3d Brain Atlas Reconstructor Documentation

Release 0.1

Piotr Majka, Jakub M. Kowalski

October 05, 2011

CONTENTS

Warning: This is only a draft of documentation!

Contents:

CHAPTER

ONE

3D BRAIN ATLAS RECONSTRUCTOR README FILE

In this file you can find a brief description of the 3D Brain Atlas Reconstructor software instalation and usage. Please visit the http://www.3dbar.org website for more detailed description.

If you haven't done this already, please let us know that you are using 3D Brain Atlas Reconstructor by filling out short form available on http://service.3dbar.org/downloadForm . Thanks a lot.

1.1 Installing required packages (Ubuntu)

- Installation in Ubuntu 9.10
  1. Install the Visualization Toolkit and other graphics libraries:

     ```
     sudo apt-get install libvtk5.2 libvtk5-dev libvtk5.2-qt4 \
     libvtk5-qt4-dev tk8.5 tk8.5-dev python-vtk libgtkgl2.0-1 \
     libgtkgl2.0-dev libgtkglext1 librsvg2-2 python-nifti
     ```

  2. Install python related packages:

     ```
     sudo apt-get install python-gtkglext1 python-rsvg python-opengl \
     python-numpy python-scipy python-wxgtk2.6
     ```

  3. Other packages:

     ```
     sudo apt-get install potrace pstoedit python-setuptools python-epydoc
     ```

  4. Install Sphinx:

     ```
     sudo easy_install -U Sphinx
     ```

- Installation in Ubuntu 10.04
  1. Install the Visualization Toolkit and other graphics libraries:

     ```
     sudo apt-get install libvtk5.2 libvtk5-dev libvtk5.2-qt4 \
     libvtk5-qt4-dev tk8.5 tk8.5-dev python-vtk libgtkgl2.0-1 \
     libgtkgl2.0-dev libgtkglext1 librsvg2-2 python-nifti
     ```

  2. Install python related packages:

     ```
     sudo apt-get install python-gtkglext1 python-rsvg python-opengl \
     python-numpy python-scipy python-wxgtk2.6
     ```

3. Other packages:

```
sudo apt-get install potrace pstoedit python-setuptools python-epydoc
```

4. Install Sphinx:

```
sudo easy_install -U Sphinx
```

- Installation in Ubuntu 10.10 and Ubuntu 11.04

1. Install the Visualization Toolkit and other graphics libraries:

```
sudo apt-get install libvtk5.4 libvtk5-dev libvtk5.4-qt4 \
libvtk5-qt4-dev tk8.5 tk8.5-dev python-vtk libgtkgl2.0-1 \
libgtkgl2.0-dev libgtkglext1 librsvg2-2 python-nifti
```

2. Install python related packages:

```
sudo apt-get install python-gtkglext1 python-rsvg python-opengl \
python-numpy python-scipy python-wxgtk2.8
```

3. Other packages:

```
sudo apt-get install potrace pstoedit python-setuptools python-epydoc
```

4. Install Sphinx:

```
sudo easy_install -U Sphinx
```

1.2 Generating CAF datasets

Once the software is installed, you need to generate CAF representations of data of interest. For this you need to use parsers. We provide here the following parsers:

1. ScalableBrainAtlas DB08 template (http://scalablebrainatlas.incf.org/main/coronal3d.php?template=DB08)
2. ScalableBrainAtlas PHT00 template (http://scalablebrainatlas.incf.org/main/coronal3d.php?template=PHT00)
3. ScalableBrainAtlas WHS09 template (http://scalablebrainatlas.incf.org/main/coronal3d.php?template=WHS09)
4. ScalableBrainAtlas WHS10 template (http://scalablebrainatlas.incf.org/main/coronal3d.php?template=WHS10)
5. ScalableBrainAtlas LPBA40_on_SRI24 template (http://scalablebrainatlas.incf.org/main/coronal3d.php?template=LPBA40_on_S
6. ScalableBrainAtlas RM_on_F99 template (http://scalablebrainatlas.incf.org/main/coronal3d.php?template=RM_on_F99)
7. the Waxholm Space Atlas (the source mouse brain volumetric dataset)
8. the Waxholm Space Atlas (the source mouse brain volumetric dataset), another delineation (http://software.incf.org/software/waxholm-space/waxholm-space/LabeledAtlas0.5.1/file_download?file_field=file)
9. Paxinos and Watson "The Rat Brain in Stereotaxic Coordinates" atlas.
10. Franklin and Paxinos "The Mouse Brain in Stereotaxic Coordinates" atlas.
11. The Allen Mouse Brain Atlas (http://mouse.brain-map.org/atlas/index.html)

To generate CAF dataset for data from ScalableBrainAtlas DB08 template execute the following commands in the root directory of the software:

```
$ source setbarenv.sh
$ make sba_DB08
```

The first line sets the path to the API and uses appropriate parser to download the data from SBA and do the transformation into the CAF dataset.

You can also generate that way CAF dataset for any of following SBA templates: `PHT00`, `WHS09`, `WHS10`, `LPBA40_on_SRI24` and `RM_on_F99` just by replacing `DB08` with the name of the source template.

In order to generate that way CAF dataset for the Waxholm Space Atlas replace `sba_DB08` with `whs_0.5` or `whs_0.51` (for another WHS delineation).

To generate CAF dataset from Paxinos and Watson atlas (Paxinos, G. and Watson, C. (2007). The Rat Brain In Stereotaxic Coordinates. Elsevier, 6th edition.) you have to supply the parser with PDF file delivered with printed copy of the atlas. Execute the following command in the root directory of the software:

```
$ bash bin/parsers/paxinos_watson_rbisc/make_svg_from_pdf_rat.sh <PDF path>
```

You have to replace *<PDF path>* with a valid path to the file mantioned above. The CAF dataset will be stored in the `atlases/paxinos_watson_rbisc/caf-src` directory.

If the result of parsing does not satisfy you, you can edit slides derived from the PDF atlass manually with your favourite SVG editor.

The slides are located in `atlases/paxinos_watson_rbisc/caf-src` directory and named `N_pretrace_v1.svg` where N is the slide number. Once you have your slides edited execute in the root directory of the software:

```
$ make -f make_pw_rbisc.mk
```

to reparse the edited slides.

Similarly for Paxinos and Frnklin atlas (Paxinos, G. and Franklin, K. B. J. (2008). The Mouse Brain In Stereotaxic Coordinates. Elsevier, 3rd edition.) you have to execute:

```
$ bash bin/parsers/paxinos_franklin_mbisc/make_svg_from_pdf_mouse.sh <PDF path>
```

in the root directory of the software. The CAF dataset will be stored in the `atlases/paxinos_franklin_mbisc/caf-src` directory.

To reparse the edited slides execute:

```
$ make -f make_pf_mbisc.mk
```

in the root directory of the software.

Generation of CAF dataset for The Allen Mouse Brain Atlas requires the Advanced Normalization Tools (ANTS; http://picsl.upenn.edu/ANTS/) installed. ANTS have to be availiable as shell commands (for an example by adding ANTS `bin` directory to environment value `PATH`).

To generate CAF dataset from The Allen Mouse Brain Atlas execute:

```
$ source setbarenv.sh
$ make aba
```

in the root directory of the software.

1.3 Generating 3-D models

Once you have a CAF of any dataset you can test the GUI for structure creation. To do it, in the main directory run:

```
$ ./3dbar.sh
```

and choose in the menu Atlas/Open and select *index.xml* file of chosen CAF dataset.

To test, click the topmost label on the tree in the left panel and press *Perform reconstruction* button in the right panel. The reconstruction process will start. When it is finished, chose in the menu *Edit/Save Model*. It allows you to put it later in context by right click on the ontology tree.

1.4 Generating documentation

In order to generate documentation execute:

```
$ make doc
```

The documentation for API can be viewed by opening *doc/api/html/index.html* and the documentation for 3dBAR graphic interface can be viewed by opening *doc/gui/html/index.html*.

CHAPTER

TWO

BASICS OF CAF DATASETS

2.1 Basics of CAF slide

How to create *CAF slide*?

In order to use CAF API you need to import `bar` module:

```
>>> import bar
```

2.1.1 Creating empty CAF slide

```
>>> slide=bar.barCafSlide(slideNumber=0)
```

2.1.2 Creating delineations of brain regions

Note that, by definition, SVG paths representing brain regions are closed paths in sense of *SVG closepath* command and they should be expressed using SVG absolute coordinates. By default API believes that provided paths are valid and does not check their syntax thus following invalid path definition will be accepted:

```
>>> path=bar.barPath("structure_s1_AA","M 100,100 l 100 200 L 200 200", "#ff0000")
```

In order to force path validation of path definition set `clearPathDef` to `True`. If invalid path is provided an exception is raised.

```
>>> path=bar.barPath("structure_s1_AA","M 100,100 L 100 200 L 200 200", "#ff0000", clearPathDef=True)
Traceback (most recent call last):
ValueError: Invalid path definition provided
```

Important: Providing invalid path definitions will surely lead to errors in further steps. If you are not sure about path definition - validate it.

```
>>> path=bar.barPath("structure_s1_AA","M 100,100 L 100 200 L 200 200 Z", "#ff0000", clearPathDef=Tru
>>> print path
<path bar:growlevel="0" d="M100.0,100.0 L100.0,200.0 L200.0,200.0 Z " fill="#ff0000" id="structure_s1
```

Path id convention

Additional properties

Cusomizing paths

2.1.3 Adding labels

Automatic label generation

```
>>> slide.generateLabels()
<bar.base.barTracedSlideRenderer object at 0x...>
```

2.1.4 Adding information about spatial coordinate system

2.1.5 Putting pieces togeater

```
>>> import bar
>>> import datetime
>>> from random import gauss

>>> slideRange    = range(50)
>>> coronalCoords = map(lambda x: 50 - x - gauss(1, 0.25), slideRange)

>>> slides = []
>>> indexer = bar.barIndexer()

>>> for i in slideRange:
...     pathId  = "structure_s%02d_AA" % i
...     pathDef = "M %d,100 L %d,500 L 500,500 Z" % (100+2*i, 100+2*i)
...     path=bar.barPath(pathId, pathDef ,"#ff0000", clearPathDef=True)
...     structure = bar.barGenericStructure("AA", "#ff0000", [path])
...     slide=bar.barCafSlide(slideNumber=i)
...     slide.addStructures(structure)
...     slide.updateMetadata(\
...             [bar.barTransfMatrixMetadataElement((1.0,0,1.0,0)),
...              bar.barBregmaMetadataElement(coronalCoords[i])])
...     slide.writeXMLtoFile("%02d_traced_v0.svg" % i)
...     indexer.indexSingleSlide(slide, i)

>>> indexerProperties = {\
...      'ReferenceWidth'    : str(slide._rendererConf['imageSize'][0]),\
...      'ReferenceHeight'   : str(slide._rendererConf['imageSize'][1]),\
...      'FilenameTemplate' : '%02d_traced_v%d.svg',\
...      'RefCords'   : "0,0,1.0,1.0",\
...      'CAFName'    : "caf_test",\
...      'CAFComment': 'Exemplary CAF dataset',\
...      'CAFCreator': 'Put your fullname here',\
...      'CAFCreatorEmail': 'your.email@here.org',\
...      'CAFCompilationTime': datetime.datetime.utcnow().strftime("%F %T"),\
...      'CAFSlideUnits':'mm'}

# Set indexer properties
>>> indexer.updateProperties(indexerProperties)
>>> indexer.createFlatHierarchy()
```

```
>>> indexer.fullNameMapping = {}
>>> indexer.colorMapping = {}

>>> indexer.writeXMLtoFile('index.xml')
```

Just remove the temporary file containing test slide.

```
>>> import os
>>> os.system("rm -f *.svg  *.xml")
0
```

CHAPTER

THREE

CAF TESTS: CRISP-EDGES PROPERTY

Crisp edges property determines the rendering settings of the SVG drawing. Note: By now (and probably also by the future) the 'shape-rendering' property is run-time property only and is not stored or loaded from SVG files.

3.1 Tests

```
>>> import bar
```

Create empty slide and check its 'crispEdges' property

```
>>> testSlide = bar.barCafSlide()
>>> print testSlide.crispEdges
True
```

Let's set `crispEdges` to `False`

```
>>> testSlide.crispEdges = False
```

Now, lets validate 'crispEdges' property on the `path` object. Path element has assigned corresponding attribute defined

```
>>> testPath = bar.barPath("structure_test_test", "M 100 100 L 100 200 L 200 200 Z", "#ff0000")
>>> testPath.crispEdges = True
>>> print testPath
<path bar:growlevel="0" d="M 100 100 L 100 200 L 200 200 Z" fill="#ff0000" id="structure_test_test" s

>>> testPath.crispEdges = False
>>> print testPath
<path bar:growlevel="0" d="M 100 100 L 100 200 L 200 200 Z" fill="#ff0000" id="structure_test_test" s
```

Note that `crispEdges` property accepts only boolean values. Providing type other than boolean will cause exception.

```
>>> testPath.crispEdges = 'string is invalid'
Traceback (most recent call last):
AssertionError: Boolean value expected

>>> testSlide.crispEdges = 'string is invalid'
Traceback (most recent call last):
AssertionError: Boolean value expected
```

`crispEdgest` should also always return boolean value:

```
>>> type(testPath.crispEdges) == type(True)
True

>>> type(testSlide.crispEdges) == type(True)
True
```

CHAPTER

FOUR

CAF TESTS: `TYPE` PROPERTY (BARGENERICSTRUCTURE, BARPATH)

Attribute holding type of the feature delineated by given path. For example it can be like *gray matter*, *white matter*, *single cell*, *ventricle*, and other... This property tries to mimic INCF DAI *feature* attribute would be extended after establishing the INCF DAI common metadata set.

4.1 Tests

```
>>> import bar
```

Let's create simple path:

```
>>> path = bar.barPath("structure_test00_test", "M 100 100 L 100 200 L 200 200 Z", "#00ff00")
```

By default, path has no *type* assigned:

```
>>> print path.type
None
```

We can assingn any consistent string as feature type:

```
>>> path.type="white-matter"
```

But inconsistent string will not work:

```
>>> path.type="invalid:  hite-mat ter"
Traceback (most recent call last):
AssertionError: Invalid feature type name provided: invalid:  hite-mat ter
```

Also, any non-string value (except None) will raise an exception:

```
>>> path.type= []
Traceback (most recent call last):
AssertionError: String or 'None' value expected
>>> path.type = None
```

Similarly as for single path, you can define the type attribute for the whole structure Let's create exemplary structure and assign a type:

```
>>> structure = bar.barGenericStructure("test", "#00ff00")
```

By default, structure *type* property is also *None*:

```
>>> print structure.type
None
```

You can assign any reasonable string or *None* value:

```
>>> structure.type = "test-type"
>>> structure.type = None
```

But you cannot assign any other type than string or *None*:

```
>>> structure.type = ['dsf']
Traceback (most recent call last):
AssertionError: String or 'None' value expected

>>> structure.type = True
Traceback (most recent call last):
AssertionError: String or 'None' value expected
```

You can assign a path to a structure:

```
>>> structure.addPaths(path)
```

After that, their types match. Type from structure is copied to the all paths belonging to this structure.

```
>>> path.type, structure.type
(None, None)
>>> path.type == structure.type
True
```

After altering type of the structure, the type of its path is also altered:

```
>>> structure.type="test-type"

>>> path.type, structure.type
('test-type', 'test-type')
>>> path.type == structure.type
True
```

But it doesn't work at the opposite direction:

```
>>> path.type = "other-type"

>>> path.type, structure.type
('other-type', 'test-type')
>>> path.type == structure.type
False
```

Thus after adding given path to the structure it is recommended to define type of the path indirectly - thought the structure. *Type* property is stored with CAF slide and can be loaded from XML.

CHAPTER

FIVE

CAF API TESTS: SIZE AND BITMAPSIZE PROPERTY

Obviously, we need to import bar module

```
>>> import bar
```

Let's create an exemplary CAF slide with default settings

```
>>> slide = bar.barCafSlide()
```

We can read the dimensions of the SVG drawing representing CAF slide

```
>>> slide.size
(1200, 900)
```

Size is provided as tuple of two Intergers

```
>>> map(type, slide.size) == map(type, (1,1))
True
```

Also the rendering size of the slide (which is not the same property as dimensions of the CAF slide) can be set

```
>>> slide.bitmapSize
(1200, 900)
>>> map(type, slide.bitmapSize) == map(type, (1,1))
True
```

Size, as well as bitmapsize can be altered. Only tuple of two integers is accepted as new value Altering slide dimensions:

```
>>> slide.size = (1000, 500)
>>> slide.size
(1000, 500)

>>> slide.size = ("1000", 500)
Traceback (most recent call last):
AssertionError: (int,int) Tuple of two integers required

>>> slide.bitmapSize = 'invalid slide dimensions'
Traceback (most recent call last):
AssertionError: (int,int) Tuple of two integers required
```

Size and *bitmap size* are related with some internal properies that can be edited by advanded user to better customize the slides. Changing the slide size alters the slide template as well as tracing and rendering properties

```
>>> slide._rendererConf
{'ReferenceHeight': 500, 'ReferenceWidth': 1000, 'imageSize': (1200, 900)}
>>> slide._tracingConf
```

Changing the slide size alters the slide template as well as tracing and rendering properties

```
>>> slide.size = (1000,500)
>>> slide.bitmapSize = (2000, 1000)
>>> slide._rendererConf
{'ReferenceHeight': 500, 'ReferenceWidth': 1000, 'imageSize': (2000, 1000)}
```

Slide with customized size can be obviously saved to and loaded from XML file:

```
>>> from xml.dom import minidom as dom
>>> slide.writeXMLtoFile('testSlide.svg')
>>> testSlide = bar.barCafSlide.fromXML('testSlide.svg')
>>> testSlide._rendererConf
{'ReferenceHeight': 500, 'ReferenceWidth': 1000, 'imageSize': (2000, 1000)}
```

All slide size settings are preserved:

```
>>> testSlide.size
(1000, 500)
>>> slide.bitmapSize
(2000, 1000)
```

Just remove the temporary file containing test slide.

```
>>> import os
>>> os.remove('testSlide.svg')
```

CHAPTER

SIX

CAF API TESTS: BARMARKER CLASS AND ITS SUBCLASSES

Locating slide in spatial reference system (SRS)

The last thing to make the CAF slide complete is to locate it in a spatial coordinate system. CAF supports orthogonal, 3-D coordinate systems which seem to cover the majority of SRS used in brain atlases (e.g. Stereotactic coordinate system, the Waxholm Space, Talairach Space). In a CAF dataset infotmation about used SRS and its units of measurements is stored in the CAF index file. Wherever it is possible, link to the INCF DAI SRS is provided offering more precise description of the given SRS.

Each slide contains information about its own location in th SRS. This information is stored within the CAF slide. This information if expressed using two sets of numbers:

- Intra-plane transofrmation allpowith the software to express location of exery point of the slide using SRS coordinates
- stack coordinate (coordinate perpendicular to the slide plane).

E.g. the first set consists of 4 numbers (a, b, c, d) while the second contains only one number (z). The formula to convert SVG coordinates into SRS coordinates is following:

$$\begin{cases} x' &= ax + b \\ y' &= cy + d \\ z' &= z \end{cases}$$

Where (x,y,z) are SVG coordinates while (x', y', z') are related SRS coordinates.

In the opposite direction:

$$\begin{cases} x &= (x' - b)/a \\ y &= (y' - d)/c \\ z &= z' \end{cases}$$

Each CAF slide can carry different SVG to SRS coordinates transformation. However, if the purpose of the given dataset is to serve as as source to 3-D model generation this variability is limited and all slides has to carry the same parameters.

Embedding SRS information into CAF slide:

Mentioned coefficients can be embedded into CAF slide in two ways:

1. By explicitly defining transformation,
2. By calculating transformation from provided data with the help of `markers`

In order to calculate SRS <=> SVG transformation one have to establish three markers. Two in-plane markers and single intra-plane marker.

Obviously, we need to import bar module

```
>>> import bar
>>> m1 = bar.barCoordinateMarker((0,0),(10,10))
>>> m2 = bar.barCoordinateMarker((5,5),(500,500))
>>> bm = bar.barCoronalMarker(6.0, (800, 800))
```

Let's display the XML/SVG representation of the markers:

```
>>> print m1
<text fill="#000000" font-family="Helvetica,sans-serif" font-size="18px" stroke="none" x="10" y="10">
>>> print m2
<text fill="#000000" font-family="Helvetica,sans-serif" font-size="18px" stroke="none" x="500" y="500
>>> print bm
<text fill="#000000" font-family="Helvetica,sans-serif" font-size="18px" stroke="none" x="800" y="800
```

Create empty CAF slide and locate it in SRS using created markers. By default, the slide does not contain any infotmation abou its location is spatial coordinate system.

```
>>> slide = bar.barCafSlide()
>>> slide.metadata
{}
```

SVG to SRS transformation can be calucated and put into the slide using methods available in CAF API:

```
>>> bar.base.processMarkers(m1,m2,bm)
(<bar.base.barTransfMatrixMetadataElement object at 0x...>, <bar.base.barBregmaMetadataElement object
```

Add indormation about SRS to the slide:

```
>>> slide.updateMetadata(bar.base.processMarkers(m1,m2,bm))
```

From this moment one can express location on the slide using both, SVG and SRS coordinates. To convert between these coordinates use `svg2srs((svgx,svgy), ndims = 2)` method. Where (svgx,svgy) are SVG coordinate system to be converted into spatial coordinates. While `ndims` is number of dimensions of returned value (determines if output value will be 2 or 3 dimensional; when `ndims == 3` inter-plane coordinate is also included):

```
>>> slide.svg2srs((0,0))
(-0.1020408163265306, -0.1020408163265306)
>>> slide.svg2srs((30,20), ndims=3)
(0.20408163265306117, 0.1020408163265306, 6.0)
```

Composition of `svg2srs` and `srs2svg` should result in initial coordinate.

```
>>> slide.srs2svg(slide.svg2srs((30,20)))
(29.999999999999996, 20.0)
>>> slide.svg2srs(slide.srs2svg((-1,2)))
(-0.99999999999999989, 2.0)
```

Full functionality is preserved after loading and saving the slide:

```
>>> slide.writeXMLtoFile('test_markers.svg')
>>> testSlide = bar.barCafSlide.fromXML('test_markers.svg')
>>> testSlide.srs2svg(testSlide.svg2srs((30,20)))
(30.0, 20.0)
>>> testSlide.svg2srs(testSlide.srs2svg((-1,2)))
(-0.99999999999999989, 2.0)
```

Just remove all the temporary files.

```
>>> import os
>>> os.remove('test_markers.svg')
```

CHAPTER

SEVEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*