# **Human Brain Project Active Repository Use Cases**

**Human Brain Project Active Repository Use Cases** 1

Introduction 2

Use Case I – Remote interactive multiresolution visualization of large volumetric datasets 2

Use Case II – Feature extraction and analysis of large volumetric datasets 3

Appendix I – BBIC Service Description 5

Appendix II – BBIC File Format 13

## Introduction

The aim of the Human Brain Project (HBP) is to accelerate our understanding of the human brain by integrating global neuroscience knowledge and data into supercomputer-based models and simulations. This will be achieved, in part, by engaging the European and global research communities using six collaborative ICT platforms: Neuroinformatics, Brain Simulation, High Performance Computing, Medical Informatics, High Performance Computing, Neuromorphic Computing and Neurorobotics.

In the United States, the **B**rain **R**esearch through **A**dvancing **I**nnovative **N**eurotechnologies (BRAIN) Initiative aims to accelerate the development of new technologies to create large-scale measurements of the structure and function of the brain. The aim is to enable researchers to acquire, analyze and disseminate massive amounts of data about the dynamics nature of the brain from cells to circuits and the whole brain.

For the HBP Neuroinformatics Platform, a key capability is to deliver multi-level brain atlases that enable the analysis and integration of many different types of data into common semantic and spatial coordinate frameworks. Because the data to be integrated is large and widely distributed an infrastructure that enables “in place” visualization and analysis with data services co-located with data storage is requisite. Providing a standard set of services for such large data sets will enhance data sharing and collaboration in neuroscience initiatives around the world.

## Use Case I – Remote interactive multiresolution visualization of large volumetric datasets

Large amounts of image stacks or volumetric data are produced daily at brain research sites around the world. This includes human brain imaging data in clinics, connectome data in research studies, whole brain imaging with light-sheet microscopy and tissue clearing methods or micro-optical sectioning techniques, two-photon imaging, array tomography, and electron beam microscopy.

A key challenge in make such data available is to make it accessible without moving large amounts of data. Typical dataset sizes can reach in the terabyte range, while a researcher may want to only view or access a small subset of the entire dataset.

***An active repository*** The ability to easily deploy an active repository that combines large data storage with a set of computational services for accessing and viewing large volume datasets would address a key challenge present in modern neuroscience and across other domains.

This use case would enable a neuroscientist user to deploy their data in a specified repository where it would be accessible for web-based viewing and annotation.

This use case would require:

1. A multi-terabyte storage capacity. Each image will typically range from 1-10TB.
2. A compute node with fast IO bandwidth to storage device (to be specified shortly)
3. The ability to deploy a Python-based service (BBIC, see appendix I) and supporting libraries (HDF5, etc).
4. High performance internet connectivity for web service
5. A standardized authentication/authorization/identity mechanism (first version could provide public access, current version uses HBP AAI).
6. Web client code for interactively viewing dataset via BBIC service (provided by HBP)
7. Modern web client (Chrome/Safari/Firefox) for interactive 2D/3D viewing using WebGL and/or OpenLayers.
8. Sample neuroscience-based volumetric datasets including electron microscopy, light microscopy, two-photon imaging, light sheet microscopy, etc ranging from subcellular to whole brain – provided by HBP, OpenConnectome, Allen Institute and others.

## Use Case II – Feature extraction and analysis of large volumetric datasets

Neurons are essential building blocks of the brain and key to its information processing ability. The three-dimensional shape of a neuron plays a major role in determining its connectivity, integration of synaptic input and cellular firing properties. Thus, characterization of the 3D morphology of neurons is fundamentally important in neuroscience and related applications. Digitization of the morphology of neurons and other tree-shape biological structures (e.g. glial cells, brain vasculatures) has been studied in the last 30 years. Recent big neuroscience initiatives worldwide, e.g. USA’s BRAIN initiative and Europe’s Human Brain Project, highlight the importance to understand the types of cells in nervous systems. Current reconstruction techniques (both manual and automated) show tremendous variability in the quality and completeness of the resulting morphology. Yet, building a large library of high quality 3D cell morphologies is essential to comprehensively cataloging the types of cells in a nervous system. Furthermore, enabling comparisons of neuron morphologies across species will provide additional sources of insight into neural function.

Automated reconstruction of neuron morphology has been studied by many research groups. Methods including fitting tubes or other geometrical elements, ray casting, spanning tree, shortest paths, deformable curves, pruning, etc., have been proposed. Commercial software packages such as Neurolucida also start to include some of the automated neuron reconstruction methods. The DIADEM Challenge ( <http://diademchallenge.org/> ), a worldwide neuron reconstruction contest, was organized in 2010 by several major institutions as a way to stimulate progress and attract new computational researchers to join the technology development community.

A new effort, called BigNeuron (<http://www.alleninstitute.org/bigneuron>) aims to bring the latest automated neuron morphology reconstruction algorithms to bear on large image stacks from around the world.

The second use case would entail deploying Vaa3D ([www.vaa3d.org](http://www.vaa3d.org)) as an additional service to the active repository described in Use Case I. Vaa3D is open source and provides a plugin architecture into which any type of neuron reconstruction algorithm can be adapted. The second use case would require additional computational resources (and could benefit from multithreaded and parallel compute resources) for the reconstruction process.

In this use case, a neuroscientist user would provide via a web service input parameters to a Vaa3D instance which would trace any recognized neuron structures using a selected algorithm. The output file would be returned via the webservice.

This use case would require:

1. The active repository developed for Use Case I.
2. The additional deployment of Vaa3D adapted for use with BBIC. A beta version of this is currently available. The REST API may need development.
3. A multiprocessor compute node with high speed access to the storage device.
4. Additional datasets including image stacks/volumes of clearly labeled single or multiple neurons – provided by HBP, Allen Institute and others.

# Appendix I – BBIC Service Description

**BBIC Web Services Documentation**

Author: Stanislaw Adaszewski

Blue Brain Project Image Container

**Table of Contents**

1. Glossary
2. Foreword
3. Web Services
   1. BBIC URI format
   2. Meta Information Query
   3. Stack Service
   4. Volume Service
   5. Image Manipulation Service
4. Closing Remarks
5. FAQ

**Glossary**

* BBIC - Blue Brain Image Container
* HCL - [Hyperdimensional Compression Library](http://github.com/sadaszewski/hcl)
* L1 - Level 1 Structure
* L2 - Level 2 Structure
* L3 - Level 3 Structure
* LOD - Level Of Detail
* REST - Representational State Transfer
* SEC2J / H5FD\_sec2j - [SEC2 journaled](http://github.com/sadaszewski/H5FD_sec2j)

**Foreword**

BBIC Web Services is a collection of REST services for providing imaging and meta-information extracted from the BBIC files. At the moment it's implemented in Python using [Tornado](http://www.tornadoweb.org/) web server. An alternative version using Python's minimalist [SimpleHTTPServer](https://docs.python.org/2/library/simplehttpserver.html) is also available.

Noteworthy, the Tornado implementation is single-threaded, while the SimpleHTTPServer one is multi-threaded. Depending on a particular service this can translate into significant performance benefits.

In the following document, the APIs for accessing image stack and volume data are explained.

In many places, references to *bbic\_format.md* are made, which explains the structure of the underlying BBIC file format. In particular *L1*, *L2* and *L3* are such references, referring to respectively Level 1, 2 and 3 structure of the file format.

**Web Services**

**BBIC URI format**

BBIC shares the semantics of regular URIs using the fragment part of URI to specify the path within BBIC container. A couple of examples:

* file:///home/adaszews/Downloads/BigBrain.h5#/volumes/0
* http://127.0.0.1/download/1234-5678#/volumes/0

This notation is used by various scripts creating BBIC datasets out of other BBIC datasets to fill in the *original\_filenames* field and therefore creating a provenance hierarchy.

Please note that this URI format refers to accessing BBIC files directly/locally. Further along this chapter, REST web services are described, for retrieving meta-information and data from the BBIC files across the Web using HTTP queries.

The syntax for referencing BBIC files within those services is slightly different. In particular a query parameter is used instead of URI fragment to specify the stack/volume.

**Meta information query**

/bbic?mode=meta&fname=<fname>

/bbic\_vol?mode=meta&fname=<fname>

This request returns meta information about a given BBIC file. The response is in JSON format. The top-level entity is a dictionary with keys named *attributes*, *stacks* and *volumes*. Values for the latter two are arrays containing meta-information about consecutive stacks/volumes stored in the file. Value for the first one contains a dump of attributes from the */bbic* group.

An overview of the reply:

{'attributes': { ... }, 'stacks': [ .... ], 'volumes': [ .... ]}

Attributes are dumped as simple strings or numbers with an exception of *local\_to\_world* matrix which is dumped as an array of 16 numbers in C (column-major) order.

An example of file attributes dump:

{'version': 1, 'num\_stacks': 1, 'num\_volumes': 1}

**Stacks**

The *stacks* array contains *num\_stacks* dictionaries with the following keys: *attributes*, *levels* and *projections*. The value for first one is a dump of attributes from */bbic/stacks/n*, where *n* is the stack index. For the second one the value contains an array of dictionaries describing consecutive levels of detail for a given stack. The last one contains dictionary specifying types of projection acceleration data available for the given stack.

Example of *stacks* array element:

{'attributes': { ... }, 'levels': [ ... ], 'projections': [ ... ]}

Example of stack attributes dump:

{'width': 512, 'height': 512, 'num\_slices': 256, 'tile\_size': 256, 'local\_to\_world': [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1], ...}

*m*-th level dictionary contains a dump of attributes from */bbic/stacks/n/levels/m*.

Example of stack level attributes dump:

{'num\_x\_tiles': 2, 'num\_y\_tiles': 2, 'num\_slices': 256}

Projections dictionary contains one key per projection type. Currently these are either 'min', 'max' or both. The corresponding value is an array of numbers specifying numbers of slices used for calculating intermediaty projection acceleration tiles. For a file containing *min* and *max* projection data with numbers of slices equal to 10, 20, 50 and 100, the returned dictionary would look like the following:

{'min': [10, 20, 50, 100], 'max': [10, 20, 50, 100]}

Please refer to projection documentation in the L2 chapter for details.

**Volumes**

The *volumes* array contains *num\_volumes* dictionaries with the following keys: *attributes* and *levels* analogous to the same entries for stacks. The level attributes for volume differ by indicating *num\_z\_tiles* instead of *num\_slices* reflecting the fact that volumes are cut into *tile\_size*^3 blocks rather than tiles.

Example of volume level attributes dump:

{'num\_x\_tiles': 2, 'num\_y\_tiles': 2, 'num\_z\_tiles': 1}

**Stack Service**

**Slice meta information query**

/bbic?mode=slice\_meta&fname=<fname>&stack=<n>(&slice=<m>|&slice\_orig\_id=<id>)

This query returns slice-specific meta information for slice specified either using slice number *m* or slice *id* in the stack number *n* in file *fname*. If *id* is used slice number *m* is determined using */bbic/stacks/n/slice\_orig\_ids* attributes if present. Otherwise an exception is thrown.

Slice meta information correspond to attributes of group */bbic/stacks/n/slices/m* if present. If not, slice meta information will be synthesized using *width*, *height* and *local\_to\_world* attributes from */bbic/stacks/n*. In such case, *local\_to\_world* matrix is multiplied by:

|1 0 0 0| |0 1 0 0| |0 0 1 m| |0 0 0 1|

to include the slice position. Thanks to this slice meta information can be used as the only source of transformation when displaying slices, disregarding stack level attributes.

**Tile query**

/bbic?mode=tile&fname=<fname>&stack=<n>(&slice=<m>|&slice\_orig\_id=<id>)&level=k&x=<x>&y=<y>&proj=<proj>&proj\_d=<d>&extend\_tiles=<e>

This query returns a tile image specified by BBIC filename *fname*, stack number *n*, slice number *m* or slice *id*, detail level *k*, "x" position *x*, "y" position *y* and optional parameters: *proj* specifying projection type, *proj\_d* specifying projection depth and *extend\_tiles* specifying handling of last row/column/diagonal tiles, which are usually smaller than declared *tile\_size*.

If *id* is used slice number *m* is determined using */bbic/stacks/n/slice\_orig\_ids* attributes if present. Otherwise an exception is thrown.

If *proj* parameter is absent, a regular slice tile is used. If it's present, a projection image will be constructed using projection acceleration structures if present. In such case *m* defines the starting slice position and *proj\_d* defines the projection depth. The default value for *proj\_d* is 10 and will be used if the parameter is absent.

The *extend\_tiles* parameters if present and set to 1 tells the service to pad tiles with zeros up to *tile\_size*^2 size. In the future *extend\_tiles* will be removed and all tiles will be padded to the declared *tile\_size*.

In the future, *format* and *quality* parameters will be added to specify the resulting image format. For the moment it defaults to JPEG at 85% quality which seems to be optimal for the quality/bandwidth ratio.

**Palette query**

/bbic?mode=palette&stack=<n>&source=<source>&labels=<labels>

This query returns a palette dictionary for the given stack based on the specified of colors. Currently only the "labels" source is supported indicating atlas labels-based coloring. Parameter specified labels naming in case of multiple atlas colorings stored in the BBIC stack. The "default" labels must always point to the default atlas labels for the given stack and thus is always safe to use it as parameter.

**Related query**

/bbic?mode=related&stack=<n>

This query returns stacks related to the one specified. The most important relation is defined by the returned *coreg\_matrix* attribute, which defines a matrix that needs to be applied along with *local\_to\_world* matrix of the target stack to make it coregistered with the current one. If no such matrix has been calculated, this attribute is absent.

The answer format is the following:

[{"fname": "/path/to/related/bbic.h5", "stack": "0", "coreg\_matrix": [4x4 matrix], "description": "foo bar"}]

**Lookup query**

/bbic?mode=lookup&fname=<fname>&stack=<n>&slice=<slice>&x=<x>&y=<y>&labels=<labels>

This query returns region ID and name based on embedded atlas labels (if present). If not specified, assumes the value of "default". and specify absolute coordinates and not tile coordinates as opposed to other Stack Service queries. and respectively specify the stack and slice number in the BBIC file .

The answer format is the following:

{"value": "40", "name": "caudate putamen"}

**Volume Service**

**Plane query**

/bbic\_vol?mode=plane&fname=<fname>&vol=<n>&level=<k>&size=<size>(&fill=<fill>){3}(&x=<x>&y=<y>&z=<z>){3}

This query retrieves an arbitrary rectangular slice through the volume.

**Parameters**

| **Parameter** | **Description** | |---------------------------|--------------------------------------------------------------------------| | *fname* | BBIC file name | | *vol* | Volume number within the specified BBIC file | | *level* | Detail level, where 0 corresponds to the highest detail, ranges from 0 to (*num\_levels*-1) | | *size* | The resulting output image will be *size* x *size* | | *fill* | If present the fill color for outside of the volume, otherwise (0, 0, 0) is assumed | | *x*, *y*, *z* | Coordinates of the top-left, top-right, bottom-left points of the rectangle in world coordinates |

**Remarks**

The {3} notation means that previous part of the URL enclosed in round braces should be repeated 3 times.

The resulting image is always square. Client software should account for that when providing *x*, *y* and *z* coordinates to ensure correct aspect ratio (1:1).

The *fill* value specifies normalized R, G, B values ranged [0, 1].

The triple occurence of *x*, *y* and *z* parameters in order top-left, top-right, bottom-left, specify the following rectangle in world space coordinates:

top-left ------------ top-right | | | | | | | | | | | | | | bottom-left -------------------

which is mapped into a *size* x *size* image.

**Image Manipulation Service**

**Rationale**

Processing images on the client side consumes more CPU and bandwidth (in case for example when multiple images need to be blended). With an Image Manipulation Service, multiple images can be processed on the server side and only single result needs to be downloaded.

**Query**

Method: POST

/bbic\_ims?fname=<fname1>[&fname=<fname2>[&fname=...]]

Files indicated as parameters will be the only ones accessible via the subsequent image loading commands in the program body.

The body can contain for example the following statements:

TILE 0 0 0 200 0 0 ASTYPE f4 MUL 0 0 MIN 0 MAX 0 COPY -1 SUB 3 1 SUB 0 1 PUSH 255 DIV 0 3 MUL 0 4 POP 4 ASTYPE u1

which will square and stretch histogram of tile (0, 0) from slice 200 of stack 0 of the first file specified in parameter.

The program must make sure that at the end of execution an unsigned 8-bit grayscale or RGB image is at the top of the stack.

**Available commands**

| \*Command\* | \*Description\* | |----------------|-------------------------------------------------------------------------------------| | TILE | Parameters: <file\_idx> <stack> <level> <slice> <x> <y> <proj> <proj\_d> | | | <extend\_tiles> | | Executes the Stack Service tile command on the given file index and pushes the | | | resulting image on the stack as NumPy array | |----------------|-------------------------------------------------------------------------------------| | PALETTE | Parameters: <file\_idx> <stack> <source> [<labels>] | | | Loads palette from the Stack Service for the given file index, stack and source. | | | Resulting palette can be then applied using MAP command. | | | Palette comes as a dictionary of the form {(grayscale,): (r, g, b)} or | | | {(r, g, b): (r, g, b)}. Currently only the "labels" source is supported. | | | Optional argument <labels> specifies labels name, otherwise "default" is used. | |----------------|-------------------------------------------------------------------------------------| | ASTYPE | Parameters: <type> | | | Converts top of the stack array to given numeric type. The type can be one of: | | | u1, u2, u4, u8, i1, i2, i4, i8, f4, f8. | |----------------|-------------------------------------------------------------------------------------| | COPY | Parameters: <stack\_pos> | | | Pushes to the top of the stack the results of np.copy() on the indicated | | | position of the stack. | |----------------|-------------------------------------------------------------------------------------| | PUSH | Parameters: <number> | | | Pushes to the top of the stack the given number | |----------------|-------------------------------------------------------------------------------------| | PUSH\_PALETTE | Parameters: <json\_encoded\_palette> | | | Pushes to the top of the stack a given palette defined as JSON-encoded dictionary | | | with the following content. IDs must be unsigned 32-bit integers. | | | {id\_1: (0, 0, 0), id\_2: (255, 255, 255)} | |----------------|-------------------------------------------------------------------------------------| | POP | Parameters: <number> | | | Pops given number of elements from the top of the stack. | |----------------|-------------------------------------------------------------------------------------| | DEL | Parameters: <stack\_pos> | | | Removes given position from the stack. | |----------------|-------------------------------------------------------------------------------------| | TOP | Parameters: <stack\_pos> | | | Brings indicated stack position to the top of the stack. | |----------------|-------------------------------------------------------------------------------------| | SWAP | Parameters: <stack\_pos> | | | Swaps current top of the stack with the indicated stack position. | |----------------|-------------------------------------------------------------------------------------| | MIN, MAX | Paremeters: <stack\_pos> | | | Pushes to the top of the stack the result of np.min() or np.max() respectively on | | | stack position indicated by <stack\_pos>. | |----------------|-------------------------------------------------------------------------------------| | ADD, SUB, | Parameters: <stack\_pos1> <stack\_pos2> | | MUL, DIV | Executes given mathematical operation (respectively: +, -, \*, /) on the two stack | | | positions indicated in parameters. Result overwrites the first position. | |----------------|-------------------------------------------------------------------------------------| | RGB, GRAYSCALE | Paremeters: <stack\_pos> | | | Converts respectively to RGB/grayscale given stack position and pushes the result | | | to the top of the stack. The command is a noop if the image is already in | | | RGB/grayscale respectively. | |----------------|-------------------------------------------------------------------------------------| | MAP | Parameters: <image\_pos> <palette\_pos> | | | Maps given image using given palette. RGB/grayscale conversions are handled | | | automatically. | |----------------|-------------------------------------------------------------------------------------| | UNIQUE\_IDS | Parameters: <image\_pos> | | | Returns a list of unique IDs for the image at given stack position. This is the | | | first command from the new version allowing for non-image output. | |----------------|-------------------------------------------------------------------------------------| | ZOOM | Parameters: <image\_pos> <sx> <sy> <order> | | | Pushes a zoomed version of the selected image to the stack. sx and sy determine | | | scaling for first and second dimension respectively. Order refers to the spline | | | used for interpolation, 0 - nearest neighbor, 1 - linear, 2 - cubic. Higher | | | Higher orders generally don't make sense here. | |----------------|-------------------------------------------------------------------------------------| | CROP | Parameters: <image\_pos> <x1> <y1> <x2> <y2> | | | Pushed cropped version of the selected image to the stack. x1, y1, x2, y2 determine | | | respectively the top-left and bottom-right corners. |

**Closing Remarks**

For information about the file format itself, please see *bbic\_format.md*.

For information about Python libraries provided for handling the BBIC format, please see *bbic\_py.md*.

**FAQ**

What's the license?

BBIC Web Services are BSD-licensed.

# Appendix II – BBIC File Format

**BBIC File Format Documentation**

Author: Stanislaw Adaszewski, 2014

Blue Brain Image Container File Format

**Table of Contents**

1. Glossary
2. Definitions
3. Foreword
4. Structure
   1. Level 1 (Disk Storage)
   2. Level 2 (Logical Data Sets)
   3. Level 3 (Image/Volume Codecs)
5. Tools
   1. Stack Tools
   2. Volume Tools
   3. Others
6. Closing Remarks
7. FAQ

**Glossary**

* BBIC - Blue Brain Image Container
* HCL - [Hyperdimensional Compression Library](http://github.com/sadaszewski/hcl)
* L1 - Level 1 Structure
* L2 - Level 2 Structure
* L3 - Level 3 Structure
* LOD - Level Of Detail
* SEC2J / H5FD\_sec2j - [SEC2 journaled](http://github.com/sadaszewski/H5FD_sec2j)

**Definitions**

*Nothing here for the moment*.

**Foreword**

Blue Brain Image Container (BBIC) file format is an effort to provide a flexible container for multidimensional datasets with implicit support for level of detail organization and other performance improving techniques.

In particular, two distinctive data types are recognized by the standard:

* Stacks of Images
* 3D Volumes

The former defining a collection of semantically related slices through a 3D object which be organized either regularly or arbitrarily in space.

The latter referring to a three dimensional grid of uniformly sized voxels describing a scalar field in world space.

Thanks to hierarchical organization and reuse of existing image, video and volume storage formats, BBIC is capable of accomodating many scenarios such as multiple color channels, vector field volumes, time axis, etc. in an elastic application-specific manner.

Software libraries, web services and tools are provided for handling basic scalar gray scale image stacks and volumes which are easily extensible to the desired application-specific needs.

**Structure**

**Level 1 Structure (Disk Storage)**

The Level 1 Structure is the disk storage format. Currently it's [HDF5](http://hdfgroup.org/). In principle any format supporting the notion of groups/datasets and attributes similar to those of HDF5 would be suitable.

Current APIs are mostly written in Python and will support any format handled by a library with syntax consistent with [h5py](http://h5py.org/).

Since many BBIC datasets are multi-terabyte in size, the underlying storage method should support such "big data".

Conceivable alternatives (not implemented):

* Filesystems supporting extended attributes
* ZIP64

**SEC2J file driver**

HDF5 is susceptible to file corruption due to premature process termination while writing meta data. For reliability reasons a journaled flavour of HDF5's SEC2 (POSIX) file driver is recommended for performing write operations. The one currently in use by certain tools is [H5FD\_sec2j](http://github.com/sadaszewski/HDF5_sec2j). Version 1.10.0 of HDF5 implementation was announced to contain a built-in journaling mechanism. Upon its release, existing tools should migrate to use the native journalling functionality.

**Level 2 Structure (Logical Data Sets)**

**Group /bbic**

The root group must contain a group named *bbic* with the following content:

**Attributes**

| **Attribute** | **Description** | |-------------------|-----------------------------------------------------------------| | *version* | Specifies version of the BBIC format used. Currently: 1 | | *num\_stacks* | Specifies the number of stacks in the file | | *num\_volumes* | Specifies the number of volumes in the file |

**Groups**

| **Group** | **Description** | |-------------------|-----------------------------------------------------------------| | *stacks* | Group containing the number of stacks specified by num*stacks | | \*volumes\* | Group containing the number of volumes specified by num*volumes |

**Remarks**

Version will be bumped when non-backwards-compatible changes are introduced. As long as existing tools can handle new revisions of the format without crashing and provide their intended output, even if they are unable to make use of some of the new features introduced, the version will remain the same.

A good example of this policy was the introduction of projection acceleration structures, which didn't break existing projection software using individual slices. Please see projection documentation in L2 chapter for details.

If software is looking for some specific data introduced in a new revision of the format it should do so directly instead of checking the version number.

**Group /bbic/stacks**

This group holds one group per stack.

**Attributes**

*None*

**Groups**

| **Group** | **Description** | |-------------------|-----------------------------------------------------------------| | *0* | The first stack group | | *1* | The second stack group | | ... | | | *(num\_stacks-1)* | The last stack group |

**Group /bbic/stacks/n**

This group stores attributes and data for the n-th stack. It's content is as follows:

**Attributes**

| **Attribute** | **Description** | |----------------------|-----------------------------------------------------------------| | *original\_filenames* | Pattern of original filenames from which the stack was created, e.g. "/home/adaszews/Downloads/BigBrain/BigBrain*%04d.jpg" or "/home/adaszews/Downloads/BigBrain/list.txt" if filenames were taken from a list | | \*description\* | Stack description, e.g. "Big Brain 2um slices" | | \*modify*time\* | Modification date and time, e.g. "2014-06-28 13:14:00.025 UTC" | | *type* | MIME type specifying the codec used for the underlying datasets | | *width* | Width of each slice | | *height* | Height of each slice | | *tile\_size* | Tile size, defaults to 256 in most tools | | *local\_to\_world* | 4x4 local to world transformation matrix | | *orientation* | String describing orientation of the slices, e.g. "saggital" | | *num\_levels* | Number of LOD levels | | *num\_slices* | Number of slices | | *is\_video* | Boolean indicating whether video codec was used to store the slices | | *fps* | Frame rate of the video codec | | *num\_frames* | Number of time frames, if the data contains a time axis |

**Groups**

| **Group** | **Description** | |--------------------|-----------------------------------------------------------------| | *slices* | Group holding per-slice attributes | | *slice\_orig\_ids* | Group holding original id to slice number mapping | | *levels* | This group holds the actual LOD tiles | | *projections* | This optional group holds the projection acceleration tiles |

**Remarks**

The *type* attribute determines how the data is going to be interpreted. For discussion of supported data codecs, please see *L3* documentation.

The *orientation* attribute is descriptive only. In case of oblique slices it's advised to put the string "oblique" in there.

If slices differ in size, the conversion tool can either keep their original sizes or pad them to match the common maximum width/height. If original sizes are maintained the information about them will be stored in the *slices* group on a per-slice basis.

In both cases the *width* and *height* attributes of the stack will contain the maximum width/height across all the slices.

Similarly per-slice transformation matrix may be necessary to maintain slices alignment if their original sizes are used.

The per-slice transformation matrix will be also useful if slices are irregularly placed in world coordinates.

When applying the transformation matrices x and y coordinates are assumed in the slice image coordinates, while the z coordinate is set to the slice number.

If *is\_video* attribute is set to True, the last stage of level hierarchy (slices) is contained within the video frames.

Please note that if both *num\_frames* is present and greater than one and *is\_video* is set to True, then both the slices dimension and time dimension are contained in the video stream, in the C (column-major) order. Time dimension is assumed to be higher dimension than slices dimension, therefore all slices for frame 0 will precede all slices for frame 1.

Tables explaining all possible settings for *is\_video* and *num\_frames* attributes follow:

**Tile Dataset Path**

| | **num*frames == 1*** *|* ***num*frames > 1** | |-----------------------|------------------------|------------------------------| | **is*video == False*** *| /bbic/stacks/n/levels/\*k\*/\*x\*/\*y\*/\*slice\* | /bbic/stacks/\*n\*/levels/\*k\*/\*x\*/\*y\*/\*slice\* | |* ***is*video == True** | /bbic/stacks/n/levels/*k*/*x*/*y* | /bbic/stacks/*n*/levels/*k*/*x*/*y* |

**Dataset Logical Dimensions**

| | **num*frames == 1*** *|* ***num*frames > 1** | |-----------------------|------------------------|------------------------------| | **is*video == False*** *| \*tile*size\* x *tile\_size* | *tile\_size* x *tile\_size* x *num\_frames* | | **is*video == True*** *| \*tile*size\* x *tile\_size* x *num\_slices* | *tile\_size* x *tile\_size* x *num\_slices* x *num\_frames* |

If stack-specific facilities of BBIC are not used, it's recommended to store N-dimensional data as volumes due to simpler organization.

**Group /bbic/stacks/n/slices**

This group contains one group per each slice of stack n, which in turn holds slice-specific attributes.

**Attributes**

*None*

**Groups**

| **Group** | **Description** | |--------------------|-----------------------------------------------------------------| | *0* | The first slice-specific attributes group | | *1* | The second slice-specific attributes group | | ... | | | *(num\_slices-1)* | The last slice-specific attributes group |

**Group /bbic/stacks/n/slices/m**

This group hold attributes of m-th slice of n-th stack.

**Attributes**

| **Attribute** | **Description** | |----------------------|-----------------------------------------------------------------| | *width* | Slice width | | *height* | Slice height | | *local\_to\_world* | Slice 4x4 local to world transformation matrix |

**Groups**

*None*

**Group /bbic/stacks/n/slice*orig*ids**

This group contains only attributes mapping arbitrary strings to slice numbers.

**Attributes**

| **Attribute** | **Description** | |----------------------|-----------------------------------------------------------------| | *arbitrary\_id* | Holds slice number corresponding to *arbitrary\_id* |

**Groups**

*None*

**Group /bbic/stacks/n/levels**

Nested within this group are the actual tile images depending of the level of detail. The group itself contains one subgroup per LOD. The levels are numbered from 0 to (*num\_levels*-1). Zero means the original image size and each consecutive level halves the width and height. When width or height reach 1 the scaling is halted.

**Attributes**

*None*

**Groups**

| **Group** | **Description** | |--------------------|-----------------------------------------------------------------| | *0* | The first LOD group | | *1* | The second LOD group | | ... | | | *(num\_levels-1)* | The last LOD group |

**Group /bbic/stacks/n/levels/k**

Nested within this group are the tiles for level *k*.

The group contains subgroups corresponding to "x" coordinate of the tile, ranging from 0 to (*num\_x\_tiles*-1).

**Attributes**

| **Attribute** | **Description** | |----------------------|-----------------------------------------------------------------| | *num\_x\_tiles* | Number of tiles in the "x" direction for this level | | *num\_y\_tiles* | Number of tiles in the "y" direction for this level | | *num\_slices* | Number of slices for this level, same as the number for the stack |

**Groups**

| **Group** | **Description** | |--------------------|-----------------------------------------------------------------| | *0* | The group for x position equal to 0 | | *1* | The group for x position equal to 1 | | ... | | | *(num\_x\_tiles-1)* | The group for x position equal to (*num\_x\_tiles*-1) |

**Group /bbic/stacks/n/levels/k/x**

Nested within this group are tiles for level *k* and "x" coordinate equal to *x*.

The group contains subgroups corresponding to "y" coordinate of the tile, ranging from 0 to (*num\_y\_tiles*-1).

**Attributes**

*None*

**Groups**

| **Group** | **Description** | |--------------------|-----------------------------------------------------------------| | *0* | The group for y position equal to 0 | | *1* | The group for y position equal to 1 | | ... | | | *(num\_y\_tiles-1)* | The group for y position equal to (*num\_y\_tiles*-1) |

**Group /bbic/stacks/n/levels/k/x/y**

Nested within this group are tiles for level *k*, "x" coordinate equal to *x* and "y" coordinate equal to *y*.

The group contains datasets corresponding to slice number, ranging from 0 to (*num\_slices*-1).

**Attributes**

*None*

**Datasets**

| **Dataset** | **Description** | |--------------------|-----------------------------------------------------------------| | *0* | The dataset for slice 0 | | *1* | The dataset for slice 1 | | ... | | | *(num\_slices-1)* | The dataset for slice (*num\_slices*-1) |

**Remarks**

Depending on the *is\_video* attribute, all slices may be packed into a video stream. In such case the path */bbic/stacks/n/levels/k/x/y* already refers to the respective video stream dataset, where consecutive frames correspond to slices and the stream length is *num\_slices* frames.

Logical dimensions of such video dataset are *tile\_size* x *tile\_size* x *num\_slices*.

**Dataset /bbic/stacks/n/levels/k/x/y/slice**

This dataset holds slice tile data for LOD *k*, "x" coordinate equal to *x*, "y" coordinate equal to *y* and slice number equal to *slice*.

Logical dimensions of this dataset are *tile\_size* x *tile\_size*. Physical representation depends on the MIME type declared. See *Remarks* section and *Image/Volume Codecs* chapter for further discussion.

**Remarks**

Depending on the declared MIME type, the dataset can be either a regular HDF5 dataset with corresponding dimensionality (for *application/vnd.hdf5.DataSet*) or a one dimensional unsigned 8-bit integer dataset containing data bytes stream produced by the relevant codec.

Depending on the *is\_video* attribute, all slices may be packed into a video stream. In such case the path */bbic/stacks/n/levels/k/x/y* already refers to the respective video stream dataset.

**Group /bbic/stacks/n/projections**

Nested within this group are projection acceleration tiles for aggregate *aggr* (e.g. *min*, *max*). The immediate child groups correspond to numbers of slices used to calculate the aggregates.

**Groups**

Child group names depend on the calculated projection aggregates. For a file containing acceleration data for *min* and *max* projections it will be:

| **Group** | **Description** | |-------------------|-----------------------------------------------------------------| | *min* | Group nesting *min* aggregates | | *max* | Group nesting *max* aggregates |

**Group /bbic/stacks/n/projections/aggr**

Nested within this group are projection acceleration tiles for aggregate *aggr*.

Child group names depend on the numbers of slices picked for creating the acceleration data. For a file containing 10-, 20-, 50-, ..., 2000-slice aggregates it will be:

**Groups**

| **Group** | **Description** | |-------------------|-----------------------------------------------------------------| | *10* | Group nesting 10-slice aggregates | | *20* | Group nesting 20-slice aggregates | | *50* | Group nesting 50-slice aggregates | | ... | | | *2000* | Group nesting 2000-slice aggregates |

**Group /bbic/stacks/n/projections/aggr/d**

Nested within this group are projection acceleration tiles for aggregate *aggr* calculated across *d* slices each.

Children correspond to level of detail.

**Groups**

| **Group** | **Description** | |-------------------|-----------------------------------------------------------------| | *0* | Level 0 (original detail) | | *1* | Level 1 (half the size) | | *2* | Level 2 (quarter the size) | | ... | | | *(num\_levels-1)* | Last level (width and/or height equal to 1) |

**Group /bbic/stacks/n/projections/aggr/d/k**

Nested within this group are projection acceleration tiles for aggregate *aggr* calculated across *d* slices each, for detail level *k*.

Children correspond to starting slice position.

**Groups**

| **Group** | **Description** | |-------------------|-----------------------------------------------------------------| | *0* | Starting slice equal to 0 | | *d* | Starting slice equal to *d* | | *2d* | Starting slice equal to *2d* | | ... | | | *floor(num\_slices/d)* \* *d* | Starting slice equal to floor(*num\_slices*/*d*) \* *d* |

**Group /bbic/stacks/n/projections/aggr/d/k/z**

Nested within this group are projection acceleration tiles for aggregate *aggr* calculated across *d* slices each, starting at slice *z*, for detail level *k*.

Children correspond to "x" position analogous to the regular */bbic/stacks/n/level/k* children.

**Groups**

| **Group** | **Description** | |-------------------|-----------------------------------------------------------------| | *0* | "x" position equal to 0 | | *1* | "x" position equal to 1 | | ... | | | *(num\_x\_tiles-1)* | "x" position equal to (*num\_x\_tiles*-1) |

**Group /bbic/stacks/n/projections/aggr/d/k/z/x**

This group contains projection acceleration tiles for aggregate *aggr* calculated across *d* slices each, starting at slice *z*, for detail level *k* and tile "x" position equal to *x*.

Children correspond to "y" position analogous to the regular */bbic/stacks/n/level/k/x* children.

**Datasets**

| **Dataset** | **Description** | |-------------------|-----------------------------------------------------------------| | *0* | "y" position equal to 0 | | *1* | "y" position equal to 1 | | ... | | | *(num\_y\_tiles-1)* | "y" position equal to (*num\_y\_tiles*-1) |

**Dataset /bbic/stacks/n/projections/aggr/d/k/z/x/y**

This dataset contains projection acceleration tile for aggregate *aggr* calculated across *d* slices each, starting at slice *z*, for detail level *k* and tile "x" position equal to *x* and tile "y" position equal to *y*.

Similarly to regular LOD tiles, logical dimensions of this dataset are *tile\_size* x *tile\_size*. Physical representation depends on the MIME type declared. See *Remarks* section and *Image/Volume Codecs* chapter for further discussion.

**Remarks**

Projection acceleration data can be illustrated in the following manner:

| | 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 | 120 |

|-----------|----|----|----|----|----|----|----|----|----|--------|-----|-----|

| | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |

| | 20 | 20 | 20 | 20 | 20 | 20 |

| | 50 | 50 | 50... |

| | 100 | 100... |

| | 200... |

| Example 1 | 50 | 10 | |

| Example 2 | | 10 | 20 | 10 | 90, 91 | |

| | | | | | 92, 93 | |

| | | | | | 94, 95 | |

The numbers above the ---- line indicate slice number at the beginning of the cell. Cells below correspond to acceleration images and indicate how many slices were used to create each such image.

When a projection across arbitrary slice interval is requested, the intermediary projections are used to speed up calculations.

Example 1. Projection across 60 slices was requested starting at slice 0. Slice 0 is aligned with multiple of 50-slice intermediaries, so projection across first 50 slices is sorted out using that acceleration structure. Slice 50 is aligned with another 50-slice intermediary but only 10 slices are remaining. It's not aligned with a 20-slicer which would have been too much anyway. And luckily it's aligned with a 10-slicer which is the exact amount necessary. In the end, a 60-slice projection was achieved doing a 2-image projection.

Example 2. A 45-slice projection was requested starting at slice 50. Slice 50 is aligned with a 50-slicer but it's too much. It's not aligned with a 20-slicer. It's aligned with a 10-slicer which we use. Then we end up with 35 slices remaining and aligned with a 20-slicer and a 10-slicer. We prefer a 20-slicer. We have 15 slices remaining and are aligned with a 20-slicer and a 10-slicer. 20-slicer is too big, therefore we use a 10-slicer. We are left with 5 slices remaining and aligned only with a 10-slicer which is too big. Therefore we read the remaining 5 **original slices**. A 45-slice projection done as 8-image projection.

If the smallest acceleration image is across 10 slices, the maximum number of projections that will ever have to be done is 10.

**Group /bbic/volumes**

This group holds one group per volume. Volume definitions are similar to stacks in many respects, therefore in the following sections only differences will be discussed.

**Attributes**

*None*

**Groups**

| **Group** | **Description** | |-------------------|-----------------------------------------------------------------| | *0* | The first volume group | | *1* | The second volume group | | ... | | | *(num\_volumes-1)* | The last volume group |

**Group /bbic/volumes/n**

This group stores attributes and data for the n-th *volume*. It's content is identical to that used for stacks, except for:

**Attributes**

| **Attribute** | **Description** | |----------------------|-----------------------------------------------------------------| | *width* | Width of the **volume** | | *height* | Height of the **volume** | | *tile\_size* | **Block** size | | *orientation* | If created from slices, their orientation can be described here | | *num\_slices* | **Depth** of the volume |

**Groups**

| **Group** | **Description** | |--------------------|-----------------------------------------------------------------| | ~~slices~~ | N/A | | ~~slice*orig*ids~~ | N/A | | *levels* | This group holds the LOD **blocks** | | ~~projections~~ | N/A |

**Remarks**

*local\_to\_world* matrix in the case of a volume is applied directly to the volume coordinates.

In case of non-scalar fields, the *local\_to\_world* matrix should be applied also to the stored vector values. This is left to the application-level code and doesn't constitute part of the BBIC file format specification.

**Group /bbic/volumes/n/levels/k**

Nested within this group are **blocks** for level *k*.

The group contains subgroups corresponding to "x" coordinate of the block, ranging from 0 to (*num\_x\_tiles*-1).

**Attributes**

| **Attribute** | **Description** | |----------------------|-----------------------------------------------------------------| | *num\_x\_tiles* | Number of **blocks** in the "x" direction for this level | | *num\_y\_tiles* | Number of **blocks** in the "y" direction for this level | | **num*z*tiles** | Number of blocks in the "z" direction for this level | | ~~num\_slices~~ | N/A |

**Group /bbic/volumes/n/levels/k/x**

Nested within this group are **blocks** for level *k* and "x" coordinate equal to *x*.

**Group /bbic/volumes/n/levels/k/x/y**

Nested within this group are **blocks** for level *k*, "x" coordinate equal to *x* and "y" coordinate equal to *y*.

**Dataset /bbic/volumes/n/levels/k/x/y/z**

This dataset holds **block** data for LOD *k*, "x" coordinate equal to *x*, "y" coordinate equal to *y* and "z" coordinate equal to *z*.

Logical dimensions of this dataset are *tile\_size* x *tile\_size* x *tile\_size*. Physical representation depends on the MIME type declared. See *Remarks* section and *Image/Volume Codecs* chapter for further discussion.

**Remarks**

Depending on the declared MIME type, the dataset can be either a regular HDF5 dataset with corresponding dimensionality (for *application/vnd.hdf5.DataSet*) or a one dimensional unsigned 8-bit integer dataset containing data bytes stream produced by the relevant codec.

**Level 3 Structure (Image/Volume Codecs)**

In the current BBIC ecosystem, the most frequently encountered and supported MIME types are:

* application/vnd.hdf5.DataSet
* image/jpeg, image/png, image/tiff
* video/avi, video/x-matroska
* application/vnd.hcl.JpegSequence

A MIME type defines how to handle the datasets in stacks and volumes to reconstruct their original dimensionality and format.

**MIME types**

**Virtual formats**

**application/vnd.hdf5.DataSet**

This MIME-type indicates that dataset is supposed to be treated as a native HDF5 dataset, 2-d in case of stacks, 3-d in case of volumes, possibly N-d for future data types. Gzip, Szip, LZF, n-bit and shuffling compression is still possible using HDF5 filters. **Note**: In case of non-HDF5 L1 format, this data type doesn't make sense and cannot be used. An alternative type to store raw N-dimensional arrays may be introduced in the future. The DataSet format is currently used by the oblique plane reslicing API because of its performance.

**Standard formats**

**image/jpeg, image/png and image/tiff**

These three well-known image formats are supported in particular using the PILLOW library in Python. They are suitable for stack datasets. A sequential version of image/jpeg for compressing N-dimensional data is available via HCL.

**video/avi, video/x-matroska**

Unboxed video container formats can be used to store 2-dimensional data with a time axis. This format can also be used to store slices as indicated by *is\_video* attribute.

**image/x-exr**

The [OpenEXR](http://en.wikipedia.org/wiki/OpenEXR) format is a high dynamic range imaging format. It supports arbitrary number of channels, 16-bit floating point data type, lossless and lossy compression. Its support for multiresolution data is redundant to the one provided by BBIC and shouldn't be used in combination with BBIC.

**Formats with non-standard MIME types**

**application/x-nifti, image/nii, image/gznii**

All these non-standard MIME types try to reference the popular [NIFTI-1](http://nifti.nimh.nih.gov/nifti-1) N-dimensional data format. Nifti supports different data types and up to 7 dimensions each with a maximum size of 16383 (signed short is used for storage). Due to the size limitation it's not suitable for storing "Big Data", but it's perfectly suitable for storing blocks in a BBIC volume LOD pyramid. NIFTI-1 could be a good standard for storing volume blocks across different L1 formats.

[NIFTI-2](http://nifti.nimh.nih.gov/nifti-2) is the 64-bit update of the NIFTI-1 format announced in 2011. It could be considered as a more powerful alternative to NIFTI-1 although LOD blocks exceeding NIFTI-1 limits are hard to imagine.

**HCL formats**

**application/vnd.hcl.Raw**

Raw N-dimensional array storage format. Data is stored as-is with header information specifying numeric type and dimensions of the dataset. Data is required to be Little Endian by the HCL library and BBIC layer is responsible for making appropriate conversions if running on a Big Endian machine.

**application/vnd.hcl.JpegSequence**

This MIME type is produced by the Hyperdimensional Compression Library (HCL) from N-dimensional data by taking 2-dimensional hyperslabs, compressing them as JPEG images and writing out sequentially.

**application/vnd.hcl.Ffmpeg**

This format contains a video stream compressed using ffmpeg. The default codec is libx264 and container defaults to [Matroska](http://en.wikipedia.org/wiki/Matroska). Similarly to JpegSequence, the frames are taken as two-dimensional slabs of an N-dimensional dataset.

**application/vnd.hcl.NdWavelet**

An N-dimensional wavelet compression algorithm is used to produce this type of dataset. Excellent compression rate is possible with few to no visually discernible artifacts. Part of the HCL family of compression methods.

**Time axis support**

BBIC was historically a space-oriented file format supposed to provide efficient data partitioning and LOD functionality. This minimalistic approach leaves higher level data interpretation to *L3* codecs and application-level code for the sake of keeping the specification simple.

That being said, due to its flexibility BBIC can easily accommodate a time axis. This can be achieved by using appropriate *L3* data codecs, which support the notion of time (i.e. video formats) or multi-dimensional data.

Note that for example image formats such as JPEG/PNG cannot be used in this scenario.

If the *num\_frames* attribute is present in addition to *width*, *height* and *num\_slices* it indicates that the first dimension after the spatial ones is supposed to be treated as time. In case of video formats it referes to video frames.

For storing an image stack with time axis, a video or multi-dimensional format must be used.

**Tools**

**Stack Tools**

**bbic\_create.py**

Usage: bbic\_create.py filename\_pattern tile\_size format output\_filename [orientation [description [slice\_positions]]]

This tool will create or append to an existing BBIC file an image stack specified by the following parameters.

**Parameters**

| **Parameter** | **Description** | |---------------------------|--------------------------------------------------------------------------| | *filename\_pattern* | pattern of filenames, e.g. foo*%03d*bar.png or a text file with a list | | *tile\_size* | tile size used for the stack | | *format* | target image format, e.g. JPEG | | *output\_filename* | target file name | | *orientation* | description of slices orientation, defaults to "axial" | | *description* | description of the stack | | *slice\_positions* | text file containing slice positions one per row |

**Remarks**

*format* is specified as in the PILLOW library.

**bbic\_extract.py**

Usage: bbic\_extract.py filename stack level slice output\_filename

This tool will extract a slice from a stack in BBIC file specified by the following parameters. Used for debugging/verification purposes.

**Parameters**

| **Parameter** | **Description** | |---------------------------|--------------------------------------------------------------------------| | *filename* | BBIC file name | | *stack* | stack index | | *level* | level of detail, 0 for highest, (*num\_levels*-1) for lowest | | *slice* | slice number from 0 to (*num\_slices*-1) | | *output\_filename* | target file name, e.g. "slice123.jpg" |

**Remarks**

The output filename determines the output image format.

**bbic\_projection.py**

Usage: bbic\_projection.py input.h5 stack\_nr format kind [spacings...]

This tool will create the projection acceleration structures for the given stack in a BBIC file. Numbers of slices used can be specified, as well as projection type and target image format.

**Parameters**

| **Parameter** | **Description** | |---------------------------|--------------------------------------------------------------------------| | *input.h5* | input/output BBIC file name | | *stack\_nr* | input/output stack index | | *format* | output image format, e.g. "JPEG" | | *kind* | type of the projection, e.g. *min* or *max* | | *spacings...* | number of slices used for creating the acceleration tiles |

**Remarks**

Format is specified as for the PILLOW library, e.g. "JPEG", "PNG", "TIFF".

*spacings...* refers to specifying multiple numeric arguments corresponding to the desired number of slices used in projection acceleration structures. It defaults to 10 20 50 100 200 500 1000.

**bbic\_reslice.py**

Usage: bbic\_reslice.py input.h5 output.h5 stack\_nr format axis [orientation [description]]

This tool will reslice a chosen image stack in a BBIC file along either X or Y axis. Output image format can be specified as well as the name for new orientation and new stack description.

**Parameters**

| **Parameter** | **Description** | |---------------------------|--------------------------------------------------------------------------| | *input.h5* | input BBIC file name | | *output.h5* | output BBIC file name | | *stack\_nr* | input stack index | | *format* | output image format, e.g. "JPEG" | | *axis* | axis to reslice along, X or Y | | *orientation* | optional description of new orientation | | *description* | optional stack description, copied from the input stack if not provided |

**Remarks**

If *orientation* is set to X the new stack will have *width* equal to input stack's *num\_slices*, *height* equal to input stack's *height* and *num\_slices* equal to input stack's *width*.

If *orientation* is set to X the new stack will have *width* equal to input stack's *num\_slices*, *height* equal to input stack's *width* and *num\_slices* equal to input stack's *height*.

^ Z

/ num\_slices-1

/

/ X

|---------------->

|0 width-1

|

|

|

| height-1

v Y

The generated orientation description reads "... - resliced along <axis>-axis", where ... stands in for the input stack orientation description.

**h264\_tiles.py**

Usage: h264\_tiles.py filename\_pattern tile\_size output\_filename [orientation [description [slice\_positions]]]

This tool is similar to *bbic\_create.py* as its purpose is to create a BBIC stack out of a collection of slices. The difference comes from using [avconv](https://libav.org/avconv.html) to compress individual tiles into video across slices. The resulting BBIC stack will have MIME type set to *video/x-matroska*, *is\_video* set to *True* and a non-zero *fps* attribute (25 by default). See *Group /bbic/stacks/n* Remarks in the L2 section for further details how video data is organized within the resulting stack.

**Parameters**

| **Parameter** | **Description** | |---------------------------|--------------------------------------------------------------------------| | *filename\_pattern* | filename or filename pattern, e.g. foo*%03d*bar.png' | | *tile\_size* | size used for generation of tiles | | *output\_filename* | name of output BBIC file | | *orientation* | optional description of the slices orientation, defaults to 'axial' | | *description* | optional stack description, default to 'Imported image stack' | | *slice\_positions* | text file containing slice positions one per row |

**Remarks**

In case of non-uniform slice placement, the *slice\_positions* argument can be used to specify slice positions individually.

This tool is heavy on filesystem, because it creates multiple temporary files before starting the conversion process. Later removal of those files might be slow.

**Volume Tools**

**bbic\_vol.py**

Usage: bbic\_vol.py input.h5 output.h5 stack\_nr

This tool creates a BBIC volume from a BBIC stack.

**Parameters**

| **Parameter** | **Description** | |---------------------------|--------------------------------------------------------------------------| | *input.h5* | input BBIC file name, must contain an image stack and uniform per-slice data | | *output.h5* | output BBIC file name, the reconstructed volume will be appended here | | *stack\_nr* | input stack index |

**Remarks**

This tool benefits from the already existing image stack structure to create a volume LOD pyramid with better performance than from raw slice images. To obtain this benefits however, the effective per-slice meta information must be uniform for the input image stack because the slices are treated as aligned pixel-wise.

Output file will be created if it doesn't exist.

*stack\_nr* must specify an existing stack index in the input BBIC file

This tool performs linear interpolation across slices to obtain LOD levels >0. The result should be identical to performing a 3-linear interpolation on a volume.

**bbic*vol*test.py**

Usage: bbic\_vol\_test.py input.h5 vol\_nr axis pos level out.img

Volume testing tool. Reconstructs a slice placed at specified position along given axis.

**Parameters**

| **Parameter** | **Description** | |---------------------------|--------------------------------------------------------------------------| | *input.h5* | input BBIC file name | | *vol\_nr* | volume number, must exist in the above file | | *axis* | X, Y or Z to specify the axis | | *pos* | position along the given axis to reconstruct slice at | | *level* | detail level, 0 - original, consecutive ones are at half resolution each | | *out.img* | output image file name |

**Remarks**

Output file format is determined by extension.

**Deprecated tools**

Some tools have been deprecated during the development process:

* bbic\_dpcm.py - Differential pulse-code modulation compression for images
* bbic\_subdivide.py - Change tile size of previously generated stack
* img\_reslice.py - Reslices given set of images in along X and Y axis, generating raw image files as output
* tif\_reslice.py - Dedicated tool for reslicing TIFF images
* tif\_reslice2.py - Another tool for reslicing TIFF images

**Work in progress**

There are some tools which are work in progress, e.g. for tiling SVG images.

**Closing Remarks**

For information about BBIC Web Services, please see *bbic\_services.md*.

For information about Python libraries provided for handling BBIC files, please see *bbic\_py.md*.

**FAQ**

What's the license?

The file format specification is Creative Commons. The libraries are BSD-licensed.