SBFSEM-tools Documentation

Sara Patterson, Neitz Lab

Contents

1 Introduction

SBFSEM-tools is a Matlab toolbox developed for serial EM data and connectomics in the Neitz Lab at University of Washington. While SBFSEM-tools was built around the Viking annotation software, many aspects are quite general and could apply easily to other programs and imaging methods.
SBFSEM-tools imports annotation data through Viking's OData service and parses the results into Matlab data types. This happens behind the scenes so the average user can work with familiar objects (neuron, synapse, etc).
Other features include:

- Single neuron analysis: dendritic field area, dendrite diameter, soma size, stratification, synapse distribution
- Group analysis: density recovery profile, nearest neighbor, synapse statistics
- 3D volume rendering of polygon annotations and free-form traces over a stack of EM images. 2D projections of dendritic fields.
- Generate surfaces from IPL boundary markers, compensate for Z-axis misalignments.
- Misc UIs for visualizing EM images and annotations

2 Install

SBFSEM-tools is available on Github: sbfsem-tools. Clone or download the repository and make sure to add it to your MATLAB path.
Make sure sbfsem-tools is on your MATLAB path either by adding the following to your startup file, or running it from the cmd line. You'll need to fill in the '...' with the full file path

```
addpath(genpath('C:\...\sbfsem-tools'));
```

Importing the base sbfsem-tools package is helpful too.

```
% Without importing the package
c207 = sbfsem.Neuron(207, 't');
% Import the package
import sbfsem.*;
% Now you don't have to type "sbfsem." each time
c207 = Neuron(207, 't');
```

The sample code in the documentation assumes the base package has been imported.
See `Tutorial.m` in the main sbfsem-tools folder for more information on setup.

3 Neuron

Most work revolves around the Neuron class, which is a basic representation of a neuron (called a Structure in Viking). A neuron is created with two inputs: the Cell ID and the volume name:

```
% Cell 6800 from NeitzTemporalMonkey
c6800 = Neuron(6800, 't');
% Cell 2795 from NeitzInferiorMonkey
c2795 = Neuron(2795, 'NeitzInferiorMonkey');
% Same cell but using the abbreviated volume name
c2795 = Neuron(2795, 'i');
```

The full volume names are: 'NeitzTemporalMonkey', 'NeitzInferiorMonkey' and 'MarcRC1'. These can be abbreviated to 't', 'i' and 'r', respectively.

3.1 Neuron Properties

Neuron has the following publicly accessible properties:

1. `viking` - struct - cell info from Viking
2. `nodes` - table - all annotations
3. `edges` - table - links between annotations
4. `volumeScale` - vector - units are nm/pix for X and Y, nm/section for Z.
5. `synapses` - table - all child structures
6. `geometries` - table - closed curve geometries
7. `analysis` - containers.Map - keys for each NeuronAnalysis
8. `lastModified` - datestr - last update of neuron from OData

3.2 Neuron Methods

The Neuron class has the following publicly available methods. Most are convenience functions to simplify data access.

1. **update**
   Description: Updates the underlying data from OData. Useful when working with a neuron you are currently annotating.
   Syntax: `obj.update();`
2. **graph**
   Convert the neuron into an undirected or directed graph. For more information on what this enables, see MATLAB's graph class documentation.
   Syntax: `G = graph(obj);`
3. **printSyn**
   Print a summary of the neuron's synapses to the command line.
   Syntax: `obj.printSyn;`
4. **synapseNames**
   Outputs a list of synapse types associated with the neuron
   Syntax: `names = obj.synapseNames;`

5. **getCellNodes**
   The node table contains both synapse and cell body annotations. This method returns only the cell body annotations.
   Syntax: `T = obj.getCellNodes;`

6. **getSynapseNodes**
   Same idea as getCellNodes, returns only the synapse annotations.
   Syntax: `T = obj.getSynapseNodes(onlyUnique);`

7. **getSynapseXYZ**
   Returns the XYZ locations of all annotations associated with a given synapse.
   Syntax: `xyz = getSynapseXYZ('synapseName');`

8. **getCellXYZ**
   Returns the XYZ locations of all cell body annotations
   Syntax: `xyz = obj.getCellXYZ;`

9. **setGeometries**
   Retrieve closed curve annotation control points from OData - helpful when you need to update frequently but don't want to reload the entire cell.
   Syntax: `obj.setGeometries();`

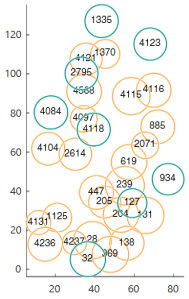
10. **synapseIDs**
    Returns the location IDs for annotations of a given synapse type.
    Syntax: `IDs = obj.synapseIDs('synapseName');`

11. **saveNeuron**
    The speed of the OData import should make saving Neurons unnecessary, however, this is an option just in case.
    Syntax: `obj.save();`

3.3 NeuronGroup

A single data object to hold related Neurons. Inputs can be ID numbers or existing Neurons.

```
h1hc = sbfsem.NeuronGroup([28, 447, 619]);
% Add a neuron to an existing group
h1hc.add(4568);
% Remove a neuron
h1hc.remove(4568);
```



The NeuronGroup class is still a work-in-progress. The current methods include:

somaPlot Plot the mosaic of somas

```
h1hc.somaPlot();
h1hc.somaPlot('addLabel',true); % Label with ID
h1hc.somaPlot('ax',gca); % Add to existing axis
% Two methods for controlling plot color:
h1hc.somaPlot('Color', [0 0.8 0.3]);
h1hc.setPlotColor([0 0.8 0.3]); h1hc.somaPlot;
```

getSomaSizes Return statistics on the soma sizes

3.4 NeuronAnalysis

The NeuronAnalysis class helps keep population data organized by managing input parameters and results of common analyses. To create a new analysis, subclass NeuronAnalysis and edit the `doAnalysis` and `visualize` methods. See `Tutorial.m` for information on these existing analysis classes:

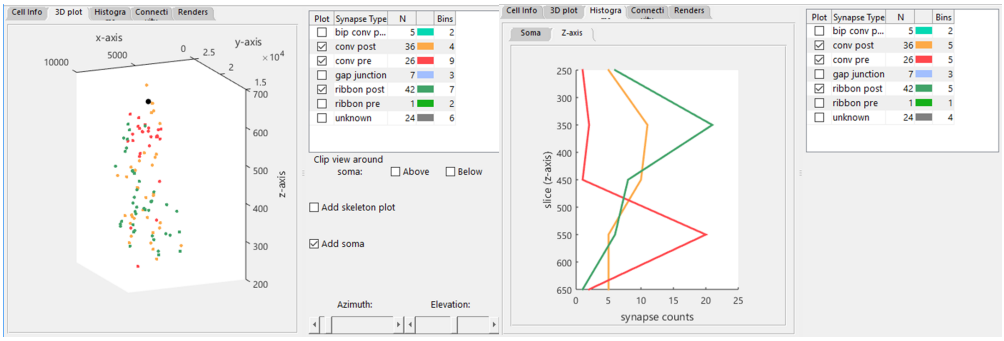
- **DendriticFieldHull** - uses convex hull to estimate dendritic field area, includes methods for removing axons prior to analysis.
- **PrimaryDendriteDiameter** - returns the median dendrite diameter at a given distance from the soma.

4 Views

Neurons can be passed to several UIs with the following syntax:

```
% VIEWNAME(NeuronObject);
c6800 = Neuron(6800, 't');
StratificationView(c6800);
```

- **NodeView** - 3D scatter plot of cell and synapse annotations associated with a cell.
- **StratificationView** - Z-axis histogram of dendrites and synapses.
- **SomaDistanceView** - proxmial-distal distribution of dendrites and synapses.

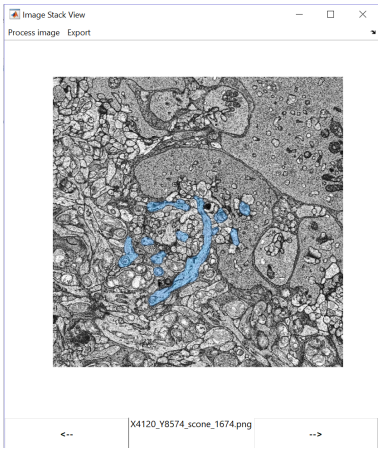


The checkboxes in the synapse table allow toggling the visibility of each synapse type. If the bin numbers in the synapse table are edited, the histogram plots will automatically update.

5 Images

5.1 ImageStackApp

The ImageStackApp was built to scan through a series of images exported using Viking's Export Frames option (although this will work with any folder of images, provided there is some numbering in the file names). Pass a folder name to browse through the images (using left and right arrow keys or buttons). If the filenames contain numbers (Viking's export frames appends a frame number automatically), these will be used to order the images.




```
ImageStackApp('C:\...\foldername');
```

To crop the set of images, use `Process Image` → `Crop` and draw a rectangle on the image. Use `Full size` to return to the original size. Both changes require moving to another image to apply.

5.2 ImageStack class

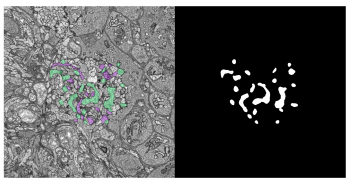
The ImageStackApp creates an ImageStack object. To create a GIF, create an instance of ImageStack alone and use the `stack2gif` function.

```
I = sbfsem.images.ImageStack('C:\...\foldername');
I.stack2gif();
```

5.3 Image Segmentation

5.3.1 Color ROIs

Renders can also be generated from a stack of EM images with color annotations. See `algorithms/segmentColorROI.m` for more information. Eventually I will make this more user-friendly. For now, new cell types and their thresholds must be added manually. To the right is an example of an L/M-cone with the ON- and OFF-midgets colored in Photoshop. On the right is the binary matrix output where 1/true (white) indicates a pixel within the color threshold limits.



5.3.2 Outlines

`getImageRGB.m` provides a quick tool for extracting data points from an image, given the outlines were drawn with a distinct color (red, green or blue in Paint works just fine). The first argument is the image, the second is the RGB value used to draw outlines.

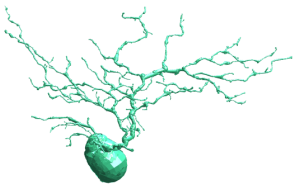
```
pts = getImageRGB(im, [0 1 0]);

```

If more than one outline is present, MATLAB's `bwboundaries` can probably separate them into distinct objects.

6 Renders

6.1 Polygon Meshes

Unlike the volumetric renders, the polygon meshes retain their absolute XYZ locations. This greatly simplifies the process of rendering multiple neurons to a single scene.



```
r1441 = sbfsem.render.Cylinder(Neuron(1441, 'i'));
% If no axis handle is provided, creates new figure
r1441.render();
% Keep axis handle to direct next render to same
 figure
ax = gca;
r1411 = sbfsem.render.Cylinder(Neuron(1411, 'i'));
r1411.render('ax', ax, 'FaceColor', [0 0.8 0.3]);
% Export as COLLADA .dae file
r1411.dae();
```

A brief outline of the steps involved:

1. Query OData service for the all annotations in a given cell ID. Save the XYZ coordinates, radius and location ID of each annotation.

2. Query OData service for all links between annotations.

3. Apply XY transforms to compensate for major shifts in section alignment.

4. Query OData service for the volume dimensions - use these to convert XYZ coordinates and radius from pixels to microns.

5. Convert the annotations and links between each annotation into the nodes and edges of an undirected graph.

6. Create an undirected graph from all nodes (annotations/locations) and edges (location links).

7. Use depth-first search (`dfsearch`) to divide nodes into segments of degree = 2. This avoids rendering dendrite branches.

8. Link each segment back to the parent node of degree ¿ 2. This way some nodes are repeated but each edge is represented only once.

9. Find the magnitude and angle of the distances between all connected nodes. Use this to get a quaternion rotation matrix for each edge.

10. Input the radii to MATLAB's built-in 3D `cylinder` function, which creates a cylinder orthogonal to the XY plane.

11. Rotate each node's cylinder by it's quaternion rotation matrix.

12. Render each segment using surf - no lighting or edge color.

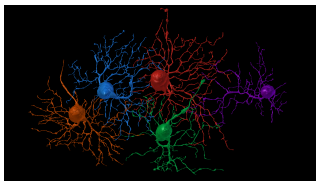
6.1.1 2D Projection

Rendering the above without any lighting creates a 3D figure with a 2D projection appearance.

6.1.2 Blender Export

To export the render to a COLLADA file, use:

```
r1411 = sbfsem.render.Cylinder(c1411);
r1411.dae();
```



This function will open up a File Explorer window that allows you to select where to save the .dae file.

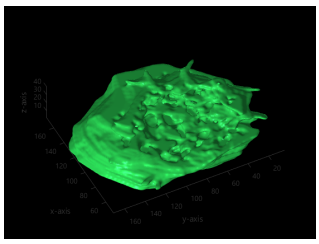
In Blender, each .dae file must be imported separately through the `File` → `Import` → `Collada (default) .dae`. The relative positions of each neuron will be preserved as long as you import all the cells before rotating/transposing/scaling.
The COLLADA export file is very minimal compared to VikingPlot. You will have to add a new Material for each cell (the + `New` button in the Materials tab on the right panel). To optimize the final render, I then select `Smooth` for the Shading option in the Tools tab of the left panel. On the `Object Data` tab (directly to the left of the `Materials` tab), check both `Auto Smooth` and `Double Sided`. I move the `Angle` up to 50-60degree if the soma has especially sharp angles. Switching the view to Ortho (numpad 5) is helpful too.

6.2 Volumetric Renders

6.2.1 Closed Curve

The closed curve renders support cutouts and multiple annotations per section. Mixing closed curve and disc renders is the next goal.

```
c2542 = Neuron(2542, 'i');
lmcone = renderClosedCurve(c2542);
% Decrease the resolution to speed up the renders
lmcone = renderClosedCurve(c2542, 'resizeFactor',
 0.5);
```



A basic outline of the code:

1. Import control points for each Closed Curve annotation and parse into a 2xN matrix.

2. Calculate a CatmullRom interpolating spline from the control points.

3. Calculate a bounding box (the minimal volume containing all annotations)

4. For each section, use `patch` to plot closed curves as white, cutouts as black and convert to a 2D binary matrix (similar to Figure 5.3.1).

5. Resample

6. Pad the 2D images as needed (occasionally the sizes will differ by 1 pixel) and stack into a 3D binary volume.

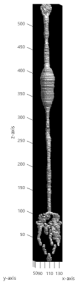
7. Smooth the data (`smooth3` for now but there's much room for improvement here).

8. Render using `isosurface` and `isonormals`.

9. Set axis dimensions (`daspect`) according to volume dimensions pulled from OData service.

6.2.2 Discs

The same method can be applied to Disc annotations as well for an accurate view of the annotations as discs in 3D. As is, the result is not as good as the polygon mesh rendering but could have some advantages in Blender.



```
c1893 = Neuron(1983, 'i');
r1893 = sbfsem.render.Disc(c1893);
r1893.dae();
```

6.3 VikingPlot

Editing the MATLAB figure generated by VikingPlot can be a good alternative to Blender. A few useful commands:

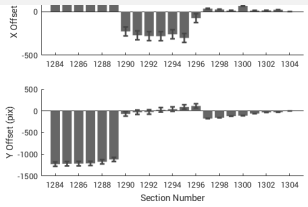
```
% Make sure the figure is the active window.
ax = gca; % Create a handle to the axis
ax.Children % Returns a list of child structures
% There should be PATCH and LIGHT objects.
% Use numerical indexing to generate handles to the objects
% NOTE: The numbering might be different on your figure
lightObj = ax.Children(1);
renderObj = ax.Children(2);
% A few useful properties to edit for PATCH objects:
renderObj.FaceColor = [0, 0.8, 0.3];
renderObj.FaceAlpha = 0.7;
```

7 Image Registration

7.1 Section Misalignment

The function `xyRegistration.m` calculates the XY offset through a range of Z sections and outputs statistics on the offsets relative to the most sclerad section. The first output is an Nx3 matrix (columns = Section Number, XOffset, YOffset). A data file (`data/XY_OFFSET_NEITZINFERIORMONKEY.txt`) tracks these offsets and can be updated using `updateRegistration.m`. The units are Viking's pixel coordinates. The transform is applied when the Neuron class first pulls X,Y coordinates from the OData service.

```
[data, S] = xyRegistration('i', [1284 1309], true)
 ;
updateRegistration('i', data);
```



This code is very new and needs quite a few improvements. However, the preliminary results are making a large difference in renders. See the code comments for more info.

7.2 IPL Boundary Markers

The INLBoundary and GCLBoundary objects retrieve all boundary markers in a given volume and create a surface to represent the slope of the tissue.

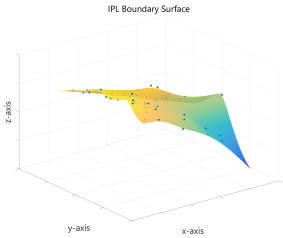
```
inl = INLBoundary('i');
% Update marker locations from OData
inl.refresh();
% Create the surface
inl.doAnalysis();
% Plot the surface
plot(inl);
```

Warping the Z-axis to account for the gradually sloping tissue is a goal. See IPL Boundary functions for generation of a surface based on depth markers. Soon I hope to have code to apply the transform - right now I'm thinking of adapting a commonly used method for warping neurons based on CHaT bands (Github repository) [1].

8 Appendix

8.1 Matlab 101

MATLAB tips are currently scattered throughout the documentation.. consolidate them here.
Data structures like table, struct.
Methods are class-specific functions. A list of all methods can be obtained by `methods(className)` (so, for example, `methods('Neuron')`).



8.2 Git 101

Git can be downloaded here. Once downloaded, you can open the terminal by right clicking anywhere in the whitespace of a File Explorer window and selecting `Git Bash here`. Do this in the folder you want sbfsem-tools to live in.

```
git clone http://github.com/sarastokes/sbfsem-tools.git
```

There should be a folder now called "sbfsem-tools". At any point, you can update the code to the latest version by right clicking on the folder itself, selecting `Git bash here` and typing in:

```
git pull
```

This saves the step of deleting the existing sbfsem-tools folder and re-downloading it every time an update is available.

References

[1] Sumbul, U., Song, S., McCulloch, K., Becker, M., Lin, B., Sanes, J.R., Masland, R.H. & Seung, H (2014) A genetic and computation approach to structurally classify neuronal types. *Nature Comm cations*, 5, 3512
Github repository: https://github.com/uygarsumbul/rgc