Group 6 – Maintenance Guide

This guide contains a description of the important methods and classes implemented in the project. The goal is to briefly explain what each class does, where it is used and how it fits in the context of the project as a whole. Any important design decisions relating to the class are also explained. This is to allow someone unfamiliar with the code to get a general idea of the existing framework. This document does not cover every method, instead focusing on the important and most frequently used methods.

CommonDataTypes.py

This file contains the definitions of the several important objects used throughout the code. They are: `EulerAngle`, `SixPosition`, `Candidate`, `TiltedTemplate` and `Tomogram`.

`EulerAngle`:

This class is used to represent Euler angles, which fully specify the orientation of a rigid body in 3D space. This class also contains a static list of Euler Angles called `Tilts`, which contains all the Euler Angles used during template generation. Instead of saving an Euler Angle object, we instead save the Euler Angle's index in Tilts. This field is usually denoted by the name `tilt_id`.

`SixPosition`:

This class contains the position and orientation of a macromolecular-complex. The position is saved in a three tuple called `COM_position` containing the (x,y,z) position of the macromolecular-complex (if using 2D, then z = 0). The orientation is saved indirectly by saving the `tilt_id` corresponding to the Euler Angles of the template.

`Candidate`:

This class is used to represent a macromolecular-complex (template) in the tomogram. In addition to the location and orientation of the macromolecular-complex, additional fields are also saved. The fields are the label, the suggested label, and the matching ground truth candidate. This class is mainly used in two contexts: to aggregate ground truth data of a macromolecular-complex in a tomogram, or to represent the possible location of a reconstructed macromolecular-complex. The first type is generated during tomogram generation, while the second is created by the Candidate Selector during analysis. The various fields are defined as follows:

- `six_position`: this field is a `SixPosition` object representing the candidate's position and orientation in the tomogram.
- `label`: this field is used to define the type of macromolecular-complex that the candidate represents. This is the `template_ID` of the template matched to the candidate. The `label`, together with the `tilt_id`, is enough to specify which tilted template this candidate represents (see `TiltedTemplate`). For ground truth candidates the `label` is set during tomogram generation, while for reconstructed candidates it is set by the `Labeler`.
- `suggested label`: this field is deprecated and currently not in use.
- For reconstructed candidates, the `ground_truth_match` field holds the matching ground truth candidate found by the position labeler.

`TiltedTemplate`:

This class represents a template rotated by a specific Euler angle. It contains the density map of the template, as well as `tilt_id` and `template_id` metadata. The `tilt_id` can be used to find by which Euler Angle the template was rotated (see `Tilts` list in `EulerAngle`), while the `template_id` tells us which 'base' template the `TiltedTemplate` was generated from. When creating the `TiltedTemplate`, the density map can either be given explicitly as a ndarray or by specifying a path from which the ndarray can be loaded.

`Tomogram`:

This object represents a tomogram. It contains both a density map (represented as a numpy ndarray object) and the tomogram's `composition` - a list of `candidate` objects which represent the templates that make up the tomogram.

This list is important because it represents the "ground truth" data of the tomogram. It is used by the position labeler during SVM training to match the candidates found by the candidate selector to their "real" counterpart. It is also used in a similar manner by the various testers to compare the reconstructed candidates to their true counterparts. The composition is meaningful only when we use generated tomograms or use tomograms with a known ground truth. If the tomogram is loaded from a PBD file, the composition will be `None`. If the tomogram is created from templates using one of the functions in Tomogram Generator, then the composition is automatically filled.

TomogramGenerator.py:

This file contains all the methods that are used to generate and load tomograms. There are three main ways to generate a single tomogram object:

- `generate_tomogram_tomogram_with_given_candidates`: This method generates a tomogram according to a list of candidates supplied by the user. The candidate object defines each macromolecular-complex's position and orientation, and the tomogram's density map is filled accordingly. The resulting tomogram's composition is a copy of the supplied list.
- `generate_random_tomogram`: This method generates a randomized tomogram. It does this by randomly generating candidates in random positions and random orientations (i.e. tilts), then using them to fill the tomogram's density map. The user specifies how many templates of each type to generate passing a list of integers called `criteria` such that `criteria[i]` is the number of tomograms of type `i` that need to be generated. There is also an option to add noise to the generated tomogram. Currently the points are distributed using a simple randomized algorithm (see naive_spaced_randomizer.py)
- `tomogram_loader`: This function lets you load tomogram objects from pickle files. The same function can also be used to save tomogram objects in a pickle file.

It is also possible to create a set of tomograms using a generator. For example `generate_random_tomogram_set` creates a set of random tomograms. Similarly to `generate_random_tomogram`, the user specifies the criteria of the tomograms, and whether or

not to add noise. The user can also specify a random seed, which is useful for testing benchmarks, since results can be recreated and tested on various configurations.

naive_spaced_randomizer.py:

This is the algorithm used to distribute the candidates when generating a random tomogram. The algorithm needs to distribute n points at random such that no two points are closer than some minimum separation.

The method `distribute_points` does this by randomly generating a point, and accepting it only if it is sufficiently far from the previously accepted points. If n points were accepted, then the algorithms returns a list of the accepted points. However, If the algorithm rejects too many points (defined during initialization), than an empty list is returned instead.

`randomize_spaced_points` is the function used to generate the set of randomized points. The method calls `distributes_points` either until it succeeds in distributing n points or until it fails a certain number of times (which is again defined during initialization). This method is used over distribute points because it is less prone to failure. In `distribute_points`, if the first few points are poorly chosen, then it is sometimes impossible to distribute any more points, even if the number of iterations is increased. Calling the distribute points several times from `randomize_spaced_points` avoids this problem since if bad initial points were chosen, then `randomize_spaced_points` will just start over.

Because the initial separation needs to be larger than the width of the largest template so that we get no overlaps, the resulting tomograms do not have adjacent templates. This is clearly not representative of real test data since in a real tomogram, the macromolecular-complexes are tightly packed. If the randomization could be improved then the efficiency of the SVM could be improved as well. One possibility is to allow the tomogram to be added as long as the area is empty. This can be check by checking if the density of the tomogram is sufficiently small. This will allow templates to be much closer together than was previously allowed.

Noise.py

This file contains functions that add noise to tomograms. Currently there are three types of noise supported:

`gauss_noise`: adds Normally distributed noise to the density map. The mean and the standard deviation can be modified. By default the mean is zero, and the standard deviation is 0.3 times the maximum value of the density map. The noise is multiplied by `NOISE_GAUSS_PARAM` before being added to the original density map.

`linear_noise`: adds uniformly distributed noise to the density map. The noise is distributed between $[0, x]$ where x is the standard deviation of the tomogram times some parameter supplied by the user. By default this parameter is equal to `NOISE_LINEAR_PARAM`

`blur_filter`: adds blur to a density map by convolving it with a Gaussian. The parameters of the Gaussian are defined in the config file under `NOISE_GAUSSIAN_SIZE` and `NOISE_GAUSSIAN_STDEV`.

The method `make_noisy_tomogram` receives a tomogram and applies a combination of the previous three methods to a copy of the tomogram. This is used in TomogramGeneration.py when we want to create a tomogram with noise.

TemplateGenerator.py:

This file contains all the methods used to generate and load templates from files.

Template Generation:

In 2D, template generation and initialization are handled by the same method - `generate_tilted_templates_2d`. It receives the angular resolution α, and generates $\left\lfloor \frac{360^\circ}{\alpha} \right\rfloor$ templates of each template type. The currently supported template types are square, circular, L-shaped and flipped L-shaped templates. The tilted templates are generated by taking the density map of the 'base' template and rotating it by α degrees. The method then initializes a tuple of tuples of tilted templates (see section about template data structures) and returns it. It also initializes the Tilt list in the `EulerAngle` class.

To add more templates, all you need to do is add a function that fills an empty ndarray with the template density map, then add it to `generate_tilted_templates_2d` in the same format as the other density maps.

In 3D, template generation is handled by `generate_templates_3d`. The method receives the angular resolution, the type of templates (either geometric, PBD, or all) and an output path. Templates of the chosen type are generated then saved in files in the specified path. For more information, see the section on `generate_templates_3d` in the user guide.

To add more types of geometric templates, you would need to modify chimera_template_generator.py in the ChimeraUtils folder. You would need to add a function which generates the desired density map, and then modify `create_geometric_model` to support the new template type.

Template Initialization:

After being generated, the templates need to be loaded onto a data structure so that they can be accessed by the various other methods. In 2D, this is also done by `generate_tilted_templates_2d`.

In 3D, the function `load_templates_3d` is used. The method receives the path to the templates generated by `generate_templates_3d`. It creates a data structure that allows us to access the templates in the designated path. It also initializes the Tilt list in the `EulerAngle` class. Both procedures are done by reading the metadata file created during template generation, which has information on the number of template types, and the Euler Angles used to generate the templates.

Template Data Structure:

The data structure returned by `generate_tilted_templates_2d` and `load_templates_3d` is accessed like a tuple of tuples of Tilted Templates. Each element of the tuple corresponds to a template type, and contains all the tilted templates created from that template type (i.e. `templates[i][j]` is the Tilted Template with `tilt_id = j` and `template_id = i`).

The actual implementation is different in the case of 2D and 3D. In 2D, the data structure is implemented as a simple tuple of tuples of Tilted Template. However in 3D, because the three dimensional density maps require more memory than two dimensional maps, and because many more tilted templates are used[1], loading all the density maps onto a tuple of tuples is very RAM intensive, especially if we only need one or two Tilted Template at a time. Instead, we access the templates lazily, by loading the density map from the file only when needed. To this end, instead of a tuple of tuples, we use an instance of the `BidimentionalLazyFileDAO` object defined in TemplatesDataAccessObject.py. It is accessed just like a regular tuple of tuples, so the interface in 2D and 3D is identical.

Template types:

In 2D there are four types of supported templates: circular, square, L-shaped and reversed L-shaped templates. In 3D we support 3 types of geometric templates: spherical, cubic and L-shaped templates. In 3D we also have the ability to load templates using pdb files.

TomogramAnalyzer.py

This object is used to encapsulate the analysis process. It is given the tomogram to be analyzed, the template data structure (see TemplateGenerator) and an instance of a Labeler object. By calling `analyze`, the tomogram is analyzed in the following steps:

1. Initialize a `TemplateMaxCorrelation` object using the tomogram and the templates
2. Initialize a `CandidateSelector` object and use it to find the candidate locations.
3. Initialize a `FeatureExtractor` object and use it to calculate each candidate's features.
4. Label the candidates using the Labeler
5. Find the best orientation and location for each candidate using the `suggest_best_3pos_and_tilt_in_neighborhood` method from `TemplateMaxCorrelation`

The method then returns the analysis results (namely the list of reconstructed candidates, their features and their labels).

[1] This is because the number of templates in 3D scales like $O\left(\frac{360^\circ}{\alpha}\right)^3$ instead of $O\left(\frac{360^\circ}{\alpha}\right)$ in 2D (α is the angular resolution)

TemplateMaxCorrelations.py

This object is created for each tomogram that we analyze, and stores the data from the various correlation calculations. Calculating the cross correlation between the tomogram and the various templates is very computationally expensive and is by far the slowest part of the code, even using FFT (in 3D this can take several minutes depending on the machine). The cross correlations are calculated when initializing this object and can be accessed when necessary. Since the cross correlation values are needed in various places throughout the code, this object allows us to avoid repeating expensive cross correlation calculations.

It is implemented as follows: the object stores two lists, one called `correlation_values` and another called `best_tilts`. The length of both lists is equal to the number of template types. Each element of `correlation_values` is a matrix of floats whose dimensions are equal to those of the input tomogram. This matrix contains the maximum cross correlation at each point of the tomogram, taken over all orientations of the template (e.g. `correlation_values[i][j,k]` is the highest cross correlation calculated at position `[j,k]` of the input tomogram, using all the `TiltedTemplates` with `template_id = i`). Similarly, each element `best_tilts` is a tomogram sized matrix of integers, which contain the `tilt_id` which resulted in the maximum correlation (see `EulerAngle` class in CommonDataTypes).

One important class method is `suggest_best_3pos_and_tilt_in_neighborhood`, which given a candidate, searches a small neighborhood around the candidate's location for the position and orientation that result in the highest correlation. Note that this does not change the candidates tilt. This is used because sometimes the Gaussian blur used in Candidate Selector moves the location of the local maximum. The size of the neighborhood used is defined in the configuration file as `NEIGHBORHOOD_HALF_SIDE`. The method itself is used in TomogramAnalyzer.

CandidateSelector.py

The candidate selector is used to find the initial candidates for each tomogram by finding the local maxima of the cross correlation between the tomogram and all of the tilted templates.

The `CandidateSelector` object is initialized using a `TemplateMaxCorrelation` object, the template shape and the dimension of the tomogram (either 2 for 2D or 3 for 3D).

In order to select the candidate, the `select` method needs to be called. For each point in the tomogram, we use the `TemplateMaxCorrelation` object to find the maximum cross correlation over all template types and tilts. In order to avoid very close maximums resulting from the same peak, we apply a Gaussian blur to the correlations. We then use a simple peak detection algorithm to find the location of the local maximums greater than some user defined threshold called `CORRELATION_THRESHOLD`. The method returns a list of candidates corresponding to the detected peaks.

FeatureExtractor.py

This object uses the `TemplateMaxCorrelation` object to find the Feature vector for each candidate. For each template we associate a feature which is equal to the max correlation over all the different tilts of that template. Thus the feature vector is equal to the number of templates. Currently these are the only implemented features, and the class can be expanded to facilitate the calculation of other features. As an example of a future improvement, one could add the option calculate the spherical harmonic components of the candidate.

Labeler.py

This object is used to assign a label to each candidate. There are two types of Labelers - the Position Labeler and the SVM Labeler:

Position labeler:

This Labeler is used during training, and assigns a label to each detected candidate using a list of known ground truth candidates. The reconstructed candidate's label is set to the closest ground truth candidate whose distance is less than some user defined constant (`DISTANCE_THRESHOLD`). If all ground truth candidates are farther than that constant, then candidate will be labeled as junk- i.e. a maximum not corresponding to any ground truth candidate. The labels assigned by the position labeler are used to train the SVM.

SVM Labeler:

As the name suggests, this labeler uses a trained SVM to assign a label to each candidate. This labeler is used when in evaluation mode.

SvmTrain.py

This file contains the `svm_train` method which is used to train an SVM. It receives a list of tomograms, and the templates that the tomograms are comprised of. Each tomogram is analyzed, and all the reconstructed candidates are used to train the SVM. A tuple called svm_and_templates containing the SVM, the templates and the Tilts list in `EulerAngle` are returned. This tuple is received by `svm_eval` as an input, and can be used to analyze future candidates.

SvmEval.py

This file contains the `svm_eval` method, which uses the output of `svm_train` to reconstruct tomograms. It receives a tomogram iterator, and the tuple returned by `svm_train`. The function returns a list of lists of candidates. Each element in the output is the reconstructed candidates for each tomogram in the input.