gala Documentation

Release 0.3.2

Juan Nunez-Iglesias

November 24, 2015

Contents

Gala is a python library for performing and evaluating image segmentation, distributed under the open-source, BSD-like Janelia Farm license. It implements the algorithm described in Nunez-Iglesias et al., PLOS ONE, 2013.

If you use this library in your research, please cite:

> Nunez-Iglesias J, Kennedy R, Plaza SM, Chakraborty A and Katz WT (2014) Graph-based active learning of agglomeration (GALA): a Python library to segment 2D and 3D neuroimages. *Front. Neuroinform. 8:34.* doi:10.3389/fninf.2014.00034

If you use or compare to the GALA algorithm in your research, please cite:

> Nunez-Iglesias J, Kennedy R, Parag T, Shi J, Chklovskii DB (2013) Machine Learning of Hierarchical Clustering to Segment 2D and 3D Images. *PLoS ONE 8(8): e71715.* doi:10.1371/journal.pone.0071715

Gala supports n-dimensional images (images, volumes, videos, videos of volumes...) and multiple channels per image.

Contents:

CHAPTER 1

Installation

1.1 Requirements

- Python 3.4 or 2.7
- numpy 1.7+
- scipy 0.10+
- Image (a.k.a. Python Imaging Library or PIL) 1.1.7 or Pillow 2.5+
- networkx 1.6+
- HDF5 and h5py 1.5+
- cython 0.17+
- scikit-learn 0.15+
- matplotlib 1.2+
- scikit-image 0.11+
- viridis 0.3+

1.1.1 Optional dependencies

- progressbar 2.3 (Python 2.7 only, currently)
- vigra/vigranumpy (1.9.0)
- libtiff 0.4+

In its original incarnation, this project used Vigra for the random forest classifier. Installation is less simple than scikit-learn, which has emerged in the last few years as a truly excellent implementation and is now recommended. Tests in the test suite expect scikit-learn rather than Vigra. You can also use any of the scikit-learn classifiers, including their newly-excellent random forest.

1.1.2 Installing with distutils

Gala is a Python library with limited Cython extensions and can be installed in two ways:

- Use the command `python setup.py build_ext -i` in the gala directory, then add the gala directory to your PYTHONPATH environment variable, or

- Install it into your preferred python environment with `python setup.py install`.

1.1.3 Installing requirements

Though you can install all the requirements yourself, as most are available in the Python Package Index (PyPI) and can be installed with simple commands, the easiest way to get up and running is to use the Continuum Anaconda Python distribution. An example of installing all dependencies for gala on linux is shown in the `.travis.yml` test recipe.

1.1.4 Installing with Buildem

Alternatively, you can use Janelia's own buildem system to automatically download, compile, test, and install requirements into a specified buildem prefix directory. (You will need CMake.)

```
$ cmake -D BUILDEM_DIR=/path/to/platform-specific/build/dir <gala directory>
$ make
```

You might have to run the above steps twice if this is the first time you are using the buildem system.

On Mac, you might have to install compilers (such as gcc, g++, and gfortran).

1.1.5 Testing

The test coverage is rather tiny, but it is still a nice way to check you haven't completely screwed up your installation. The tests do cover the fundamental functionality of agglomeration learning.

We use pytest for testing. Run the tests by building gala in-place and running the `py.test` command. (You need to have installed pytest and pytest-cov for this to work. Both are readily available in PyPI.)

Alternatively, you can run individual test files independently:

```
$ cd tests
$ python test_agglo.py
$ python test_features.py
$ python test_watershed.py
$ python test_optimized.py
$ python test_gala.py
```

CHAPTER 2

Getting Started

An example script, `example.py`, exists in the `tests/example-data` directory. We step through it here for a quick rundown of gala's capabilities.

First, import gala's submodules:

```
from gala import imio, classify, features, agglo, evaluate as ev
```

Next, read in the training data: a ground truth volume (`gt_train`), a probability map (`pr_train`) and a superpixel or watershed map (`ws_train`).

```
gt_train, pr_train, ws_train = (map(imio.read_h5_stack,
                                    ['train-gt.lzf.h5', 'train-p1.lzf.h5',
                                     'train-ws.lzf.h5']))
```

A *feature manager* is a callable object that computes feature vectors from graph edges. The object has the following responsibilities, which it can inherit from `classify.base.Null`:

- create a (possibly empty) *feature cache* on each edge and node, precomputing some of the calculations needed for feature computation;
- maintain the feature cache throughout node merges during agglomeration; and,
- compute the feature vector from the feature caches when called with the inputs of a graph and two nodes.

Feature managers can be chained through the `features.Composite` class.

```
fm = features.moments.Manager()
fh = features.histogram.Manager()
fc = features.base.Composite(children=[fm, fh])
```

With the feature manager, and the above data, we can create a *region adjacency graph* or *RAG*, and use it to train the agglomeration process:

```
g_train = agglo.Rag(ws_train, pr_train, feature_manager=fc)
(X, y, w, merges) = g_train.learn_agglomerate(gt_train, fc)[0]
y = y[:, 0] # gala has 3 truth labeling schemes, pick the first one
```

`X` and `y` above have the now-standard scikit-learn supervised dataset format. This means we can use any classifier that satisfies the scikit-learn API. Below, we use a simple wrapper around the scikit-learn `RandomForestClassifier`.

```
rf = classify.DefaultRandomForest().fit(X, y)
```

The composition of a feature map and a classifier defines a *policy* or *merge priority function*, which will determine the agglomeration of a volume of hereby unseen data (the *test* volume).

```
learned_policy = agglo.classifier_probability(fc, rf)

pr_test, ws_test = (map(imio.read_h5_stack,
                        ['test-p1.lzf.h5', 'test-ws.lzf.h5']))
g_test = agglo.Rag(ws_test, pr_test, learned_policy, feature_manager=fc)
```

The best expected segmentation is obtained at a threshold of 0.5, when a merge has even odds of being correct or incorrect, according to the trained classifier.

```
g_test.agglomerate(0.5)
```

The RAG is a *model* for the segmentation. To extract the segmentation itself, use the `get_segmentation` function. This is a map of labels of the same shape as the original image.

```
seg_test1 = g_test.get_segmentation()
```

Gala transparently supports multi-channel probability maps. In the case of EM images, for example, one channel may be the probability that a given pixel is part of a cell boundary, while the next channel may be the probability that it is part of a mitochondrion. The feature managers work identically with single and multi-channel features.

```
# p4_train and p4_test have 4 channels
p4_train = imio.read_h5_stack('train-p4.lzf.h5')
# the existing feature manager works transparently with multiple channels!
g_train4 = agglo.Rag(ws_train, p4_train, feature_manager=fc)
(X4, y4, w4, merges4) = g_train4.learn_agglomerate(gt_train, fc)[0]
y4 = y4[:, 0]
rf4 = classify.DefaultRandomForest().fit(X4, y4)
learned_policy4 = agglo.classifier_probability(fc, rf4)
p4_test = imio.read_h5_stack('test-p4.lzf.h5')
g_test4 = agglo.Rag(ws_test, p4_test, learned_policy4, feature_manager=fc)
g_test4.agglomerate(0.5)
seg_test4 = g_test4.get_segmentation()
```

For comparison, gala allows the implementation of many agglomerative algorithms, including mean agglomeration (below) and LASH.

```
g_testm = agglo.Rag(ws_test, pr_test,
                    merge_priority_function=agglo.boundary_mean)
g_testm.agglomerate(0.5)
seg_testm = g_testm.get_segmentation()
```

2.1 Evaluation

The gala library contains numerous evaluation functions, including edit distance, Rand index and adjusted Rand index, and our personal favorite, the variation of information (VI):

```
gt_test = imio.read_h5_stack('test-gt.lzf.h5')
import numpy as np
results = np.vstack((
    ev.split_vi(ws_test, gt_test),
    ev.split_vi(seg_testm, gt_test),
    ev.split_vi(seg_test1, gt_test),
    ev.split_vi(seg_test4, gt_test)
    ))
print(results)
```

This should print something like:

```
[[ 0.1845286   1.64774412]
 [ 0.18719817  1.16091003]
 [ 0.38978567  0.28277887]
 [ 0.39504714  0.2341758 ]]
```

Each row is an evaluation, with the first number representing the undersegmentation error or false merges, and the second representing the oversegmentation error or false splits, both measured in bits.

(Results may vary since there is some randomness involved in training a random forest, and the datasets are small.)

As mentioned earlier, many other evaluation functions are available. See the documentation for the `evaluate` package for more information.

```
# rand index and adjusted rand index
ri = ev.rand_index(seg_test1, gt_test)
ari = ev.adj_rand_index(seg_test1, gt_test)
# Fowlkes-Mallows index
fm = ev.fm_index(seg_test1, gt_test)
```

2.2 Other options

Gala supports a wide array of merge priority functions to explore your data. We can specify the median boundary probability with the `merge_priority_function` argument to the RAG constructor:

```
g_testM = agglo.Rag(ws_test, pr_test,
                    merge_priority_function=agglo.boundary_median)
```

A user can specify their own merge priority function. A valid merge priority function is a callable Python object that takes as input a graph and two nodes, and returns a real number.

2.3 To be continued...

That's a quick summary of the capabilities of Gala. There are of course many options under the hood, many of which are undocumented... Feel free to push me to update the documentation of your favorite function!

CHAPTER 3

API Reference

3.1 gala.agglo: RAG Agglomeration

class gala.agglo.**Rag**(*watershed=array([], dtype=int64), probabilities=array([], dtype=float64), merge_priority_function=<function boundary_mean>, gt_vol=None, feature_manager=<gala.features.base.Null object>, mask=None, show_progress=False, connectivity=1, channel_is_oriented=None, orientation_map=array([], dtype=float64), normalize_probabilities=False, exclusions=array([], dtype=float64), isfrozennode=None, isfrozenedge=None*)

Region adjacency graph for segmentation of nD volumes.

Parameters **watershed** : array of int, shape (M, N, ..., P)

The labeled regions of the image. Note: this is called *watershed* for historical reasons, but could refer to a superpixel map of any origin.

probabilities : array of float, shape (M, N, ..., P[, Q])

The probability of each pixel of belonging to a particular class. Typically, this has the same shape as *watershed* and represents the probability that the pixel is part of a region boundary, but it can also have an additional dimension for probabilities of belonging to other classes, such as mitochondria (in biological images) or specific textures (in natural images).

merge_priority_function : callable function, optional

This function must take exactly three arguments as input (a Rag object and two node IDs) and return a single float.

feature_manager : `features.base.Null` object, optional

A feature manager object that controls feature computation and feature caching.

mask : array of bool, shape (M, N, ..., P)

A mask of the same shape as *watershed*, *True* in the positions to be processed when making a RAG, *False* in the positions to ignore.

show_progress : bool, optional

Whether to display an ASCII progress bar during long- -running graph operations.

connectivity : int in {1, ..., *watershed.ndim*}

When determining adjacency, allow neighbors along *connectivity* dimensions.

channel_is_oriented : array-like of bool, shape (Q,), optional

For multi-channel images, some channels, for example some edge detectors, have a specific orientation. In conjunction with the *orientation_map* argument, specify which channels have an orientation associated with them.

orientation_map : array-like of float, shape (Q,)

Specify the orientation of the corresponding channel. (2D images only)

normalize_probabilities : bool, optional

Divide the input *probabilities* by their maximum to ensure a range in [0, 1].

exclusions : array-like of int, shape (M, N, ..., P), optional

Volume of same shape as *watershed*. Mark points in the volume with the same label (>0) to prevent them from being merged during agglomeration. For example, if *exclusions[45, 92] == exclusions[51, 105] == 1*, then segments *watershed[45, 92]* and *watershed[51, 105]* will never be merged, regardless of the merge priority function.

isfrozennode : function, optional

Function taking in a Rag object and a node id and returning a bool. If the function returns `True`, the node will not be merged, regardless of the merge priority function.

isfrozenedge : function, optional

As *isfrozennode*, but the function should take the graph and *two* nodes, to specify an edge that cannot be merged.

Attributes

Methods

agglomerate(*threshold=0.5*, *save_history=False*)

Merge nodes hierarchically until given edge confidence threshold.

This is the main workhorse of the `agglo` module!

Parameters **threshold** : float, optional

The edge priority at which to stop merging.

save_history : bool, optional

Whether to save and return a history of all the merges made.

Returns **history** : list of tuple of int, optional

The ordered history of node pairs merged.

scores : list of float, optional

The list of merge scores corresponding to the *history*.

evaluation : list of tuple, optional

The split VI after each merge. This is only meaningful if a ground truth volume was provided at build time.

Notes

This function returns `None` when *save_history* is `False`.

agglomerate_count(*stepsize=100*, *save_history=False*)
Agglomerate until 'stepsize' merges have been made.

This function is like `agglomerate`, but rather than to a certain threshold, a certain number of merges are made, regardless of threshold.

Parameters **stepsize** : int, optional

The number of merges to make.

save_history : bool, optional

Whether to save and return a history of all the merges made.

Returns **history** : list of tuple of int, optional

The ordered history of node pairs merged.

scores : list of float, optional

The list of merge scores corresponding to the *history*.

evaluation : list of tuple, optional

The split VI after each merge. This is only meaningful if a ground truth volume was provided at build time.

See also:

agglomerate

Notes

This function returns `None` when *save_history* is `False`.

agglomerate_ladder(*min_size=1000*, *strictness=2*)
Merge sequentially all nodes smaller than *min_size*.

Parameters **min_size** : int, optional

The smallest allowable segment after ladder completion.

strictness : {1, 2, 3}, optional

strictness == 1: all nodes smaller than *min_size* are merged according to the merge priority function. *strictness == 2*: in addition to *1*, small nodes can only be merged to big nodes. *strictness == 3*: in addition to *2*, nodes sharing less than one pixel of boundary are not agglomerated.

Returns None

Notes

Nodes that are on the volume boundary are not agglomerated.

at_volume_boundary(*n*)
Return True if node n touches the volume boundary.

build_boundary_map(*ebunch=None*)
Return a map of the current merge priority.

Parameters **ebunch** : iterable of (int, int), optional

The list of edges for which to build a map. Use all edges if not provided.

Returns **bm** : array of float

The image of the edge weights.

build_graph_from_watershed(*idxs=None*)
Build the graph object from the region labels.

The region labels should have been set ahead of time using `set_watershed()`.

Parameters **idxs** : array-like of int, optional

Linear indices into raveled volume array. If provided, the graph is built only for these indices.

build_merge_queue()
Build a queue of node pairs to be merged in a specific priority.

Parameters **None**

Returns **mq** : MergeQueue object

A MergeQueue is a Python `deque` with a specific element structure: a list of length 4 containing:

- the merge priority (any ordered type)
- a 'valid' flag
- and the two nodes in arbitrary order

The valid flag allows one to "remove" elements from the queue in O(1) time by setting the flag to `False`. Then, one checks the flag when popping elements and ignores those marked as invalid.

One other specific feature is that there are back-links from edges to their corresponding queue items so that when nodes are merged, affected edges can be invalidated and reinserted in the queue with a new priority.

build_volume(*nbunch=None*)
Return the segmentation induced by the graph.

Parameters **nbunch** : iterable of int (node id), optional

A list of nodes for which to build the volume. All nodes are used if this is not provided.

Returns **seg** : array of int

The segmentation implied by the graph.

Notes

This function is very similar to `get_segmentation`, but it builds the segmentation from the bottom up, rather than using the currently-stored segmentation.

cluster_by_labels(*labels*, *nodes=None*)
Merge all superpixels with the same label (1 label per 1 sp)

compute_W(*merge_priority_function*, *sigma=5100.0*, *nodes=None*)
Computes the weight matrix for clustering

compute_feature_caches()
Use the feature manager to compute node and edge feature caches.

Parameters None

Returns None

compute_orphans()
Find all the segments that do not touch the volume boundary.

Parameters None

Returns **orphans** : list of int (node id)

A list of node ids.

Notes

This function differs from `orphans` in that it does not use the graph, but rather computes orphans directly from the segmentation.

copy()
Return a copy of the object and attributes.

get_edge_coordinates(*n1*, *n2*, *arbitrary=False*)
Find where in the segmentation the edge (n1, n2) is most visible.

get_segmentation(*threshold=None*)
Return the unpadded segmentation represented by the graph.

Remember that the segmentation volume is padded with an "artificial" segment that envelops the volume. This function simply removes the wrapping and returns a segmented volume.

Parameters **threshold** : float, optional

Get the segmentation at the given threshold. If no threshold is given, return the segmentation at the current level of agglomeration.

Returns **seg** : array of int

The segmentation of the volume presently represented by the graph.

is_traversed_by_node(*n*)
Determine whether a body traverses the volume.

This is defined as touching the volume boundary at two distinct locations.

Parameters **n** : int (node id)

The node being inspected.

Returns **tr** : bool

Whether the segment "traverses" the volume being segmented.

learn_agglomerate(*gts*, *feature_map*, *min_num_samples=1*, *learn_flat=True*, *learning_mode='strict'*, *labeling_mode='assignment'*, *priority_mode='active'*, *memory=True*, *unique=True*, *random_state=None*, *max_num_epochs=10*, *min_num_epochs=2*, *max_num_samples=inf*, *classifier='random forest'*, *active_function=<function classifier_probability>*, *mpf=<function boundary_mean>*)

Agglomerate while comparing to ground truth & classifying merges.

Parameters **gts** : array of int or list thereof

The ground truth volume(s) corresponding to the current probability map.

feature_map : function (Rag, node, node) -> array of float

The map from node pairs to a feature vector. This must consist either of uncached features or of the cache used when building the graph.

min_num_samples : int, optional

Continue training until this many training examples have been collected.

learn_flat : bool, optional

Do a flat learning on the static graph with no agglomeration.

learning_mode : {'strict', 'loose'}, optional

In 'strict' mode, if a "don't merge" edge is encountered, it is added to the training set but the merge is not executed. In 'loose' mode, the merge is allowed to proceed.

labeling_mode : {'assignment', 'vi-sign', 'rand-sign'}, optional

How to decide whether two nodes should be merged based on the ground truth segmentations. `'assignment'` means the nodes are assigned to the ground truth node with which they share the highest overlap. `'vi-sign'` means the the VI change of the switch is used (negative is better). `'rand-sign'` means the change in Rand index is used (positive is better).

priority_mode : string, optional

One of:

`'active'`: Train a priority function with the data from previous epochs to obtain the next.

`'random'`: Merge edges at random. `'mixed'`: Alternate between epochs of `'active'`

and `'random'`.

`'mean'`: Use the mean boundary value. (In this case, training is limited to 1 or 2 epochs.)

`'custom'`: Use the function provided by *mpf*.

memory : bool, optional

Keep the training data from all epochs (rather than just the most recent one).

unique : bool, optional

Remove duplicate feature vectors.

random_state : int, optional

If provided, this parameter is passed to *get_classifier* to set the random state and allow consistent results across tests.

max_num_epochs : int, optional

Do not train for longer than this (this argument *may* override the *min_num_samples* argument).

min_num_epochs : int, optional

Train for no fewer than this number of epochs.

max_num_samples : int, optional

Train for no more than this number of samples.

classifier : string, optional

Any valid classifier descriptor. See `gala.classify.get_classifier()`

active_function : function (feat. map, classifier) -> function, optional

Use this to create the next priority function after an epoch.

mpf : function (Rag, node, node) -> float

A merge priority function to use when `priority_mode` is `'custom'`.

Returns **data** : list of array

Four arrays containing:

- the feature vectors, shape `(n_samples, n_features)`.
- the labels, shape `(n_samples, 3)`. A value of *-1* means "should merge", while *1* means "should not merge". The columns correspond to the three labeling methods: assignment, VI sign, or RI sign.
- the VI and RI change of each merge, `(n_edges, 2)`.
- the list of merged edges `(n_edges, 2)`.

alldata : list of list of array

A list of lists like *data* above: one list for each epoch.

See also:

Rag

Notes

The gala algorithm [1] uses the default parameters. For the LASH algorithm [2], use:

•*learning_mode*: `'loose'`

•*labeling_mode*: `'rand-sign'`

•*memory*: `False`

References

[R1], *[R2]*

learn_edge(*edge*, *ctables*, *assignments*, *feature_map*)

Determine whether an edge should be merged based on ground truth.

Parameters **edge** : (int, int) tuple

An edge in the graph.

ctables : list of array

A list of contingency tables determining overlap between the current segmentation and the ground truth.

assignments : list of array

Similar to the contingency tables, but each row is thresholded so each segment corresponds to exactly one ground truth segment.

feature_map : function (Rag, node, node) -> array of float

The map from node pairs to a feature vector.

Returns **features** : 1D array of float

The feature vector for that edge.

labels : 1D array of float, length 3

The labels determining whether the edge should be merged. A value of *-1* means "should merge", while *1* means "should not merge". The columns correspond to the three labeling methods: assignment, VI sign, or RI sign.

weights : 1D array of float, length 2

The VI and RI change of the merge.

nodes : tuple of int

The given edge.

learn_epoch(*ctables*, *feature_map*, *learning_mode='permissive'*, *labeling_mode='assignment'*)

Learn the agglomeration process using various strategies.

Parameters **ctables** : array of float or list thereof

One or more contingency tables between own segments and gold standard segmentations

feature_map : function (Rag, node, node) -> array of float

The map from node pairs to a feature vector. This must consist either of uncached features or of the cache used when building the graph.

learning_mode : {'strict', 'permissive'}, optional

If `'strict'`, don't proceed with a merge when it goes against the ground truth. For historical reasons, 'loose' is allowed as a synonym for 'strict'.

labeling_mode : {'assignment', 'vi-sign', 'rand-sign'}, optional

Which label to use for *learning_mode*. Note that all labels are saved in the end.

Returns **data** : list of array

Four arrays containing:

- the feature vectors, shape `(n_samples, n_features)`.
- the labels, shape `(n_samples, 3)`. A value of *-1* means "should merge", while *1* means "should not merge". The columns correspond to the three labeling methods: assignment, VI sign, or RI sign.
- the VI and RI change of each merge, `(n_edges, 2)`.
- the list of merged edges `(n_edges, 2)`.

learn_flat(*gts*, *feature_map*)

Learn all edges on the graph, but don't agglomerate.

Parameters **gts** : array of int or list thereof

The ground truth volume(s) corresponding to the current probability map.

feature_map : function (Rag, node, node) -> array of float

The map from node pairs to a feature vector. This must consist either of uncached features or of the cache used when building the graph.

Returns **data** : list of array

Four arrays containing:

- the feature vectors, shape `(n_samples, n_features)`.
- the labels, shape `(n_samples, 3)`. A value of *-1* means "should merge", while *1* means "should not merge". The columns correspond to the three labeling methods: assignment, VI sign, or RI sign.
- the VI and RI change of each merge, `(n_edges, 2)`.
- the list of merged edges `(n_edges, 2)`.

See also:

learn_agglomerate

merge_edge_properties(*src*, *dst*)
Merge the properties of edge src into edge dst.

Parameters **src, dst** : (int, int)

Edges being merged.

Returns None

merge_nodes(*n1*, *n2*, *merge_priority=0.0*)
Merge two nodes, while updating the necessary edges.

Parameters **n1, n2** : int

Nodes determining the edge for which to update the UCM.

merge_priority : float, optional

The merge priority of the merge.

Returns **node_id** : int

The id of the node resulting from the merge.

Notes

Additionally, the RIG (region intersection graph), the contingency matrix to the ground truth (if provided) is updated.

merge_subgraph(*subgraph=None*, *source=None*)
Merge a (typically) connected set of nodes together.

Parameters **subgraph** : agglo.Rag, networkx.Graph, or list of int (node id)

A subgraph to merge.

source : int (node id), optional

Merge the subgraph to this node.

Returns None

ncut(*num_clusters=10*, *kmeans_iters=5*, *sigma=5100.0*, *nodes=None*, ***kwargs*)
Run normalized cuts on the current set of superpixels. Keyword arguments:

num_clusters – number of clusters to compute kmeans_iters – # iterations to run kmeans when clustering sigma – sigma value when setting up weight matrix

Return value: None

non_traversing_bodies()
List bodies that are not orphans and do not traverse the volume.

orphans()
List all the nodes that do not touch the volume boundary.

Parameters **None**

Returns **orphans** : list of int (node id)

A list of node ids.

Notes

"Orphans" are not biologically plausible in EM data, so we can flag them with this function for further scrutiny.

raveler_body_annotations(*traverse=False*)
Return JSON-compatible dict formatted for Raveler annotations.

real_edges(**args*, ***kwargs*)
Return edges internal to the volume.

The RAG actually includes edges to a "virtual" region that envelops the entire volume. This function returns the list of edges that are internal to the volume.

Parameters ***args, **kwargs** : arbitrary types

Arguments and keyword arguments are passed through to the `edges()` function of the `networkx.Graph` class.

Returns **edge_list** : list of tuples

A list of pairs of node IDs, which are typically integers.

See also:

real_edges_iter, `networkx.Graph.edges`

real_edges_iter(**args*, ***kwargs*)
Return iterator of edges internal to the volume.

The RAG actually includes edges to a "virtual" region that envelops the entire volume. This function returns the list of edges that are internal to the volume.

Parameters ***args, **kwargs** : arbitrary types

Arguments and keyword arguments are passed through to the `edges()` function of the `networkx.Graph` class.

Returns **edges_iter** : iterator of tuples

An iterator over pairs of node IDs, which are typically integers.

rebuild_merge_queue()
Build a merge queue from scratch and assign to self.merge_queue.

See also:

build_merge_queue

remove_inclusions()
Merge any segments fully contained within other segments.

In 3D EM images, inclusions are not biologically plausible, so this function can be used to remove them.

Parameters None

Returns None

remove_obvious_inclusions()
Merge any nodes with only one edge to their neighbors.

rename_node(*old*, *new*)
Rename node *old* to *new*, updating edges and weights.

Parameters **old** : int

The node being renamed.

new : int

The new node id.

replay_merge_history(*merge_seq*, *labels=None*, *num_errors=1*)
Agglomerate according to a merge sequence, optionally labeled.

Parameters **merge_seq** : iterable of pair of int

The sequence of node IDs to be merged.

labels : iterable of int in {-1, 0, 1}, optional

A sequence matching *merge_seq* specifying whether a merge should take place or not. -1 or 0 mean "should merge", 1 otherwise.

Returns **n** : int

Number of elements consumed from *merge_seq*

e : (int, int)

Last merge pair observed.

Notes

The merge sequence and labels *must* be generators if you don't want to manually keep track of how much has been consumed. The merging continues until *num_errors* false merges have been encountered, or until the sequence is fully consumed.

set_exclusions(*excl*)
Set an exclusion volume, forbidding certain merges.

Parameters **excl** : array of int

Exclusions work as follows: the volume *excl* is the same shape as the initial segmentation (see `set_watershed`), and consists of mostly 0s. Any voxels with *the same* non-zero label will not be allowed to merge during agglomeration (provided they were not merged in the initial segmentation).

This allows manual separation *a priori* of difficult-to- -segment regions.

Returns None

set_feature_manager(*feature_manager*)
Set the feature manager and ensure feature caches are computed.

Parameters **feature_manager** : `features.base.Null` object

The feature manager to be used by this RAG.

Returns None

set_ground_truth(*gt=None*)
Set the ground truth volume.

This is useful for tracking segmentation accuracy over time.

Parameters **gt** : array of int

A ground truth segmentation of the same volume passed to `set_watershed`.

Returns None

set_orientations(*orientation_map*, *channel_is_oriented*)
Set the orientation map of the probability image.

Parameters **orientation_map** : array of float

A map of angles of the same shape as the superpixel map.

channel_is_oriented : 1D array-like of bool

A vector having length the number of channels in the probability map.

Returns None

set_probabilities(*probs=array([], dtype=float64)*, *normalize=False*)
Set the *probabilities* attributes of the RAG.

For various reasons, including removing the need for bounds checking when looking for neighboring pixels, the volume of pixel-level probabilities is padded on all faces. In addition, this function adds an attribute *probabilities_r*, a raveled view of the padded probabilities array for quick access to individual voxels using linear indices.

Parameters **probs** : array

The input probabilities array.

normalize : bool, optional

If `True`, the values in the array are scaled to be in [0, 1].

Returns None

set_watershed(*ws=array([], dtype=int64)*, *connectivity=1*)
Set the initial segmentation volume (watershed).

The initial segmentation is called *watershed* for historical reasons only.

Parameters **ws** : array of int

The initial segmentation.

connectivity : int in {1, ..., *ws.ndim*}, optional

The pixel neighborhood.

Returns None

split_node(*u*, *n=2*, ***kwargs*)
Use normalized cuts [1] to split a node/segment.

Parameters **u** : int (node id)

Which node to split.

n : int, optional

How many segments to split it into.

Returns None

References

[R3]

traversing_bodies()
List all bodies that traverse the volume.

update_merge_queue(*u*, *v*)
Update the merge queue item for edge (u, v). Add new by default.

Parameters **u, v** : int (node id)

Edge being updated.

Returns None

write_plaza_json(*fout*, *synapsejson=None*, *offsetz=0*)
Write graph to Steve Plaza's JSON spec.

gala.agglo.**approximate_boundary_mean**(*g*, *n1*, *n2*)
Return the boundary mean as computed by a MomentsFeatureManager.

The feature manager is assumed to have been set up for g at construction.

gala.agglo.**best_possible_segmentation**(*ws*, *gt*)
Build the best possible segmentation given a superpixel map.

gala.agglo.**compute_local_rand_change**(*s1*, *s2*, *n*)
Compute change in rand if we merge disjoint sizes s1,s2 in volume n.

gala.agglo.**compute_local_vi_change**(*s1*, *s2*, *n*)
Compute change in VI if we merge disjoint sizes s1,s2 in a volume n.

gala.agglo.**compute_true_delta_rand**(*ctable*, *n1*, *n2*, *n*)
Compute change in RI obtained by merging rows n1 and n2.

This function assumes ctable is normalized to sum to 1.

gala.agglo.**conditional_countdown**(*seq*, *start=1*, *pred=<type 'bool'>*)
Count down from 'start' each time pred(elem) is true for elem in seq.

Used to know how many elements of a sequence remain that satisfy a predicate.

Parameters **seq** : iterable

Any sequence.

start : int, optional

The starting element.

pred : function, type(next(seq)) -> bool

A predicate acting on the elements of *seq*.

Examples

```
>>> seq = range(10)
>>> cc = conditional_countdown(seq, start=5, pred=lambda x: x % 2 == 1)
>>> next(cc)
5
>>> next(cc)
4
>>> next(cc)
4
>>> next(cc)
3
```

gala.agglo.**get_edge_coordinates**(*g*, *n1*, *n2*, *arbitrary=False*)
Find where in the segmentation the edge (n1, n2) is most visible.

3.2 gala.classify: Classifier tools

gala.classify.**boundary_overlap_threshold**(*boundary_idxs*, *gt*, *tol_false*, *tol_true*)
Return -1, 0 or 1 by thresholding overlaps between boundaries.

gala.classify.**concatenate_data_elements**(*alldata*)
Return one big learning set from a list of learning sets.

A learning set is a list/tuple of length 4 containing features, labels, weights, and node merge history.

gala.classify.**default_classifier_extension**(*cl*, *use_joblib=True*)
Return the default classifier file extension for the given classifier cl.

Returns: String of file extension

gala.classify.**get_classifier**(*name='random forest'*, **args*, ***kwargs*)
Return a classifier given a name.

Parameters **name** : string

The name of the classifier, e.g. 'random forest' or 'naive bayes'.

***args, **kwargs :**

Additional arguments to pass to the constructor of the classifier.

Returns **cl** : classifier

A classifier object implementing the scikit-learn interface.

Raises **NotImplementedError**

If the classifier name is not recognized.

Examples

```
>>> cl = get_classifier('random forest', n_estimators=47)
>>> isinstance(cl, RandomForestClassifier)
True
>>> cl.n_estimators
```

```
    47
    >>> from numpy.testing import assert_raises
    >>> assert_raises(NotImplementedError, get_classifier, 'perfect class')
```

gala.classify.**label_merges**(*g*, *merge_history*, *feature_map_function*, *gt*, *loss_function*)
Replay an agglomeration history and label the loss of each merge.

gala.classify.**load_classifier**(*fn*)
Load a classifier previously saved to disk, given a filename.

Supported classifier types are: - scikit-learn classifiers saved using either pickle or joblib persistence - vigra random forest classifiers saved in HDF5 format

Parameters **fn** : string

Filename in which the classifier is stored.

Returns **cl** : classifier object

cl is one of the supported classifier types; these support at least the standard scikit-learn interface of *fit()* and *predict_proba()*

gala.classify.**make_thresholded_boundary_overlap_loss**(*tol_false*, *tol_true*)
Return a merge loss function based on boundary overlaps.

gala.classify.**sample_training_data**(*features*, *labels*, *num_samples=None*)
Get a random sample from a classification training dataset.

Parameters **features: np.ndarray [M x N]**

The M (number of samples) by N (number of features) feature matrix.

labels: np.ndarray [M] or [M x 1]

The training label for each feature vector.

num_samples: int, optional

The size of the training sample to draw. Return full dataset if *None* or if num_samples >= M.

Returns feat: np.ndarray [num_samples x N]

The sampled feature vectors.

lab: np.ndarray [num_samples] or [num_samples x 1]

The sampled training labels

gala.classify.**save_classifier**(*cl*, *fn*, *use_joblib=True*, ***kwargs*)
Save a classifier to disk.

Parameters **cl** : classifier object

Pickleable object or a classify.VigraRandomForest object.

fn : string

Writeable path/filename.

use_joblib : bool, optional

Whether to prefer joblib persistence to pickle.

kwargs : keyword arguments

Keyword arguments to be passed on to either *pck.dump* or *joblib.dump*.

Returns None

Notes

For joblib persistence, *compress=3* is the default.

3.3 gala.features: Feature definitions

3.3.1 Features

All the features used in agglomeration should be put here.

3.4 gala.morpho: Morphological operations

`gala.morpho.`**`damify`**(*a*, *in_place=False*)
Add dams to a borderless segmentation.

`gala.morpho.`**`get_neighbor_idxs`**(*ar*, *idxs*, *connectivity=1*)
Return indices of neighboring voxels given array, indices, connectivity.

Parameters **ar** : ndarray

The array in which neighbors are to be found.

idxs : int or container of int

The indices for which to find neighbors.

connectivity : int in {1, 2, ..., `ar.ndim`}

The number of orthogonal steps allowed to be considered a neighbor.

Returns **neighbor_idxs** : 2D array, shape (nidxs, nneighbors)

The neighbor indices for each index passed.

Examples

```
>>> ar = np.arange(16).reshape((4, 4))
>>> ar
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> get_neighbor_idxs(ar, [5, 10], connectivity=1)
array([[ 9,  6,  1,  4],
       [14, 11,  6,  9]])
>>> get_neighbor_idxs(ar, 9, connectivity=2)
array([[13, 10,  5,  8, 14, 12,  6,  4]])
```

`gala.morpho.`**`hminima`**(*a*, *thresh*)
Suppress all minima that are shallower than thresh.

gala.morpho.**hollowed**(*ar*, *skinsize=1*)
Return a copy of ar with the center zeroed out.

'skinsize' determines how thick of a crust to leave untouched.

gala.morpho.**imhmin**(*a*, *thresh*)
Suppress all minima that are shallower than thresh.

gala.morpho.**impose_minima**(*a*, *minima*, *connectivity=1*)
Transform 'a' so that its only regional minima are those in 'minima'.

Parameters: 'a': an ndarray 'minima': a boolean array of same shape as 'a' 'connectivity': the connectivity of the structuring element used in morphological reconstruction.

Value: an ndarray of same shape as a with unmarked local minima paved over.

gala.morpho.**manual_split**(*probs*, *seg*, *body*, *seeds*, *connectivity=1*, *boundary_seeds=None*)
Manually split a body from a segmentation using seeded watershed.

Input:

- probs: the probability of boundary in the volume given.
- seg: the current segmentation.
- body: the label to be split.
- seeds: the seeds for the splitting (should be just two labels).

[-connectivity: the connectivity to use for watershed.] [-boundary_seeds: if not None, these locations become inf in probs.]

Value:

- the segmentation with the selected body split.

gala.morpho.**minimum_seeds**(*current_seeds*, *min_seed_coordinates*, *connectivity=1*)
Ensure that each point in given coordinates has its own seed.

gala.morpho.**morphological_reconstruction**(*marker*, *mask*, *connectivity=1*)
Perform morphological reconstruction of the marker into the mask.

See the Matlab image processing toolbox documentation for details: http://www.mathworks.com/help/toolbox/images/f18-16264.html

gala.morpho.**non_traversing_segments**(*a*)
Find segments that enter the volume but do not leave it elsewhere.

gala.morpho.**orphans**(*a*)
Find all the segments that do not touch the volume boundary.

This function differs from agglo.Rag.orphans() in that it does not use the graph, but rather computes orphans directly from a volume.

gala.morpho.**raveled_steps_to_neighbors**(*shape*, *connectivity=1*)
Compute the stepsize along all axes for given connectivity and shape.

Parameters **shape** : tuple of int

The shape of the array along which we are stepping.

connectivity : int in {1, 2, ..., `len(shape)`}

The number of orthogonal steps we can take to reach a "neighbor".

Returns **steps** : array of int64

The steps needed to get to neighbors from a particular raveled index.

Examples

```
>>> shape = (5, 4, 9)
>>> steps = raveled_steps_to_neighbors(shape)
>>> sorted(steps)
[-36, -9, -1, 1, 9, 36]
>>> steps2 = raveled_steps_to_neighbors(shape, 2)
>>> sorted(steps2)
[-45, -37, -36, -35, -27, -10, -9, -8, -1, 1, 8, 9, 10, 27, 35, 36, 37, 45]
```

gala.morpho.**refined_seeding**(*a*, *maximum_height=0*, *grey_close_radius=1*, *binary_open_radius=1*, *binary_close_radius=1*, *minimum_size=0*)

Perform morphological operations to get good segmentation seeds.

gala.morpho.**regional_minima**(*a*, *connectivity=1*)

Find the regional minima in an ndarray.

gala.morpho.**relabel_connected**(*im*, *connectivity=1*)

Ensure all labels in *im* are connected.

Parameters **im** : array of int

The input label image.

connectivity : int in {1, ..., *im.ndim*}, optional

The connectivity used to determine if two voxels are neighbors.

Returns **im_out** : array of int

The relabeled image.

Examples

```
>>> image = np.array([[1, 1, 2],
...                   [2, 1, 1]])
>>> im_out = relabel_connected(image)
>>> im_out
array([[1, 1, 2],
       [3, 1, 1]])
```

gala.morpho.**remove_merged_boundaries**(*labels*, *connectivity=1*)

Remove boundaries in a label field when they separate the same region.

By convention, the boundary label is 0, and labels are positive.

Parameters **labels** : array of int

The label field to be processed.

connectivity : int in {1, ..., labels.ndim}, optional

The morphological connectivity for considering neighboring voxels.

Returns **labels_out** : array of int

The same label field, with unnecessary boundaries removed.

Examples

```
>>> labels = np.array([[1, 0, 1], [0, 1, 0], [2, 0, 3]], np.int)
>>> remove_merged_boundaries(labels)
array([[1, 1, 1],
       [0, 1, 0],
       [2, 0, 3]])
```

gala.morpho.**seg_to_bdry**(*seg*, *connectivity=1*)
Given a borderless segmentation, return the boundary map.

gala.morpho.**split_exclusions**(*image*, *labels*, *exclusions*, *dilation=0*, *connectivity=1*, *standard_seeds=False*)
Ensure that no segment in 'labels' overlaps more than one exclusion.

gala.morpho.**undam**(*seg*)
Assign zero-dams to nearest non-zero region.

gala.morpho.**watershed**(*a*, *seeds=None*, *connectivity=1*, *mask=None*, *smooth_thresh=0.0*, *smooth_seeds=False*, *minimum_seed_size=0*, *dams=False*, *override_skimage=False*, *show_progress=False*)
Perform the watershed algorithm of Vincent & Soille (1991).

Parameters **a** : np.ndarray, arbitrary shape and type

The input image on which to perform the watershed transform.

seeds : np.ndarray, int or bool type, same shape as *a* (optional)

The seeds for the watershed. If provided, these are the only basins allowed, and the algorithm proceeds by flooding from the seeds. Otherwise, every local minimum is used as a seed.

connectivity : int, {1, ..., a.ndim} (optional, default 1)

The neighborhood of each pixel, defined as in *scipy.ndimage*.

mask : np.ndarray, type bool, same shape as *a*. (optional)

If provided, perform watershed only in the parts of *a* that are set to *True* in *mask*.

smooth_thresh : float (optional, default 0.0)

Local minima that are less deep than this threshold are suppressed, using *hminima*.

smooth_seeds : bool (optional, default False)

Perform binary opening on the seeds, using the same connectivity as the watershed.

minimum_seed_size : int (optional, default 0)

Remove seed regions smaller than this size.

dams : bool (optional, default False)

Place a dam where two basins meet. Set this to True if you require 0-labeled boundaries between different regions.

override_skimage : bool (optional, default False)

skimage.morphology.watershed is used to implement the main part of the algorithm when *dams=False*. Use this flag to use the separate pure Python implementation instead.

show_progress : bool (optional, default False)

Show a cute little ASCII progress bar (using the progressbar package)

Returns **ws** : np.ndarray, same shape as *a*, int type.

The watershed transform of the input image.

gala.morpho.**watershed_sequence**(*a*, *seeds=None*, *mask=None*, *axis=0*, ***kwargs*)

Perform a watershed on a plane-by-plane basis.

See documentation for *watershed* for available kwargs.

The watershed algorithm views image intensity as "height" and finds flood basins within it. These basins are then viewed as the different labeled regions of an image.

This function performs watershed on an ndarray on each plane separately, then concatenate the results.

Parameters **a** : numpy ndarray, arbitrary type or shape.

The input image on which to perform the watershed transform.

seeds : bool/int numpy.ndarray, same shape as a (optional, default None)

The seeds for the watershed.

mask : bool numpy.ndarray, same shape as a (optional, default None)

If provided, perform watershed only over voxels that are True in the mask.

axis : int, {1, ..., a.ndim} (optional, default: 0)

Which axis defines the plane sequence. For example, if the input image is 3D and axis=1, then the output will be the watershed on a[:, 0, :], a[:, 1, :], a[:, 2, :], ... and so on.

Returns **ws** : numpy ndarray, int type

The labeled watershed basins.

Other Parameters ****kwargs** : keyword arguments passed through to the *watershed* function.

3.5 gala.evaluate: Segmentation evaluation

gala.evaluate.**adapted_rand_error**(*seg*, *gt*, *all_stats=False*)

Compute Adapted Rand error as defined by the SNEMI3D contest [1]

Formula is given as 1 - the maximal F-score of the Rand index (excluding the zero component of the original labels). Adapted from the SNEMI3D MATLAB script, hence the strange style.

Parameters **seg** : np.ndarray

the segmentation to score, where each value is the label at that point

gt [np.ndarray, same shape as seg] the groundtruth to score against, where each value is a label

all_stats [boolean, optional] whether to also return precision and recall as a 3-tuple with rand_error

Returns **are** : float

The adapted Rand error; equal to $1 -

rac{2pr}{p + r}$,

where p and r are the precision and recall described below.

prec [float, optional] The adapted Rand precision. (Only returned when *all_stats* is `True`.)

rec [float, optional] The adapted Rand recall. (Only returned when *all_stats* is `True`.)

`gala.evaluate.`**`adj_rand_index`**(*x*, *y=None*)

Return the adjusted Rand index.

The Adjusted Rand Index (ARI) is the deviation of the Rand Index from the expected value if the marginal distributions of the contingency table were independent. Its value ranges from 1 (perfectly correlated marginals) to -1 (perfectly anti-correlated).

Parameters **x, y** : np.ndarray

Either x and y are provided as equal-shaped np.ndarray label fields (int type), or y is not provided and x is a contingency table (sparse.csc_matrix) that is *not* normalised to sum to 1.

Returns **ari** : float

The adjusted Rand index of *x* and *y*.

`gala.evaluate.`**`bin_values`**(*a*, *bins=255*)

Return an array with its values discretised to the given number of bins.

Parameters **a** : np.ndarray, arbitrary shape

The input array.

bins : int, optional

The number of bins in which to put the data. default: 255.

Returns **b** : np.ndarray, same shape as a

The output array, such that values in bin X are replaced by mean(X).

`gala.evaluate.`**`contingency_table`**(*seg, gt, ignore_seg=[0], ignore_gt=[0], norm=True*)

Return the contingency table for all regions in matched segmentations.

Parameters **seg** : np.ndarray, int type, arbitrary shape

A candidate segmentation.

gt : np.ndarray, int type, same shape as *seg*

The ground truth segmentation.

ignore_seg : list of int, optional

Values to ignore in *seg*. Voxels in *seg* having a value in this list will not contribute to the contingency table. (default: [0])

ignore_gt : list of int, optional

Values to ignore in *gt*. Voxels in *gt* having a value in this list will not contribute to the contingency table. (default: [0])

norm : bool, optional

Whether to normalize the table so that it sums to 1.

Returns **cont** : scipy.sparse.csc_matrix

A contingency table. *cont[i, j]* will equal the number of voxels labeled *i* in *seg* and *j* in *gt*. (Or the proportion of such voxels if *norm=True*.)

gala.evaluate.**divide_columns**(*matrix*, *row*, *in_place=False*)

Divide each column of *matrix* by the corresponding element in *row*.

The result is as follows: out[i, j] = matrix[i, j] / row[j]

Parameters **matrix** : np.ndarray, scipy.sparse.csc_matrix or csr_matrix, shape (M, N)

The input matrix.

column : a 1D np.ndarray, shape (N,)

The row dividing *matrix*.

in_place : bool (optional, default False)

Do the computation in-place.

Returns **out** : same type as *matrix*

The result of the row-wise division.

gala.evaluate.**divide_rows**(*matrix*, *column*, *in_place=False*)

Divide each row of *matrix* by the corresponding element in *column*.

The result is as follows: out[i, j] = matrix[i, j] / column[i]

Parameters **matrix** : np.ndarray, scipy.sparse.csc_matrix or csr_matrix, shape (M, N)

The input matrix.

column : a 1D np.ndarray, shape (M,)

The column dividing *matrix*.

in_place : bool (optional, default False)

Do the computation in-place.

Returns **out** : same type as *matrix*

The result of the row-wise division.

gala.evaluate.**edit_distance**(*aseg*, *gt*, *size_threshold=1000*, *sp=None*)

Find the number of splits and merges needed to convert *aseg* to *gt*.

Parameters **aseg** : np.ndarray, int type, arbitrary shape

The candidate automatic segmentation being evaluated.

gt : np.ndarray, int type, same shape as *aseg*

The ground truth segmentation.

size_threshold : int or float, optional

Ignore splits or merges smaller than this number of voxels.

sp : np.ndarray, int type, same shape as *aseg*, optional

A superpixel map. If provided, compute the edit distance to the best possible agglomeration of *sp* to *gt*, rather than to *gt* itself.

Returns **(false_merges, false_splits)** : float

The number of splits and merges needed to convert aseg to gt.

gala.evaluate.**fm_index**(*x*, *y=None*)
Return the Fowlkes-Mallows index. [1]

Parameters **x, y** : np.ndarray

Either x and y are provided as equal-shaped np.ndarray label fields (int type), or y is not provided and x is a contingency table (sparse.csc_matrix) that is *not* normalised to sum to 1.

Returns **fm** : float

The FM index of *x* and *y*. 1 is perfect agreement.

References

[1] EB Fowlkes & CL Mallows. (1983) A method for comparing two hierarchical clusterings. J Am Stat Assoc 78: 553

gala.evaluate.**get_stratified_sample**(*ar*, *n*)
Get a regularly-spaced sample of the unique values of an array.

Parameters **ar** : np.ndarray, arbitrary shape and type

The input array.

n : int

The desired sample size.

Returns **u** : np.ndarray, shape approximately (n,)

Notes

If *len(np.unique(ar)) <= 2*n*, all the values of *ar* are returned. The requested sample size is taken as an approximate lower bound.

Examples

```
>>> ar = np.array([[0, 4, 1, 3],
...                [4, 1, 3, 5],
...                [3, 5, 2, 1]])
>>> np.unique(ar)
array([0, 1, 2, 3, 4, 5])
>>> get_stratified_sample(ar, 3)
array([0, 2, 4])
```

gala.evaluate.**make_synaptic_functions**(*fn*, *fcts*)
Make evaluation functions that only evaluate at synaptic sites.

Parameters **fn** : string

Filename containing synapse coordinates, in Raveler format. [1]

fcts : function, or iterable of functions

Functions to be converted to synaptic evaluation.

Returns **syn_fcts** : function or iterable of functions

Evaluation functions that will evaluate only at synaptic sites.

Raises **ImportError** : if the *syngeo* package [2, 3] is not installed.

References

[1] https://wiki.janelia.org/wiki/display/flyem/synapse+annotation+file+format [2] https://github.com/janelia-flyem/synapse-geometry [3] https://github.com/jni/synapse-geometry

gala.evaluate.**make_synaptic_vi**(*fn*)
Shortcut for *make_synaptic_functions(fn, split_vi)*.

gala.evaluate.**pixel_wise_boundary_precision_recall**(*pred*, *gt*)
Evaluate voxel prediction accuracy against a ground truth.

Parameters **pred** : np.ndarray of int or bool, arbitrary shape

The voxel-wise discrete prediction. 1 for boundary, 0 for non-boundary.

gt : np.ndarray of int or bool, same shape as *pred*

The ground truth boundary voxels. 1 for boundary, 0 for non-boundary.

Returns **pr** : float

rec : float

The precision and recall values associated with the prediction.

Notes

Precision is defined as "True Positives / Total Positive Calls", and Recall is defined as "True Positives / Total Positives in Ground Truth".

This function only calculates this value for discretized predictions, i.e. it does not work with continuous prediction confidence values.

gala.evaluate.**rand_by_threshold**(*ucm*, *gt*, *npoints=None*)
Compute Rand and Adjusted Rand indices for each threshold of a UCM

Parameters **ucm** : np.ndarray, arbitrary shape

An Ultrametric Contour Map of region boundaries having specific values. Higher values indicate higher boundary probabilities.

gt : np.ndarray, int type, same shape as ucm

The ground truth segmentation.

npoints : int, optional

If provided, only compute values at npoints thresholds, rather than all thresholds. Useful when ucm has an extremely large number of unique values.

Returns **ris** : np.ndarray of float, shape (3, len(np.unique(ucm))) or (3, npoints)

The rand indices of the segmentation induced by thresholding and labeling *ucm* at different values. The 3 rows of *ris* are the values used for thresholding, the corresponding Rand Index at that threshold, and the corresponding Adjusted Rand Index at that threshold.

gala.evaluate.**rand_index**(*x*, *y=None*)
Return the unadjusted Rand index. [1]

Parameters **x, y** : np.ndarray

Either x and y are provided as equal-shaped np.ndarray label fields (int type), or y is not provided and x is a contingency table (sparse.csc_matrix) that is *not* normalised to sum to 1.

Returns **ri** : float

The Rand index of *x* and *y*.

References

[1] WM Rand. (1971) Objective criteria for the evaluation of clustering methods. J Am Stat Assoc. 66: 846–850

gala.evaluate.**rand_values**(*cont_table*)

Calculate values for Rand Index and related values, e.g. Adjusted Rand.

Parameters **cont_table** : scipy.sparse.csc_matrix

A contingency table of the two segmentations.

Returns **a, b, c, d** : float

The values necessary for computing Rand Index and related values. [1, 2]

References

[1] Rand, W. M. (1971). Objective criteria for the evaluation of clustering methods. J Am Stat Assoc. [2] http://en.wikipedia.org/wiki/Rand_index#Definition on 2013-05-16.

gala.evaluate.**raw_edit_distance**(*aseg*, *gt*, *size_threshold=1000*)

Compute the edit distance between two segmentations.

Parameters **aseg** : np.ndarray, int type, arbitrary shape

The candidate automatic segmentation.

gt : np.ndarray, int type, same shape as *aseg*

The ground truth segmentation.

size_threshold : int or float, optional

Ignore splits or merges smaller than this number of voxels.

Returns **(false_merges, false_splits)** : float

The number of splits and merges required to convert aseg to gt.

gala.evaluate.**reduce_vi**(*fn_pattern='testing/%i/flat-single-channel-tr%i-%i-%.2f.lzf.h5', iterable=[(0, 1, 0), (0, 2, 0), (0, 3, 0), (0, 4, 0), (0, 5, 0), (0, 6, 0), (0, 7, 0), (1, 0, 1), (1, 2, 1), (1, 3, 1), (1, 4, 1), (1, 5, 1), (1, 6, 1), (1, 7, 1), (2, 0, 2), (2, 1, 2), (2, 3, 2), (2, 4, 2), (2, 5, 2), (2, 6, 2), (2, 7, 2), (3, 0, 3), (3, 1, 3), (3, 2, 3), (3, 4, 3), (3, 5, 3), (3, 6, 3), (3, 7, 3), (4, 0, 4), (4, 1, 4), (4, 2, 4), (4, 3, 4), (4, 5, 4), (4, 6, 4), (4, 7, 4), (5, 0, 5), (5, 1, 5), (5, 2, 5), (5, 3, 5), (5, 4, 5), (5, 6, 5), (5, 7, 5), (6, 0, 6), (6, 1, 6), (6, 2, 6), (6, 3, 6), (6, 4, 6), (6, 5, 6), (6, 7, 6), (7, 0, 7), (7, 1, 7), (7, 2, 7), (7, 3, 7), (7, 4, 7), (7, 5, 7), (7, 6, 7)], thresholds=array([0. , 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1 , 0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19, 0.2 , 0.21, 0.22, 0.23, 0.24, 0.25, 0.26, 0.27, 0.28, 0.29, 0.3 , 0.31, 0.32, 0.33, 0.34, 0.35, 0.36, 0.37, 0.38, 0.39, 0.4 , 0.41, 0.42, 0.43, 0.44, 0.45, 0.46, 0.47, 0.48, 0.49, 0.5 , 0.51, 0.52, 0.53, 0.54, 0.55, 0.56, 0.57, 0.58, 0.59, 0.6 , 0.61, 0.62, 0.63, 0.64, 0.65, 0.66, 0.67, 0.68, 0.69, 0.7 , 0.71, 0.72, 0.73, 0.74, 0.75, 0.76, 0.77, 0.78, 0.79, 0.8 , 0.81, 0.82, 0.83, 0.84, 0.85, 0.86, 0.87, 0.88, 0.89, 0.9 , 0.91, 0.92, 0.93, 0.94, 0.95, 0.96, 0.97, 0.98, 0.99, 1.])*)

Compile evaluation results embedded in many .h5 files under "vi".

Parameters **fn_pattern** : string, optional

A format string defining the files to be examined.

iterable : iterable of tuples, optional

The (partial) tuples to apply to the format string to obtain individual files.

thresholds : iterable of float, optional

The final tuple elements to apply to the format string. The final tuples are the product of *iterable* and *thresholds*.

Returns **vi** : np.ndarray of float, shape (3, len(thresholds))

The under and over segmentation components of VI at each threshold. *vi[0, :]* is the threshold, *vi[1, :]* the undersegmentation and *vi[2, :]* is the oversegmentation.

gala.evaluate.**relabel_from_one**(*label_field*)

Convert labels in an arbitrary label field to {1, ... number_of_labels}.

This function also returns the forward map (mapping the original labels to the reduced labels) and the inverse map (mapping the reduced labels back to the original ones).

Parameters **label_field** : numpy ndarray (integer type)

Returns **relabeled** : numpy array of same shape as ar

forward_map : 1d numpy array of length np.unique(ar) + 1

inverse_map : 1d numpy array of length len(np.unique(ar))

The length is len(np.unique(ar)) + 1 if 0 is not in np.unique(ar)

Examples

```
>>> import numpy as np
>>> label_field = np.array([1, 1, 5, 5, 8, 99, 42])
>>> relab, fw, inv = relabel_from_one(label_field)
>>> relab
array([1, 1, 2, 2, 3, 5, 4])
>>> fw
array([0, 1, 0, 0, 0, 2, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
           0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0,
           0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
           0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
           0, 0, 0, 0, 0, 0, 0, 5])
    >>> inv
    array([ 0,  1,  5,  8, 42, 99])
    >>> (fw[label_field] == relab).all()
    True
    >>> (inv[relab] == label_field).all()
    True
```

gala.evaluate.**sem**(*ar*, *axis=None*)

Calculate the standard error of the mean (SEM) along an axis.

Parameters **ar** : np.ndarray

The input array of values.

axis : int, optional

Calculate SEM along the given axis. If omitted, calculate along the raveled array.

Returns **sem** : float or np.ndarray of float

The SEM over the whole array (if *axis=None*) or over the chosen axis.

gala.evaluate.**sorted_vi_components**(*s1, s2, ignore1=[0], ignore2=[0], compress=False*)

Return lists of the most entropic segments in s1|s2 and s2|s1.

Parameters **s1, s2** : np.ndarray of int

Segmentations to be compared. Usually, *s1* will be a candidate segmentation and *s2* will be the ground truth or target segmentation.

ignore1, ignore2 : list of int, optional

Labels in these lists are ignored in computing the VI. 0-labels are ignored by default; pass empty lists to use all labels.

compress : bool, optional

The 'compress' flag performs a remapping of the labels before doing the VI computation, resulting in memory savings when many labels are not used in the volume. (For example, if you have just two labels, 1 and 1,000,000, 'compress=False' will give a vector of length 1,000,000, whereas with 'compress=True' it will have just size 2.)

Returns **ii1** : np.ndarray of int

The labels in *s1* having the most entropy. If *s1* is the automatic segmentation, these are the worst false merges.

h2g1 : np.ndarray of float

The conditional entropy corresponding to the labels in *ii1*.

ii2 : np.ndarray of int

The labels in *s1* having the most entropy. These correspond to the worst false splits.

h2g1 : np.ndarray of float

The conditional entropy corresponding to the labels in *ii2*.

gala.evaluate.**sparse_csr_row_max**(*csr_mat*)

Compute maximum over each row of a CSR format sparse matrix.

Parameters **csr_mat** : scipy.sparse.csr_matrix

The input matrix.

Returns **mx** : np.ndarray of shape *(mat.shape[0],)*

The maximum along every row.

gala.evaluate.**sparse_max**(*mat*, *axis=None*)

Compute the maximum value in a sparse matrix (optionally over an axis).

This function mimics the numpy.max() API for sparse.CSC or CSR matrices.

Parameters **mat** : a scipy.sparse csc or csr matrix

The matrix for which to compute the max.

axis : int in {0, 1}, optional

Compute the maximum over each column (*axis=0*) or over each row (*axis=1*). By default, compute over entire matrix.

Returns **mx** : mat.dtype (if *axis=None*) or np.ndarray of shape (mat.shape[1-axis],)

The maximum value in the array or along an axis.

gala.evaluate.**sparse_min**(*mat*, *axis=None*)

Compute the minimum value in a sparse matrix (optionally over an axis).

This function mimics the numpy.min() API for sparse.CSC or CSR matrices.

Parameters **mat** : a scipy.sparse csc or csr matrix

The matrix for which to compute the min.

axis : int in {0, 1}, optional

Compute the minimum over each column (*axis=0*) or over each row (*axis=1*). By default, compute over entire matrix.

Returns **mn** : mat.dtype (if *axis=None*) or np.ndarray of shape (mat.shape[1-axis],)

The minimum value in the array or along an axis.

gala.evaluate.**special_points_evaluate**(*eval_fct*, *coords*, *flatten=True*, *coord_format=True*)

Return an evaluation function to only evaluate at special coordinates.

Parameters **eval_fct** : function taking at least two np.ndarray of equal shapes as args

The function to be used for evaluation.

coords : np.ndarray of int, shape (n_points, n_dim) or (n_points,)

The coordinates at which to evaluate the function. The coordinates can either be subscript format (one index into each dimension of input arrays) or index format (a single index into the linear array). For the latter, use *flatten=False*.

flatten : bool, optional

Whether to flatten the coordinates (default) or leave them untouched (if they are already in raveled format).

coord_format : bool, optional

Format the coordinates to a tuple of np.ndarray as numpy expects. Set to False if coordinates are already in this format or flattened.

Returns **special_eval_fct** : function taking at least two np.ndarray of equal shapes

The returned function is the same as the above function but only evaluated at the coordinates specified. This can be used, for example, to subsample a volume, or to evaluate only whether synapses are correctly assigned, rather than every voxel, in a neuronal image volume.

gala.evaluate.**split_components**(*idx*, *cont*, *num_elems=4*, *axis=0*)

Return the indices of the bodies most overlapping with body idx.

Parameters **idx** : int

The segment index being examined.

cont : sparse.csc_matrix

The normalized contingency table.

num_elems : int, optional

The number of overlapping bodies desired.

axis : int, optional

The axis along which to perform the calculations. Assuming *cont* has the automatic segmentation as the rows and the gold standard as the columns, *axis=0* will return the segment IDs in the gold standard of the worst merges comprising *idx*, while *axis=1* will return the segment IDs in the automatic segmentation of the worst splits comprising *idx*.

Value:

comps : list of (int, float, float) tuples

num_elems indices of the biggest overlaps comprising *idx*, along with the percent of *idx* that they comprise and the percent of themselves that overlaps with *idx*.

gala.evaluate.**split_vi**(*x, y=None, ignore_x=[0], ignore_y=[0]*)

Return the symmetric conditional entropies associated with the VI.

The variation of information is defined as VI(X,Y) = H(X|Y) + H(Y|X). If Y is the ground-truth segmentation, then H(Y|X) can be interpreted as the amount of under-segmentation of Y and H(X|Y) is then the amount of over-segmentation. In other words, a perfect over-segmentation will have H(Y|X)=0 and a perfect under-segmentation will have H(X|Y)=0.

If y is None, x is assumed to be a contingency table.

Parameters **x** : np.ndarray

Label field (int type) or contingency table (float). *x* is interpreted as a contingency table (summing to 1.0) if and only if *y* is not provided.

y : np.ndarray of int, same shape as x, optional

A label field to compare to *x*.

ignore_x, ignore_y : list of int, optional

Any points having a label in this list are ignored in the evaluation. Ignore 0-labeled points by default.

Returns **sv** : np.ndarray of float, shape (2,)

The conditional entropies of Y|X and X|Y.

See also:

vi

gala.evaluate.**split_vi_threshold**(*tup*)
Compute VI with tuple input (to support multiprocessing).

Parameters **tup** : a tuple, (np.ndarray, np.ndarray, [int], [int], float)

The tuple should consist of::

- the UCM for the candidate segmentation,
- the gold standard,
- list of ignored labels in the segmentation,
- list of ignored labels in the gold standard,
- threshold to use for the UCM.

Returns **sv** : np.ndarray of float, shape (2,)

The undersegmentation and oversegmentation of the comparison between applying a threshold and connected components labeling of the first array, and the second array.

gala.evaluate.**vi**(*x, y=None, weights=array([1., 1.]), ignore_x=[0], ignore_y=[0]*)
Return the variation of information metric. [1]

VI(X, Y) = H(X | Y) + H(Y | X), where H(.|.) denotes the conditional entropy.

Parameters **x** : np.ndarray

Label field (int type) or contingency table (float). *x* is interpreted as a contingency table (summing to 1.0) if and only if *y* is not provided.

y : np.ndarray of int, same shape as x, optional

A label field to compare to *x*.

weights : np.ndarray of float, shape (2,), optional

The weights of the conditional entropies of *x* and *y*. Equal weights are the default.

ignore_x, ignore_y : list of int, optional

Any points having a label in this list are ignored in the evaluation. Ignore 0-labeled points by default.

Returns **v** : float

The variation of information between *x* and *y*.

References

[1] Meila, M. (2007). Comparing clusterings - an information based distance. Journal of Multivariate Analysis 98, 873-895.

gala.evaluate.**vi_by_threshold**(*ucm, gt, ignore_seg=[], ignore_gt=[], npoints=None, nprocessors=None*)
Compute the VI at every threshold of the provided UCM.

Parameters **ucm** : np.ndarray of float, arbitrary shape

The Ultrametric Contour Map, where each 0.0-region is separated by a boundary. Higher values of the boundary indicate more confidence in its presence.

gt : np.ndarray of int, same shape as *ucm*

The ground truth segmentation.

ignore_seg : list of int, optional

The labels to ignore in the segmentation of the UCM.

ignore_gt : list of int, optional

The labels to ignore in the ground truth.

npoints : int, optional

The number of thresholds to sample. By default, all thresholds are sampled.

nprocessors : int, optional

Number of processors to use for the parallel evaluation of different thresholds.

Returns **result** : np.ndarray of float, shape (3, npoints)

The evaluation of segmentation at each threshold. The rows of this array are:

- the threshold used
- the undersegmentation component of VI
- the oversegmentation component of VI

gala.evaluate.**vi_pairwise_matrix**(*segs*, *split=False*)

Compute the pairwise VI distances within a set of segmentations.

If 'split' is set to True, two matrices are returned, one for each direction of the conditional entropy.

0-labeled pixels are ignored.

Parameters **segs** : iterable of np.ndarray of int

A list or iterable of segmentations. All arrays must have the same shape.

split : bool, optional

Should the split VI be returned, or just the VI itself (default)?

Returns **vi_sq** : np.ndarray of float, shape (len(segs), len(segs))

The distances between segmentations. If *split==False*, this is a symmetric square matrix of distances. Otherwise, the lower triangle of the output matrix is the false split distance, while the upper triangle is the false merge distance.

gala.evaluate.**vi_statistics**(*vi_table*)

Descriptive statistics from a block of related VI evaluations.

Parameters **vi_table** : np.ndarray of float

An array containing VI evaluations of various samples. The last axis represents the samples.

Returns **means, sems, medians** : np.ndarrays of float

The statistics of the given array along the samples axis.

gala.evaluate.**vi_tables**(*x, y=None, ignore_x=[0], ignore_y=[0]*)

Return probability tables used for calculating VI.

If y is None, x is assumed to be a contingency table.

Parameters **x, y** : np.ndarray

Either x and y are provided as equal-shaped np.ndarray label fields (int type), or y is not provided and x is a contingency table (sparse.csc_matrix) that may or may not sum to 1.

ignore_x, ignore_y : list of int, optional

Rows and columns (respectively) to ignore in the contingency table. These are labels that are not counted when evaluating VI.

Returns **pxy** : sparse.csc_matrix of float

The normalized contingency table.

px, py, hxgy, hygx, lpygx, lpxgy : np.ndarray of float

The proportions of each label in *x* and *y* (*px*, *py*), the per-segment conditional entropies of *x* given *y* and vice-versa, the per-segment conditional probability p log p.

gala.evaluate.**wiggle_room_precision_recall**(*pred*, *boundary*, *margin=2*, *connectivity=1*)

Voxel-wise, continuous value precision recall curve allowing drift.

Voxel-wise precision recall evaluates predictions against a ground truth. Wiggle-room precision recall (WRPR, "warper") allows calls from nearby voxels to be counted as correct. Specifically, if a voxel is predicted to be a boundary within a dilation distance of *margin* (distance defined according to *connectivity*) of a true boundary voxel, it will be counted as a True Positive in the Precision, and vice-versa for the Recall.

Parameters **pred** : np.ndarray of float, arbitrary shape

The prediction values, expressed as probability of observing a boundary (i.e. a voxel with label 1).

boundary : np.ndarray of int, same shape as pred

The true boundary map. 1 indicates boundary, 0 indicates non-boundary.

margin : int, optional

The number of dilations that define the margin. default: 2.

connectivity : {1, ..., pred.ndim}, optional

The morphological voxel connectivity (defined as in SciPy) for the dilation step.

Returns **ts, pred, rec** : np.ndarray of float, shape *(len(np.unique(pred)+1),)*

The prediction value thresholds corresponding to each precision and recall value, the precision values, and the recall values.

gala.evaluate.**xlogx**(*x*, *out=None*, *in_place=False*)

Compute x * log_2(x).

We define 0 * log_2(0) = 0

Parameters **x** : np.ndarray or scipy.sparse.csc_matrix or csr_matrix

The input array.

out : same type as x (optional)

If provided, use this array/matrix for the result.

in_place : bool (optional, default False)

Operate directly on x.

Returns **y** : same type as x

Result of x * log_2(x).

3.6 gala.imio: Image IO

gala.imio.**apply_segmentation_map**(*superpixels*, *sp_to_body_map*)
Return a segmentation from superpixels and a superpixel to body map.

Parameters **superpixels** : numpy ndarray, arbitrary shape, int type

A superpixel (or supervoxel) map (aka label field).

sp_to_body_map : numpy ndarray, shape (NUM_SUPERPIXELS, 2), int type

An array of (superpixel, body) map pairs.

Returns **segmentation** : numpy ndarray, same shape as 'superpixels', int type

The segmentation induced by the superpixels and map.

gala.imio.**compute_sp_to_body_map**(*sps*, *bodies*)
Return unique (sp, body) pairs from a superpixel map and segmentation.

Parameters **sps** : numpy ndarray, arbitrary shape

The superpixel (supervoxel) map.

bodies : numpy ndarray, same shape as sps

The corresponding segmentation.

Returns **sp_to_body** : numpy ndarray, shape (NUM_SPS, 2)

Notes

No checks are made for sane inputs. This means that incorrect input, such as non-matching shapes, or superpixels mapping to more than one segment, will result in undefined behavior downstream with no warning.

gala.imio.**extract_segments**(*seg*, *ids*)
Get a uint8 volume containing only the specified segment ids.

Parameters **seg** : array of int

The input segmentation.

ids : list of int, maximum length 255

A list of segments to extract from *seg*.

Returns **segs** : array of uint8

A volume with 1, 2, ..., `len(ids)` labels where the required segments were, and 0 elsewhere.

Notes

This function is designed to output volumes to VTK format for viewing in ITK-SNAP

Examples

```
    >>> segments = array([[45, 45, 51, 51],
    ...                   [45, 83, 83, 51]])
    >>> extract_segments(segments, [83, 45])
    array([[2, 2, 0, 0],
           [2, 1, 1, 0]], dtype=uint8)
```

gala.imio.**pil_to_numpy**(*img*)

Convert an Image object to a numpy array.

Parameters **img** : Image object (from the Python Imaging Library)

Returns **ar** : numpy ndarray

The corresponding numpy array (same shape as the image)

gala.imio.**raveler_body_annotations**(*orphans*, *non_traversing=None*)

Return a Raveler body annotation dictionary of orphan segments.

Orphans are labeled as body annotations with *not sure* status and a string indicating *orphan* in the comments field.

Non-traversing segments have only one contact with the surface of the volume, and are labeled *does not traverse* in the comments.

Parameters **orphans** : iterable of int

The ID numbers corresponding to orphan segments.

non_traversing : iterable of int (optional, default None)

The ID numbers of segments having only one exit point in the volume.

Returns **body_annotations** : dict

A dictionary containing entries for 'data' and 'metadata' as specified in the Raveler body annotations format [1, 2].

References

[1] https://wiki.janelia.org/wiki/display/flyem/body+annotation+file+format and: [2] https://wiki.janelia.org/wiki/display/flyem/generic+file+format

gala.imio.**raveler_output_shortcut**(*svs*, *seg*, *gray*, *outdir*, *sps_out=None*)

Compute the Raveler format and write to directory, all at once.

Parameters **svs** : np.ndarray, int, shape (M, N, P)

The supervoxel map.

seg : np.ndarray, int, shape (M, N, P)

The segmentation map. It is assumed that no supervoxel crosses any segment boundary.

gray : np.ndarray, uint8, shape (M, N, P)

The grayscale EM images corresponding to the above segmentations.

outdir : string

The export directory for the Raveler volume.

sps_out : np.ndarray, int, shape (M, N, P) (optional)

The precomputed serial section 2D superpixel map. Output will be much faster if this is provided.

Returns **sps_out** : np.ndarray, int, shape (M, N, P)

The computed serial section 2D superpixel map. Keep this when making multiple calls to *raveler_output_shortcut* with the same supervoxel map.

gala.imio.**raveler_rgba_to_int**(*im*, *ignore_alpha=True*)

Convert a volume using Raveler's RGBA encoding to int. [1]

Parameters **im** : np.ndarray, shape (M, N, P, 4)

The image stack to be converted.

ignore_alpha : bool, optional

By default, the alpha channel does not encode anything. However, if we ever need 32 bits, it would be used. This function supports that with *ignore_alpha=False*. (default is True.)

Returns **im_int** : np.ndarray, shape (M, N, P)

The label volume.

References

[1] https://wiki.janelia.org/wiki/display/flyem/Proofreading+data+and+formats

gala.imio.**raveler_serial_section_map**(*nd_map*, *min_size=0*, *do_conn_comp=False*, *globally_unique_ids=True*)

Produce *serial_section_map* and label one corner of each plane as 0.

Raveler chokes when there are no pixels with label 0 on a plane, so this function produces the serial section map as normal but then adds a 0 to the [0, 0] corner of each plane, IF the volume doesn't already have 0 pixels.

Notes

See *serial_section_map* for more info.

gala.imio.**raveler_to_labeled_volume**(*rav_export_dir*, *get_glia=False*, *use_watershed=False*, *probability_map=None*, *crop=None*)

Import a raveler export stack into a labeled segmented volume.

Parameters **rav_export_dir** : string

The directory containing the Raveler stack.

get_glia : bool (optional, default False)

Return the segment numbers corresponding to glia, if available.

use_watershed : bool (optional, default False)

Fill in 0-labeled voxels using watershed.

probability_map : np.ndarray, same shape as volume to be read (optional)

If *use_watershed* is True, use *probability_map* as the landscape. If this is not provided, it uses a flat landscape.

crop : tuple of int (optional, default None)

A 6-tuple of [xmin, xmax, ymin, ymax, zmin, zmax].

Returns **output_volume** : np.ndarray, shape (Z, X, Y)

The segmentation in the Raveler volume.

glia : list of int (optional, only returned if *get_glia* is True)

The IDs in the segmentation corresponding to glial cells.

gala.imio.**read_h5_stack**(*fn, group='stack', crop=[None, None, None, None, None, None], **kwargs*)

Read a volume in HDF5 format into numpy.ndarray.

Parameters **fn** : string

The filename of the input HDF5 file.

group : string, optional (default 'stack')

The group within the HDF5 file containing the dataset.

crop : list of int, optional (default '[None]*6', no crop)

A crop to get of the volume of interest. Only available for 2D and 3D volumes.

Returns **stack** : numpy ndarray

The stack contained in fn, possibly cropped.

gala.imio.**read_image_stack**(*fn*, **args*, ***kwargs*)

Read a 3D volume of images in image or .h5 format into a numpy.ndarray.

This function attempts to automatically determine input file types and wraps specific image-reading functions.

Parameters **fn** : filename (string)

A file path or glob pattern specifying one or more valid image files. The file format is automatically determined from this argument.

***args** : filenames (string, optional)

More than one positional argument will be interpreted as a list of filenames pointing to all the 2D images in the stack.

****kwargs** : keyword arguments (optional)

Arguments to be passed to the underlying functions. A 'crop' keyword argument is supported, as a list of length 6: [xmin, xmax, ymin, ymax, zmin, zmax]. Use 'None' for no crop in that coordinate.

Returns **stack** : 3-dimensional numpy ndarray

Notes

If reading in .h5 format, keyword arguments are passed through to read_h5_stack().

Automatic file type detection may be deprecated in the future.

gala.imio.**read_mapped_segmentation**(*fn*, *sp_group='stack'*, *sp_to_body_group='transforms'*)

Read a volume in mapped HDF5 format into a numpy.ndarray pair.

Parameters **fn** : string

The filename to open.

sp_group : string, optional (default 'stack')

The group within the HDF5 file where the superpixel map is stored.

sp_to_body_group : string, optional (default 'transforms')

The group within the HDF5 file where the superpixel to body map is stored.

Returns **segmentation** : numpy ndarray, same shape as 'superpixels', int type

The segmentation induced by the superpixels and map.

gala.imio.**read_mapped_segmentation_raw**(*fn*, *sp_group='stack'*, *sp_to_body_group='transforms'*)

Read a volume in mapped HDF5 format into a numpy.ndarray pair.

Parameters **fn** : string

The filename to open.

sp_group : string, optional (default 'stack')

The group within the HDF5 file where the superpixel map is stored.

sp_to_body_group : string, optional (default 'transforms')

The group within the HDF5 file where the superpixel to body map is stored.

Returns **sp_map** : numpy ndarray, arbitrary shape

The superpixel (or supervoxel) map.

sp_to_body_map : numpy ndarray, shape (NUM_SUPERPIXELS, 2)

The superpixel to body (segment) map, as (superpixel, body) pairs.

gala.imio.**read_multi_page_tif**(*fn, crop=[None, None, None, None, None, None]*)

Read a multi-page tif file into a numpy array.

Parameters **fn** : string

The filename of the image file being read.

Returns **ar** : numpy ndarray

The image stack in array format.

Notes

Currently, only grayscale images are supported.

gala.imio.**read_multi_page_tif_libtiff**(*fn*)

Read a multi-page tif file into a numpy array.

Parameters **fn** : string

The filename of the image file being read.

Returns **ar** : numpy ndarray

The image stack in array format.

Notes

Currently, only grayscale images are supported.

gala.imio.**read_prediction_from_ilastik_batch**(*fn*, ***kwargs*)

Read the prediction produced by Ilastik from batch processing.

Parameters **fn** : string

The filename to read from.

group : string (optional, default '/volume/prediction')

Where to read from in the HDF5 file hierarchy.

single_channel : bool (optional, default True)

Read only the 0th channel (final dimension) from the volume.

Returns None

gala.imio.**read_vtk**(*fin*)

Read a numpy volume from a VTK structured points file.

Code adapted from Erik Vidholm's readVTK.m Matlab implementation.

Parameters **fin** : string

The input filename.

Returns **ar** : numpy ndarray

The array contained in the file.

gala.imio.**segs_to_raveler**(*sps*, *bodies*, *min_size=0*, *do_conn_comp=False*, *sps_out=None*)

Return a Raveler tuple from 3D superpixel and body maps.

Parameters **sps** : numpy ndarray, shape (M, N, P)

The supervoxel map.

bodies : numpy ndarray, shape (M, N, P)

The body map. Superpixels should not map to more than one body.

min_size : int, optional (default: 0)

Superpixels smaller than this size on a particular plane are blacked out.

do_conn_comp : bool (default: False)

Whether to do a connected components operation on each plane. This is required if we want superpixels to be contiguous on each plane, since 3D-contiguous superpixels are not guaranteed to be contiguous along a slice.

sps_out : numpy ndarray, shape (M, N, P), optional (default: None)

A Raveler-compatible superpixel map, meaning that superpixels are unique to each plane along axis 0. (See *superpixels* in the return values.) If provided, this saves significant computation time.

Returns **superpixels** : numpy ndarray, shape (M, N, P)

The superpixel map. Non-zero superpixels are unique to each plane. That is, *np.unique(superpixels[i])* and *np.unique(superpixels[j])* have only 0 as their intersection.

sp_to_segment : numpy ndarray, shape (Q, 3)

The superpixel to segment map. Segments are unique to each plane. The first number on each line is the plane number.

segment_to_body : numpy ndarray, shape (R, 2)

The segment to body map.

gala.imio.**serial_section_map**(*nd_map*, *min_size=0*, *do_conn_comp=False*, *globally_unique_ids=True*)

Produce a plane-by-plane superpixel map with unique IDs.

Raveler requires sps to be unique and different on each plane. This function converts a fully 3D superpixel map to a serial-2D superpixel map compatible with Raveler.

Parameters **nd_map** : np.ndarray, int, shape (M, N, P)

The original superpixel map.

min_size : int (optional, default 0)

Remove superpixels smaller than this size (on each plane)

do_conn_comp : bool (optional, default False)

In some cases, a single supervoxel may result in two disconnected superpixels in 2D. Set to True to force these to have different IDs.

globally_unique_ids : bool (optional, default True)

If True, every plane has unique IDs, with plane n having IDs {i1, i2, ..., in} and plane n+1 having IDs {in+1, in+2, ..., in+ip}, and so on.

Returns **relabeled_planes** : np.ndarray, int, shape (M, N, P)

A volume equal to nd_map but with superpixels relabeled along axis 0. That is, the input volume is reinterpreted as M slices of shape (N, P).

gala.imio.**ucm_to_raveler**(*ucm*, *sp_threshold=0.0*, *body_threshold=0.1*, ***kwargs*)

Return Raveler map from a UCM.

Parameters **ucm** : numpy ndarray, shape (M, N, P)

An ultrametric contour map. This is a map of scored segment boundaries such that if A, B, and C are segments, then score(A, B) = score(B, C) >= score(A, C), for some permutation of A, B, and C. A hierarchical agglomeration process produces a UCM.

sp_threshold : float, optional (default: 0.0)

The value for which to threshold the UCM to obtain the superpixels.

body_threshold : float, optional (default: 0.1)

The value for which to threshold the UCM to obtain the segments/bodies. The condition *body_threshold* >= *sp_threshold* should hold in order to obtain sensible results.

****kwargs** : dict, optional

Keyword arguments to be passed through to *segs_to_raveler*.

Returns **superpixels** : numpy ndarray, shape (M, N, P)

The superpixel map. Non-zero superpixels are unique to each plane. That is, *np.unique(superpixels[i])* and *np.unique(superpixels[j])* have only 0 as their intersection.

sp_to_segment : numpy ndarray, shape (Q, 3)

The superpixel to segment map. Segments are unique to each plane. The first number on each line is the plane number.

segment_to_body : numpy ndarray, shape (R, 2)

The segment to body map.

gala.imio.**write_h5_stack**(*npy_vol*, *fn*, *group='stack'*, *compression=None*, *chunks=None*, *shuffle=None*)

Write a numpy.ndarray 3D volume to an HDF5 file.

Parameters **npy_vol** : numpy ndarray

The array to be saved to HDF5.

fn : string

The output filename.

group : string, optional (default: 'stack')

The group within the HDF5 file to write to.

compression : {None, 'gzip', 'szip', 'lzf'}, optional (default: None)

The compression to use, if any. Note that 'lzf' is only available through h5py, so implementations in other languages will not be able to read files created with this compression.

chunks : tuple, True, or None (default: None)

Whether to use chunking in the HDF5 dataset. Default is None. True lets h5py choose a chunk size automatically. Otherwise, use a tuple of int of the same length as *npy_vol.ndim*. From the h5py documentation: "In the real world, chunks of size 10kB - 300kB work best, especially for compression. Very small chunks lead to lots of overhead in the file, while very large chunks can result in inefficient I/O."

shuffle : bool, optional

Shuffle the bytes on disk to improve compression efficiency.

Returns None

gala.imio.**write_ilastik_batch_volume**(*im*, *fn*)

Write a volume to an HDF5 file for Ilastik batch processing.

Parameters **im** : np.ndarray, shape (M, N[, P])

The image volume to be saved.

fn : string

The filename in which to save the volume.

Returns None

gala.imio.**write_ilastik_project**(*images*, *labels*, *fn*, *label_names=None*)

Write one or more image volumes and corresponding labels to Ilastik.

Parameters **images** : np.ndarray or list of np.ndarray, shapes (M_i, N_i[, P_i])

The grayscale images to be saved.

labels : np.ndarray or list of np.ndarray, same shapes as *images*

The label maps corresponding to the images.

fn : string

The filename to save the project in.

label_names : list of string (optional)

The names corresponding to each label in *labels*. (Not implemented!)

Returns None

Notes

Limitations: Assumes the same labels are used for all images. Supports only grayscale images and volumes, and a maximum of 8 labels. Requires at least one unlabeled voxel in the label field.

gala.imio.**write_image_stack**(*npy_vol*, *fn*, ***kwargs*)

Write a numpy.ndarray 3D volume to a stack of images or an HDF5 file.

Parameters **npy_vol** : numpy ndarray

The volume to be written to disk.

fn : string

The filename to be written, or a format string when writing a 3D stack to a 2D format (e.g. a png image stack).

****kwargs** : keyword arguments

Keyword arguments to be passed to wrapped functions. See corresponding docs for valid arguments.

Returns **out** : None

Examples

```
>>> import numpy as np
>>> from gala.imio import write_image_stack
>>> im = 255 * np.array([
... [[0, 1, 0], [1, 0, 1], [0, 1, 0]],
... [[1, 0, 1], [0, 1, 0], [1, 0, 1]]], dtype=uint8)
>>> im.shape
(2, 3, 3)
>>> write_image_stack(im, 'image-example-%02i.png', axis=0)
>>> import os
>>> fns = sorted(filter(lambda x: x.endswith('.png'), os.listdir('.')))
>>> fns # two 3x3 images
['image-example-00.png', 'image-example-01.png']
>>> os.remove(fns[0]); os.remove(fns[1]) # doctest cleanup
```

gala.imio.**write_json**(*annot*, *fn='annotations-body.json'*, *directory=None*)

Write an annotation dictionary in Raveler format to a JSON file.

The annotation file format is described in: https://wiki.janelia.org/wiki/display/flyem/body+annotation+file+format and: https://wiki.janelia.org/wiki/display/flyem/generic+file+format

Parameters **annot** : dict

A body annotations dictionary (described in pages above).

fn : string (optional, default 'annotations-body.json')

The filename to which to write the file.

directory : string (optional, default None, or '.')

A directory in which to write the file.

Returns None

gala.imio.**write_mapped_segmentation**(*superpixel_map*, *sp_to_body_map*, *fn*, *sp_group='stack'*, *sp_to_body_group='transforms'*)

Write a mapped segmentation to an HDF5 file.

Parameters **superpixel_map** : numpy ndarray, arbitrary shape

sp_to_body_map : numpy ndarray, shape (NUM_SPS, 2)

A many-to-one map of superpixels to bodies (segments), specified as rows of (superpixel, body) pairs.

fn : string

The output filename.

sp_group : string, optional (default 'stack')

the group within the HDF5 file to store the superpixel map.

sp_to_body_group : string, optional (default 'transforms')

the group within the HDF5 file to store the superpixel to body map.

Returns None

gala.imio.**write_png_image_stack**(*npy_vol*, *fn*, *axis=-1*, *bitdepth=None*)

Write a numpy.ndarray 3D volume to a stack of .png images.

Parameters **npy_vol** : numpy ndarray, shape (M, N, P)

The volume to be written to disk.

fn : format string

The file pattern to which to write the volume.

axis : int, optional (default = -1)

The axis along which output the images. If the input array has shape (M, N, P), and axis is 1, the function will write N images of shape (M, P) to disk. In keeping with Python convention, -1 specifies the last axis.

Returns **None** : None

No value is returned.

Notes

Only 8-bit and 16-bit single-channel images are currently supported.

gala.imio.**write_to_raveler**(*sps*, *sp_to_segment*, *segment_to_body*, *directory*, *gray=None*, *raveler_dir='/usr/local/raveler-hdf'*, *nproc_contours=16*, *body_annot=None*)

Output a segmentation to Raveler format.

Parameters **sps** : np.ndarray, int, shape (nplanes, nx, ny)

The superpixel map. Superpixels can only occur on one plane.

sp_to_segment : np.ndarray, int, shape (nsps + nplanes, 3)

Superpixel-to-segment map as a 3 column list of (plane number, superpixel id, segment id). Segments must be unique to a plane, and each plane must contain the map {0: 0}

segment_to_body: np.ndarray, int, shape (nsegments, 2)

The segment to body map.

directory: string

The directory in which to write the stack. This directory and all necessary subdirectories will be created.

gray: np.ndarray, uint8 or uint16, shape (nplanes, nx, ny) (optional)

The grayscale images corresponding to the superpixel maps.

raveler dir: string (optional, default '/usr/local/raveler-hdf')

Where Raveler is installed.

nproc_contours: int (optional, default 16)

How many processes to use when generating the Raveler contours.

body_annot: dict or np.ndarray (optional)

Either a dictionary to write to JSON in Raveler body annotation format, or a numpy ndarray of the segmentation from which to compute orphans and non traversing bodies (which then get written out as body annotations).

Returns None

Notes

Raveler is the EM segmentation proofreading tool developed in-house at Janelia for the FlyEM project.

`gala.imio.`**`write_vtk`**(*ar, fn, spacing=[1.0, 1.0, 1.0]*)

Write 3D volume to VTK structured points format file.

Code adapted from Erik Vidholm's writeVTK.m Matlab implementation.

Parameters **ar** : a numpy array, shape (M, N, P)

The array to be written to disk.

fn : string

The desired output filename.

spacing : iterable of float, optional (default: [1.0, 1.0, 1.0])

The voxel spacing in x, y, and z.

Returns **None** : None

This function does not have a return value.

3.7 gala.viz: Visualization tools

`gala.viz.`**`add_nats_to_plot`**(*ars*, *tss*, *stops=0.5*, *colors='k'*, *markers='o'*, ***kwargs*)

In an existing active split-vi plot, add the natural stopping point.

By default, a circle marker is used.

Parameters **ars** : list of numpy arrays

Each array has shape (2, N) and represents a split-VI curve, with *ars[i][0]* holding the undersegmentation and *ars[i][1]* holding the oversegmentation for each *i*.

tss : list of numpy arrays

Each array has shape (N,) and represents the algorithm threshold that gave rise to the VI measurements in *ars*.

stops : float, optional

The natural stopping point for the algorithm. For example, if an algorithm merges segments according to a merge probability, the natural stopping point is at $p=0.5$, when there are even odds of the merge being a true merge.

colors : string, list of string, or list of float tuple, optional

A color specification or list of color specifications. If there are fewer colors than split-VI arrays, the colors are cycled.

markers : string, or list of string, optional

Point marker specification (as defined in matplotlib) or list thereof. As with colors, if there are fewer markers than VI arrays, the markers are cycled.

****kwargs** : dict (string keys), optional

Keyword arguments to be passed through to *matplotlib.pyplot.scatter*.

Returns **points** : list of *matplotlib.collections.PathCollection*

The points returned by each of the calls to *scatter*.

gala.viz.**add_opts_to_plot**(*ars*, *colors='k'*, *markers='^'*, ***kwargs*)

In an existing active split-vi plot, add the point of optimal VI.

By default, a star marker is used.

Parameters **ars** : list of numpy arrays

Each array has shape (2, N) and represents a split-VI curve, with *ars[i][0]* holding the undersegmentation and *ars[i][1]* holding the oversegmentation for each *i*.

colors : string, list of string, or list of float tuple, optional

A color specification or list of color specifications. If there are fewer colors than split-VI arrays, the colors are cycled.

markers : string, or list of string, optional

Point marker specification (as defined in matplotlib) or list thereof. As with colors, if there are fewer markers than VI arrays, the markers are cycled.

****kwargs** : dict (string keys), optional

Keyword arguments to be passed through to *matplotlib.pyplot.scatter*.

Returns **points** : list of *matplotlib.collections.PathCollection*

The points returned by each of the calls to *scatter*.

gala.viz.**display_3d_segmentations**(*segs*, *image=None*, *probability_map=None*, *axis=0*, *z=None*, *fignum=None*)

Show slices of multiple 3D segmentations.

Parameters **segs** : list or tuple of np.ndarray of int, shape (M, N, P)

The segmentations to be examined.

image : np.ndarray, shape (M, N, P[, 3]), optional

The image corresponding to the segmentations.

probability_map : np.ndarray, shape (M, N, P), optional

The segment boundary probability map.

axis : int in {0, 1, 2}, optional

The axis along which to show a slice of the segmentation.

z : int in [0, *(M, N, P)[axis]*), optional

The slice to display. Defaults to the middle slice.

fignum : int, optional

Which figure number to use. Uses the default (new figure) if none is provided.

Returns **fig** : plt.Figure

The figure handle.

gala.viz.**draw_seg**(*seg*, *im*)

Return a segmentation map matching the original image color.

Parameters **seg** : np.ndarray of int, shape (M, N, ...)

The segmentation to be displayed

im : np.ndarray, shape (M, N, ..., C)

The image corresponding to the segmentation.

Returns **out** : np.ndarray, same shape and type as *im*.

An image where each segment has uniform color.

Examples

```
>>> a = np.array([[1, 1, 2, 2],
...               [1, 2, 2, 3],
...               [2, 2, 3, 3]])
>>> g = np.array([[0.5, 0.2, 1.0, 0.9],
...               [0.2, 0.8, 0.9, 0.6],
...               [0.9, 0.9, 0.4, 0.5]])
>>> draw_seg(a, g)
array([[ 0.3,  0.3,  0.9,  0.9],
       [ 0.3,  0.9,  0.9,  0.5],
       [ 0.9,  0.9,  0.5,  0.5]])
```

gala.viz.**imshow_grey**(*im*)

Show a segmentation using a gray colormap.

Parameters **im** : np.ndarray of int, shape (M, N)

The segmentation to be displayed.

Returns **fig** : plt.Figure

The image shown.

gala.viz.**imshow_jet**(*im*)

Show a segmentation using a jet colormap.

Parameters **im** : np.ndarray of int, shape (M, N)

The segmentation to be displayed.

Returns **fig** : plt.Figure

The image shown.

gala.viz.**imshow_rand**(*im*, *labrandom=True*)

Show a segmentation using a random colormap.

Parameters **im** : np.ndarray of int, shape (M, N)

The segmentation to be displayed.

labrandom : bool, optional

Use random points in the Lab colorspace instead of RGB.

Returns **fig** : plt.Figure

The image shown.

gala.viz.**plot_split_vi**(*ars*, *best=None*, *colors='k'*, *linespecs='-'*, ***kwargs*)

Make a split-VI plot.

The split-VI plot was introduced in Nunez-Iglesias et al, 2013 [1]

Parameters **ars** : array or list of arrays of float, shape (2, N)

The input VI arrays. *ars[i][0]* should contain the undersegmentation and *ars[i][1]* the oversegmentation.

best : array-like of float, len=2, optional

Agglomerative segmentations can't get to (0, 0) VI if the starting superpixels are not perfectly aligned with the gold standard segmentation. Therefore, there is a point of best achievable VI. *best* should contain the coordinates of this point.

colors : matplotlib color specification or list thereof, optional

The color of each line being plotted. If there are fewer colors than arrays, they are cycled.

linespecs : matplotlib line type spec, or list thereof, optional

The line type to plot with ('-', '–', '-.', etc).

kwargs : dict, string keys, optional

Additional keyword arguments to pass through to plt.plot.

Returns **lines** : matplotlib Lines2D object(s)

The lines plotted.

gala.viz.**plot_vi**(*g*, *history*, *gt*, *fig=None*)

Plot the VI from segmentations based on Rag and sequence of merges.

Parameters **g** : agglo.Rag object

The region adjacency graph.

history : list of tuples

The merge history of the RAG.

gt : np.ndarray

The ground truth corresponding to the RAG.

fig : plt.Figure, optional

Use this figure for plotting. If not provided, a new figure is created.

Returns None

gala.viz.**plot_vi_breakdown**(*seg*, *gt*, *ignore_seg=[]*, *ignore_gt=[]*, *hlines=None*, *subplot=False*, *figsize=None*, ***kwargs*)

Plot conditional entropy H(Y|X) vs P(X) for both seg|gt and gt|seg.

Parameters **seg** : np.ndarray of int, shape (M, [N, ..., P])

The automatic (candidate) segmentation.

gt : np.ndarray of int, shape (M, [N, ..., P]) (same as *seg*)

The gold standard/ground truth segmentation.

ignore_seg : list of int, optional

Ignore segments in this list from the automatic segmentation during evaluation and plotting.

ignore_gt : list of int, optional

Ignore segments in this list from the ground truth segmentation during evaluation and plotting.

hlines : int, optional

Plot this many isoclines between the minimum and maximum VI contributions.

subplot : bool, optional

If True, plot oversegmentation and undersegmentation in separate subplots.

figsize : tuple of float, optional

The figure width and height, in inches.

****kwargs** : dict

Additional keyword arguments for *matplotlib.pyplot.plot*.

Returns None

gala.viz.**plot_vi_breakdown_panel**(*px*, *h*, *title*, *xlab*, *ylab*, *hlines*, *scatter_size*, ***kwargs*)

Plot a single panel (over or undersegmentation) of VI breakdown plot.

Parameters **px** : np.ndarray of float, shape (N,)

The probability (size) of each segment.

h : np.ndarray of float, shape (N,)

The conditional entropy of that segment.

title, xlab, ylab : string

Parameters for *matplotlib.plt.plot*.

hlines : iterable of float

Plot hyperbolic lines of same VI contribution. For each value *v* in *hlines*, draw the line $h = v/px$.

scatter_size : int, optional

****kwargs** : dict

Additional keyword arguments for *matplotlib.pyplot.plot*.

Returns None

CHAPTER 4

Release notes

4.1 0.3

4.1.1 0.3.2

- Bug fix: missing import in `test_gala.py`. This was caused by rebasing commits from post-0.3 onto 0.3.

4.1.2 0.3.1

This is a major bug fix release addressing issue #63 on GitHub. You can read more there and in the related mailing list thread, but the gist is that the "learning mode" parameter did nothing in previous releases of gala. The gala library in fact was not implementing the algorithm described in the GALA paper, but rather, a variant of LASH with memory across epochs. (LASH only retains data from the most recent learning epoch.) It remains to be determined whether the "strict" learning mode described in our paper indeed yields improvements in segmentation accuracy.

Note that the included tests pass when using scikit-learn 0.16, but not with the recently-released 0.17, because of changes in the implementation of `GaussianNB`.

4.1.3 0.3.0

Announcing the third release of gala!

I want to thank Paul Watkins, Sean Colby, Larissa Heinrich, Joergen Kornfeld, and Jan Funke for their bug reports and mailing list discussions, which prompted almost all of the improvements in this release.

I must also thank the Saalfeld lab for financial support while I was making these improvements.

This release focuses on performance improvements, but also includes some API and behavior changes.

This is the last release of gala supporting Python 2. Upcoming work will focus on asynchronous learning to enable interactive proofreading, for which Python 3.4 and 3.5 offer compelling features and libraries. If you absolutely *need* Python 2.7 support in gala, get in touch!

On to the changes in this version!

4.1.4 Major changes:

- 2x memory reduction and 3x RAG construction speedup.

- Add support for masked volumes: use a boolean array of the same shape as the image to inspect only `True` positions.
- **API break:** The label "0" is no longer considered a boundary label; volumes with a single-voxel-thick boundary are no longer supported.
- **API break:** The Ultrametric Contour Map (UCM) is gone, because it is inaccurate without a voxel-thick boundary, and was computationally expensive to maintain.

4.1.5 Minor changes:

- Add `paper_em` and `snemi3d` default feature managers (in `gala.features.default`) to reproduce previous gala results.
- Bug fix: passing a label array of type floating point no longer causes a crash. (But you really should use integers for labels!)

4.2 0.2

4.2.1 0.2.3

Minor feature addition: enable exporting segmentation results *after* agglomeration is complete.

4.2.2 0.2.2

This maintenance release contains several bug fixes:

- package Cython source files (.pyx) for PyPI
- package the gala-segment command-line interface for PyPI
- include viridis in `requirements.txt`
- update libtiff usage

4.2.3 0.2

This release owes much of its existence to Neal Donnelly (@NealJMD on GitHub), who bravely delved into gala and reduced its memory and time footprints by over 20% each. The other highlights are Python 3 support and much better continuous integration.

4.2.4 Major changes:

- gala now uses an ultrametric tree backend to represent the merge hierarchy. This speeds up merges and will allow more sophisticated editing operations in the future.
- gala is now **fully compatible with Python 3.4**! That's a big tick in the "being a good citizen of the Python community" box. =) The downside is that a lot of the operations are slower in Py3.
- As mentioned above, gala is 20% faster and 20% smaller than before. That's thanks to extensive benchmarking and Cythonizing by @NealJMD
- We are now measuring code coverage, and although it's a bit low at 40%, the major gala functions (RAG building, learning, agglomerating) are covered. And we're only going up from here!

- We now have documentation on ReadTheDocs!

4.2.5 Minor changes:

- @anirbanchakraborty added the concepts of "frozen nodes" and "frozen edges", which are never merged. This is useful to temporarily ignore mitochondria during the first stages of agglomeration, which can dramatically reduce errors. (See A Context-aware Delayed Agglomeration Framework for EM Segmentation.)
- @anirbanchakraborty added the inclusiveness feature, a measure of how much a region is "surrounded" by another.
- The *gala.evaluate* module now supports the Adapted Rand Error, as used by the SNEMI3D challenge.
- Improvements to the *gala.morphology* module.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

[R1] Nunez-Iglesias et al, Machine learning of hierarchical clustering to segment 2D and 3D images, PLOS ONE, 2013.

[R2] Jain et al, Learning to agglomerate superpixel hierarchies, NIPS, 2011.

[R3] Shi, J., and Malik, J. (2000). Normalized cuts and image segmentation. Pattern Analysis and Machine Intelligence.

g

A

B

C

D

E

F

G

H

I

T

U

V

W

X