GSCbins README

Updates

3.0

This update mostly focused on changing the terminology used in the code so it better fits what's going on linguistically (e.g. changing references to constraints to references to weights). It also added basic three dimensional plots.

- Language.constraints is now called Language.weights
- Language.set_of_constraints() is now Language.set_of_weights()
- Language.count_constraints() is now Language.count_weights()
- Language.language() is now Language.weights_in()
    - Language.weights_in() now checks the weights of the base language against the convex hull of the argument language, instead of vice versa
- adding a third constraint dimension to Language.plot() now creates a three dimensional plot
- Bin.constraint() is now Bin.set_of_weights()
- Bin.plot_bin() and Bin.plot_languages (formerly Bin.plot_language()) also support a third dimension when an additional constraint index is used as an argument.
    - *offset* now also offsets the drawn convex hulls as well as the points.
- collapsed all plotting functionality (used in Language.plot(), Bin.plot_bin(), and Bin.plot_languages()) into a single function called master_plot(), which is called by the three previously mentioned functions
- plot legends are no longer cut off

2.2

- added an optional offset argument to Bin.plot_bin() and Bin.plot_language()

2.1

Added more plotting functions to Bin().

- plots are now generated using paths instead of lines
- changed name of Bin.plot() to Bin.plot_bin() and added legend to plot
- added Bin.plot_language() which plots a specific language or list of languages in a bin
- added Language.plot() to plot a language
- added legends to all plotting functions
- added background color to all plots

2.0

Lots of changes since the first edition. The main change is the integration of convex hulls. This allows us to treat arrays of constraints as shapes, instead of point clouds. Note that for convex hulls to be calculated, there must be more sets of constraints than there are constraints, and those sets cannot form a "plane" or the equivalent in higher dimensions. All loads and saves are compatible with 1.0.

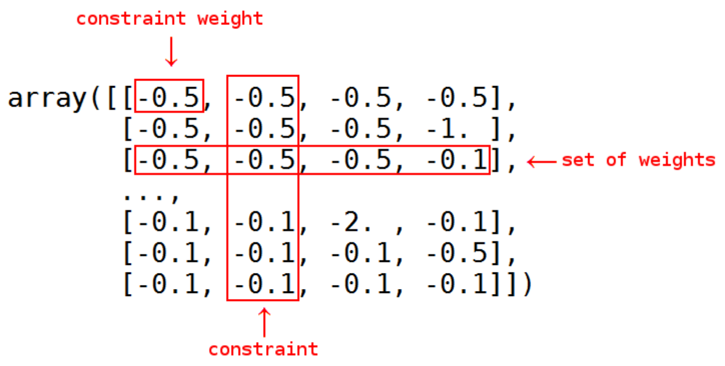
- old Language.__contains__() changed to Language.constraint()
- new Language.__contains__() now determines if point is in the convex hull of the language
- Language.language() returns the set of constraints of a language found in the conxev hull of the base language
- Language.volume() calculates volume of convex hull
- Bin.plot() plots the languages in a bin using two dimensions

---

Guide

Terminology

The entire array of constraint weights for a language is colloquially referred to as a language's "weights." Within a language's weights, the following terminology is used.



```
array([[-0.5, -0.5, -0.5, -0.5],
       [-0.5, -0.5, -0.5, -1. ],
       [-0.5, -0.5, -0.5, -0.1],
       ...,
       [-0.1, -0.1, -2. , -0.1],
       [-0.1, -0.1, -0.1, -0.5],
       [-0.1, -0.1, -0.1, -0.1]])
```

When discussing plots, "point" always refers to a set of weights. Spyder is used for all of the examples in this readme.

Importing GSCbins

The easy way to work with GSCbins is to import all of the classes from the file, instead of importing just the file. This way, every call to a class doesn't have to start with "GSCbins."

```
In [1]: from GSCbins import *
```

The Language Class

To create (or initialize) a language, it must have a *token* and a *set of weights* tuple (tuples and lists are interchangeable in all of these examples).

```
In [2]: first_language = Language('LHO', (0, 0, 0, 0))
```

A language can also be created with an optional *description*.

```
In [3]: second_language = Language('SHO', (0, 0, 0, 1), 'lowering')
```

The *token*, *weights*, and *description* of a language can be returned at any time by using that language's identifier (e.g. "first_language") and it's attribute.

```
In [4]: first_language.token
Out[4]: 'LHO'
In [5]: first_language.weights
Out[5]: array([[0, 0, 0, 0]])
In [6]: first_language.description
In [7]: second_language.description
Out[7]: 'lowering'
```

Note that since "first_language" wasn't given a description, nothing is returned when prompted for its description.

More sets of weights can be added to one at a time to a specific language using `lshift` (<<).

```
In [8]: first_language << (1, 0, 0, 0)
In [9]: first_language << (-1, 0, 2, 0)
In [10]: first_language.weights
Out[10]:
array([[ 0,  0,  0,  0],
       [ 1,  0,  0,  0],
       [-1,  0,  2,  0]])
```

A specific set of weights can be returned by using the *weights* attribute and the set of weights index in brackets. A specific constraint weight can be returned by adding the constraint index in brackets after the set of weights index.

```
In [11]: first_language.weights[1]
Out[11]: array([1, 0, 0, 0])
In [12]: first_language.weights[1][0]
Out[12]: 1
```

The number of sets of weights in a language can be returned by using `count_weights`.

```
In [13]: first_language.count_weights()
Out[13]: 3
```

A language can be checked to see if it has a specific *set of weights* in its *weights* attribute. Be sure to enter the *set of weights* as a tuple (or list).

```
In [14]: first_language.set_of_weights((1, 0, 0, 0))
Out[14]: True
In [15]: second_language.set_of_weights((1, 0, 0, 0))
Out[15]: False
```

The range of a language's constraints can be returned using `constraint_range`. To return the range of only one constraint, enter the index of the constraint as an argument.

```
In [16]: first_language.constraint_range()
Out[16]: ([-1, 0, 0, 0], [1, 0, 2, 0])
In [17]: first_language.constraint_range(0)
Out[17]: (-1, 1)
```

For the next few examples, more weights are needed to correctly generate convex hulls. The new example language is only in 3 dimensions for simplicity.

```
In [18]: rectangular_language = Language('prism', (0, 0, 0))
In [19]: rectangular_language << (0, 0, 1)
In [20]: rectangular_language << (0, 1, 0)
In [21]: rectangular_language << (0, 1, 1)
In [22]: rectangular_language << (2, 0, 0)
In [23]: rectangular_language << (2, 0, 1)
In [24]: rectangular_language << (2, 1, 0)
In [25]: rectangular_language << (2, 1, 1)
In [26]: rectangular_language << (0.5, 0.5, 0.5)
```

This new language's weights create a convex hull (in this case a 2 by 1 by 1 rectangular prism) with a point (set of weights) inside of it.

The "volume" of a language's convex hull can be returned using the function (not attribute) `volume`.

```
In [27]: rectangular_language.volume()
Out[27]: 2.0
```

Whether or not a point is in a language's convex hull can be returned using the magic function `in`.

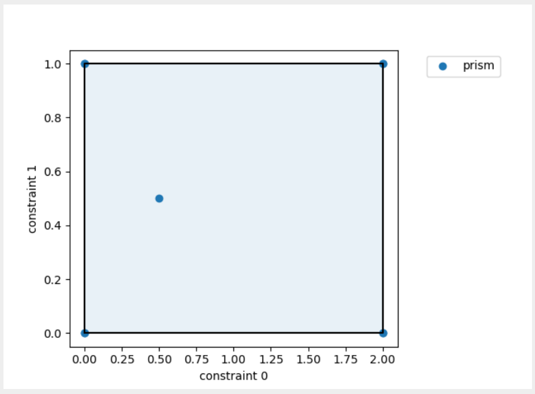
```
In [27]: (1, 0.5, 0.5) in rectangular_language
Out[27]: True
In [28]: (2, 0.5, 0.5) in rectangular_language
Out[28]: True
In [29]: (3, 0.5, 0.5) in rectangular_language
Out[29]: False
```

You can return the points of a language that lie in another language's convex hull by using the function `weights_in`. For this example, a "cube" is initialized. Then, the points of cube_language that are in rectangular_language's convex hull are returned.

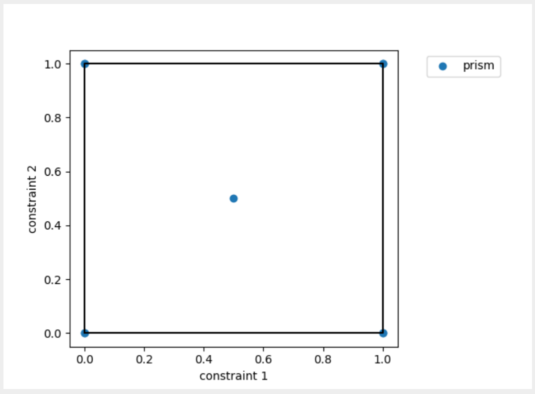
```
In [30]: cube_language = Language('box', (0.5, 0.5, 0.5)
In [31]: cube_language << (1.5, 0.5, 0.5)
In [32]: cube_language << (1.5, 1.5, 0.5)
In [33]: cube_language << (0.5, 1.5, 0.5)
In [34]: cube_language << (0.5, 0.5, 1.5)
In [35]: cube_language << (1.5, 0.5, 1.5)
In [36]: cube_language << (1.5, 1.5, 1.5)
In [37]: cube_language << (0.5, 1.5, 1.5)
In [38]: cube_language.weights_in(rectangular_language)
Out[38]: [[0.5, 0.5, 0.5], [1.5, 0.5, 0.5]]
```

You can plot two dimensions (or constraints) of a language's weights by using `plot`. `plot` takes two constraints arguments, one for each dimension. The border of the convex hull is automatically drawn in those dimensions, and by default the area of the language is shaded. The shading can be changed by using the argument *alpha*.

```
In [39]: rectangular_language.plot(0, 1)
```



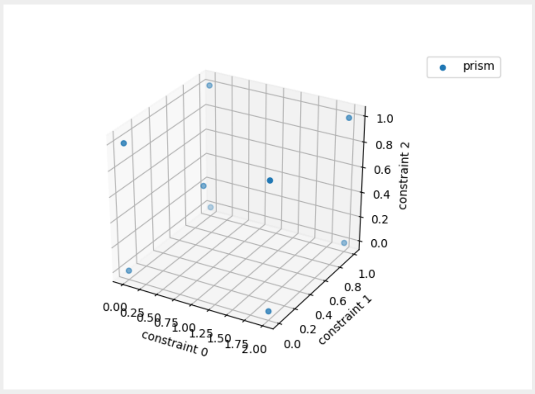
```
In [40]: rectangular_language.plot(1, 2, alpha=0)
```



Note: If using IPython, be sure to enter `%matplotlib` before using `plot`. If using Spyder, the plot will be drawn in the IDLE. To draw the plot outside of the IDLE, change Graphics backend from "Inline" to "Automatic" under Preferences/Ipython/Graphics.

You can also plot three dimensions now, by adding a third constraint as an argument to `plot`. Currently, drawing the convex hull and shading is not supported. You can rotate the plot by clicking and dragging on it.

```
In [41]: rectangular_language.plot(0, 1, 2)
```



The Bin Class

A bin is initialized with no arguments.

```
In [42]: the_bin = Bin()
```

To add a language to a bin, you use `lshift`. You can either add a previously initialized language by it's identified, or you can initialize a language into a bin. When two languages with the same tokens are added to a bin, the languages "collapse" into one language with both weights from each former language. This prevents duplicate tokens from being in a bin.

```
In [43]: the_bin << rectangular_language
In [44]: the_bin << cube_language
In [45]: the_bin << Language('no identifier', (1, 1, 1))
In [46]: the_bin << Language('no identifier', (1, 2, 1))
```

A list of tokens in the bin can be returned by using the function `tokens_list`.

```
In [47]: the_bin.tokens_list()
Out[47]: ['prism', 'box', 'no identifier']
```

To see if a language is in a bin, you can search for its token.

```
In [48]: 'box' in the_bin
Out[48]: True
In [49]: 'triangle' in the_bin
Out[49]: False
```

The number of languages in a bin can be returned by using `count`.

```
In [50]: the_bin.count()
Out[50]: 3
```

The address of a language in a bin can be returned by either token or set of weights. To return the address of a language by its token, use the function `token`. To return the address of a language by a set of weights it contains, use the function `set_of_weights`.

```
In [51]: the_bin.token('no identifier')
Out[51]: <GSCbins.Language at 0x146a29cdeb8>
In [52]: the_bin.set_of_weights((1, 1, 1))
Out[52]: <GSCbins.Language at 0x146a29cdeb8>
```

The addresses here are identical because they refer to the same language. Addresses vary from session to session. Returning a language allows you to use all of the attributes and functions of the Language class on the language returned. In this example, this allows you to modify the ‘no identifier’ language, even though it has no identifier.

```
In [53]: the_bin.token('no identifier') << (1, 2, 2)
In [54]: the_bin.token('no identifier').weights
Out[54]:
array([[1, 1, 1],
       [1, 2, 1],
       [1, 2, 2]])
In [55]: the_bin.set_of_weights((1, 2, 2)) << (1, 1, 2)
In [56]: the_bin.token('no identifier').weights
Out[56]:
array([[1, 1, 1],
       [1, 2, 1],
       [1, 2, 2],
       [1, 1, 2]])
In [57]: the_bin.token('no identifier').count_weights()
Out[57]: 4
```

You can save the languages in a bin to a text file by using the `save` function and providing a file name as an argument. To load languages to a bin from a text file, use `load`.

```
In [58]: the_bin.save("the_file.txt")
In [59]: new_bin = Bin()
In [60]: new_bin.load("the_file.txt")
In [61]: new_bin.tokens_list()
Out[61]: ['prism', 'box', 'no identifier']
```

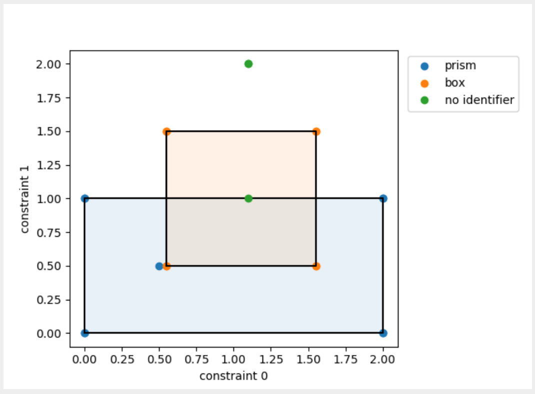
To “empty” a bin (by dereferencing the languages in its language list), use `empty`.

```
In [62]: the_bin.empty()
In [63]: the_bin.count()
Out[63]: 0
```

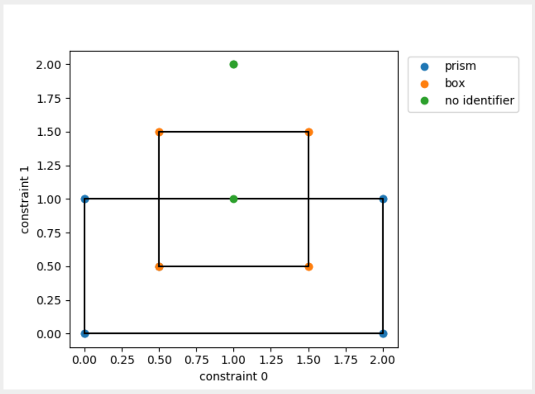
Note: Loading a file to a bin does not overwrite a bin’s contents. Loading the same file to a bin twice without emptying it will create duplicate weights for each language.

All of the languages in a bin can be plotted on the same plot using `plot_bin`. There is an additional optional argument that can be used in addition to *alpha*, called *offset*. By default, each successive language is offset on the x-axis by 0.05 from the previous language, in order to make shared points easily visible. To turn off *offset*, use the argument *offset=0*.

```
In [64]: new_bin.plot_bin(0, 1)
Languages with no convex hull in these dimensions: no identifier
```

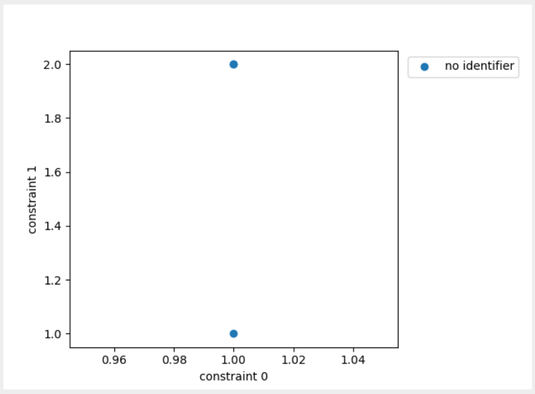


```
In [65]: new_bin.plot_bin(0, 1, alpha=0, offset=0)
Languages with no convex hull in these dimensions: no identifier
```

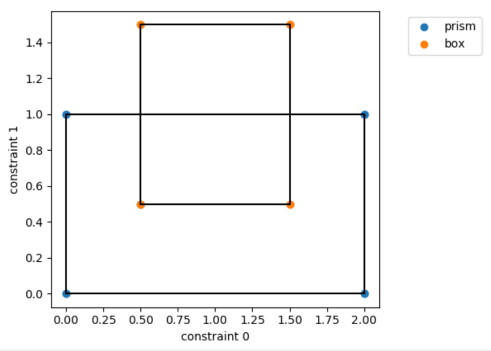


To plot specific languages in a bin, as opposed to all languages in a bin, use `plot_languages`. In contrast to the other plotting functions, the first argument of `plot_languages` must be either a language or a list of languages. The next two (or 3 for three dimensional plots) arguments are constraints. The optional arguments *alpha* and *offset* can also be used.

```
In [66]: new_bin.plot_languages(new_bin.token("no identifier"), 0, 1)
Languages with no convex hull in these dimensions: no identifier
```

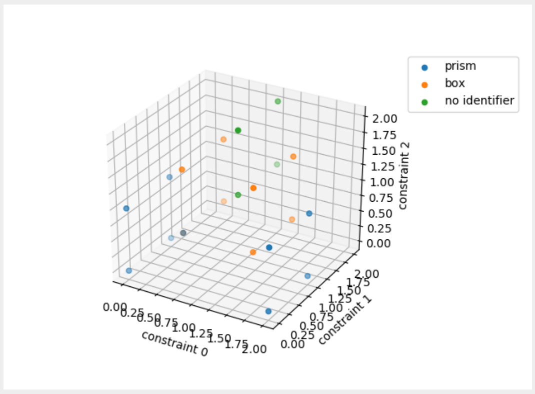


```
In [67]: new_bin.plot_languages((rectangular_language,
cube_language), 0, 1, alpha=0, offset=0)
```

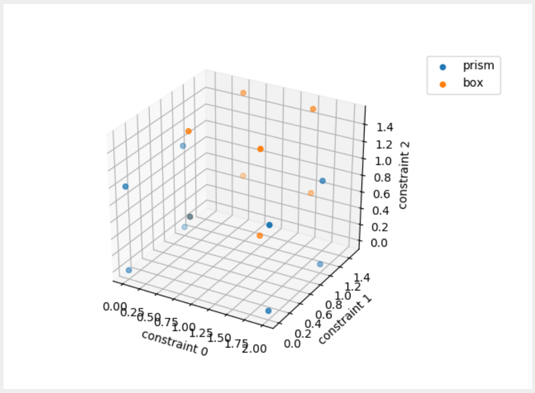

prism
box
constraint 0
constraint 1
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4

You can also use `plot_bin` and `plot_languages` in three dimensions by adding an additional constraint argument. As before, *alpha* and *offset* are currently not supported in three dimensions.

```
In [68]: new_bin.plot_bin(0, 1, 2)
```



```
In [69]: new_bin.plot_languages((rectangular_language,
cube_language), 0, 1, 2)
```



The bin_language Function

The `bin_language` function is a novelty function most useful for entering large amounts of languages into bins via loops. `bin_language` automatically sorts a language into one of three bins, depending on which bin the language's token is associated with. If `bin_language` doesn't know which bin to put a language into, it prompts the user for input. It then "remembers" which bin to put that language into if it comes up later, including across sessions. This function may or may not be useful, depending on how GSCbins is used.

```
In [70]: good_bin = Bin()
In [71]: okay_bin = Bin()
In [72]: trash_bin = Bin()
In [73]: bin_language(first_language, good_bin, okay_bin, trash_bin)
Token "LHO" not recognized. Add to bin? (y/n): y
Language quality (good, okay, trash): good
Language description: totally faithful
In [74]: good_bin.tokens_list()
Out[74]: ['LHO']
In [75]: good_bin.token('LHO').description
Out[75]: 'totally faithful'
In [76]: bin_language(Language('LHO', (5, 5, 5, 5)), good_bin,
okay_bin, trash_bin)
In [77]: good_bin.token('LHO').weights
Out[77]:
array([[ 0,  0,  0,  0],
       [ 1,  0,  0,  0],
       [-1,  0,  2,  0],
       [ 5,  5,  5,  5]])
```