LIPGLOSS Notes

Pieter Mostert

April 28, 2018

1 Introduction

LIPGLOSS is a collection of Python scripts designed to construct ceramic recipes that satisfy a number of constraints. In particular, it allows users to construct recipes with a given oxide composition, when possible.

It employs a linear programming algorithm to calculate bounds on oxides and ingredients, hence its full name: *LInear Programming GLaze Oxide Software System*. Since that's a terrible name, it's better to just stick with the acronym LIPGLOSS.

The plan is that LIPGLOSS will eventually be released as a desktop app that users can customize with their own ingredients and restrictions. This will be designed to import and export recipes from Glazy, as well as other glaze calculation software; alternatively, part of the code could be incorporated within Glazy.

NOTE: This document refers to an older version of the code. The functionality is still mostly the same, but the code has been refactored to fit the MVC pattern (well, sort of...).

2 Using LIPGLOSS

Currently, to use LIPGLOSS, you need to instal Python 3, the linear programming API PuLP, and the GLPK library[†] (and possibly the tkinter Python package). Running the GUI.py script opens the graphical user interface.

When the gui opens, you'll be presented with a window that shows a list of ingredients and 'other restrictions' in the panel on the left. Some of these will have already been selected by default, but you can add or remove items by clicking on the buttons.

In the panel on the right, the selected ingredients and other restrictions will be shown, as well as any oxides that appear in the selected ingredients. The two columns on the left, which contain values in blue, are for entering the desired lower and upper bounds on the oxide, ingredient, or other restriction. Bounds on oxides can be expressed as UMF, percent weight or percent mole by selecting the appropriate radio button above.

[†]If installing the GLPK library is a problem, comment out 'solver = GLPK()' and uncomment 'solver = PULP_CBC_CMD()' in pulp2dim.py. This will result in the COIN-OR Cbc solver being used, which comes with PuLP. Note, however, that it is released under the Eclipse Public Licence, which is incompatible with the GPL.

When you click the 'Calculate restrictions' button, the two columns on the right will be filled with lower and upper bounds for each oxide, ingredient, or other restriction, printed in red (assuming there is at least one recipe that satisfies the bounds; otherwise, an error message will appear). These are lower and upper bounds for the corresponding item among all possible recipes using the ingredients selected that satisfy the bounds you've imposed. Naturally, the bounds on the right are at least as restrictive as the bounds on the left.

You can now progressively strengthen the bounds on the left, based on the calculated bounds. This may be as simple as unselecting an ingredient whose calculated lower bound is zero. Given a single oxide, ingredient, or other restriction, you can narrow it's range to any interval that overlaps the calculated interval, and be guaranteed that the problem still has a solution[‡]. However, it's important to note that if you restrict the bounds for two or more items simultaneously, a solution is not guaranteed even if the restricted intervals both overlap their respective calculated intervals[§].

If you want to restrict the bounds for two variables simultaneously, you have the option of plotting the set of possible values of those two variables. That is to say, for each point in the region shown, there is at least one recipe where the variables take on the x and y values of the point, and which satisfies all the constraints[¶]. The x and y variables to plot can be selected by clicking on the left and right hand labels in the right hand panel, respectively. An asterisk should appear next to the label when you click on it.

A collection of ingredient and other restrictions with lower and upper bounds will be referred to as a bounded recipe. Bounded recipes can be saved, or saved as new bounded recipes. If you start with the default bounds and narrow them down to a recipe - i.e. all upper and lower bounds are effectively the same - you should save this as a new bounded recipe, otherwise you'll have to undo all your restrictions when you start working on a new problem. Saved bounded recipes can be opened through the File menu.

LIPGLOSS will come pre-installed with common types of ingredients, but most of these should be updated to reflect the analyses of the ingredients in your studio. To do this, click Options > Edit Ingredients to open the ingredient editor. The order in which ingredients are displayed can be changed by dragging (with a right-click) and dropping the names of ingredients.

3 Code Structure

In an attempt to make things clearer, any text below that is used as code will be printed in blue. File names will be printed in purple.

[‡]Actually, this is a lie, since the calculated bounds are rounded to a certain number of decimal places, so, for example, setting the upper bound equal to the calculated lower bound won't work if the calculated lower bound has been rounded down.

[§]For example, if only two ingredients are selected and no restrictions are imposed, the bounds on the percentages of these ingredients will be 0 and 100. If you then impose the bounds 0 to 40 on both these ingredient percentages, there will be no solution, simply because the sum of the ingredient percentages must add up to 100.

[¶]If you take the example in the previous footnote, the region corresponding to the two ingredient percentages will be a straight line from $(0, 100)$ to $(100, 0)$. Typically, the regions that arise will be two dimensional, and will always have boundaries that are either straight lines or sections of hyperbolas.

As noted before, the GUI.py file launches the gui. The following diagram shows the import relations between the modules.

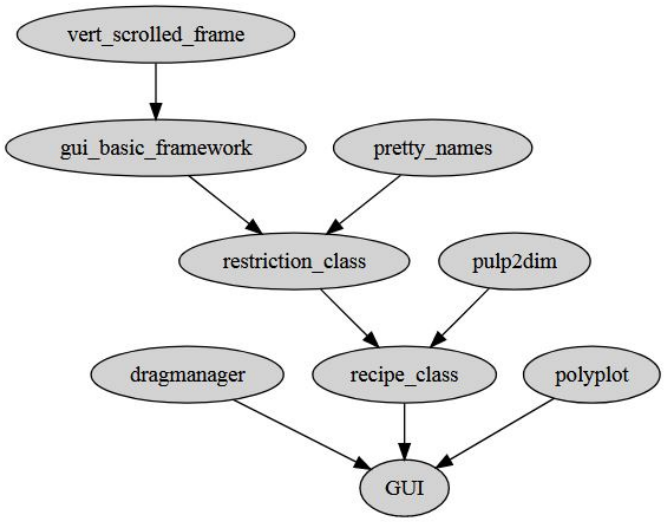

Figure 1: Module dependencies

The files ingredientfile.py and oxidefile.py are usually not used, and can be omitted from any stable releases. See section 5 in restrictions.py for when they are needed.

The following are the most important classes: Oxide, Ingredient, Other, Restriction (see restrictions.py) and Recipe (see recipes.py). To keep track of the instances of these classes, we use the dictionaries oxide_dict, ingredient_dict, other_dict, restr_dir, and recipe_dict. The keys for the ingredient_dict, other_dict, and recipe_dict dictionaries are strings of non-negative integers. The keys for oxide_dict are strings of the form 'SiO2', 'Al2O3', etc. There are five types of Restriction instances, so keys are of five different types:

- 'umf_'+ox, 'mass_perc_'+ox, and 'mole_perc_'+ox, for each key ox in oxide_dict,
- 'ingredient_'+k for each key k in ingredient_dict,
- 'other_'+k for each key k in other_dict.

We could easily do without the Oxide class, but I've introduced it so that if (in some future version of the program) we decide to allow users to assign attributes to oxides, then this can be done in the same way as for ingredients. See the 'To do' list.

3.1 Setting up the linear programming problem

The code relating to the computation of the bounds is mostly found in recipes.py, although parts of the linear programming problem are set up in restrictions.py, section 5. To set up the LP problem, we define the following variables / families of variables:

- lp_var['mass_'+ox], for each key ox in oxide_dict, representing the mass (in grams) of that oxide in a batch.
- lp_var['mole_'+ox], for each key ox in oxide_dict, representing the number (in moles) of that oxide in a batch.
- lp_var['ingredient_'+k] for each key k in ingredient_dict, representing the mass (in grams) of the ingredient in a batch.
- lp_var['other_'+k] for each key k in other_dict.
- lp_var['ingredient_total'], representing the total mass (in grams) of all ingredients in the batch.
- lp_var['fluxes_total'], representing the total number (in moles) of all flux oxides in the batch.
- lp_var['ox_mass_total'], representing the total mass (in grams) of all oxides in the batch.
- lp_var['ox_mole_total'], representing the total number (in moles) of all oxides in the batch.

These variables are related as follows:

- For each oxide ox,

  $$\text{lp\_var['mass\_'+ox]} = M \times \text{lp\_var['mole\_'+ox]},$$

  where M is the molar mass of that oxide.
- For each oxide ox,

  $$\text{lp\_var['mass\_'+ox]} = \sum_{\text{k} \in \mathcal{I}} \frac{P_{\text{k,ox}}}{100} \times \text{lp\_var['ingredient\_'+k]},$$

  where $\mathcal{I}$ is the set of all ingredients, and $P_{\text{k,ox}}$ is the percent mass of ox in ingredient k.
- For each index k in other_dict,

  $$\text{lp\_var['other\_'+k]} = \begin{array}{l}\text{a linear combination of the variables} \\ \text{corresponding to ingredients and oxides}\end{array}$$

- $$\text{lp\_var['ingredient\_total']} = \textstyle\sum_{\text{k} \in \mathcal{I}} \text{lp\_var['ingredient\_'+k]}$$

  ,
- $$\text{lp\_var['fluxes\_total']} = \sum_{\text{ox} \in \mathcal{F}} \text{lp\_var['oxide\_mol\_'+ox]},$$

  where $\mathcal{F}$ is the set of all fluxes,

- $$\texttt{lp\_var['ox\_mass\_total']} = \sum_{\texttt{ox}\in\mathcal{O}} \texttt{lp\_var['mass\_'+ox]},$$

  where $\mathcal{O}$ is the set of all oxides,

- $$\texttt{lp\_var['ox\_mole\_total']} = \sum_{\texttt{ox}\in\mathcal{O}} \texttt{lp\_var['mole\_'+ox]}.$$

Note that the quantities we'd like to optimize are not the basic variables lp_var[...], but certain *ratios* of these variables. For example, the UMF value of an oxide ox is

$$\frac{\texttt{lp\_var['mole'+ox]}}{\texttt{lp\_var['fluxes\_total']}},$$

while the percentage of an ingredient with index k is

$$100 \times \frac{\texttt{lp\_var['ingredient'+k]}}{\texttt{lp\_var['ingredient\_total']}}.$$

In general, given a restriction res, the associated quantity we want to optimize is of the form lp_var[numerator]/res.normalization where 'numerator' and res.normalization are defined below:

restriction	numerator	res.normalization
'umf_'+ox	'mole_'+ox	lp_var['fluxes_total']
'mass_perc_'+ox	'mass_'+ox	0.01×lp_var['oxide_mass_total']
'mole_perc_'+ox	'mole_'+ox	0.01×lp_var['oxide_mole_total']
'ingredient_'+k	'ingredient_'+k	0.01×lp_var['ingredients_total']
'other_'+k	'other_'+k	Custom

When recipe.calc_restrictions(...) is run, for each restriction res, we impose the constraints

$$\texttt{lp\_var[numerator]} < U \times \texttt{res.normalization}$$

and

$$\texttt{lp\_var[numerator]} > L \times \texttt{res.normalization},$$

where U and L are the upper and lower bounds imposed by the user respectively.

To maximize or minimize the variable associated with a restriction res, we set the 'normalization' constraint res.normalization=1, and maximize or minimize lp_var[numerator].

Remark: We could eliminate all the variables other than the ones corresponding to the selected ingredients before we hand the problem over to the solver. It remains to be seen by how much this would speed up the computation time. Since the computation time for typical problems is currently only about 2 seconds, implementing this is not a high priority.

3.2 Storing user-defined data

To save data entered by the user, I've been using the shelve module. This results in several data files: IngredientShelf.dat, OtherShelf.dat, OxideShelf.dat, and OrderShelf.dat (these all have associated bak and dir files). These are stored in the 'data' folder. Derek Au is in the process of switching to JSON instead. So far, only the list of recipe bounds is stored as a JSON file, JSONRecipeShelf.

At the moment, we could do without the OxideShelf files, since there's no way for the user to modify any data related to the oxides, but it may be possible in future versions.

The IngredientShelf, OtherShelf, OxideShelf files are essentially copies of the dictionaries ingredient_dict, other_dict, and oxide_dict. The order in which ingredients are displayed is saved in OrderShelf; the code

```
with shelve.open("./data/OrderShelf") as order_shelf:
    print(order_shelf['ingredients'])
```

will print a list of all ingredient indices, in the order the ingredients are displayed in the ingredient selection window and the ingredient editor. Once users are able to change the order of oxides and other restrictions, OrderShelf will record these orders too, using the keys 'oxides' and 'other_restr'.

4 Known bugs

- On my computer, the PuLP API causes a window to open and close each time a linear function is optimized, resulting in a delay of around 10 seconds, even though the actual calculation usually takes under 2 seconds. There may be a fix for this, but I haven't been able to implement it.

5 To do

- Fix bugs above.
- For the 2d graphing, have a button to the right of the restriction panel that when clicked on, opens a panel with instructions on how to select the two variables, as well as a button that calculates only the region corresponding to two variables, not all the other lower and upper bounds (Before updating the restrictions, you may want to compare several pairs of variables. In this case there's no need to recalculate all the other bounds). Include labels on the axes when the region is calculated, and make sure the numbers along the axes don't overlap.
- Implement the ability to plot the feasible region corresponding to any two restrictions. EDIT: This has been done, but using a brute-force approach, which may not give accurate results.
- Clicking Options > Edit restriction settings, should open a window that allows you to set default lower and upper bounds for each oxide, ingredient and other restriction.

- Allow users to define custom 'other restrictions'. Users would need to specify an objective function, and a normalization.
- Implement the ability to add other attributes to ingredients, using the restriction editor. Each other attribute will be assigned a variable, and an 'other restriction' corresponding to that attribute will automatically be added. For example, an other attribute could be the percentage of an ingredient having particle diameters between two bounds.
- Make the headings in the ingredient editor align with their respective columns. Add a horizontal scrollbar, and fix the ingredient names when scrolling sideways.
- Import recipes (or oxide compositions of recipes) from external software, and automatically generate a recipe problem where the oxides are constrained to have the same composition as the imported recipe.
- Convert recipe bounds to actual recipes, and export these to external software. This should only be used when the upper and lower bounds on all ingredients are sufficiently close together.
- Include an 'about box'.