bfio

Release 2.3.0

Nicholas J Schaub

Aug 10, 2022

CONTENTS:

The `bfio` utility is an easy to use, image input/output utility optimized to scalably read and write OME TIFF and OME Zarr images as well as all formats supported by Bioformats . Reading of data makes direct usage of BioFormats (when using the *bioformats* install option) using JPype, giving `bfio` the ability to read any of the 150+ formats supported by BioFormats.

For file outputs, only two file formats are supported: tiled OME TIFF, and OME Zarr. These two file formats are supported because they are scalable, permitting tiled writing. This package was created out of a need for speed and scalability so that plugins could be created for the Web Image Processing Pipeline (WIPP).

`bfio` has a simple to use interface that allows programs to access images as if they were memory mapped numpy arrays operating on the original data file. There are a lot of caveats to data reading and writing, but the goal of this package was to lower the barrier to working with image data that may not fit into memory.

CHAPTER

ONE

INSTALL

The `bfio` package can be installed with a variety of options designed in consideration of containerization.

1.1 Base Package

The base package can only read/write tiled, OME TIFF images. It can be installed using:

```
pip install bfio
```

To install all reading and writing options, install with all dependencies:

```
pip install bfio[all]
```

NOTE: See the Java and Bioformats note about licensing considerations when installing using the *bfio[all]* option.

1.2 Java and Bioformats

To use Bioformats, it is necessary to install Java 8 or later. Once Java is installed, `bfio` can be installed with support for Java using:

pip install bfio['bioformats']

NOTE: The *bioformats_jar* package and BioFormats are licensed under GPL, while *bfio* is licensed under MIT. This may have consequences when packaging any software that uses *bfio* as a dependency when this option is used.

1.3 Zarr

To use the zarr backend, you need to have Zarr already installed, or you can install it when installing bfio using:

pip install bfio['zarr']

CHAPTER

TWO

EXAMPLES

2.1 Create a Tiled Tiff Converter

2.1.1 Introduction

The bfio package is designed to make it easy to process arbitrarily sized images in a fast, scalable way. The two core classes, *bfio.BioReader* and *bfio.BioWriter*, can use one of three different backends depending on the file type that will be read. Usually, the proper backend will be selected when opening a file, but periodically it will not select the proper backend so it is useful to mention them here:

1. `backend="python"` can only be used to read/write OME tiled tiff images. Tiled tiff is the preferred file format for reading/writing arbitrarily sized images.
2. `backend="bioformats"` can be used to read any any format supported by Bioformats. The BioWriter with java backend will only save images as OME tiled tiff.
3. `backend="zarr"` can be used to read/write a subset of Zarr files following the OME Zarr spec..

The advantage to using the `python` and `zarr` backends are speed and scalability at the expense of a rigid file structure, while the `bioformats` backend provides broad access to a wide array of file types but is considerably slower.

In this example, the basic useage of two core classes are demonstrated by creating a "universal" converter from any Bioformats supported file format to OME tiled tiff.

2.1.2 Getting Started

Install Dependencies

To run this example, a few dependencies are required. First, ensure Java is installed. Then install `bfio` using:

```
pip install bfio['bioformats']
```

Note: JPype and Bioformats require Java 8 or later. `bfio` has been tested using Java 8.

Note: Bioformats is licensed under GPL. If you plan to package `bfio` using this option, make sure to consider the licensing ramifications.

Download a Test Image

Download a test image from the Web Image Processing Pipeline (WIPP) github set of test images.

```
from pathlib import Path
import requests

""" Get an example image """
# Set up the directories
PATH = Path("data")
PATH.mkdir(parents=True, exist_ok=True)

# Download the data if it doesn't exist
URL = "https://github.com/usnistgov/WIPP/raw/master/data/PyramidBuilding/inputCollection/
↪"
FILENAME = "img_r001_c001.ome.tif"
if not (PATH / FILENAME).exists():
        content = requests.get(URL + FILENAME).content
        (PATH / FILENAME).open("wb").write(content)
```

2.1.3 Reading Images with the BioReader

The first step in loading an image with the BioReader is to initialize a BioReader object. The BioReader tries to predict what backend to use based on the file extension, and it will throw an error if it predicts incorrectly. In this case, the test file downloaded from the WIPP repository is not in OME tiled tiff format even though it has the `.ome.tif` extension, meaning it will try to use the Python backend:

```
from bfio import BioReader

br = BioReader(PATH / FILENAME)
```

The output should be something like this:

```
TypeError: img_r001_c001.ome.tif is not a tiled tiff. The python backend of
the BioReader only supports OME tiled tiffs. Use the bioformats backend to load
this image.
```

Manually Specifying a Backend

Following the error message above, we need to use the `bioformats` backend. To do this, just use the keyword argument `backend=bioformats` when initializing the BioReader.

```
# Set up the BioReader
br = BioReader(PATH / FILENAME,backend='bioformats')

# Print off some information about the image before loading it
print('br.shape: {}'.format(br.shape))
print('br.dtype: {}'.format(br.dtype))

br.close()
```

Behind the scenes, what happens is that JPype starts Java and loads the Bioformats jar. If Java is not installed, the above code will raise an error.

Note: Currently, the mechanism for starting Java is likely not thread safe.

Using the BioReader

In the above code, a `BioReader` object is initialized, the shape and data type is printed, and then the BioReader object is closed. The closing of the `BioReader` object is necessary to ensure that the Java object is cleaned up properly. To ensure that this happens, it is recommended to put image access into a `with` block, which will automatically perform file cleanup.

```
from bfio import BioReader

# Initialize the BioReader inside a ``with`` block to handle file cleanup
with BioReader(PATH / FILENAME,backend='bioformats') as br:

    # Print off some information about the image before loading it
    print('br.shape: {}'.format(br.shape))
    print('br.dtype: {}'.format(br.dtype))
```

To read an entire image, use the *read* method without any arguments.

```
I = br.read()
```

Alternatively, the *X*, *Y*, *Z*, *C*, and *T* values can be specified to load only a subsection of the image. If the BioReader is reading from an OME tiled tiff, then the file reading should be faster and require less memory than other formats. This has to do with how data is stored in the OME tiled tiff.

For the current file, to load only the first 100x100 pixels:

```
I = br.read(X=[0,100],Y=[0,100])
```

The above code will return a 5-dimensional numpy array with `shape=(100,100,1,1,1)`. If this file had multiple z-slices, channels, or timepoint information stored in it, then the first 100x100 pixels in every z-slice, channel, and timepoint would all be loaded since Z, C, and T were not included as keyword arguments.

To make it easier to load data, data can be fetch from a file using NumPy like indexing. However, there are some caveats. Step sizes in slices are ignored for the first three indices. Thus, the following three lines of code will load data exactly the same as the above line using `read` to load the first 100 rows and columns of pixels:

```
I = br[0:100,0:100,:,:,:]
I = br[:100,:100,...]
I = br[:100:2,:100:2]

print(I.shape) # Should return (100,100)
```

Note: When using NumPy like indexing, trailing dimensions with size=1 are squeezed. So, while `read` will always return a 5-dimensional array, the NumPy indexing in this case will return a 2-dimensional array since there are no Z, C, or T dimensions.

2.1.4 Writing Images With the BioWriter

Initializing the BioWriter

Initializing the *bfio.BioWriter* requires a little more thought than the `BioReader` because the properties of the file have to be set prior to writing any data. In many cases, starting the BioWriter with the same metadata as the BioReader will get you most of the way there.

```
from bfio import BioWriter

bw = BioWriter(PATH / 'out.ome.tif',metadata=br.metadata)
```

The above code copies all the metadata from a BioReader object to the BioWriter object. If the data type needs to be changed for the file, simply set the object property.

```
bw.dtype = np.uint8 # Must be a numpy data type
bw.X = 1000 # set the image width
bw.Y = 500  # set the image height
bw.channel_names = ['PI','phalloidin','DAPI'] # if your image has three channels, name␣
↪each of them
```

For more information on the settable properties, see the *bfio.BioWriter* documentation.

Writing the Image

As with the BioReader, the BioWriter needs to be properly closed using the *close* method. Closing the BioWriter finalizes the file, and if code exits without a file being close then the image may not open properly. To help prevent this scenario, use a `with` block.

```
with BioWriter(PATH / 'out.ome.tif',metadata=br.metadata) as bw:

    original_image = br[:]
    bw.write(original_image)
```

This code reads an image and savess it as an OME tiled tiff!

As with the BioReader, it is possible to use numpy-like indexing. An alternative to the above code block would be:

```
with BioWriter(PATH / 'out.ome.tif',metadata=br.metadata) as bw:

    bw[:] = br[:]
```

Note: After the first `write` call, most BioWriter attributes become *read_only*.

2.1.5 An Efficient, Scalable, Tiled Tiff Converter

In the above example, the demo image was relatively small, so opening the entire image and saving it was trivial. However, the `bfio` classes can be used to to convert an arbitrarily large image on a resource constrained system. This is done by reading/writing images in subsections and controlling the number of threads used for processing. Both the BioReader and BioWriter use multiple threads to read/write data, one thread per tile on individual tiles. By default, the number of threads is half the number of detected CPU cores, and this can be changed when a BioReader or BioWriter object is created by using the `max_workers` keyword argument.

To get started, let's transform the previous examples into something more scalable. Something more scalable will read in a small part of one image, and save it into the tiled tiff format.

Note: The BioWriter always saves images in 1024x1024 tiles. So, it is important to save images in multiples of 1024 (height or width) in order for the image to save correctly. In the future, the tiled tiff tile size may become a user defined parameter, but for now the WIPP OME tiled tiff standard of 1024x1024 tile size is used exclusively.

```
# Number of tiles to process at a time
# This value squared is the total number of tiles processed at a time
tile_grid_size = 1

# Do not change this, the number of pixels to be saved at a time must
# be a multiple of 1024
tile_size = tile_grid_size * 1024

with BioReader(PATH / 'file.czi',backend='bioformats') as br, \
    BioWriter(PATH / 'out.ome.tif',backend='bioformats',metadata=br.metadata) as bw:

    # Loop through timepoints
    for t in range(br.T):

        # Loop through channels
        for c in range(br.C):

            # Loop through z-slices
            for z in range(br.Z):

                # Loop across the length of the image
                for y in range(0,br.Y,tile_size):
                    y_max = min([br.Y,y+tile_size])

                    # Loop across the depth of the image
                    for x in range(0,br.X,tile_size):
                        x_max = min([br.X,x+tile_size])

                        bw[y:y_max,x:x_max,z:z+1,c,t] = br[y:y_max,x:x_max,z:z+1,c,t]
```

The above code has a lot of for loops. What makes the above code more scalable than just a simple piece of code like `bw[:] = br[:]`? The for loops and the `tile_size` variable make it so that only a small portion of the image is loaded into memory. In the above code, `tile_grid_size = 1`, meaning that individual tiles are being stored one by one, which is the most memory efficient way of converting to tiled tiff.

One thing to note in the above example is that both the BioReader and BioWriter are using the Java backend. This ensures a direct, 1-to-1 file conversion can take place. While the Python backend for both the BioReader and BioWriter

can read OME TIFF files with any number of XYZCT dimensions, the WIPP platform expects each file to only contain XYZ data. To make the above tiled tiff converter export WIPP compliant files, the code should be changed as follows:

```
# Number of tiles to process at a time
# This value squared is the total number of tiles processed at a time
tile_grid_size = 1

# Do not change this, the number of pixels to be saved at a time must
# be a multiple of 1024
tile_size = tile_grid_size * 1024

with BioReader(PATH / 'file.czi',backend='bioformats') as br:

    # Loop through timepoints
    for t in range(br.T):

        # Loop through channels
        for c in range(br.C):

            with BioWriter(PATH / 'out_c{c:03d}_t{t:03d}.ome.tif',
                           backend='bioformats',
                           metadata=br.metadata) as bw:
                bw.C = 1
                bw.T = 1

                # Loop through z-slices
                for z in range(br.Z):

                    # Loop across the length of the image
                    for y in range(0,br.Y,tile_size):
                        y_max = min([br.Y,y+tile_size])

                        # Loop across the depth of the image
                        for x in range(0,br.X,tile_size):
                            x_max = min([br.X,x+tile_size])

                            bw[y:y_max,x:x_max,z:z+1,0,0] = br[y:y_max,x:x_max,z:z+1,c,t]
```

2.1.6 Complete Example Code

Self Contained Example

```
from bfio import BioReader, BioWriter
from pathlib import Path
import requests
import numpy as np

""" Get an example image """
# Set up the directories
PATH = Path("data")
PATH.mkdir(parents=True, exist_ok=True)
```

(continues on next page)

(continued from previous page)

```
# Download the data if it doesn't exist
URL = "https://github.com/usnistgov/WIPP/raw/master/data/PyramidBuilding/inputCollection/
↪"
FILENAME = "img_r001_c001.ome.tif"
if not (PATH / FILENAME).exists():
    content = requests.get(URL + FILENAME).content
    (PATH / FILENAME).open("wb").write(content)

""" Convert the tif to tiled tiff """
# Set up the BioReader
with BioReader(PATH / FILENAME,backend='bioformats') as br, \
    BioWriter(PATH / 'out.ome.tif',metadata=br.metadata,backend='python') as bw:

    # Print off some information about the image before loading it
    print('br.shape: {}'.format(br.shape))
    print('br.dtype: {}'.format(br.dtype))

    # Read in the original image, then save
    original_image = br[:]
    bw[:] = original_image

# Compare the original and saved images using the Python backend
br = BioReader(PATH.joinpath('out.ome.tif'))

new_image = br.read()

br.close()

print('original and saved images are identical: {}'.format(np.array_equal(new_image,
↪original_image)))
```

Scalable Tiled Tiff

```
from bfio import BioReader, BioWriter
import math
from pathlib import Path
from multiprocessing import cpu_count

""" Define the path to the file to convert """
# Set up the directories
PATH = Path("path/to/file").joinpath('file.tif')


""" Convert the tif to tiled tiff """
# Number of tiles to process at a time
# This value squared is the total number of tiles processed at a time
tile_grid_size = math.ceil(math.sqrt(cpu_count()))

# Do not change this, the number of pixels to be saved at a time must
# be a multiple of 1024
```

(continues on next page)

(continued from previous page)

```
tile_size = tile_grid_size * 1024

# Set up the BioReader
with BioReader(PATH,backend='bioformats',max_workers=cpu_count()) as br:

    # Loop through timepoints
    for t in range(br.T):

        # Loop through channels
        for c in range(br.C):

            with BioWriter(PATH.with_name(f'out_c{c:03}_t{t:03}.ome.tif'),
                           backend='python',
                           metadata=br.metadata,
                           max_workers = cpu_count()) as bw:

                # Loop through z-slices
                for z in range(br.Z):

                    # Loop across the length of the image
                    for y in range(0,br.Y,tile_size):
                        y_max = min([br.Y,y+tile_size])

                        # Loop across the depth of the image
                        for x in range(0,br.X,tile_size):
                            x_max = min([br.X,x+tile_size])

                            bw[y:y_max,x:x_max,z:z+1,0,0] = br[y:y_max,x:x_max,z:z+1,c,t]
```

CHAPTER

THREE

REFERENCE

3.1 bfio.BioReader

class BioReader(*file_path*, *max_workers=None*, *backend=None*, *clean_metadata=True*)

Bases: BioBase

Read supported image formats using Bioformats.

This class handles file reading of multiple formats. It can read files from any Bioformats supported file format, but is specially optimized for handling the OME tiled tiff format.

There are three backends: `bioformats`, `python`, and `zarr`. The `bioformats` backend directly uses Bio-Formats for file reading, and can read any forma that is supported by Bio-Formats. The `python` backend will only read images in OME Tiff format with tile tags set to 1024x1024, and is significantly faster than the "bioformats" backend for reading these types of tiff files. The `zarr` backend will only read OME Zarr files.

File reading and writing are multi-threaded by default, except for the `bioformats` backend which does not currently support threading. Half of the available CPUs detected by multiprocessing.cpu_count() are used to read an image.

For for information, visit the Bioformats page: https://www.openmicroscopy.org/bio-formats/

Note: In order to use the `bioformats` backend, jpype must be installed.

Initialize the BioReader.

Parameters

- **file_path** (Union[str, Path]) – Path to file to read
- **max_workers** (Optional[int]) – Number of threads used to read and image. *Default is half the number of detected cores.*
- **backend** (Optional[str]) – Can be `python`, `bioformats`, or `zarr`. If None, then BioReader will try to autodetect the proper backend. *Default is python.*
- **clean_metadata** (bool) – Will try to reformat poorly formed OME XML metadata if True. If False, will throw an error if the metadata is poorly formed. *Default is True.*

property X

Setter
X is *read_only* in *BioReader*

Getter
Number of pixels in the x-dimension (width)

Type
int

property Y

Setter
Y is *read_only* in *BioReader*

Getter
Number of pixels in the y-dimension (height)

Type
int

property Z

Setter
Z is *read_only* in *BioReader*

Getter
Number of pixels in the z-dimension (depth)

Type
int

property C

Setter
C is *read_only* in *BioReader*

Getter
Number of pixels in the c-dimension

Type
int

property T

Setter
T is *read_only* in *BioReader*

Getter
Number of pixels in the t-dimension

Type
int

__getitem__(*keys*)

Image loading using numpy-like indexing.

This is an abbreviated method of accessing the *read* method, where a portion of the image will be loaded using numpy-like slicing syntax. Up to 5 dimensions can be designated depending on the number of available dimensions in the image array (Y, X, Z, C, T).

Note: Not all methods of indexing can be used, and some indexing will lead to unexpected results. For example, logical indexing cannot be used, and step sizes in slice objects is ignored for the first three indices. This means and index such as `[0:100:2,0:100:2,0,0,0]` will return a 100x100x1x1x1 numpy array.

Parameters
keys (Union[tuple, slice]) – numpy-like slicing used to load a section of an image.

Return type
ndarray

Returns
A numpy.ndarray where trailing empty dimensions are removed.

Example

```
import bfio

# Initialize the bioreader
br = bfio.BioReader('Path/To/File.ome.tif')

# Load and copy a 100x100 array of pixels
a = br[:100,:100,:1,0,0]

# Slice steps sizes are ignored for the first 3 indices, so this
# returns the same as above
a = br[0:100:2,0:100:2,0:1,0,0]

# The last two dimensions can receive a tuple or list as input
# Load the first and third channel
a = br[:100,100,0:1,(0,2),0]

# If the file is 3d, load the first 10 z-slices
b = br[...,:10,0,0]
```

read(*X=None*, *Y=None*, *Z=None*, *C=None*, *T=None*)

Read the image.

Read the all or part of the image. A n-dimmensional numpy.ndarray is returned such that all trailing empty dimensions will be removed.

For example, if an image is read and it represents an xz plane, then the shape will be [1,m,n].

Parameters

- **X** (Union[list, tuple, None]) – The (min,max) range of pixels to load along the x-axis (columns). If None, loads the full range. *Defaults to None.*
- **Y** (Union[list, tuple, None]) – The (min,max) range of pixels to load along the y-axis (rows). If None, loads the full range. *Defaults to None.*
- **Z** (Union[list, tuple, int, None]) – The (min,max) range of pixels to load along the z-axis (depth). Alternatively, an integer can be passed to select a single z-plane. If None, loads the full range. *Defaults to None.*
- **C** (Union[list, tuple, int, None]) – Values indicating channel indices to load. If None, loads the full range. *Defaults to None.*
- **T** (Union[list, tuple, int, None]) – Values indicating timepoints to load. If None, loads the full range. *Defaults to None.*

Return type
ndarray

Returns
A 5-dimensional numpy array.

__call__(*tile_size*, *tile_stride=None*, *batch_size=None*, *channels=[0]*)

Iterate through tiles of an image.

The BioReader object can be called, and will act as an iterator to load tiles of an image. The iterator buffers the loading of pixels asynchronously to quickly deliver images of the appropriate size.

Parameters

- **tile_size** (Union[list, tuple]) – A list/tuple of length 2, indicating the height and width of the tiles to return.
- **tile_stride** (Union[list, tuple, None]) – A list/tuple of length 2, indicating the row and column stride size. If None, then tile_stride = tile_size. *Defaults to None.*
- **batch_size** (Optional[int]) – Number of tiles to return on each iteration. *Defaults to None, which is the smaller of 32 or the maximum_batch_size*
- **channels** (List[int]) – A placeholder. Only the first channel is ever loaded. *Defaults to [0].*

Return type
Iterable[Tuple[ndarray, tuple]]

Returns
A tuple containing a 4-d numpy array and a tuple containing a list of X,Y,Z,C,T indices. The numpy array has dimensions [tile_num,tile_size[0],tile_size[1],channels]

Example

```
from bfio import BioReader
import matplotlib.pyplot as plt

br = BioReader('/path/to/file')

for tiles,ind in br(tile_size=[256,256],tile_stride=[200,200]):
    for i in tiles.shape[0]:
        print(
            'Displaying tile with X,Y coords: {},{}'.format(
                ind[i][0],ind[i][1]
            )
        )
        plt.figure()
        plt.imshow(tiles[ind,:,:,0].squeeze())
        plt.show()
```

classmethod image_size(*filepath*)

image_size Read image width and height from header.

This class method only reads the header information of tiff files or the zarr array json to identify the image width and height. There are instances when the image dimensions may want to be known without actually loading the image, and reading only the header is considerably faster than loading bioformats just to read simple metadata information.

If the file is not a TIFF or OME Zarr, returns width = height = -1.

This code was adapted to only operate on tiff images and includes additional to read the header of little endian encoded BigTIFF files. The original code can be found at: https://github.com/shibukawa/imagesize_py

Parameters
filepath (Path) – Path to tiff file

Returns
Tuple of ints indicating width and height.

property bpp

Same as *bytes_per_pixel*.

property bytes_per_pixel: int

Number of bytes per pixel.

Return type
int

property channel_names: List[str]

Get the channel names for the image.

Return type
List[str]

close()

Close the image.

property cnames: List[str]

Same as *channel_names*.

Return type
List[str]

property dtype: dtype

The numpy pixel type of the data.

Return type
dtype

maximum_batch_size(*tile_size*, *tile_stride=None*)

maximum_batch_size Maximum allowable batch size for tiling.

The pixel buffer only loads at most two supertiles at a time. If the batch size is too large, then the tiling function will attempt to create more tiles than what the buffer holds. To prevent the tiling function from doing this, there is a limit on the number of tiles that can be retrieved in a single call. This function determines what the largest number of retreivable batches is.

Parameters

- **tile_size** (List[int]) – The height and width of the tiles to retrieve
- **tile_stride** (Optional[List[int]]) – If None, defaults to tile_size. *Defaults to None.*

Return type
int

Returns

Maximum allowed number of batches that can be retrieved by the
iterate method.

property metadata: OME

Get the metadata for the image.

This function calls the Bioformats metadata parser, which extracts metadata from an image. This returns a reference to an OMEXML class, which is a convenient handler for the complex xml metadata created by Bioformats.

Most basic metadata information have their own BioReader methods, such as image dimensions(i.e. x, y, etc). However, in some cases it may be necessary to access the underlying metadata class.

Minor changes have been made to the original OMEXML class created for python-bioformats, so the original OMEXML documentation should assist those interested in directly accessing the metadata. In general, it is best to assign data using the object properties to ensure the metadata stays in sync with the file.

For information on the OMEXML class: https://github.com/CellProfiler/python-bioformats/blob/master/bioformats/omexml.py

Return type
OME

Returns
OMEXML object for the image

property physical_size_x: Tuple[float, str]

Physical size of pixels in x-dimension.

Return type
Tuple[float, str]

Returns
Units per pixel, Units (i.e. "cm" or "mm")

property physical_size_y: Tuple[float, str]

Physical size of pixels in y-dimension.

Return type
Tuple[float, str]

Returns
Units per pixel, Units (i.e. "cm" or "mm")

property physical_size_z: Tuple[float, str]

Physical size of pixels in z-dimension.

Return type
Tuple[float, str]

Returns
Units per pixel, Units (i.e. "cm" or "mm")

property ps_x: Tuple[float, str]

Same as *physical_size_x*.

Return type
Tuple[float, str]

property ps_y

Same as *physical_size_y*.

property ps_z

Same as *physical_size_z*.

property read_only: bool

Returns true if object is ready only.

Return type
bool

property samples_per_pixel: int

Number of samples per pixel.

Return type
int

property shape: Tuple[int, int, int, int, int]

The 5-dimensional shape of the image.

Return type
Tuple[int, int, int, int, int]

Returns
(*Y*, *X*, *Z*, *C*, *T*) shape of the image

property spp

Same as *samples_per_pixel*.

3.2 bfio.BioWriter

class BioWriter(*file_path*, *max_workers=None*, *backend=None*, *metadata=None*, *image=None*, ***kwargs*)

Bases: BioBase

BioWriter Write OME tiled tiff images.

This class handles the writing OME tiled tif images. There is a Java backend version of this tool that directly interacts with the Bioformats java library directly, and is primarily used for testing. It is currently not possible to change the tile size (which is set to 1024x1024).

Unlike the BioReader class, the properties of this class are settable until the first time the `write` method is called.

For for information, visit the Bioformats page: https://www.openmicroscopy.org/bio-formats/

Note: In order to use the `bioformats` backend, jpype must be installed.

Initialize a BioWriter.

Parameters

- **file_path** (Union[str, Path]) – Path to file to read
- **max_workers** (Optional[int]) – Number of threads used to read and image. *Default is half the number of detected cores.*
- **backend** (Optional[str]) – Must be `python` or `bioformats`. *Default is python.*
- **metadata** (Optional[OME]) – This directly sets the ome tiff metadata using the OMEXML class if specified. *Defaults to None.*
- **image** (Optional[ndarray]) – The metadata will be set based on the dimensions and data type of the numpy array specified by this keyword argument. Ignored if metadata is specified. *Defaults to None.*
- **kwargs** – Most BioWriter object properties can be passed as keyword arguments to initialize the image metadata. If the metadata argument is used, then keyword arguments are ignored.

property X

Setter
Set the number of pixels in the x-dimension (width)

Getter
Number of pixels in the x-dimension (width)

Type
int

property Y

Setter
Set the number of pixels in the y-dimension (height)

Getter
Number of pixels in the y-dimension (height)

Type
int

property Z

Setter
Set the number of pixels in the z-dimension (depth)

Getter
Number of pixels in the z-dimension (depth)

Type
int

property C

Setter
Set the number of pixels in the c-dimension

Getter
Number of pixels in the c-dimension

Type
int

property T

Setter
Set the number of pixels in the t-dimension

Getter
Number of pixels in the t-dimension

Type
int

property spp

Same as *samples_per_pixel*.

property dtype: dtype

The numpy pixel type of the data.

Return type
dtype

__setitem__(*keys*, *value*)

Image saving using numpy-like indexing.

This is an abbreviated method of accessing the *write* method, where a portion of the image will be saved using numpy-like slicing syntax. Up to 5 dimensions can be designated depending on the number of available dimensions in the image array (Y, X, Z, C, T).

Note: Not all methods of indexing can be used, and some indexing will lead to unexpected results. For example, logical indexing cannot be used, and step sizes in slice objects is ignored for the first three indices. This means and index such as `[0:100:2,0:100:2,0,0,0]` will save a 100x100x1x1x1 numpy array.

Parameters

- **keys** (Union[tuple, slice]) – numpy-like slicing used to save a section of an image.
- **value** (ndarray) – Image chunk to save.

Return type
None

Example

```
import bfio

# Initialize the biowriter
bw = bfio.BioWriter('Path/To/File.ome.tif',
                    X=100,
                    Y=100,
                    dtype=numpy.uint8)

# Load and copy a 100x100 array of pixels
bw[:100,:100,0,0,0] = np.zeros((100,100),dtype=numpy.uint8)

# Slice steps sizes are ignored for the first 3 indices, so this
# does the same as above
bw[0:100:2,0:100:2] = np.zeros((100,100),dtype=numpy.uint8)

# The last two dimensions can receive a tuple or list as input
# Save two channels
bw[:100,100,0,:2,0] = np.ones((100,100,1,2),dtype=numpy.uint8)

# If the file is 3d, save the first 10 z-slices
br[...,:10,0,0] = np.ones((100,100,1,2),dtype=numpy.uint8)
```

Return type
None

Parameters

- **keys** (*Union[tuple, slice]*) –
- **value** (*ndarray*) –

write(*image*, *X=None*, *Y=None*, *Z=None*, *C=None*, *T=None*)

write_image Write the image.

Write all or part of the image. A 5-dimmensional numpy.ndarray is required as the image input.

Parameters

- **image** (ndarray) – a 5-d numpy array
- **X** (Optional[int]) – The starting index of where to save data along the x-axis (columns). If None, loads the full range. *Defaults to None.*
- **Y** (Optional[int]) – The starting index of where to save data along the y-axis (rows). If None, loads the full range. *Defaults to None.*
- **Z** (Optional[int]) – The starting index of where to save data along the z-axis (depth). If None, loads the full range. *Defaults to None.*
- **C** (Union[list, tuple, int, None]) – Values indicating channel indices to load. If None, loads the full range. *Defaults to None.*
- **T** (Union[list, tuple, int, None]) – Values indicating timepoints to load. If None, loads the full range. *Defaults to None.*

Return type
None

close()

Close the image.

This function should be called when an image will no longer be written to. This allows for proper closing and organization of metadata.

Return type
None

property bpp

Same as *bytes_per_pixel*.

property bytes_per_pixel: int

Number of bytes per pixel.

Return type
int

property channel_names: List[str]

Get the channel names for the image.

Return type
List[str]

property cnames: List[str]

Same as *channel_names*.

Return type
List[str]

maximum_batch_size(*tile_size*, *tile_stride=None*)

maximum_batch_size Maximum allowable batch size for tiling.

The pixel buffer only loads at most two supertiles at a time. If the batch size is too large, then the tiling function will attempt to create more tiles than what the buffer holds. To prevent the tiling function from

doing this, there is a limit on the number of tiles that can be retrieved in a single call. This function determines what the largest number of retreivable batches is.

Parameters

- **tile_size** (List[int]) – The height and width of the tiles to retrieve
- **tile_stride** (Optional[List[int]]) – If None, defaults to tile_size. *Defaults to None*.

Return type
int

Returns

Maximum allowed number of batches that can be retrieved by the
iterate method.

property metadata: OME

Get the metadata for the image.

This function calls the Bioformats metadata parser, which extracts metadata from an image. This returns a reference to an OMEXML class, which is a convenient handler for the complex xml metadata created by Bioformats.

Most basic metadata information have their own BioReader methods, such as image dimensions(i.e. x, y, etc). However, in some cases it may be necessary to access the underlying metadata class.

Minor changes have been made to the original OMEXML class created for python-bioformats, so the original OMEXML documentation should assist those interested in directly accessing the metadata. In general, it is best to assign data using the object properties to ensure the metadata stays in sync with the file.

For information on the OMEXML class: https://github.com/CellProfiler/python-bioformats/blob/master/bioformats/omexml.py

Return type
OME

Returns
OMEXML object for the image

property physical_size_x: Tuple[float, str]

Physical size of pixels in x-dimension.

Return type
Tuple[float, str]

Returns
Units per pixel, Units (i.e. “cm” or “mm”)

property physical_size_y: Tuple[float, str]

Physical size of pixels in y-dimension.

Return type
Tuple[float, str]

Returns
Units per pixel, Units (i.e. “cm” or “mm”)

property physical_size_z: Tuple[float, str]

Physical size of pixels in z-dimension.

Return type
Tuple[float, str]

Returns
Units per pixel, Units (i.e. "cm" or "mm")

property ps_x: Tuple[float, str]

Same as *physical_size_x*.

Return type
Tuple[float, str]

property ps_y

Same as *physical_size_y*.

property ps_z

Same as *physical_size_z*.

property read_only: bool

Returns true if object is ready only.

Return type
bool

property samples_per_pixel: int

Number of samples per pixel.

Return type
int

property shape: Tuple[int, int, int, int, int]

The 5-dimensional shape of the image.

Return type
Tuple[int, int, int, int, int]

Returns
(*Y*, *X*, *Z*, *C*, *T*) shape of the image

INDEX

Symbols

B

C

D

I

M

P

R

S

T

W

X

Y

Z