Material Database Manual

Usage & Extensibility

by Nathaniel Starkman

This is a Manual for the Material Database. It details all the functions in the database. If any function is added or modified this database should be updated to reflect that. Do your part.

Using the Database:

Using the Database with the Calculator

Open the Material_Database_Calculator. call the desired function with the correct inputs. For more information on a function, find its specific information section in this document. The necessary steps to reference a material are also listed below.

Using the Database via Terminal

First navigate into the correct directory. This is done with the commands ‘cd’ followed by the directory path. ‘ls’ is a useful tool for listing the contents of the current directory. For more information, click this link.

Once in the correct directory access i-python or any interactive python console. The exact command depends on the operating system. On a mac, simply type ‘python’. Once an interactive python console has been initialized, type (or copy & paste), excluding the quotes, “from Material_Database_Data import *”. This will import all the modules of the Material_Database into the current terminal session.

Next call the function desired with all the correct inputs. For more information on a function, find its specific information section in this document. The necessary steps to reference a material are also listed below.

Referencing a Material

Materials need to be referenced via their handle. If the material is a temporary one, then the handle is whatever was assigned.

```
material_handle = Mat_Class(inputs)
```

Else, the material handle is the materials place in the Material Database. The Database stores all materials in a dictionary intuitively called *materials*. Entries in this dictionary are accessed via normal python syntax.

```
materials[‘material name’]
```

If the material is stored in a nested dictionary then it is referenced like this —

```
materials['sub-dictionary name here']['material name']
```

If there are more nested dictionaries just use more [' '] to get to the desired material.

If you don't know the material's name or what dictionary it is in, find it in Material_Database_Data.py.

Creating a Material:

Making a Material for Temporary Use

** Note: more detailed instructions in the usage of Mat_Class() may be found it it's specific section.

This section details how to create a material without adding it to the Material Database. This can be useful in such situations where only an inexact approximate is needed and as such should not be added to the database incase someone mistakes it for a good model.

Depending on how the Database is being accessed, the following instruction will vary.

From Terminal:

Double check that the correct directory is selected, an interactive python environment is running, and all the modules have been imported. To do this, or to double check that everything has been done correctly, read "Using the Database via Terminal Section".

From Material_Database_Calculator:

Continue on. Everything is already set up.

Create the material by calling Mat_Class() with a name as an input.

```
material = Mat_Class('name')
```

Next call the add_model sub-function with the correct inputs, detailed subsequently.

```
material.add_model(input)
```

The input should be a dictionary. Of the following keys in the dictionary, only 'matProp' and 'name' are mandatory (though a material is not much use without an equation).
Keys:

- 'matProp': the material property, such as thermal conduction. The entry should be a string.
- 'name': the model name since each material property can have many different fit models. The entry should be a string. Common names are 'NIST', 'data', and 'mixed'.
- 'eq': the equation for the model. Should be a function. Some usable functions are stored in Materia_Database_Functions and are listed later. Else, a lambda function is

perfectly acceptable. It is best practice to define the lambda function beforehand and call its handle here. For more information on lambda functions, go here.

- 'eqinput': the input to the equation. Can be coefficients in form [coeffs], or raw data in form [[temps], [data]], or a mixture in form [[coeffs], [[temps], [data]], [[coeffs_min, coeffs_max], [data_min, data_max]]]
- 'eqrange': the range of validity for the model. Must be given as a list in the form [min, max].
- 'd_eq': the error in the equation. Can be a constant or a one-variable function.

Example:

```
material = Mat_Class('material name', {'matProp': 'material property', 'name': 'model name', 'eq': equation, 'eqr': [min, max], 'd_eq': lambda x: 0.1*x}, fullname='material fullname')
```

The 'eqr' (equation range), and 'd_eq' (equation error) are optional arguments. If no 'eqr' is given the program will raise a warning but will work normally otherwise. If no 'd_eq' is given it will be assumed to be 0.

Remember, items in a dictionary do not need a particular order, nor do kwargs. The only order of importance is that the *material name* precedes the *material property*'s dictionary.

Since the material is not being added to the Database, it need not be put into any dictionary, so referencing it will be very easy.

Adding a Material to the Database

This cannot easily be done from terminal. The best way to add a material to the Database is to open Material_Database_Data in a text editor.

Creating a Material

ONE LINER:

A material can be created in one line by calling Mat_Class, feeding it a name, an arbitrary number of material properties and models, and some optional kwargs.

Example of 1 model (numbers and names only stand-ins):

```
material = Mat_Class('material name', {'matProp': 'material property', 'name': 'model name', 'eq': equation, 'eqr': [min, max], 'd_eq': lambda x:
```

```
0.1*x}, UNS='UNS', fullname='material fullname', area=1, d_area=0.1,
length=1, d_length=0.1, info='info')
```

Example of 3 models (numbers and names only stand-ins)

```
material = Mat_Class('material name', {'matProp': 'material property',
'name': 'model name', 'eq': equation, 'eqr': [min, max], 'd_eq': lambda x:
0.1*x}, {'matProp': 'material property', 'name': 'model name', 'eq':
equation, 'eqr': [min, max], 'd_eq': lambda x: 0.1*x}, {'matProp':
'material property', 'name': 'model name', 'eq': equation, 'eqr': [min,
max], 'd_eq': lambda x: 0.1*x}, UNS='UNS', fullname='material fullname',
area=1, d_area=0.1, length=1, d_length=0.1, info='info')
```

Giving the same material property but different model will not result in an overwrite of an existing model for that material property (only giving the same material property and model will overwrite a previous model), so many models may be added for one material property. For more information on Mat_Class(), see the Mat_Class information section here.

The 'eqr' (equation range), and 'd_eq' (equation error) are optional arguments. If no 'eqr' is given the program will raise a warning but will work normally otherwise. If no 'd_eq' is given it will be assumed to be 0.

Remember, items in a dictionary do not need a particular order, nor do kwargs. The only order of importance is that the *material name* precedes the *material property*'s dictionary.

IN PARTS:
A material can be created in many parts using the whole suite of tools available.

First call Mat_Class() with as many arguments / kwargs as desired. The model's dictionary does not need to be fully populated as any values may be added in later.

```
material = Mat_Class('name', any desired inputs)
```

Choice Menu:

- If a non-model property needs to be defined / overwritten: see section *Modifying an Existing Material*, subsection *Adding / Overwriting a non-Model Property*
- If a material property and model were not yet defined: see section *Modifying an Existing Material*, subsection *Adding a Material Property to a Material*
- If a material property and model was already defined:
  - If a new material property and model is required: see section *Modifying an Existing Material*, subsection *Adding a Material Property to a Material*

- If a new model for a material property is required: see section *Modifying an Existing Material*, subsection *Adding / Overwriting a Material Property Model*
- If a model needs to be overwritten: see section *Modifying an Existing Material*, subsection *Adding / Overwriting a Material Property Model*
- If fields need to be added to a model: see section *Modifying an Existing Material*, subsection *Adding to / Overwriting a Model's Specific Fields*

Putting the Material in a Dictionary

materials can be stored in dictionaries for organizational purposes and easy access.

NEW DICTIONARY:

To put a material in a dictionary, call make_dict() with the material's handle as an argument and a kwarg — name='dictionary name' (the name in " " is a stand-in).

Example:

```
example_dictionary = make_dict(material, name='example_dictionary')
```

for more information see the make_dict information section here

EXISTING DICTIONARY:

To put a material in an existing dictionary, call make_dict() with the material's handle as an argument and a kwarg — dict='dictionary name' (the name in " " is a stand-in).

Example:

```
example_dictionary = make_dict(material, dict='example_dictionary')
```

Having the same handle as in the kwarg dict='' will just add the material to the already existing dictionary. Having a different handle as in the kwarg dict='' will create a new dictionary with the new material appended to the old dictionary.

Putting the Dictionary in a Dictionary

Dictionaries may be put inside of dictionaries. An instance where this is useful is having a Stainless Steel dictionary as a sub-dictionary of the Steel dictionary.

NEW DICTIONARY:

To put a dictionary in an new dictionary, call make_dict() with the material's handle as an argument and a kwarg — name='dictionary name' (the name in " " is a stand-in).

Example:

```
new_dictionary = make_dict(sub_dictionary, name='new_dictionary')
```

EXISTING DICTIONARY:

To put a dictionary in an existing dictionary, call make_dict() with the dictionary's handle as an argument and a kwarg — dict='dictionary name' (the name in " " is a stand-in).

Example:

```
example_dictionary = make_dict(sub_dictionary, dict='example_dictionary')
```

Having the same handle as in the kwarg dict='' will just add the dictionary to the already existing dictionary. Having a different handle as in the kwarg dict='' will create a new dictionary with the new dictionary appended to the old dictionary.

Modifying an Existing Material:

Note: this section will give instructions for how to add a Material Property to a Material, however many of the listed functions are not completely detailed below. For complete information on all of their options and capabilities, see their respective sections.

All materials may be modified or overwritten. Some functions will overwrite previous data, some others will not. This is detailed below.

Changing Information in a Model

The easiest way is to find the material and change the information. If the material is a temporary one made in terminal, overwrite it.

To see another method of changing information in a model go to the *Adding / Overwriting a Material Property Model* section.

Further methods are detailed in the *Function References section;* specifically any function that starts with 'add_'.

Adding a Material Property to a Material

Note: this section will give instructions for how to add a Material Property to a Material, however many of the listed functions are not completely detailed below. For complete information on all of their options and capabilities, see their respective sections.

To add a material property to a material, a model for that material property is also needed. Call add_model(), with the correct inputs, after the material handle.

```
material.add_model(input)
```

If the material was a temporary one, then the handle is whatever you gave it. If it is in the Database then it should be referenced as materials['material name'] — if it is a stand-alone item — or materials['dictionary name']['material name'] — if it is in a sub-dictionary of the materials dictionary. For more information on material handles, see the section on referencing materials here.

The input should be a dictionary. Of the following keys for the dictionary, only ‘matProp’ and ‘name’ are mandatory (though the model & material property are not much use without an equation).

- ‘matProp’: the material property, such as thermal conduction. The entry should be a string.
- ‘name’: the model name since each material property can have many different fit models. The entry should be a string. Common names are ‘NIST’, ‘data’, and ‘mixed’.
- ‘eq’: the equation for the model. Should be a function. Some usable functions are stored in Materia_Database_Functions and are listed in the *Material Properties* section. Else, a lambda function is perfectly acceptable. It is best practice to define the lambda function beforehand and call its handle here. For more information on lambda functions, click this link. For more information on ‘eq’ see the add_equation section.
- ‘eqinput’: the input to the equation. Can be coefficients in form [coeffs], or raw data in form [[temps], [data]], or a mixture in form [[coeffs], [[temps], [data]], [[coeffs_min, coeffs_max], [data_min, data_max]]]. For more information on ‘eqinput’ see the add_eqinput section.
- ‘eqrange’: the range of validity for the model. Must be given as a list in the form [min, max]. For more information on ‘eqrange’ see the add_eqrange section.
- ‘d_eq’: the error in the equation. Can be a constant or a one-variable function. for more information on ‘d_eq’ see the add_eqerror section.

Example:

```
material.add_model({‘matProp’: ‘material property', ‘name': ‘model name', ‘eq': equation, 'eqr': [min, max], 'd_eq': lambda x: 0.1*x})
```

The ‘eqr’ (equation range), and ‘d_eq’ (equation error) are optional arguments. If no ‘eqr’ is given the program will raise a warning but will work normally otherwise. If no ‘d_eq’ is given it will be assumed to be 0.

Remember, items in a dictionary do not need a particular order, nor do kwargs. The only order of importance is that the *material name* precedes the *material property*’s dictionary.

Adding / Overwriting a Material Property Model

Note: this section will give instructions for how to add a Material Property to a Material, however many of the listed functions are not completely detailed below. For complete information on all of their options and capabilities, see their respective sections.

To add a model to a material property, call add_model(), with the correct inputs, after the material handle.

```
material.add_model(input)
```

If the material was a temporary one, then the handle is whatever was given. If it is in the Database then it should be referenced as materials['material name'] — if it is a stand-alone item — or materials['dictionary name']['material name'] — if it is in a sub-dictionary of the materials dictionary. For more information on material handles, see the section on referencing materials here.

The input should be a dictionary. Of the following keys for the dictionary, only 'matProp' and 'name' are mandatory (though the model & material property are not much use without an equation).

- 'matProp': the material property. Call an existing material property. The entry should be a string.
- 'name': the model name since each material property can have many different fit models. The entry should be a string. Common names are 'NIST', 'data', and 'mixed'. **If the name of an existing model (for the given material property) is called, add_model() will overwrite the all the previous data in that model.**
- 'eq': the equation for the model. Should be a function. Some usable functions are stored in Materia_Database_Functions and are listed later. Else, a lambda function is perfectly acceptable. It is best practice to define the lambda function beforehand and call its handle here. For more information on lambda functions, click this link. For more information on 'eq' see the add_equation section.
- 'eqinput': the input to the equation. Can be coefficients in form [coeffs], or raw data in form [[temps], [data]], or a mixture in form [[coeffs], [[temps], [data]], [[coeffs_min, coeffs_max], [data_min, data_max]]]. For more information on 'eqinput' see the add_eqinput section.
- 'eqrange': the range of validity for the model. Must be given as a list in the form [min, max]. For more information on 'eqrange' see the add_eqrange section.

- ‘d_eq’: the error in the equation. Can be a constant or a one-variable function. for more information on ‘d_eq’ see the add_eqerror section.

Example:

```
material.add_model({'matProp': ‘material property', ‘name': ‘model name', ‘eq': equation, 'eqr': [min, max], 'd_eq': lambda x: 0.1*x})
```

Adding to / Overwriting a Model’s Specific Fields

Any existing *material property*’s *model*’s fields may be overwritten (as they should all be initialized as None).

The method of overwriting a specific field is to call a *add_X* function, where X is a stand-in for a specific function name. The available *add_X* functions are:

- *add_model*
- *add_equation*
- *add_eqrange*
- *add_eqerror*
- *add_eqinput*
- *add_anything*

For more information on any one of these functions see its section in the *Function Reference* section of this manual.

Adding / Overwriting a non-Model Property

Any existing *material property*’s fields may be overwritten (as they should all be initialized as None).

The method of overwriting a specific field is to call a *add_X* function, where X is a stand-in for a specific function name. The available *add_X* functions are:

- *add_area*
- *add_area_error*
- *add_length*
- *add_length_error*
- *add_info*
- *add_UNS*
- *add_fullname*
- *add_anything*

For more information on any one of these functions see its section in the *Function Reference* section of this manual.

Function Reference

This section gives details on many of the functions in the Material Database. Model-specific function information is given in the *Material Properties* section. Additionally, this section may reference previous sections as they were written for easy use and there is not much point writing the same thing here as in a more useful section.

Mat_Class()

Mat_Class is the class which all materials in the Material Database belong to. Mat_Class holds all the functions beginning with '*add_'* and their sections should be read for specifics. Mat_Class must be called with an input string which will be assigned as the material's name. Mat_Class also has its own __init__ function which may take, besides the material's name, an unlimited number of models (each one it's own dictionary) and kwargs with such properties as area, info, fullname, etc. For specifics on how to use the init function see the section *Adding a Material to the Database*, subsection *Creating a Material*, subsection *One Liner.*

add_model

This function can add a material property &/or model to a material. Good instructions for using this function are provided in the *Modifying an Existing Material* section. It should be noted that any item in the dictionary can alternatively be given as a kwarg. kwargs take precedence over dictionary items, so be careful not to give something twice as they won't both be added, but one will be ignored.

add_anything

This function can add any/all of listed below to a material if the correct kwarg(s) is given:

- *add_area:* kwarg *area =*
- *add_area_error*: kwarg *d_area =*
- *add_length*: kwarg *length =*
- *add_length_error*: kwarg *d_length =*
- *add_info*: kwarg info *=*
- *add_UNS*: kwarg *UNS =*
- *add_fullname*: kwarg *fullname =*

- *add_model*: kwarg *model =*
- *add_equation*: kwargs *equation =* , *matProp =* , & *name =*
- *add_eqrange*: kwargs *eqrange =* , *matProp =* , & *name =*
- *add_eqerror*: kwargs *eqerror =* , *matProp =* , & *name =*
- *add_eqinput*: kwargs *eqinput =* , *matProp =* , & *name =*

See each specific function for the format of the input given in the kwargs.

Just a side-note, this function can add two complete models to a material as well as the materials area, length, d_area, d_length, info, UNS, and fullname. The first model can be added through the kwarg *model*, and the second through calling kwargs *equation*, *eqrange*, *eqerror*, and *eqinput*.

add_area

This function adds the area of a material to that material. While it technically accepts any argument, best practice is to pass it a number or None-type argument. If no argument is passed, the area is taken as None.

add_area is called like this:

```
material.add_area(inputs) # inputs is a stand-in
```

The initial *material* means the material to which the area will be added.

The only input to this function is the area as either an arg or a kwarg. The kwarg should be called by

```
area =
```

add_area_error

This function adds the error in the area of a material to that material. While it technically accepts any argument, best practice is to pass it a number or None-type argument. If no argument is passed, the area_error is taken as None.

add_area_error is called like this:

```
material.add_area_error(inputs) # inputs is a stand-in
```

The initial *material* means the material to which the area_error will be added.

The only input to this function is the area_error as either an arg or a kwarg. The kwarg should be called by

```
d_area =
```

add_length

This function adds the the length of a material to that material. While it technically accepts any argument, best practice is to pass it a number or None-type argument. If no argument is passed, the length is taken as None.

add_length is called like this:

```
material.add_length(inputs) # inputs is a stand-in
```

The initial *material* means the material to which the length will be added.

The only input to this function is the length as either an arg or a kwarg. The kwarg should be called by

```
length =
```

add_length_error

This function adds the error in the length of a material to that material. While it technically accepts any argument, best practice is to pass it a number or None-type argument. If no argument is passed, the length_error is taken as None.

adel_length_error is called like this:

```
material.add_length_error(inputs) # inputs is a stand-in
```

The initial *material* means the material to which the length_error will be added.

The only input to this function is the length_error as either an arg or a kwarg. The kwarg should be called by

```
d_length =
```

add_info

This function adds any given extra information about a material to that material. While it technically accepts any argument, best practice is to pass it a string or None-type argument. If no argument is passed, the UNS is taken as None.

add_info is called like this:

```
material.add_info(inputs) # inputs is a stand-in
```

The initial *material* means the material to which the info will be added.

The only input to this function is the info as either an arg or a kwarg. The kwarg should be called by

```
info = '    '
```

add_UNS

This function adds the UNS designation of a material to that material. While it technically accepts any argument, best practice is to pass it a string or None-type argument. If no argument is passed, the UNS is taken as None.

add_UNS is called like this:

```
material.add_UNS(inputs) # inputs is a stand-in
```

The initial *material* means the material to which the UNS will be added.

The only input to this function is the UNS as either an arg or a kwarg. The kwarg should be called by

```
UNS = '    '
```

add_fullname

This function adds the full name of a material to that material. While it technically accepts any argument, best practice is to pass it a string or None-type argument. If no argument is passed, the full name is taken as None.

add_fullname is called like this:

```
material.add_fullname(inputs) # inputs is a stand-in
```

The initial *material* means the material to which the full name will be added.

The only input to this function is the full name as either an arg or a kwarg. The kwarg should be called by

```
fullname = '    '
```

add_eqrange

This function adds the range of validity for a model's equation. The equation range is used to assess whether the equation should be used in a particular instance.

add_eqrange is called like this:

```
material.add_input(inputs)  # inputs is a stand-in
```

The initial *material* means the material to which the equation range will be added.

There must be 3 inputs to this function:

- *matProp*: the material property to whose model *eqrange* is being added.
  - must be a string.
  - can be either an arg or kwarg *matProp=*

    - If it is an arg, there can be no preceding arguments
- *name*: the name of the model to which *eqrange* is being added.
    - must be a string.
    - can be either an arg or kwarg *name=*
    - If it is an arg, the only argument that can precede it is *matProp*
- *eqrange*: the range of validity for the model's equation
    - must be a 2-item list with integer or float entries. None-type entries may be allowable, depending on the model.
        ```
        [1, 300]
        ```
    - kwarg can be called either *eqrange* or *eqr*
    - If it is an arg, the only arguments that can precede it are *matProp* and *name*

Some examples of calling *add_eqrange* to demonstrate the arg, kwarg ordering

```
material.add_eqinput(material_property, model_name, eqrange)
material.add_eqinput(material_property, eqerror, name = 'example')
material.add_eqinput(matProp='X', name='Y', eqerror=[1, 4, 7])
```

add_equation

This function adds an equation to a material property's model. An example usage of an equation is to calculate the thermal conductivity of a material.

add_equation is called like this:

```
material.add_equation(inputs)  # inputs is a stand-in
```

The initial *material* means the material to which the equation will be added.

There must be 3 inputs to this function:

- *matProp*: the material property to whose model an equation is being added.
    - must be a string.
    - can be either an arg or kwarg *matProp=*
    - If it is an arg, there can be no preceding arguments
- *name*: the name of the model to which an equation is being added.
    - must be a string.
    - can be either an arg or kwarg *name=*
    - If it is an arg, the only argument that can precede it is *matProp*
- *equation*: the equation
    - must be a function — lambda or otherwise. Depending on the model, None-type entries are allowable.

    - Some illustrative examples:
      ```
      lambda x: x**2
      thermCond_NIST_model
      ```
  - kwarg can be called either *equation* or *eq*
  - If it is an arg, the only arguments that can precede it are *matProp* and *name*

Some examples of calling *add_equation* to demonstrate the arg, kwarg ordering

```
material.add_equation(material_property, model_name, equation)
material.add_equation(material_property, equation, name = 'example')
material.add_equation(matProp='X', name='Y', equation= equation)
```

add_eqinput

This function adds an equation input to a material property's model. The equation input is a variable that is fed into the equation. Common such instances are: a list of coefficients to variables, raw data for a best fit function, or a mix of both.

add_eqinput is called like this:

```
material.add_input(inputs)  # inputs is a stand-in
```

The initial *material* means the material to which the equation input will be added.

There must be 3 inputs to this function:

- *matProp*: the material property to whose model *eqinput* is being added.
  - must be a string.
  - can be either an arg or kwarg *matProp=*
  - If it is an arg, there can be no preceding arguments
- *name*: the name of the model to which *eqinput* is being added.
  - must be a string.
  - can be either an arg or kwarg *name=*
  - If it is an arg, the only argument that can precede it is *matProp*
- *eqinput*: the inputs for the model's equation
  - may be in whatever data format the equation takes. Some common formats are a list of coefficients or raw data. Depending on the model, None-type entries may be allowable.
    - Some illustrative examples:
      ```
      [1, 2, 3, 4, 5, 6, 7, 8] # coefficients
      [[1, 10, 50, 100, 200, 300], [0.1, 3, 6, 9, 12, 15]] # raw data
      ```
    - functions will be fed the equation to produce the error bar specific to that value

- kwarg can be called either *eqinput* or *eqin*
- If it is an arg, the only arguments that can precede it are *matProp* and *name*

Some examples of calling *add_input* to demonstrate the arg, kwarg ordering

```
material.add_eqinput(material_property, model_name, eqinput)
material.add_eqinput(material_property, eqerror, name = 'example')
material.add_eqinput(matProp='X', name='Y', eqerror=[1, 4, 7])
```

add_eqerror

This function adds an equation error to a material property's model. The equation error is used to find the absolute uncertainty in the equation.

add_eqerror is called like this:

```
material.add_eqerror(inputs)  # inputs is a stand-in
```

The initial *material* means the material to which an equation error will be added.

There must be 3 inputs to this function:

- *matProp*: the material property to whose model *eqerror* is being added.
  - must be a string.
  - can be either an arg or kwarg *matProp=*
  - If it is an arg, there can be no preceding arguments
- *name*: the name of the model to which *eqerror* is being added.
  - must be a string.
  - can be either an arg or kwarg *name=*
  - If it is an arg, the only argument that can precede it is *matProp*
- *eqerror*: the error in the equation
  - may be an integer, float, single-variable equation, 1 item list with integer, float, or single-variable equation entries, or 2 item list with integer, float, or single-variable equation entries. Depending on the model, None-type entries may be allowable.
    - Some illustrative examples:

      ```
      1 or 1.0 or lambda x: x or [1] or [1.0] or [lambda x: x] or [1, 1.0]
      or [1.0, lambda x: x]
      ```

    - If it is given as a single entry, in list format or not, the error is assumed to be symmetric around the equation
    - If it is given as a 2-item list then the first entry is the lower error bound and the second entry is the upper error bound
    - numbers are assumed to be absolute errors

    - functions will be fed the equation to produce the error bar specific to that value
  - can be either an arg or a kwarg.
  - kwarg can be called either *eqerror* or *d_eq*
  - If it is an arg, the only arguments that can precede it are *matProp* and *name*

Some examples of calling *add_eqerror* to demonstrate the arg, kwarg ordering

```
material.add_eqerror(material_property, model_name, eqerror)
material.add_eqerror(material_property, eqerror, name = 'example')
material.add_eqerror(matProp='X', name='Y', eqerror=[1, 4.0])
```

make_dict

The function make_dict allows the creation or updating of a dictionary. The accepted inputs to this function are dictionaries and classes. Dictionaries must have an entry with a 'name' key and classes must have a .name field.

To make a dictionary call make_dict() with the kwarg *name='dictionary name'* (the name in " " is a stand-in). This won't really do anything since *make_dict()* has no handle.

```
make_dict(name='example_dictionary')
```

to use the dictionary later, give make_dict() a handle when it was first called:

```
example_dictionary = make_dict(name='example_dictionary')
```

This will make *example_dictionary* a dictionary with one entry — 'name': 'example_dictionary'. The handle does not need to be a new variable. Any existing object may be converted to a dictionary with *make_dict*.

There are three options for putting objects inside of *example_dictionary*. The first is to have called make_dict() with the object's handle as an argument as well as the kwarg *name='dictionary name'* (the name in " " is a stand-in).

```
example_dictionary = make_dict(object, name='example_dictionary')
```

The second is to call *make_dict()* with the object's handle as an argument as well as the kwarg *dict='example_dictionary'*. This will update an existing dictionary.

```
example_dictionary = make_dict(object, dict='example_dictionary')
```

If the dictionary's handle is not the same as in the kwarg, then make_dict() will not have updated an existing dictionary, but made a new one consisting of the concatenation of the arguments and kwarg

```
concatenated_dictionary = make_dict(object, dict='example_dictionary')
```

Remember that the input objects may be classes or dictionaries. Nesting dictionaries can be quite useful; for instance having a stainless steel dictionary as a sub-dictionary of the steel dictionary is better than separating stainless steel and normal steel into two dictionaries

is_number

This function is not implemented anywhere, but it checks whether an input is a number and returns True or False if it is or isn't.

Material Properties

Thermal Conductivity

Description here

heat_load

Calculates the heat load with equation:

```
thermCond * area * deltaT / length
```

thermCond * deltaT is calculated by calling *thermCond_integral*.

It also calculates the error in the heat load calculation by calling *thermCond_error*.

INPUTS:

- Must be given (material, modelname, bounds) in that order
  - any can be a kwarg, and so given with all the other kwargs\
  - must be strings
  - Do not give something as both an arg and a kwarg!
- May be given area and/or length as kwargs.
  - can be called as either *area* or *A*, *length* or *L*
- May be given error in area and length as kwargs.
  - can be called as either *area_error* or *dA*, *length_error* or *dL*
  - can be an upper error and lower error in form [lower, upper]

Any optional input that isn't given will be taken from the material itself. So if the material doesn't have one of the inputs and it is not given as a kwarg, then either an error will be raised or the value of the missing input will be taken as 0.

thermCond_integral

This function is used to do the integral of the thermal conductivity equation between the given bounds. *thermCond_integral* prints an error if the bounds are outside the range of validity but will still do the integral using scipy.integrate.quad. Additionally, this function will automatically feed eqinput into the equation, unless eqinput is *None*.

thermCond_integral is called by *heat_load* but can also be called by hand as in the following example.

```
integral, error = thermCond_integral(material.thermCond[modelname], [lower,
upper])
```

thermCond_integral takes the model as well as the bounds (in a list format) and returns the integral and scipy's uncertainty in the integral.

thermCond_error

This function is used to numerically find the uncertainty in the heat load. It is called by the *heat_load* function after it first calls *thermCond_integral* as the outputs of *thermCond_integral* are inputs to this function. The inputs to this function are (in the correct order): model, bounds, integral, abserror, area, d_area, length, d_length.

- *model* is material.thermCond[modelname]
- *bounds* are the bounds over which the equation was integrated
- *integral* is first output of *thermCond_integral*
- *abserror* is the second output of *thermCond_integral*
- *area* is the cross-sectional area of the materials
- *d_area* is the error in the area
- *length* is the length of the material
- *d_length* is the error in the length

Additionally, even though it is not input, this function calls *eqerror* to compare equation's error, the error generated by integrating the equation with the error, to a*bsessor* — scipy's uncertainty in the integral.

The output of this function is a 2-item list consisting of the lower and upper errors.

```
[4, 5] ==> error is -4, +5.
```

thermCond_error is called by *heat_load* but can also be called by hand.

```
error = thermCond_error(material.thermCond[modelname], [lower, upper],
integral, abserror, area, d_area, length, d_length)
```

ThermCond_error accepts errors of *None* but will replace them with 0's, so they will not contribute to the overall error.

thermCond_NIST_model

This function is used as an equation for a model. If given eqinputs in the right form, it will work without ever being called by hand. This function comes from NIST and can be found in their material database at http://cryogenics.nist.gov/MPropsMAY/materialproperties.htm.

THE MODEL:

This function uses the following equation to generate a thermal conductivity curve

```
y=10^(a+b*log(T)+c*(log(T))^2+d*(log(T))^3…
```

The letter T is the temperature and the remaining letters (excepting y) stand for coefficients to this equation. Each material has a different set of coefficients which are the eqinput.

WHAT EQINPUTS TO GIVE:

This function takes two inputs: a temperature and a list of coefficients. The temperature will be automatically supplied when this function is called by *heat_load*, but the list of coefficients needs to be made correctly and supplied as the eqinput for the model. The format of the coefficients is a normal list, as shown below.

```
[1.8743, -0.41538, -0.6018, 0.13294, 0.26426, -0.0219]
```

thermCond_log10NIST_model

This function is currently unused

thermCond_fit_UnivariateSpline_model

this function is used to fit raw data to a model. If given eqinputs in the right form, it will work without ever being called by hand.

THE MODEL:

This function uses scipy.interpolate's Univariate Spline function to create a best fit for raw data. Univariate Spline does 1D smoothing to given data. It has a spline degree, *k*, which is automatically chosen to be as large as possible within the constraints that the number of data points must be larger than *k* and that *k* can't be larger than 5. This function is also set to do extrapolations.

WHAT EQINPUTS TO GIVE:

This function takes two inputs: a temperature and the raw data. The temperature will be automatically supplied when this function is called by *heat_load*, but the raw data needs to be made correctly and supplied as the eqinput for the model. The format of the data is a list of two lists; the first list is the temperatures, the second is the corresponding thermal conductivities. An example is shown below.

```
[[0.3, 20.0, 40.0, 75.0, 150.0, 250.0],[0.0018, 0.15, 0.25, 0.7, 2.0, 4.0]]
```

thermCond_NIST_and_UnivariateSpline_model

this function is used as an equation for a model. If given eqinputs in the right form, it will work without ever being called by hand. Part of this function comes from NIST and can

be found in their material database at http://cryogenics.nist.gov/MPropsMAY/materialproperties.htm. The other part of this function comes from Univariate Spline

WHAT IT DOES & HOW IT DOES IT:

This function handles cases when both the *thermCond_fit_UnivariateSpline_model* and *thermCond_NIST_model* apply, for instance when they have different temperature ranges of validity and so different models are needed for different temperatures. This function handles this by comparing the temperature to the model's range of validity and assigning it a weighting factor of 1 if it falls within the range and e^(- abs(T - bound)) if the temperature is outside the range of validity. This is done so that estimates may be given when the two model's begin to overlap. A weighted mean is then taken of the two fits and their respective weighting factors.

It should be noted that this is a very simple method of overlapping two models since each weighting factor only consider the temperature difference between its model and given temperature and does not consider the distance of the other model.

WHAT EQINPUTS TO GIVE:

This function takes two inputs: a temperature and the raw data. The temperature will be automatically supplied when this function is called by *heat_load*, but the raw data needs to be made correctly and supplied as the eqinput for the model. The format of the data is a list of 3 lists:

1. the first list is the coefficients
2. the the second list is a list of two lists
    1. the first list is the temperatures
    2. the second is the corresponding thermal conductivities.
3. the third list is a list of two lists
    1. the first list is the range of validity for the NIST function
    2. the second list is the range of validity for the Univariate Spline function

An example is shown below.

```
[[-4.1236, 13.788, -26.068, 26.272, -14.663, 4.4954, -0.6905, 0.0397],
[[0.3, 1.4, 4.2], [1.64e-3, 2.06e-2, 6.56e-2]], [[10, 300], [0.3, 4.2]]]
```