,4]. Typically 3 markers are put together (e.g. [-3,1,0]) such that the integers are [4,-3,4, 4,1,4, 4,0,4] to mark important parts of the table.

Markers will "increment" during table reading, such that the first marker is [-1,1,0], then [-2,1,0], etc. Tables will end (or really the next table starts) when a [-1,1,0] or a [0,1,0] marker is found.

Verify the following statement... Occassionally, buffer markers will be embedded inside the marker [-3,1,0], (e.g. [4,2^16,4] <- the default BUFFSIZE), which make reading the marker more difficult.

readString(*nData*)
reads nCharacters that are assumed to be a string

readStringBlock(*debug=True*)
reads a fortran formatted block assumes that the data is made up of characters only

readTableName(*rewind=True*, *debug=True*, *stopOnFailure=True*)
peeks into a table to check it's name

rewind(*n*)
rewinds the file nBytes

> **Warning:** doesnt support a full rewind, only a partial

scan(*n*)
same as skip, but actually reads the data instead of using seek

setEndian(*endian=u'<'*)
sets the endian

Todo

hasnt been implemented

---

skip(*n*)
skips nBits

skipNextTable(*bufferSize=10000*)
skips a table

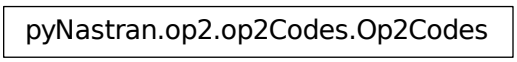
Todo

fix bugs

---

op2 Module

op2Codes Module



class pyNastran.op2.op2Codes.**Op2Codes**
Bases: object

ElementType(*eCode*)

codeInformation()
prints the general table information DMAP - page 60-63

isMagnitudePhase()

isRandomResponse()

isReal()

isRealImaginary()

isRealImaginaryOrMagnitudePhase()

isRealOrRandom()

isSort1()

isSort2()

isSortedResponse()

isStress()

isThermal()

printTableCode(*tableCode*)

op2_helper Module

Subpackages

resultObjects Package

op2_Objects Module

tableObject Module

tables Package

destab Module

pyNastran.op2.tables.destab.DESTAB

class pyNastran.op2.tables.destab.**DESTAB**
Bases: object

readDesvar(*data*)

readTable_DesTab()

ept Module

mpt Module

r1tab Module

class pyNastran.op2.tables.r1tab.**R1TAB**
Bases: object

readTable_R1TAB()

resultTable Module

Subpackages

geom Package

dit Module

dynamics Module

geom1 Module

geom2 Module

geom3 Module

geom4 Module

geometryTables Module

lama_eigenvalues Package

lama Module

lama_objects Module

oee_energy Package

oee Module

oee_objects Module

oef_forces Package

complexForces Module

oef Module

oef_Objects Module

oef_complexForceObjects Module

oef_forceObjects Module

oef_thermalObjects Module

realForces Module

pyNastran.op2.tables.oef_forces.realForces.RealForces

class pyNastran.op2.tables.oef_forces.realForces.**RealForces**

Bases: object

OEF_Beam()

OEF_Bend()

OEF_CBar()

OEF_CBar100()

OEF_CBush()

OEF_CGap()

OEF_CVisc()

OEF_ConeAx()

OEF_Force_VU()

OEF_Force_VUTRIA()

OEF_PentaPressure()

OEF_Plate()

OEF_Plate2()

OEF_Rod()

OEF_Shear()

OEF_Spring()

OEF_aCode()

getOEF_FormatStart()
Returns an i or an f depending on if it's SORT2 or not. Also returns an extraction function that is called on the first argument

thermal_elements Module

oes_stressStrain Package

oes Module

oes_nonlinear Module

oesnlxr Module

class pyNastran.op2.tables.oes_stressStrain.oesnlxr.**OESNLXR**
Bases: *pyNastran.op2.tables.oes_stressStrain.real.elementsStressStrain.RealElementsStressStrain*

Table of stresses/strains

deleteAttributes_OESNLXR()

readOESNLXR_Data_format1_sort0()

readOESNLXR_ElementTable()

readTable_OESNLXR()

readTable_OESNLXR_3(*iTable*)

readTable_OESNLXR_4_Data(*iTable*)

Subpackages

complex Package

elementsStressStrain Module

oes_bars Module

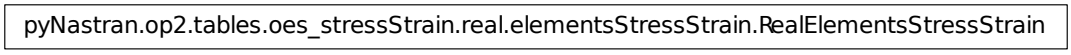
oes_plates Module

oes_rods Module

oes_springs Module

real Package

elementsStressStrain Module



class pyNastran.op2.tables.oes_stressStrain.real.elementsStressStrain.**RealElementsStr**

Bases: object

OES_CBAR_34()

OES_CBEAM_2()

OES_CBEAM_94()

OES_CHEXANL_93()
The DMAP manual says fields 3-18 repeat 9 times. but they dont. They repeat 8 times. Other DMAP cards are correct with their repeat statements.

OES_CPENTANL_91()
The DMAP manual says fields 3-18 repeat 7 times. but they dont. They repeat 6 times. Other DMAP cards are correct with their repeat statements.

OES_CQUAD4NL_90()

OES_CQUAD4_144()
GRID-ID DISTANCE,NORMAL-X,NORMAL-Y,SHEAR-XY,ANGLE,MAJOR MINOR,VONMISES

OES_CQUAD4_33()
GRID-ID DISTANCE,NORMAL-X,NORMAL-Y,SHEAR-XY,ANGLE,MAJOR MINOR,VONMISES

OES_CQUAD4_95()
GRID-ID DISTANCE,NORMAL-X,NORMAL-Y,SHEAR-XY,ANGLE,MAJOR MINOR,VONMISES composite quad

OES_CQUADR_82()
GRID-ID DISTANCE,NORMAL-X,NORMAL-Y,SHEAR-XY,ANGLE,MAJOR MINOR,VONMISES

OES_CSOLID_67()
stress is extracted at the centroid CTETRA_39 CPENTA_67 CHEXA_68

OES_CSOLID_85()
stress is extracted at the centroid CTETRA_85 CPENTA_91 ??? CHEXA_93 ???

OES_CTRIA3_74()
DISTANCE,NORMAL-X,NORMAL-Y,SHEAR-XY,ANGLE,MAJOR,MINOR,VONMISES
stress is extracted at the centroid

OES_CTRIAX6_53()

OES_QUAD4FD_139()
Hyperelastic Quad 36+4*7*4 = 148

OES_RODNL_89_92()

OES_Thermal(*debug=False*)

OES_basicElement()
genericStressReader - works on CROD_1, CELAS2_12 stress & strain formatCode=1 sortCode=0 (eid,axial,axialMS,torsion,torsionMS)

OES_field1()

dummyPass()

skipOES_Element()

oqg Module

oqg_mpcForces Module

oqg_spcForces Module

oug Package

oug Module

oug_Objects Module

oug_accelerations Module

oug_displacements Module

oug_eigenvectors Module

oug_temperatures Module

oug_velocities Module

writer Package

oesWriter Module

class pyNastran.op2.writer.oesWriter.**Oes1Writer**

Bases: object

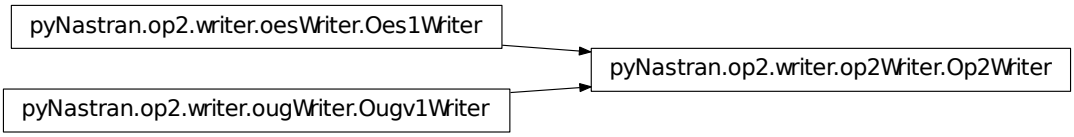
writeOES1()

writes isotropic/composite stress/strain

Todo

assumes sCode=0 (stress) or 10 (strain)

op2Writer Module



class pyNastran.op2.writer.op2Writer.**Op2Writer**

Bases: *pyNastran.op2.writer.ougWriter.Ougv1Writer*, *pyNastran.op2.writer.oesWriter.Oes1Writer*

aCode_tCode(*approachCode*, *tableCode*, *sortCode*)

combineApproachDeviceCodes(*approachCode*)

combineTableSortCodes(*tableCode*, *sortCode*)

hollerith = 'H\x02\x00\x00'

pack(*format*, **vals*)

packTitle(*iSubcase*)

printHeader(*word*, *nChars*)

writeMarkers(*markers*)

takes -5,1,0 -> [4,5,4, 4,1,4, 4,0,4] and puts it into binary

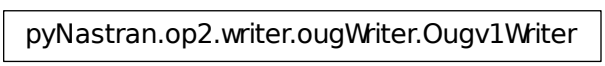
writeOp2(*op2Name*)

writeStart()

writeStringBlock(*word*, *nChars*)

'OUG' - 1 word = 4 characters 'OUGV' - 1 word = 4 characters 'OUGV1' - 2 words = 8 characters '12345678' - 2 words = 8 characters nWords = round(ceil(len(word)/4.)) nChars = nWords*4 != len(word) just set nChars and dont overthink it too much

ougWriter Module



class pyNastran.op2.writer.ougWriter.**Ougv1Writer**
Bases: object

writeOUGV1()

writeOUG_displacements(*iSubcase*, *data*, *thermal=0*)
this function writes table 3 for the OUGV1 Table

Todo

add the buffer and block caps

op4 Package

op4 Module

sovler Package

beam Module

makeFrame Module

marginChecker Module

panelBuckling Module

plate Module

pyNastranSolver Module

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

H

I

L

M

N

O

R

S

U

W

X

Z