Embryo Pre-processing Documentation

Release 0.0.1

James Brown

March 12, 2015

CONTENTS

Contents:

CHAPTER

ONE

EMBRYOPREPROCESS MODULE

Search for downloaded embryo media (OPT/uCT) and pre-process it accordingly for viewing in Internet Embryo Viewer (IEV)

Class instances are initialised by passing in a config (.yaml) file, which contains the hostname (HOST), username (USER) and password (PASS) required to connect to the database. This allows for easy switching between "prince" and "live".

To begin pre-processing, the run() method must be called. Upon establishing a successful connection to the database, the program queries phenodcc_media.media_file, finding all valid recon media:

```
SELECT cid, lid, gid, sid, pid, qid, genotype.genotype, gene_symbol, measurement_id, url, checksum, e

FROM phenodcc_media.media_file, phenodcc_overviews.genotype, phenodcc_embryo.file_extension, phenodcc

WHERE

-- get the qid for the recon parameters
-- IMPC_EOL_001_001 OPT E9.5
-- IMPC_EMO_001_001 uCT E14.5/E15.5
-- IMPC_EMA_001_001 uCT E18.5
qid = (select impress.parameter.parameter_id from impress.parameter where parameter.parameter_key = '

-- join with measurements_performed to get the latest active and valid data
and mid = measurements_performed.measurement_id

-- join with genotype to get the colony id and gene_symbol
AND genotype.genotype_id = gid

-- join with extension to get the extension
AND extension_id = file_extension.id

-- recon has been downloaded and skipped tiling
AND ((phase_id = 2) or (phase_id = 3))
AND checksum IS NOT null

LIMIT 10000000
```

Subject to certain constraints, valid data is added to a pre-processing list which is subsequently handled by the process_recons() method.

Author *James Brown*

Organization Medical Research Council (MRC) Harwell, Oxfordshire, UK

Version 0.0.1

1.1 Requirements

- Python 2.7
- Numpy 1.8.2
- PyYAML 3.11
- MySQLdb 1.2.3
- SimpleITK 0.8.1

1.2 Examples

```
>>> ep = EmbryoPreprocess('/local/folder/IMPC_media', 'phenodcc_embryo.preprocessed', 'db_connect.yaml')
>>> ep.run()
```

class EmbryoPreprocess.**EmbryoPreprocess**(*base_path*, *embryo_table*, *config_file*)

Bases: object

The __init__ initialises a number of class attributes, and parses the .yaml config file.

Parameters

- **config_file** – path to .yaml containing database connection credentials.
- **base_path** – path where the embryo 'src' and 'emb' directories are located
- **embryo_table** – name of the table where embryo pre-processing rows are to be added

db_connect()

The db_connect method attempts to connect to a database using the credentials in the specified .yaml file.

Returns MySQLdb connect object if successful, or None if connection fails

db_disconnect()

The db_disconnect method attempts to disconnect from the database, and is called at the end of processing. Raises an exception upon failure (i.e. if the connection has already been closed).

get_mip(*in_path*, *out_dir*)

The get_mip method generates three maximum intensity projections (MIP) for visual QC purposes.

MIPs are generated from the downscaled image data, so the whole image can be loaded into memory for ease. The resulting images are written to disk in .png format.

process_recons()

The process_recons method loops through the pre-processing list, and attempts to process each recon in turn.

For each recon in the pre-processing list, it is first decompressed (if necessary) according to its file extension. Unfortunately, we do not know what image format the data will be. To overcome this, the program will attempt to open the file using each of the valid file readers in turn.

If the file is successfully opened, the pixel size extracted from the database:

```
SELECT * FROM phenodcc_overviews.metadata_group_to_values
    WHERE metadata_group = "$REPLACE$"
```

The image data is then rescaled to pre-specified image resolutions, writing the results to disk as NRRD files. In addition to the rescaled images, three orthogonal maximum intensity projection (MIP) are generated for visual QC purposes (for the moment, these are written to the IMPC_media/emb/... directory)

The three possible outcomes of the pre-processing job are as follows:

1.Successfully read and resample image data (status_id 1)

2.Failed to read image data, presumably due to an invalid file extension (status_id 2)

3.Error when resampling image data (status_id 3)

Finally, the status ID, extension ID and pixel size are updated in the embryo pre-processing table:

```
UPDATE phenodcc_embryo.preprocessed SET status_id=1, extension_id=12, pixelsize=13.59 WHERE
```

query_database(*sql*, *replacement=None*)

The query_database method executes arbitrary queries and returns any results as dictionary lists.

Input queries can be either strings or SQL files. If a replacement is specified, the $REPLACE$ wildcard will be replaced with it. The query is then executed and committed, raising an exception upon failure. If there are rows to be returned, the resulting data is parsed and returned as a list of dictionaries that can be referenced by column name for convenience.

Parameters

- **sql** – either a string containing an SQL query, or a path to an SQL file
- **replacement** – optional argument for SQL files, $REPLACE$ wildcard is replaced with the specified string

Returns a list of dictionaries if there were rows returned, otherwise None

run()

The run method queries the phenodcc_media.media_file table, identifying those that require pre-processing.

The method firsts attempts to connect to the database. If this connection fails, an exception is raised and the program terminates. If the connection is successful, the program queries phenodcc_media.media_file for media submitted for each of the three "embryo reconstruction" parameters; IMPC_EOL_001_001 (OPT E9.5), IMPC_EMO_001_001 (uCT E14.5/15.5) and IMPC_EMA_001_001 (uCT E18.5).

For each of the rows returned, its unique URL is used to determine where it has already processed:

```
SELECT * FROM phenodcc_embryo.preprocessed WHERE url = "/the/url/123456.bz2"
```

If URL already exists in phenodcc_embryo.preprocessed:

1. Check its status ID and extension ID
2. If status_id != 1 (success), add job to the pre-processing list
3. Otherwise, update its metadataGroup and measurement_id.

Otherwise:

1. Create a new row in the pre-processing table
2. Add job to the pre-processing list.

Once all of the parameters have been searched against, the process_recons method is called.

CHAPTER

TWO

SLICEGENERATOR MODULE

The SliceGenerator class is an "abstract" superclass designed to be extended for specific file formats.

Subclasses call the super constructor and override the slices() that yields image slices when used as a generator. The class should also override the dtype() and shape() methods accordingly.

Instances of a subclass should be initialised by passing a valid file path as an argument. Slices may then be generated using a for-loop, calling the slices() method on the generator object.

2.1 Requirements

- h5py
- Numpy 1.8.2
- ctutils 0.2

2.2 Examples

```
>>> from SliceGenerator import NrrdSliceGenerator
>>> gen = NrrdSliceGenerator('path/to/file.nrrd')
>>> for x in gen.slices():
...     print x.shape  # do something with the slice
```

class SliceGenerator.**MincSliceGenerator**(*recon*)

Bases: SliceGenerator.SliceGenerator

The MincSliceGenerator class extends SliceGenerator, yielding slices from a single MINC (Medical Image NetCDF) file.

MINC files (.mnc) are based on the Hierarchical Data (HDF5) Format, and so they are currently handled by the h5py module (https://github.com/h5py/h5py).

dtype()

Overrides the superclass to return the data type of the MINC file i.e. 8 bit/16 bit.

shape()

Overrides the superclass to return the shape of the MINC file.

slices(*start=0*)

Slices are yielded one slice at a time from the memory mapped numpy array

class SliceGenerator.**NrrdSliceGenerator**(*recon*)
Bases: SliceGenerator.SliceGenerator

The NrrdSliceGenerator class extends SliceGenerator, yielding slices from a single NRRD (Nearly Raw Raster Data) file.

NRRDs are the most likely file type to be received at the DCC, especially for those centres that have adopted HARP. This generator has gone through several iterations, and now relies on Utah Nrrd Utilities (unu) by teem (http://teem.sourceforge.net/index.html).

dtype()
Overrides the superclass to return the data type of the NRRD file i.e. 8 bit/16 bit.

parse_header(*hdr*)
The parse_header method reads a NRRD header and extracts the data type, shape and encoding of the data.

Parameters **hdr** – path to a header (.hdr) file

shape()
Overrides the superclass to return the shape of the NRRD file.

slices(*start=0*)
Slices are yielded one slice at a time from the memory mapped NRRD file.

class SliceGenerator.**SliceGenerator**(*recon*)
Bases: object

dtype()
The dtype method should return the datatype of the memory mapped numpy array

shape()
The shape method should return the shape of the memory mapped numpy array in x, y, z order.

slices()
The slices method should yield xy image slices from a memory mapped numpy array.

class SliceGenerator.**TXMSliceGenerator**(*recon*)
Bases: SliceGenerator.SliceGenerator

The TXMSliceGenerator class extends SliceGenerator, yielding slices from TXM file (Transmission X-ray Microscopy).

This generator has not been tested comprehensively, as we only have access to one TXM file for testing. It makes use of the ctutils module (https://github.com/waveform80/ctutils)

dtype()
Overrides the superclass to return the data type of the TXM file i.e. 8 bit/16 bit.

shape()
Overrides the superclass to return the shape of the TXM file.

slices(*start=0*)
Slices are yielded one slice at a time from the TXM file.

class SliceGenerator.**TiffSliceGenerator**(*recon*)
Bases: SliceGenerator.SliceGenerator

The TiffSliceGenerator class extends SliceGenerator, yielding slices from a folder of TIFFs. The method uses the tifffile.py module, created by Christoph Gohlke.

This class is unlikely to ever be needed, as we should not receive folders of TIFFs from IMPC centres.

dtype()
Overrides the superclass to return the data type of the TIFF files i.e. 8 bit/16 bit.

shape()
Overrides the superclass to return the shape of the TIFF files as a volume.

slices(*start=0*)
Slices are yielded one slice at a time from the list of TIFFs.

class SliceGenerator.**TiffStackSliceGenerator**(*recon*)
Bases: SliceGenerator.SliceGenerator

The TiffStackSliceGenerator class extends SliceGenerator, yielding slices from a single TIFF stack.

In the unlikely event that we receive a TIFF stack from an IMPC centre, this generator will slice it for resampling. At present, TIFF stacks CANNOT be memory mampped, and will ne loaded into memory. The method uses the tifffile.py module, created by Christoph Gohlke.

dtype()
Overrides the superclass to return the data type of the TIFF stack i.e. 8 bit/16 bit.

shape()
Overrides the superclass to return the shape of the TIFF stack.

slices(*start=0*)
Slices are yielded one slice at a time from the TIFF stack.

CHAPTER

THREE

CONVERSION MODULE

conversion.**decompress_bz2**(*bz2_in*, *decompressed_out*)

The decompress_bz2 method performs sequential decompression of bzipped files on disk.

If the bz2_in path can be found, the file is open and read in chunks of ~100 Mb at a time. Each chunk is then decompressed, and written to the file specified by decompressed_out. A progress bar is used to indicate how the decompression is getting on.

Parameters

- **bz2_in** – path to bzipped file to be decompressed
- **decompressed_out** – path to output file, which can already exist

Raises IOError the decompressed file could not be found/opened

conversion.**write_xtk_nrrd**(*volume*, *nrrd_out*)

The write_xtk_nrrd method writes numpy arrays as IEV-ready NRRD files. It also works on memory mapped arrays.

IEV works using the X Toolkit (XTK), which is quite particular about the NRRD files it displays. This method ensures that the NRRD headers are written appropriately, using nrrd.py by Maarten Everts (https://github.com/mhe/pynrrd/blob/master/nrrd.py)

Parameters

- **volume** – a numpy array in memory, or a memory mapped numpy array
- **nrrd_out** – a file path to which the NRRD file is written

Raises IOError unable to write file to disk

CHAPTER

FOUR

NRRD MODULE

nrrd.py An all-python (and numpy) implementation for reading and writing nrrd files. See http://teem.sourceforge.net/nrrd/format.html for the specification.

exception nrrd.**NrrdError**
Bases: exceptions.Exception

Exceptions for Nrrd class.

nrrd.**parse_nrrdvector**(*inp*)
Parse a vector from a nrrd header, return a list.

nrrd.**parse_optional_nrrdvector**(*inp*)
Parse a vector from a nrrd header that can also be none.

nrrd.**read**(*filename*)
Read a nrrd file and return a tuple (data, header).

nrrd.**read_data**(*fields*, *filehandle*, *filename=None*)
Read the actual data into a numpy structure.

nrrd.**read_header**(*nrrdfile*)
Parse the fields in the nrrd header

nrrdfile can be any object which supports the iterator protocol and returns a string each time its next() method is called — file objects and list objects are both suitable. If csvfile is a file object, it must be opened with the 'b' flag on platforms where that makes a difference (e.g. Windows)

```
>>> read_header(("NRRD0005", "type: float", "dimension: 3"))
{'type': 'float', 'dimension': 3, 'keyvaluepairs': {}}
>>> read_header(("NRRD0005", "my extra info:=my : colon-separated : values"))
{'keyvaluepairs': {'my extra info': 'my : colon-separated : values'}}
```

nrrd.**write**(*filename*, *data*, *options={}*, *separate_header=False*)
Write the numpy data to a nrrd file. The nrrd header values to use are inferred from from the data. Additional options can be passed in the options dictionary. See the read() function for the structure of this dictionary.

To set data samplings, use e.g. *options['spacings'] = [s1, s2, s3]* for 3d data with sampling deltas *s1*, *s2*, and *s3* in each dimension.

CHAPTER

FIVE

RESAMPLER MODULE

Resampler for the embryo pre-processing script.

This module performs resampling of images of arbitrary size, using a slice generator and memory mapped raw files.

5.1 Requirements

- OpenCV 2.4.9
- Numpy 1.8.2
- python-progressbar 2.3

5.2 Examples

```
>>> from resampler import resample
>>> from SliceGenerator import NrrdSliceGenerator
>>> gen = NrrdSliceGenerator('/path/to/file.nrrd')
>>> resample(gen, 0.5, '/path/to/rescaled.nrrd')
```

resampler.**resample**(*slicegen*, *scale*, *nrrd_path*)

The resample method takes a slice generator and scaling factor as arguments, resamples the image accordingly, and writes it to disk as an IEV-ready NRRD file.

Parameters

- **slicegen** – a slice generator that provides xy slices of an image as two-dimensional numpy arrays
- **scale** – scaling factor for resampling, which should be < 1 for downscaling (e.g. 0.5)
- **nrrd_path** – outfile NRRD containing extension

CHAPTER

SIX

TEST MODULE

CHAPTER

SEVEN

TIFFFILE MODULE

Read and write image data from and to TIFF files. Image and meta-data can be read from TIFF, BigTIFF, OME-TIFF, STK, LSM, NIH, ImageJ, MicroManager, FluoView, SEQ and GEL files. Only a subset of the TIFF specification is supported, mainly uncompressed and losslessly compressed 2**(0 to 6) bit integer, 16, 32 and 64-bit float, grayscale and RGB(A) images, which are commonly used in bio-scientific imaging. Specifically, reading JPEG/CCITT compressed image data or EXIF/IPTC/GPS/XMP meta-data is not implemented. Only primary info records are read for STK, FluoView, MicroManager, and NIH image formats. TIFF, the Tagged Image File Format, is under the control of Adobe Systems. BigTIFF allows for files greater than 4 GB. STK, LSM, FluoView, SEQ, GEL, and OME-TIFF, are custom extensions defined by MetaMorph, Carl Zeiss MicroImaging, Olympus, Media Cybernetics, Molecular Dynamics, and the Open Microscopy Environment consortium respectively. For command line usage run `python tifffile.py --help` :Author:

Christoph Gohlke

Organization Laboratory for Fluorescence Dynamics, University of California, Irvine

Version 2014.02.05

7.1 Requirements

- CPython 2.7 or 3.3
- Numpy 1.7
- Matplotlib 1.3 (optional for plotting)
- Tifffile.c 2013.01.18 (recommended for faster decoding of PackBits and LZW encoded strings)

7.2 Notes

The API is not stable yet and might change between revisions. Tested on little-endian platforms only. Other Python packages and modules for reading bio-scientific TIFF files: * Imread * PyLibTiff * SimpleITK * PyLSM * PyMca.TiffIO.py * BioImageXD.Readers * Cellcognition.io * CellProfiler.bioformats Acknowledgements ———— * Egor Zindy, University of Manchester, for cz_lsm_scan_info specifics. * Wim Lewis for a bug fix and some read_cz_lsm functions. * Hadrien Mary for help on reading MicroManager files. References ———- (1) TIFF 6.0 Specification and Supplements. Adobe Systems Incorporated.

http://partners.adobe.com/public/developer/tiff/

2. TIFF File Format FAQ. http://www.awaresystems.be/imaging/tiff/faq.html
3. MetaMorph Stack (STK) Image File Format. http://support.meta.moleculardevices.com/docs/t10243.pdf
4. File Format Description - LSM 5xx Release 2.0. http://ibb.gsf.de/homepage/karsten.rodenacker/IDL/Lsmfile.doc

5. BioFormats. http://www.loci.wisc.edu/ome/formats.html
6. The OME-TIFF format. http://www.openmicroscopy.org/site/support/file-formats/ome-tiff
7. TiffDecoder.java http://rsbweb.nih.gov/ij/developer/source/ij/io/TiffDecoder.java.html
8. UltraQuant(r) Version 6.0 for Windows Start-Up Guide. http://www.ultralum.com/images%20ultralum/pdf/UQStart%20Up%20G
9. Micro-Manager File Formats. http://www.micro-manager.org/wiki/Micro-Manager_File_Formats

7.3 Examples

```
>>> data = numpy.random.rand(301, 219)
>>> imsave('temp.tif', data)
>>> image = imread('temp.tif')
>>> assert numpy.all(image == data)
>>> tif = TiffFile('test.tif')
>>> images = tif.asarray()
>>> image0 = tif[0].asarray()
>>> for page in tif:
...     for tag in page.tags.values():
...         t = tag.name, tag.value
...     image = page.asarray()
...     if page.is_rgb: pass
...     if page.is_palette:
...         t = page.color_map
...     if page.is_stk:
...         t = page.mm_uic_tags.number_planes
...     if page.is_lsm:
...         t = page.cz_lsm_info
>>> tif.close()
```

tifffile.**imsave**(*filename*, *data*, *photometric=None*, *planarconfig=None*, *resolution=None*, *description=None*, *software='tifffile.py'*, *byteorder=None*, *bigtiff=False*, *compress=0*, *extratags=()*)

Write image data to TIFF file. Image data are written in one stripe per plane. Dimensions larger than 2 or 3 (depending on photometric mode and planar configuration) are flattened and saved as separate pages. The 'sample_format' and 'bits_per_sample' TIFF tags are derived from the data type. Parameters ———- filename : str

Name of file to write.

data [array_like] Input image. The last dimensions are assumed to be image height, width, and samples.

photometric [{'minisblack', 'miniswhite', 'rgb'}] The color space of the image data. By default this setting is inferred from the data shape.

planarconfig [{'contig', 'planar'}] Specifies if samples are stored contiguous or in separate planes. By default this setting is inferred from the data shape. 'contig': last dimension contains samples. 'planar': third last dimension contains samples.

resolution [(float, float) or ((int, int), (int, int))] X and Y resolution in dots per inch as float or rational numbers.

description [str] The subject of the image. Saved with the first page only.

software [str] Name of the software used to create the image. Saved with the first page only.

byteorder [{'<', '>'}] The endianness of the data in the file. By default this is the system's native byte order.

bigtiff [bool] If True, the BigTIFF format is used. By default the standard TIFF format is used for data less than 2000 MB.

compress [int] Values from 0 to 9 controlling the level of zlib compression. If 0, data are written uncompressed (default).

extratags: sequence of tuples Additional tags as [(code, dtype, count, value, writeonce)]. code : int

The TIFF tag Id.

dtype [str] Data type of items in *value* in Python struct format. One of B, s, H, I, 2I, b, h, i, f, d, Q, or q.

count [int] Number of data values. Not used for string values.

value [sequence] *Count* values compatible with *dtype*.

writeonce [bool] If True, the tag is written to the first page only.

```
>>> data = numpy.ones((2, 5, 3, 301, 219), 'float32') * 0.5
>>> imsave('temp.tif', data, compress=6)
>>> data = numpy.ones((5, 301, 219, 3), 'uint8') + 127
>>> value = u'{"shape": %s}' % str(list(data.shape))
>>> imsave('temp.tif', data, extratags=[(270, 's', 0, value, True)])
```

tifffile.**imread**(*files*, **args*, ***kwargs*)

Return image data from TIFF file(s) as numpy array. The first image series is returned if no arguments are provided. Parameters ———- files : str or list

File name, glob pattern, or list of file names.

key [int, slice, or sequence of page indices] Defines which pages to return as array.

series [int] Defines which series of pages in file to return as array.

multifile [bool] If True (default), OME-TIFF data may include pages from multiple files.

pattern [str] Regular expression pattern that matches axes names and indices in file names.

```
>>> im = imread('test.tif', 0)
>>> im.shape
(256, 256, 4)
>>> ims = imread(['test.tif', 'test.tif'])
>>> ims.shape
(2, 256, 256, 4)
```

tifffile.**imshow**(*data*, *title=None*, *vmin=0*, *vmax=None*, *cmap=None*, *bitspersample=None*, *photometric='rgb'*, *interpolation='nearest'*, *dpi=96*, *figure=None*, *subplot=111*, *maxdim=8192*, ***kwargs*)

Plot n-dimensional images using matplotlib.pyplot. Return figure, subplot and plot axis. Requires pyplot already imported `from matplotlib import pyplot`. Parameters ———- bitspersample : int or None

Number of bits per channel in integer RGB images.

photometric [{'miniswhite', 'minisblack', 'rgb', or 'palette'}] The color space of the image data.

title [str] Window and subplot title.

figure [matplotlib.figure.Figure (optional).] Matplotlib to use for plotting.

subplot [int] A matplotlib.pyplot.subplot axis.

maxdim [int] maximum image size in any dimension.

kwargs [optional] Arguments for matplotlib.pyplot.imshow.

class tifffile.**TiffFile**(*arg*, *name=None*, *multifile=False*)
Bases: object

Read image and meta-data from TIFF, STK, LSM, and FluoView files. TiffFile instances must be closed using the close method, which is automatically called when using the 'with' statement. Attributes ———- pages : list

All TIFF pages in file.

series [list of Records(shape, dtype, axes, TiffPages)] TIFF pages with compatible shapes and types.

micromanager_metadata: dict Extra MicroManager non-TIFF metadata in the file, if exists.

All attributes are read-only. Examples ——— >>> tif = TiffFile('test.tif') ... try: ... images = tif.asarray() ... except Exception as e: ... print(e) ... finally: ... tif.close()

asarray(*key=None*, *series=None*, *memmap=False*)
Return image data of multiple TIFF pages as numpy array. By default the first image series is returned. Parameters ———- key : int, slice, or sequence of page indices

Defines which pages to return as array.

series [int] Defines which series of pages to return as array.

memmap [bool] If True, use numpy.memmap to read arrays from file if possible.

close()
Close open file handle(s).

fstat
Lazy object attribute whose value is computed on first access.

is_bigtiff
Lazy object attribute whose value is computed on first access.

is_fluoview
Lazy object attribute whose value is computed on first access.

is_imagej
Lazy object attribute whose value is computed on first access.

is_lsm
Lazy object attribute whose value is computed on first access.

is_mdgel
Lazy object attribute whose value is computed on first access.

is_mediacy
Lazy object attribute whose value is computed on first access.

is_micromanager
Lazy object attribute whose value is computed on first access.

is_nih
Lazy object attribute whose value is computed on first access.

is_ome
Lazy object attribute whose value is computed on first access.

is_palette
Lazy object attribute whose value is computed on first access.

is_rgb
Lazy object attribute whose value is computed on first access.

is_stk
Lazy object attribute whose value is computed on first access.

series
Lazy object attribute whose value is computed on first access.

class tifffile.**TiffSequence**(*files*, *imread=<class 'tifffile.TiffFile'>*, *pattern='axes'*)
Bases: object

Sequence of image files. Properties ———- files : list

List of file names.

shape [tuple] Shape of image sequence.

axes [str] Labels of axes in shape.

```
>>> ims = TiffSequence("test.oif.files/*.tif")
>>> ims = ims.asarray()
>>> ims.shape
(2, 100, 256, 256)
```

asarray(**args*, ***kwargs*)
Read image data from all files and return as single numpy array. Raise IndexError if image shapes don't match.

close()

CHAPTER

EIGHT

INDICES AND TABLES

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

c

e

n

r

s

t

T

W