PyFibreBundle

Mike Hughes

Aug 10, 2023

CONTENTS

PyFibreBundle is a Python package for processing of images captured through optical fibre bundles.

The project is hosted on github. The latest stable release can be installed via pip:

```
pip install PyFibreBundle
```

The package supports fibre core pattern removal by filtering and triangular linear interpolation, background correction and flat fielding, as well as automatic bundle location, cropping and masking. Both monochrome and colour images can be processed. The *PyBundle* class is the preferred way to access this functionality, but the lower level functions can also be used directly for greater customisation. The *Mosaic* class provides mosaicing via normalised cross correlation, and the *SuperRes* class allows multiple shifted images to be combined to improve resolution.

The package is designed to be fast enough for use in imaging GUIs as well as for offline research - frame rates of over 100 fps can be achieved on mid-level hardware, including core removal and mosaicing. The Numba just-in-time compiler is used to accelerate key portions of code (particularly triangular linear interpolation) and OpenCV is used for fast mosaicing. If the Numba package is not installed then PyFibreBundle falls back on Python interpreted code.

PyFibreBundle is developed mainly by Mike Hughes's lab in the Applied Optics Group, School of Physics and Astronomy, University of Kent. Bug reports, contributions and pull requests are welcome.

CHAPTER

ONE

CONTENTS

1.1 Installation

There are several ways to get PyFibreBundle:

- Download the latest stable release from github and unzip. (The download link is at the bottom of the linked page.)

This will give you all the examples, tests and test data.

- Download the latest files from the Github repository by clicking 'Code' and 'Download ZIP'
- Clone the Github repository using:

```
git clone https://github.com/MikeHughesKent/PyFibreBundle
```

- Install the latest stable release using:

```
pip install PyFibreBundle
```

Using pip install should find and install all the dependencies. For the other options you will need to either manually check you have the requirements installed, or navigate to the PyFibreBundle folder on your machine and run:

```
pip install -r requirements.txt
```

to install the dependencies. You may wish to create a virtual environment using Conda/venv first to avoid conflicts with your existing python setup.

Note that the pip install doesn't include the examples and tests which still need to be downloaded from Github.

Once installed, you can try running the examples in the examples folder. The examples assume they are being run from the working directory.

1.1.1 Dependencies

- matplotlib>=3.3.4
- numba>=0.55.1
- numpy>=1.18
- opencv_python>=4.5.2.54
- Pillow>=9.3.0
- scipy>=1.7.3

1.2 Basic Image Processing

PyFibreBundle includes functions for basic processing of bundle images, including locating, cropping and masking the bundle and removing the core pattern using spatial filtering or linear interpolation between cores. Both monochrome and colour images are supported. Monochrome images are stored as 2D numpy arrays and colour images are stored as 3D numpy arrays, with the colour channels along the third axis (any number of colour channels is allowed). Image types explicitly supported are uint8, uint16, float32 and float64.

The recommended way to use this functionality is via the *PyBundle* class, but it is also possible to use the low level functions directly.

Full examples are available on Github for spatial filtering and linear interpolation.

1.2.1 Getting Started

Begin by importing the PyBundle class:

```
from pybundle import PyBundle
```

We then instantiate a PyBundle object:

```
pyb = PyBundle()
```

Let's assume we have an image stored in `img`, a 2D (monochrome) or 3D (colour) numpy array. If the image is colour then the colour channels are along the third axis.

In general, to process an image we use:

```
procImage = pyb.process(img)
```

However, this will do nothing to the raw image unless we first set some processing options, either by passing optional arguments when creating the PyBundle object, or by calling setter methods on the object after creation.

1.2.2 Filtering

First we define what type of core-removal we would like, for example for Gaussian filtering with a sigma of 2.5, we pass:

```
pyb = PyBundle(coreMethod = PyBundle.FILTER, filterSize = 2.5)
```

or equivalently:

```
pyb = PyBundle()
pyb.set_core_method(PyBundle.FILTER)    # Choose to use a Gaussian filter
pyb.set_filter_size(2.5)                # Gaussian filter sigma is 2.5 pixels
```

If we then call:

```
procImage = pyb.process(img)
```

then `procImage` will be a Gaussian filtered version of `img`.

The other two options for core removal are PyBundle.TRILIN, for triangular linear interpolation, and PyBundle.EDGE_FILTER to use a custom edge filter.

See the *Linear Interpolation* page for details on how to perform linear interpolation.

The edge filter is a spatial frequency domain filter that seeks to cut off higher spatial frequencies includes those that correspond to the cores. It is used similarly to the Gaussian filter, except that the filter size is defined by passing `edgeFilterSize = (pos, slope)` or calling:

```
set_edge_filter_size(pos, slope)
```

where `pos` the defines the position of the cut-off and `slope` defines the steepness of the cut-off. `pos` should typically be around twice the core spacing and slope around 10% of this.

As for the Gaussian filter, the best speed is achieved by setting a calibration image and then calling `calibrate()`; the edge filter will be generated at this point.

1.2.3 Background and Normalisation

We can provide images which will be used for background subtraction and normalisation (flat-fielding). Essentially, an image provided as background will be subtracted from the raw image to be processed, and the raw image will be divided by the normalisation image (the exact implementation depends on the core processing method).

If we have a background image in the numpy array `backImg`, then to set the background image, pass `background = backImg` or call:

```
pyb.set_background(backimg)
```

To set a normalisation image stored in `normImg`, pass `normaliseImage = normImg` or call:

```
pyb.set_normalise_image(normImg)
```

Existing backgrounds/normalisations can be cleared by passing `None` to these functions.

1.2.4 Cropping

If we are using FILTER or EDGE_FILTER core removal methods, we might also want to crop the image to the size of the bundle (this happens intrinsically for TRILIN). Cropping can be set to happen automatically by passing `crop = True` e.g.

```
pyb = PyBundle(coreMethod = PyBundle.FILTER, filterSize = 2.5, crop = True)
```

In this case, when `process()` is called, PyBundle will attempt to locate the bundle and crop the image to a square just enclosing the bundle.

This is inefficient when processing multiple images from the same imaging setup, as the bundle location routine will run each time we call `process()`. It is faster to determine the location once from a calibration image, which is ideally an image with uniform illumination of the bundle (although it could also be one of the images to be processed).

Set the calibratation image by passing `calibImage = calibImg` or by calling:

```
pyb.set_calib_image(calibImg)
```

where `calibImg` is a 2D/3D numpy array containing the calibration image. The calibration image should be the same size as the images to be processed.

Optionally, we can then call:

```
pyb.calibrate()
```

and PyBundle will calculate and store the location of the bundle. If `calibrate()` is not called then the calibration image will still be used to find the bundle for cropping when `pyb.process()` is called for the first time. The bundle location will then be stored for future use.

Alternatively, we can explicitly tell PyBundle the location of the bundle in advance, either by passing `loc = location` or by calling:

```
pyb.set_loc(location)
```

where `location` is a tuple of (centre_x, centre_y, radius).

1.2.5 Masking

If we are using FILTER or EDGE_FILTER core removal methods, we might also want to set pixels outside the bundle to 0 (this happens intrinsically for TRILIN). This masking can be set to happen automatically by passing `applyMask = True`, e.g.

```
pyb = PyBundle(coreMethod = PyBundle.FILTER, filterSize = 2.5, applyMask = True)
```

As for cropping, PyBundle will generate a mask automatically each time we call pyb.calibrate() on an image which is generally not efficient. Again, it often better to generate the mask based on a calibration image in the same way as for cropping, i.e. by passing `calibImage = calibImg`. Calling:

```
pyb.calibrate()
```

will then allow the mask to be generated in advance, otherwise it will be created the first time we call pyb.process().

1.2.6 Image Type and Autocontrast

The default image output type is `'float64'`, this can be changed by passing, for example `outputType = 'uint8'` when creating the `PyBundle` object, or by calling

```
pyb.set_output_type('uint8')      # Output images will be 8 bit
```

where `'uint8'`, `'uint16'`, `'float32'` or `'float64'` can be used. The output will simply be cast to this format without any scaling, unless we pass `autoContrast = True` or set:

```
pyb.set_auto_contrast(True)
```

in which case the image will be first scaled to between 0 and 255 if an 8 bit output type is set, or between 0 and 65535 if a 16 bit output type is set, or between 0 and 1 if a floating point output type is set.

1.2.7 Examples

Gaussian filtering: examples/filtering_example.py.

Linear interpolation: examples/linear_interp_example.py.

Colour Gaussian filtering: examples/filtering_colour_example.py.

Colouor Linear interpolation: examples/linear_interp_colour_example.py.

1.3 Linear Interpolation

Triangular linear interpolation can be used to remove the fibre bundle core pattern. Using a calibration image, usually acquired with no object in view (i.e. a flat field), the location of each core is determined. A Delaunay triangulation is performed over the core locations. A reconstruction grid is then defined, and the enclosing triangle for each pixel is determined. Images can then be processed by interpolating the value of each pixel from the brightness of the three surrounding cores. Although calibration can take a few seconds, processing of images can then be at video rate.

There are examples for monochrome and colour images on Github.

Full details of all available options are listed on the PyBundle Class section.

1.3.1 Object Oriented Approach

Import the `PyBundle` class and instantiate an object:

```
from pybundle import PyBundle
pyb = PyBundle()
```

Set the core removal method to triangular linear interpolation:

```
pyb.set_core_method(pyb.TRILIN)
```

Set both the calibration and normalisation images to be `calibImg`, a 2D/3D numpy array:

```
pyb.set_calib_image(calibImg)
pyb.set_normalise_image(calibImg)
```

Choose the output images size:

```
pyb.set_grid_size(512)
```

If we are normalising it is best to get an output image which is auto-contrasted:

```
pyb.set_auto_contrast(True)
```

Alternatively, those options can all be set at instantiation:

```
pyb = PyBundle(coreMethod = PyBundle.TRILIN,
               calibImage = calibImg,
               normaliseImage = calibImg,
               gridSize = 512,
               autoContrast = True)
```

We then perform the calibration, which takes around a second:

```
pyb.calibrate()
```

To remove the fibre bundle pattern from an image `img`, a 2D/3D numpy array, we call:

```
imgProc = pyb.process(img)
```

For real-time processing, a speed-up of approx X4 in reconstruction can be obtained if the Numba package is installed. To disable use of Numba, call:

```
pyb.set_use_numba(False)
```

1.3.2 Lower level functions

For greater customisation, the static functions can be called directly. First perform a calibration using the calibration image `calibImg`, a 2D numpy array:

```
coreSize = 3
gridSize = 512
calib = pybundle.calib_tri_interp(calibImg, coreSize, gridSize, normalise = calibImg,␣
↪automask = True)
```

Here we have specified `coreSize = 3` which is the approximate core spacing in the image. This assists the calibration routine in finding all cores. If unknown it can be estimated using `find_core_spacing`.

The `gridSize` is the number of pixels in each dimensions of the reconstructed image, which is square.

Finally, we have specified to use the `calibImg` for normalisation. This means that the intensity extracted from each core during imaging will be normalised with respect to the intensity from the calibration image, removing effects due to non-uniform cores. If this is not done (i.e. normalise is left as the default `None`) then images may appear grainy.

To reconstruct an image `img`, a 2D/3D numpy array, we then call:

```
imgRecon = pybundle.recon_tri_interp(img, calib)
```

This returns a 2D numpy array of size `(gridSize, gridSize)` containing the image with the core pattern removed.

For all optional parameters refer to the functon reference for `calib_tri_interp` and `recon_tri_interp`.

1.3.3 Example

An example using OOP is in "examples\linear_interp_example.py".

An example using OOP and showing the difference in speed if Numba is used is in "examples\linear_interp_numba_example.py".

Linear interpolation is compared with other core method removal techniques in "examples\compare_recons.py".

1.4 PyBundle class

The PyBundle class is the recommended way to use most functionality of the package (other than Mosaicing which has its own class. All method are listed below, introductory guides on using the class are available for for Basic Processing, Linear Interpolation and Super Resolution .

1.4.1 Instantiatation

PyBundle(*optional arguments*)

Creates a PyBundle object. There are a large number of optional keyword arguments (.e.g `autoLoc = True`), which are listed below with their defaults if not set. Each option also has a setter method (e.g. `set_auto_loc`) which can be called after creating the object as an alternative to passing the keyword arguement at creation. See the documentation for each setter, below, for a detailed description of each option's meaning.

GENERAL Settings:

- autoContrast = False (`set_auto_contrast`)
- background = None (`set_background`)
- coreMethod = None (`set_core_method`)
- outputType = 'float64' (`set_output_type`)

CROP/MASK Settings:

- applyMask = False (`set_apply_mask`)
- autoMask = True (`set_auto_mask`)
- autoLoc = False (`set_auto_loc`)
- crop = False(`set_crop`)
- loc = None (`set_loc`)
- mask = None (`set_mask`)
- radius = None (`set_radius`)

CALIB/BACKGROUND/NORMALISATION Settings:

- calibImage = None (`set_calib_image`)
- backgroundImage = None (`set_background`)
- normaliseImage = none (`set_normalise_image`)

GAUSSIAN FILTER Settings:

- filterSize = None (`set_filter_size`)

EDGE_FILTER Settings:

- edgeFilterShape = None (`set_edge_filter_shape`)

LINEAR INTERPOLATION Settings:

- coreSize = 3 (`set_core_size`)
- gridSize = 512 (`set_grid_size`)
- useNumba = True (`set_use_numba`)
- whiteBalance = False (`set_white_balance`)

SUPER RESOLUTION Settings:

- superRes = False (`set_super_res`)
- srShifts = None (`set_sr_shifts`)
- srCalibImages = None (`set_sr_calib_images`)
- srNormToBackgrounds = False (`set_sr_norm_to_backgrounds`)
- srNormToImages = True (`set_sr_norm_to_images`)
- srMultiBackgrounds = False (`set_sr_multi_backgrounds`)
- srMultiNormalisation = False (`set_sr_multi_normalisation`)
- srDarkFrame = None (`set_sr_dark_frame`)
- srUseLut = False (`set_sr_use_lut`)
- srParamValue = None (`set_sr_param_value`)

1.4.2 Methods

`class` pybundle.**PyBundle**(***kwargs*)

calibrate()

Peforms calibraion steps appropriate to chosen method. A calibration image must have been set prior to calling this.

For TRILIN, creates interpolation calibration.

For FILTER, EDGE_FILTER, a filtered normalisation image is created.

For FILTER, EDGE_FILTER the bundle will be located if autoLoc has been set.

For FILTER, EDGE_FILTER the mask will be located if autoMask has been set.

calibrate_sr()

Creates calibration for TRILIN SR method. A calibration image, set of super-res shift images, coreSize and gridSize must have been set prior to calling this.

calibrate_sr_lut(*paramCalib*, *paramRange*, *nCalibrations*)

Creates calibration LUT for TRILIN SR method. A calibration image, set of super-res shift images, coreSize and griSize must have been set prior to calling this.

Parameters

- **paramCalib** – parameter shift calibration, as generated by calib_param_shift()
- **paramRange** – tuple of (min, max) defining range of parameter values to generate for
- **nCalibration** – int, number of parameter values to generate for

create_and_set_mask(*img*, ***kwargs*)

Determine mask from provided calibration image and set as mask Optionally provide a radius rather than using radius of determined bundle location.

Parameters
img – calibration image from which size of mask is determined

Keyword Arguments
radius – radius of mask, default is to determine this automtically

`get_pixel_scale()`

Returns the scaling factor between the pixel size in the raw image and the pixel size in the processed image. If the TRILIN method is selected, but a calibration has not yet been performed, returns None.

`process(img)`

Process fibre bundle image using current settings.

Returns processed image as 2D/3D numpy array.

Parameters
img – input image as 2D/3D numpy array

`set_apply_mask(applyMask)`

Determines whether areas outside the bundle are set to zero for FILTER and EDGE_FILTER method.

Parameters
applyMask – boolean, True to apply mask, False to not apply mask

`set_auto_contrast(ac)`

Determines whether images are scaled to be between 0-255.

Parameters
ac – boolean, True to autocontrast

`set_auto_loc(img)`

Sets whether the bundle is automatically located for cropping and masking, if these are turned on, depending on boolean value passed.

It is also possible to pass an image as a 2D numpy array instead of a Boolean, in which case the bundle location will be determined from this image. However, this is noq deprecated in favour of setting calibImg and then calling calibrate.

Parameters
img – boolean, True to auto-locate bundle, False to not.

`set_auto_mask(img, **kwargs)`

Set whether to automically create mask using pre-determined bundle location.

It is also possible to provide an image as a 2D numpy array, in which case the mask will be generated of the correct size for this image, but this is deprecated, use calibrate() instead. Optionally provide a radius rather than using radius of determined bundle location.

Parameters
img – boolean, True to automically create mask

Keyword Arguments
radius – optional, int, overrides automically determined radius for mask.

`set_background(background)`

Store an image to be used as background. If TRILIN is being used and a calibration has already been performed, the background will be added to the calibration.

Parameters
backgroundImage – background image as 2D/3D numpy array. Set as None to remove background.

`set_calib_image(calibImg)`

Set image to be used for calibration.

Parameters
calibImg – calibration image as 2D/3D numpy array

`set_core_method`(*coreMethod*)

Set the method to use to remove cores, FILTER, TRILIN or EDGE_FILTER

Parameters
coreMethod – PyBundle.FILTER, PyBundle.TRILIN or PyBundle.EDGE_FILTER

`set_core_size`(*coreSize*)

Set the estimated centre-centre core spacing used to help find cores as part of TRILIN method.

Parameters
coreSize – float, estimate core spacing

`set_crop`(*crop*)

Determines whether images are cropped to size of bundle for FILTER, EDGE_FILTER methods. crop is Boolean.

Parameters
crop – boolean, True to crop

`set_edge_filter_shape`(*edgePos*, *edgeSlope*)

Creates and stores filter for EDGE_FILTER method.

Parameters

- **edgePos** – float, spatial frequency of edge in pixels of FFT of image
- **edgeSlope** – float, steepness of slope (range from 10% to 90%) in pixels of FFT of image

`set_filter_size`(*filterSize*)

Set the size of Gaussian filter used if filtering method employed.

Parameters
filterSize – float, sigma of Gaussian filter

`set_grid_size`(*gridSize*)

Sets output image size if TRLIN method used. If not called prior to calling 'calibrate', the default value of 512 will be used.

Parameters
gridSize – int, size of square image output

`set_loc`(*loc*)

Store the location of the bundle. This will also set autoLoc = False.

Parameters
loc – bundle location, tuple of (centreX, centreY, radius)

`set_mask`(*mask*)

Provide a mask to be used. Mask must be a 2D numpy array of same size as images to be processed

Parameters
mask – 2D numpy array, 1 inside bundle, 0 inside bundle. Must be same size as image to be processed.

`set_normalise_image`(*normaliseImage*)

Store an image to be used for normalisation. If TRILIN is being used and a calibration has already been performed, the normalisation will be added to the calibration.

Parameters
normaliseImage – normalisation image as 2D/3D numpy array. Set as None to remove normalisation.

set_output_type(*outputType*)

Specify the data type of input images returned from 'process'. Returns False if type not valid.

Parameters
outputType – str, one of 'uint8', 'unit16' or 'float'

set_radius(*radius*)

Sets the radius of the bundle in the image. This will override any automically determined value.

Parameters
radius – int, bundle radius

set_sr_backgrounds(*backgrounds*)

Provide a set of background images for background correction of each SR shifted image.

Parameters
backgrounds – 3D numpy array, stack of background images

set_sr_calib_images(*calibImages*)

Provides the calibration images, a stack of shifted images used to determine shifts between images for super-resolution

Parameters
calibImages – 3D numpy array, stack of shifted images.

set_sr_dark_frame(*darkFrame*)

Provide a dark frame for super-resolution calibration.

Parameters
darkFrame – 2D numpy array, dark frame

set_sr_multi_backgrounds(*mb*)

Sets whether super-resolution should use individual backgrounds for each each shifted image.

Parameters
mb – boolean, True to use multiple backgrounds, False to not

set_sr_multi_normalisation(*mn*)

Sets whether super-resolution should normalise to each core in each image

Parameters
mn – boolean, True to normalise each core in each image, False to not

set_sr_norm_to_backgrounds(*normToBackgrounds*)

Sets whether super-resolution recon should normalise each input image w.r.t. a stack of backgrounds in srBackgrounds to have the same mean intensity.

Parameters
normToBackgrounds – boolean, True to normalise, False to not

set_sr_norm_to_images(*normToImages*)

Sets whether super-resolution recon should normalise each input image to have the same mean intensity.

Parameters
normToImages – boolean, True to normalise, False to not

set_sr_normalisation_images(*normalisationImages*)

Provide a set of normalisation images for normalising intensity of each SR shifted image.

Parameters
normalisationImages – 3D numpy array, stack of images

set_sr_param_value(*val*)

Sets the current value of the parameter on which the shifts dependent for SR reconstruction.

Parameters
val – parameter value

set_sr_shifts(*shifts*)

Provide shifts between SR images. If this is set then no registration will be performed.

Parameters
shifts – 2D numpy array, shifts for each image relative to first image. 1st axis is image number, 2nd axis is (x,y)

set_sr_use_lut(*useLUT*)

Enables or disables use of calibration LUT for super resoution.

Parameters
useLUT – boolean, True to use calibration LUT.

set_super_res(*sr*)

Enables or disables super resolution.

Parameters
sr – boolean, True to use super-resolution.

set_use_numba(*useNumba*)

Sets whether Numba should be used for JIT compiler acceleration for functionality which supports this.

Parameters
useNumba – boolean, True to use Numba, False to not.

set_white_balance(*whiteBalance*)

Sets whether each colour channel should be normalised independently when using linear interpolation method.

Parameters
whiteBalance – boolean, each channel normalised independently if True (default is False).

1.5 Mosaicing

The Mosaic class allows high speed mosaicing using normalised cross correlation to detect shifts between image frames, and either dead-leaf or alpha-blended insertion of images into a mosaic. The easiest way to use this functionality is to create an instance of `Mosaic` class and then use `Mosaic.add(img)` to sequentially register and add image `img` to the mosaic, and `Mosaic.getMosaic()` to get the latest mosaic image. Both `img` and the `mosaic` are 2D (monochrome) or 3D (colour) numpy arrays.

An example is provided on Github.

1.5.1 Getting Started

Instantiate an object of the `Mosaic` class using default options and with a mosaic image size of 1000x1000:

```
from pybundle import Mosaic
mMosaic = Mosaic(1000)
```

Add an image `img` to the mosaic:

```
mMosaic.add(img)
```

Request the latest mosaic image:

```
mosaicImage = mMosaic.getMosaic()
```

The `mosaicImage` will be a 2D numpy array if `img` is 2D and a 3D numpy array if `img` is 3D, in which case the third axis represents the colour channels.

1.5.2 Methods

class pybundle.**Mosaic**(*mosaicSize*, ***kwargs*)

The Mosaic class is used for producing mosaics from a sequence of images. After instantiating a Mosaic object, use add() to add in images and get_mosaic to obtain the current mosaic image.

Parameters

mosaicSize – int, square size of mosaic image

Keyword Arguments

- **resize** – int, images will be resized to a square this size, default is None meaning no resize.
- **templateSize** – int, size of square to extract to use as template for shift detection. Default is 1/4 image size.
- **refSize** – int, size of square to extract to use as image to compare template to for shift detection. Default is 1/2 image size.
- **cropSize** – int, input images are cropped to a circle of this diameter before insertion. (default is 0.9 x size of first image added)
- **imageType** – str, data type for mosaic, default is the same as first image added
- **blend** – boolean, if True (default), images will be added blended, otherwise they are added dead-leaf
- **blendDist** – int, distance in pixels from edge of inserted image to blend with mosaic, default is 40
- **minDistforAdd** – int, minimum distance moved before an image will be added to the mosaic, default is 25
- **initialX** – int, starting positon of mosaic, default is centre
- **initialY** – int, starting positon of mosaic, default is centre
- **boundaryMethod** – method to deal with reaching edge: CROP, SCROLL or EXPAND, default is CROP
- **expandStep** – int, amount to expand by if EXPAND boundaryMethod is used

- **resetThresh** – float, mosaic will reset if correlation peak is below this, default is None (ignore)
- **resetIntensity** – float, mosaic will reset if mean image value is below this value, default is None (ignore)
- **resetSharpness** – float, mosaic will reset if image sharpness (mean of gradient) drops below this value, default is None (ignore)

add(*img*)

Add image to current mosaic.

Parameters
img – image as 2D/3D numpy array

get_mosaic()

Returns current mosaic image as 2D/3D numpy array

reset()

Call to reset mosaic, clearing image. The mosaic will only be fully reset once a new image is added. Note that parameters that were already initialised will not be reset.

1.5.3 Usage Notes

The only required argument is the size of the mosaic image. By default images will be added blended, there will be no resize of the input image, no checking of input image quality and if the mosaic reaches the edge of the image it will simple run off the the edge.

Usually it is beneficial to resize the input images to prevent the need for a very large mosaic image, e.g.:

```
mMosaic = Mosaic(1000, resize = 250)
```

The reset methods (`resetThresh`, `resetIntensity` and `resetSharpness`) are normally required when used with a handheld probe to handle instances where tissue contact is lost or the probe is moved too quickly. For optical sectioning endomicroscope, a combination of correlation based thresholding (`resetThresh`) and intensity based thresholding (`resetIntensity`) works well. For non-sectioning endomicroscopes, moving out of focus does not sufficiently reduce either, and so it may be necessary to use sharpness thresholding (`resetSharpness`) as well. The best values to use must be determined empirically and will depend on pre-processing steps.

For slow moving probes, `minDistForAdd` may need to be adjusted particularly when using blending to prevent undesirable effects of the same image being blended with itself.

1.5.4 Low Level Functions

The private member functions of the Mosaic class are listed below for custom use:

class pybundle.**Mosaic**(*mosaicSize*, ***kwargs*)

The Mosaic class is used for producing mosaics from a sequence of images. After instantiating a Mosaic object, use add() to add in images and get_mosaic to obtain the current mosaic image.

Parameters
mosaicSize – int, square size of mosaic image

Keyword Arguments

- **resize** – int, images will be resized to a square this size, default is None meaning no resize.

- **templateSize** – int, size of square to extract to use as template for shift detection. Default is 1/4 image size.
- **refSize** – int, size of square to extract to use as image to compare template to for shift detection. Default is 1/2 image size.
- **cropSize** – int, input images are cropped to a circle of this diameter before insertion. (default is 0.9 x size of first image added)
- **imageType** – str, data type for mosaic, default is the same as first image added
- **blend** – boolean, if True (default), images will be added blended, otherwise they are added dead-leaf
- **blendDist** – int, distance in pixels from edge of inserted image to blend with mosaic, default is 40
- **minDistforAdd** – int, minimum distance moved before an image will be added to the mosaic, default is 25
- **initialX** – int, starting positon of mosaic, default is centre
- **initialY** – int, starting positon of mosaic, default is centre
- **boundaryMethod** – method to deal with reaching edge: CROP, SCROLL or EXPAND, default is CROP
- **expandStep** – int, amount to expand by if EXPAND boundaryMethod is used
- **resetThresh** – float, mosaic will reset if correlation peak is below this, default is None (ignore)
- **resetIntensity** – float, mosaic will reset if mean image value is below this value, default is None (ignore)
- **resetSharpness** – float, mosaic will reset if image sharpness (mean of gradient) drops below this value, default is None (ignore)

__cosine_window(*circleSize*, *circleSmooth*)

Produce a circular cosine window mask on grid of imgSize * imgSize. Mask is 0 for radius > circleSize and 1 for radius < (circleSize - circleSmooth). The intermediate region is a smooth cosine function.

Returns mask as 2D numpy array.

Parameters

- **imgSize** – int, size of square mask to generate
- **circleSize** – int, radius of mask (mask pixels outside here are 0)
- **circleSmooth** – int, size of smoothing region at the inside edge of the circle. (mask pixels with a radius less than this are 1)

__expand_mosaic(*distance*, *direction*, *currentX*, *currentY*)

Increase size of mosaic image by 'distance' in direction 'direction'. Supply currentX and currentY position so that these can be modified to be correct for new mosaic size.

Returns tuple of (newMosaic, width, height, newX, newY), where newMosaic is the larger mosaic image as 2D numpy array, width is the x-size of the new mosaic, height is the y-size of the new mosaic, newX is the x position of the last image insertion in the new mosaic, newY is the y position of the last image insertion in the new mosaic.

Parameters

- **mosaic** – input mosaic image as 2D numpy array

- **distance** – pixels to expand by
- **direction** – side to expand, one of Mosaic.Top, Mosaic.Bottom, Mosaic.Left or Mosaic.Right
- **currentX** – x position of last image insertion into mosaic
- **currentY** – y position of last image insertion into mosaic

__find_shift(*img2*, *templateSize*, *refSize*)

Calculates how far img2 has shifted relative to img1 using normalised cross correlation.

Returns tuple of (shift, max_val) where shift is a tuple of (x_shift, y_shift) and max_val is the normalised cross correlation peak value. Returns None if the shift cannot be calculated.

Parameters

- **img1** – reference image as 2D/3D numpy array
- **img2** – template image as 2D/3D numpy array
- **templateSize** – int, a square of this size is extracted from img as the template
- **refSize** – int, a square of this size is extracted from refSize as the template. Must be bigger than templateSize and the maximum shift detectable is (refSize - templateSize)/2

__initialise_mosaic(*img*)

Choose sensible values for non-specified parameters.

Parameters

img – input image to used to choose sensible parameters for mosaicing, 2D/3D numpy array

__insert_into_mosaic(*img*, *mask*, *position*)

Dead leaf insertion of image into a mosaic at specified position. Only pixels for which mask == 1 are copied.

Parameters

- **mosaic** – current mosaic image, 2D/3D numpy array
- **img** – img to insert, 2D/3D numpy array
- **mask** – 2D numpy array with values of 1 for pixels to be copied and 0 for pixels not to be copied. Must be same size as img.
- **position** – position of insertion as tuple of (x,y). This is the pixel the centre of the image will be at.

__insert_into_mosaic_blended(*img*, *mask*, *blendMask*, *cropSize*, *blendDist*, *position*)

Insertion of image into a mosaic with cosine window blending. Only pixels from image for which mask == 1 are copied. Pixels within blendDist of edge of mosaic (i.e. radius of cropSize/2) are blended with existing mosaic pixel values

Parameters

- **mosaic** – current mosaic image, 2D/3D numpy array
- **img** – img to insert, 2D/3D numpy array
- **mask** – 2D numpy array with values of 1 for pixels to be copied and 0 for pixels not to be copied. Must be same size as img.
- **blendMask** – the cosine window blending mask with weighted pixel values. If passed empty [] this will be created

- **cropSize** – size of input image.
- **blendDist** – number which controls the sptial extent of the blending
- **position** – position of insertion as tuple of (x,y). This is the pixel the centre of the image will be at.

__is_outside_mosaic(*img*, *position*)

Checks if position of image to insert into mosaic will result in part of inserted image being outside of mosaic. Returns tuple of boolean (true if outside), side it leaves (using consts defined above) and distance is has strayed over the edge. e.g. (True, Mosaic.Top, 20).

Returns tuple of (ouside, side, distance), where outside is True if part of image, side is (one of the) side(s) it has strayed out of, one of Mosaic.Top, Mosaic.Bottom, Mosaic.Left or Mosaic.Right or -1 if outside == False, distance is distance it has strayed outside mosaic in the direction specified in size (0 if outside == False).

Parameters

- **mosaic** – mosaic image (strictly can be any numpy array the same size as the mosaic)
- **img** – image to be inserted as 2D numpy array
- **position** – position of insertion as tuple of (x,y). This is the pixel the centre of the image will be at.

__scroll_mosaic(*distance*, *direction*, *currentX*, *currentY*)

Scroll mosaic to allow mosaicing to continue past edge of mosaic. Pixel values will be lost. Supply currentX and currentY position so that these can be modified to be correct for new mosaic size.

Return: tuple of (newMosaic, width, height, newX, newY), where newMosaic is the larger mosaic image as 2D numpy array, width is the x-size of the new mosaic, height is the y-size of the new mosaic, newX is the x position of the last image insertion in the new mosaic, newY is the y position of the last image insertion in the new mosaic.

Parameters

- **mosaic** – input mosaic image as 2D numpy array
- **distance** – pixels to expand by
- **direction** – side to expand, one of Mosaic.Top, Mosaic.Bottom, Mosaic.Left or Mosaic.Right
- **currentX** – x position of last image insertion into mosaic
- **currentY** – y position of last image insertion into mosaic

1.5.5 Example

An example is provided in "examples\mosaicing_example.py"

1.6 Super Resolution

Core super-resolution (i.e. overcoming the sampling limit of the fibre bundle) can be achieved by combining multiple images, with the object slightly shifted with respect to the fibre pattern. The super-resolution sub-package of PyFibreBundle provides the ability to combine multiple images and generate an enhanced resolution using triangular linear interpolation. As with single image triangular linear interpolation, calibation takes several seconds, but reconstruction is fast. For most applications this functionality is best accessed via the *PyBundle* class as shown below. This functionality is currently only available for monochrome images.

1.6.1 Getting Started

Import pybundle and instantiate a PyBundle object:

```
from pybundle import PyBundle
pyb = PyBundle()
```

To use super-resolution we must use the `TRILIN` processing method and choose the output image size:

```
pyb.set_core_method(pyb.TRILIN)
pyb.set_grid_size(512)
```

Then enable super-resolution:

```
pyb.set_super_res(True)
```

As for non-super-resolution `TRILIN`, we provide a calibration image, a uniformly illuminated image (2D numpy array) used to identify core locations:

```
pyb.set_calib_image(calibImage)
```

We can also set a background image (2D numpy array) which will be used for core-by-core background subtraction, this may be the same as `calibImage`:

```
pyb.set_background(backgroundImg)
```

A normalisation image (2D numpy array) for core-by-core normalisation is set using:

```
pyb.set_normalise_image(calibImg)
```

We also need to provide a set of calibration images from which the shifts can be determined. These can be the same as the images we wish to use to create a super-resolution image from, or a set of image that we know have the same shifts. The calibration images are provided as a 3D numpy array, with image number along the third axis:

```
pyb.set_sr_calib_images(srCalibImages)
```

We then perform the calibration, this may take several seconds:

```
pyb.calibrate_sr()
```

We can then generate a super-resolution image from a stack of shifted input images, again a 3D numpy array. This may be the same as `srCalibImages` or, if the shifts are reproducible, we can re-use the calibration for different stacks of images.:

```
srImage = pyb.process(inputImages)
```

1.6.2 Using Known Shifts

If the x and y shifts between images are known in advance, they can be specified using:

```
pyb.set_sr_shifits(shifts)
```

where `shifts` is a 2D numpy array of size (nImages, 2). When `pyb.calibrate_sr()` is called, these shifts will be used instead of calculating shifts, i.e. `pyb.set_sr_calib_images()` does not need to be called.

1.6.3 Correcting for Intensity Difference Between Images

If the shifted images are created simply by moving the bundle or the object, then the above method using only a single background/normalisation image is all that is requried. If the shifted images have different intensities (e.g. they are created from different light sources) then any global intensity differences must be corrected to avoid image artefacts. A simple way to correct this is by multiplying each shifts images by an intensity correction factor such that all the shifted images in such a way that multipying all the corresponding calibration images by the same set of factors would results in them all having the same mean intensity. This happens by default and can be explicitly turned on or off using:

```
pyb.set_sr_norm_to_images(True)
```

or:

```
pyb.set_sr_norm_to_images(False)
```

Alternatively, if a set of background images with the same relative mean intensities is available, these can be used to determine the required correction by setting:

```
pyb.set_sr_backgrounds(backgroundImgs)
pyb.set_sr_norm_to_backgrounds(True)
```

where `backgroundImgs` is a 3D numpy array of (height, width, num images) containing the set of background images.

Note that this 'normalisation' is correcting for global differences in the intensity of each of the shfited images and is distinct from the core-by-core normaliation of the TRILIN method which corrects for core-to-core variations.

These options must be set prior to calibration.

1.6.4 Using Background/Normalisation Stack

If each shifted image requires a different core-by-core background subtraction and/or normalisation, this can be specified:

```
pyb.set_sr_multi_backgrounds(True)
pyb.set_sr_backgrounds(backgroundImgs)

pyb.set_sr_multi_normalisation(True)
pyb.set_sr_normalisation_images(normalisationImgs)
```

In this case, a different TRILIN normalisation/background correction is applied to each shifted image independently. If multi-normalisation is used, this overrides `set_sr_norm_to_backgrounds` or `set_sr_norm_to_images` since it

will inherently ensure that each image is corrected to be of the same mean intensity. This may offer an improvement over correcting on the basis of the mean background image intensities (i.e. using `pyb.set_norm_to_backgrounds`) in cases where the coupling efficiency of individual cores varies across the set of shifted images.

These options must be set prior to calibration.

1.6.5 Using Lower Level Functions

First, perform the calibration. This requires a flat-field/background image `calibImg` (a 2D numpy array), a stack of shifted images `imgs` (a 3D numpy array - [x,y,n]), an estimate of the core spacing `core size`, and the output image size `gridSize`

```
calib = SuperRes.calib_multi_tri_interp(calibImg, imgs, coreSize, gridSize, normalise =␣
↪calibImg)
```

We have also specified an optional parameter, a normalisation image `calibImg`, which prevents the images becoming grainy due to core-core variations. Note that `imgs` does not need to be the actual images to be used for reconstruction, but they must have the same relative shift as the the images. Alternatively, if the shifts are known, these can be specified using the optional parameter `shifts` which should be a 2D numpy array of the form (x_shift, y_shift, image_number). If `shifts` is specified then `imgs` can be `None`.

We then perform the super-resolution reconstruction using:

```
reconImg = SuperRes.recon_multi_tri_interp(imgs, calib)
```

which returns `reconImg` a 2D numpy array representing the output image.

Additional options are described below.

1.6.6 Parameterised Shifts

In some circumstances, the shifts between the images in the stack are fixed in time but are linearly dependent on some other parameter. For example, in fibre bundle inline holographic microscopy, which uses multiple light sources in a transmission geometry, the shifts of the hologram (image) position on the bundle depend on the distance between the object and the bundle. In these cases it can be convenient to determine the dependence of the shifts on this parameter in a calibrations stage, and then to subsequently infer the shifts for all further sets of images based on the current value of the parameter rather than measuring them directl from the images.

Assuming we have acquired several stacks of shifted images for different values of the parameter, we assemble a 4D numpy array of images with dimensions (image height, image width, number of shifts, number of example param values).

We then call:

```
paramCalib = calib_param_shift(param, images, calibration)
```

where `calibration` is a single image linear interpolation calibration such as returned from `calib_tri_interp` or the one stored in `PyBundle.calibration` after calling `PyBundle.calibrate`. `param` is a 1D numpy array specifying the values of the parameter for each stack of shifted images.

The calibration is used to reconstruct each image in the stack. The x and y-components of the shifts of the nth shifted image from each stack of shifted images (i.e. across all example values of the parameter) are then fitted to the values of the parameter with a 1st order polynomial. The function returns `paramCalib` which provides the gradient and offset of the shift for each image in the stack of shifted images.

To calculate the expected shifts for a stack of shifted images for a specific value of the parameter, we then call:

```
shifts = get_param_shift(param, paramCalib)
```

Where `param` is the value of the parameter we wish to know the shifts for, and `paramCalib` is the calibration returned by `calib_param_shift`.

1.6.7 Calibration Look-up-table

In cases where the shifts between the images change in some linear way with some parameter, as discussed in detail above, it may be desirable to reconstruct resolution-enhanced images for multiple values of the parameter. For example, in inline bundle holographic microscopy, the shifts depend on the distance to the object which may change in time. A different value for the shifts requires a new SR calibration, since the calibration requires knowledge of the shifts. However, this calibration is too slow to be performed in real-time. It is therefore advantageous to compute different calibrations for different values of the parameter, store these in a look-up table (LUT), and then at run-time to use the calibration stored for the nearest value of the parameter.

To generate a LUT when using the PyBundle class, call:

```
PyBundle.calibrate_sr_LUT(self, paramCalib, paramRange, nCalibrations)
```

Here, `paramRange` is a tuple of (min, max) values of the parameter to generate the LUT for, and `nCalibrations` is the number of values of the parameter within this range to create calibrations for. Since each calibration takes typically several seconds to perform, large values of `nCalibrations` will take a long time to compute.

Prior to calling `calibrate_sr_LUT`, a single image calibration must already have been created using `pyb.calibrate`. We must also provide a parameter calibration `paramCalib`, which tells us how the image shifts are related to the value of the parameter, either created manually or using the output of `calib_param_shift`.

We then tell PyBundle to use the LUT:

```
PyBundle.set_use_sr_lut(True)
```

We must also tell PyBundle the current value of the parameter:

```
PyBundle.set_sr_param(paramValue)
```

Now, when we call `PyBundle.process`, assuming we have enabled super-resolution and provided a set of shifted images, as described above, the calibration LUT will be accessed and the calibration previously created for a value of the parameter closest to the current value will be used. This look up is much faster (by several orders of magnitude) then performing a new calibration.

Alternatively, if lower-level control is needed, an instance of the CalibrationLUT can be created directly:

```
lut = CalibrationLUT(calibImg, imgs, coreSize, gridSize, paramCalib, paramRange,␣
↪nCalibrations, [optional arguments])
```

Parameters (including optional parameters) are as for `calib_multi_tri_interp`, with the addition of `paramCalib`, which is the return from calling `calib_param_shift` and stores the mapping between the parameter and the shifts, `paramRange` which is a tuple of (min, max) values of the parameter to generate the LUT for, and `nCalibrations` which is the number of values of the parameter within this range to create calibrations for.

Once the LUT is generated, we can extract the best calibration for the current value of the parameter using:

```
calibrationSR = lut.calibrationSR(paramValue)
```

SR reconstructions can then be performed using:

```
reconImage = recon.multi_tri_interp(imageStack, calibrationSR)
```

1.6.8 Function Reference

These are the low level functions, for most purposes it is better to use an instance of the `PyBundle` class.

High Level

pybundle.SuperRes.**calib_multi_tri_interp**(*calibImg*, *imgs*, *coreSize*, *gridSize*, ***kwargs*)

Calibration step for super-resolution reconstruction. Either specify the known shifts between images, or provide an example set of images which the shifts will be calculated from.

Returns instance of BundleCalibration.

Parameters

- **calibImg** – calibration image of fibre bundle, 2D numpy array
- **imgs** – example set of images with the same set of mutual shifts as the images to later be used to recover an enhanced resolution image from. 3D numpy array. Can be None if 'shifts' is specified instead.
- **coreSize** – float, estimate of average spacing between cores
- **gridSize** – int, output size of image, supply a single value, image will be square

Keyword Arguments

- **normalise** – image used for normalisation, as 2D numpy array. Can be same as calibration image, defaults to no normalisation
- **background** – image used for background subtraction, as 2D numpy array, defaults to no background
- **shifts** – known x and y shifts between images as 2D numpy array of size (numImages,2). Will override doing registration of 'imgs' if specified as anything other than None.
- **centreX** – int, x centre location of bundle, if not specified will be determined automatically
- **centreY** – int, y centre location of bundle, if not specified will be determined automatically
- **radius** – int, radius of bundle, if not specified will be determined automatically
- **filterSize** – float, sigma of Gaussian filter applied when finding cores, defaults to no filter
- **normToImage** – boolean, if True each image will be normalised to have the same mean intensity. Defaults to False.
- **normToBackground** – optional, if true, each image will be normalised with respect to the corresponding background image from a stack of background images (one for each shift position) provided in backgroundImgs. Defaults to False.
- **backgroundImgs** – stack of images, same size as imgs which are used to normalise or for image by image background subtraction. Defaults to None.
- **multiBackgrounds** – boolean, if True and backgroundImgs is defined, each image will have its own background image subtracted rather than using backgroundImg

- **imageScaleFactor** – If normToBackground and normToImage are False (default), use this to specify the normalisation factors for each image. Provide a 1D array the same size as the number of shifted images. Each image will be multiplied by the corresponding factor prior to reconstruction. Default is None (i.e. no scaling).
- **autoMask** – boolean, mask pixels outside bundle when searching for cores. Defualts to True.
- **mask** – : boolean, when reconstructing output image will be masked outside of bundle. Defaults to True

pybundle.SuperRes.**recon_multi_tri_interp**(*imgs*, *calib*, *numba=True*)

Reconstruct image with super-resolution from set of shifted image. Requires calibration to have been performed and stored in 'calib' as instance of BundleCalibration.

Returns reconstructed image as 2D numpy array

Parameters

- **imgs** – set of shifted images
- **calib** – calibration, instance of BundleCalibration (must be created by calib_multi_tri_interp and not calib_tri_interp).

Keyword Arguments

numba – boolean, if True Numba JIT will be used (default).

pybundle.SuperRes.**multi_tri_backgrounds**(*calibIn*, *backgrounds*)

Updates a multi_tri calibration with a new set of backgrounds without requiring full recalibration.

Returns instance of BundleCalibration.

Parameters

- **calibIn** – bundle calibration, instance of BundleCalibration
- **background** – background image as 2D numpy array

Registration

pybundle.SuperRes.**get_shifts**(*imgs*, *templateSize=None*, *refSize=None*, *upsample=2*, ***kwargs*)

Determines the shift of each image in a stack w.r.t. first image

Return shifts as 2D numpy array.

Parameters

imgs – stack of images as 3D numpy array

Keyword Arguments

- **templateSize** – int, a square of this size is extracted from imgs as the template, default is 1/4 image size
- **refSize** – int, a square of this size is extracted from first image as the reference image, default is 1/2 image size. Must be bigger than templateSize and the maximum shift detectable is (refSize - templateSize)/2
- **upSample** – upsampling factor for images before shift detection for sub-pixel accuracy, default is 2.

pybundle.SuperRes.**find_shift**(*img1*, *img2*, *templateSize*, *refSize*, *upsample*, *returnMax=False*)

Determines shift between two images by Normalised Cross Correlation (NCC). A square template extracted from the centre of img2 is compared with a square region extracted from the reference image img1. The size of the template (templateSize) must be less than the size of the reference (refSize). The maximum detectable shift is (refSize - templateSize) / 2.

If returnMax is False, returns shift as a tuple of (x_shift, y_shift). If returnMax is True, returns tuple of (shift, cc. peak value).

Parameters

- **img1** – image as 2D numpy array
- **img2** – image as 2D numpy array
- **templateSize** – int, size of square region of img2 to use as template.
- **refSize** – int, size of square region of img1 to template match with
- **upsample** – int, factor to scale images by prior to template matching to allow for sub-pixel registration.

Keyword Arguments

returnMax – boolean, if true returns cc.peak value as well as shift, default is False.

Parameterisation

pybundle.SuperRes.**calib_param_shift**(*param*, *images*, *calibration*)

For use when the shifts between the images are linearly dependent on some other parameter. Provide a TRILIN calibration and a 4D stack of images of (x, y, shift, parameter), i.e. an extra dimension to provide examples of shifts for different values of the parameter. The values of the parameter corresponding to each set of images is provided in param, i.e. the fourth dimension of images should be the same length as param.

Returns a 3D array of calibration factors, giving the gradient and offset of x and y shifts of each image with respect to the parameter.

pybundle.SuperRes.**get_param_shift**(*param*, *calib*)

For use when the shifts between the images are linearly dependent on some other parameter. Assuming a prior calibration using calib_param_shift in 'calib', this function returns the current value of the parameter to obtain the image shifts.

pybundle.SuperRes.**param_calib_multi_tri_interp**(*calibImg*, *imgs*, *coreSize*, *gridSize*, *shiftsSet*, ***kwargs*)

Performs the calibration step for super-resolution reconstruction multiple times for a set of different shifts. All other parameters are the same as for calib_multi_tri_interp.

Utility

pybundle.SuperRes.**sort_sr_stack**(*stack*, *stackLength*)

Takes a stack of images and extracts an ordered set of images relative to a reference 'blank' frame which is much lower intensity than the other frames.

The blank frame can be anywhere in the stack, and the output stack will be formed cyclically from frames before and after the blank frame. For example, if we have frames:

1 2 3 B 4 5

where B is the blank frame, the function will return a stack in the following order:

4 5 1 2 3

The input stack, 'stack' should should have (stackLength + 1) frames to ensure that a and there must be stackLength + 1 images in each cycle (i.e. stackLength useful images plus one blank reference image). The blank reference image is not returned, i.e the returned stack has stackLength frames.

Input stack should have frame number in third dimension.

Parameters

- **stack** – 3D numpy array (y,x, image_number)
- **stackLength** – number of image after blank frame

1.6.9 Implementation Details

`calib_multi_tri_interp` first calls `calib_tri_interp` to perform the standard calibration for triangular linear interpolation. This obtains the core locations, using `find_cores`. If the optional parameters `normToImage` or `normToBackground` are set to `True`, then the mean image intensity for ether the images stack or the background stack (supplied as a further optional parameter `backgroundImgs`) are calculated and stored. These are then later used to normalise each of the input images to a constant mean intensity. This is important for applications where the illumination intensity will be different for each image, but in most applications would not be needed. It is also possible to provide a 1D array of normalisation factors directly as the `imageScaleFactor` parameter.

`calib_multi_tri_interp` then calculates the relative shifts between the supplied images in `imgs` using `get_shifts` via normalised cross correlation. Alternatively, shifts can be provided via the optional parameter `shifts`. For each image, the recorded core positions are then translated by the measured shifts, and a single list of shifted core positions is assembled, containing the shifted core positions from all the images. `init_tri_interp` is then called, which forms a Delaunay triangulation over this set of core positions. For each pixel in the reconstruction grid the enclosing triangle is identified and the pixel location in barycentric co-ordinates is recorded.

Reconstruction is performed using `recon_multi_tri_interp`. The intensity value from each core in each of the images are extracted, and then pixel values in the final image are interpolated linearly from the three surrounding (shifted) cores, using the pre-calculated barycentric distance weights.

The SR calibration is stored in an instane of `BundleCalibration`. This is an extension of the regular TRILIN calibration, and so this super-resolution calibration can be used for non-super-resolution reconstructions (but not vice-versa).

1.6.10 Examples

Examples are provided in "examples\super_res_example" for use via PyBundle class, and "examples\super_res_example_low_level.py" for calling lower-level functions directly.

1.7 Using the Low-Level Functions

While the `PyBundle` class is the recommended way to use most functinality. the lower-level functions can be called directly if greater control is needed.

The fuctions are documented fully in the Function Reference.

Begin by importing the package:

```
import pybundle
```

To locate the bundle in an image, we use:

```
loc = pybundle.find_bundle(img)
```

This works best for background or structureless images acquired through a bundle. The function returns `loc` which is a tuple of the (x_centre, y _entre, radius).

If we would like to mask out pixels outside of the image, we can use:

```
maskedImg = pybundle.auto_mask(img)
```

Alternatively, we can generate a mask using:

```
mask = pybundle.get_mask(img, loc)
```

and apply this mask to any future image using:

```
maskedImg = pybundle.apply_mask(img, mask)
```

This is more useful in general, since the location of the bundle is best determined using a calibration image, and the same mask can then be used for all subsequent images.

We can also crop the image to a square around the bundle using:

```
croppedImg, newloc = pybundle.crop_rect(img, loc)
```

where we have specified the bundle location `loc`, a tuple of (x_centre, y_centre, radius) as output by `find_bundle`. Note that the output is a tuple of `(image, newloc)` where `newloc` is the new location of the bundle in the cropped image.

To crop and mask an image in a single step use:

```
croppedImg = pybundle.auto_mask_crop(img)
```

Spatial filtering can be used to remove the core pattern (alternatively, linear interpolation is also available). To apply a Gaussian smoothing filter, use:

```
smoothedImg = pubundle.g_filter(img, filterSize)
```

where `filterSize` is the sigma of the 2D Gaussian smoothing kernel. A convenient function to filter, mask and crop an image is given by:

```
smoothedImg = pybundle.crop_filter_mask(img, loc, mask)
```

where `loc` is the location of the bundle, determined using `find_bundle` on a calibraton image, and `mask` is a mask created by `get_mask`.

The core spacing of the bundle can be found using:

```
coreSpacing = pybundle.get_core_spacing(img)
```

This can then be used to define a custom edge filter using:

```
filter = pybundle.edge_filter(img,  edgeLocation, edgeSlope)
```

This defines a Fourier domain filter with a cosine smoothed cut-off at the spatial frequency corresponding to the spatial distance `edgeLocation`. `edgeSlope` defines the smoothness of the cut-off; a value of 0 gives a rectangular function. `img` merely needs to be a numpy array the same size as the image(s) to be filtered. `edgeLocation` should typically be `1.6 * coreSpacing`, and `edgeSlope` is not critical, but a value of `0.1 * coreSpacing` generally works well. To apply the filter use:

```
smoothedImg = pybundle.filter_image(img, filter)
```

Note that this kind of filtering is currently quite slow.

To perform linear interpolation using the low-level functions, first perform a calibration using the calibration image `calibImg`, a 2D numpy array:

```
coreSize = 3
gridSize = 512
calib = pybundle.calib_tri_interp(calibImg, coreSize, gridSize,
                                  normalise = calibImg, automask = True)
```

Here we have specified `coreSize = 3` which is the approximate core spacing in the image. This assists the calibration routine in finding all cores. If this is unknown it can be estimate using `find_core_spacing()`.

The `gridSize` is the number of pixels in each dimensions of the reconstructed image, which is square.

Finally, we have specified to use the `calibImg` for normalisation. This means that the intensity extracted from each core during imaging will be normalised with respect to the intensity from the calibration image, removing effects due to non-uniform cores. If this is not done (i.e. normalise is left as the default `None`) then images may appear grainy.

To reconstruct an image `img`, a 2D numpy array, we then call:

```
imgRecon = pybundle.recon_tri_interp(img, calib)
```

This returns a 2D/3D numpy array of size `(gridSize, gridSize, colour channels)` containing the image with the core pattern removed.

For all optional parameters refer to the *function reference* for `calib_tri_interp` and `recon_tri_interp`.

1.8 Function Reference

A list of lower-level functions is available below, see the documentation for individual classes for class methods.

The PyBundle class implements most of the functionality of the package and is the preferred approach for most applications except for Mosaicing, which is handled by the Mosaic class.

PyFibreBundle uses numpy arrays as images throughout, wherever 'image' is specified this refers to a 2D (monochrome) or 3D (colour) numpy array. For colour images, the colour channels are along the third axis. There can be as many colour channels as needed.

1.8.1 Classes

`PyBundle()`

Provides object-oriented access to core functionality of PyFibreBundle. See PyBundle class for details.

`Mosaic()`

Provides object-oriented access to mosaicing functionality of PyFibreBundle. See Mosaic class for details.

`SuperRes()`

Functions for super-resolution. Normally these should be accessed using the PyBundle class. See Super Resolution Section for details.

BundleCalibration()

Stores a calibration for triangular linear interpolation, both normal and super-resolution.

1.8.2 Low-Level Functions for Bundle finding, cropping, masking

pybundle.**auto_mask**(*img*, *loc=None*, ***kwargs*)

Locates bundle and sets pixels outside to 0 .

Parameters
img – input image as 2D numpy array

Keyword Arguments

- **loc** – optional location of bundle as tuple of (centreX, centreY, radius), defaults to determining this using find_bundle
- **radius** – optional, int, radius of mask to use rather than the automatically determined radius
- **Others** – if loc is not specified, other optional keyword arguments will be passed to find_bundle.

pybundle.**auto_mask_crop**(*img*, *loc=None*, ***kwargs*)

Locates bundle, sets pixels outside to 0, and returns cropped image around bundle.

Parameters
img – input image as 2D numpy array

Keyword Arguments

- **loc** – optional location of bundle as tuple of (centreX, centreY, radius), defaults to determining this using find_bundle
- **Others** – if loc is not specified, other optional keyword arguments will be passed to find_bundle.

pybundle.**apply_mask**(*img*, *mask*)

Sets all pixels outside bundle to 0 using a pre-defined mask. If the image is 3D, the mask will be applied to each colour plane.

Parameters

- **img** – input image as 2D numpy array
- **mask** – mask as 2D numy array with same dimensions as img, with areas to be kept as 1 and areas to be masked as 0.

pybundle.**crop_rect**(*img*, *loc*)

Extracts a square around the bundle using specified co-ordinates. If the rectange is larger than the image then the returned image will be a rectangle, limited by the extent of the image.

Returns tuple of (cropped image as 2D numpy array, new location tuple)

Parameters

- **img** – input image as 2D numpy array
- **loc** – location to crop, specified as bundle location tuple of (centreX, centreY, radius)

pybundle.**find_bundle**(*img*, ***kwargs*)

Locate fibre bundle by thresholding and searching for largest connected region.

Returns tuple of (centreX, centreY, radius).

Parameters
img – input image of fibre bundle, 2D numpy array

Keyword Arguments
filterSize – sigma of Gaussian filter applied to remove core pattern, defaults to 4

pybundle.**find_core_spacing**(*img*)

Estimates fibre bundle core spacing using peak in 2D Fourier transform.

If the image is not square, a square will be cropped from the centre. It is therefore usually best to crop the image to the bundle before passing it to this function.

Returns core spacing as float.

Parameters
img – input image showing bundle as 2D/3D numpy array

pybundle.**get_mask**(*img*, *loc*)

Returns a circular mask, 1 inside bundle, 0 outside bundle, using specified bundle co-ordinates. Mask image has same dimensions as first two dimensions of input image (i.e. does not return a mask for each colour plane).

Returns mask as 2D numpy array.

Parameters

- **img** – img used to determine size of mask, 2D numpy array
- **loc** – location of bundle used to determine location of mask, tuple of (centreX, centreY, radius)

1.8.3 Low Level Functions for Spatial Filtering

pybundle.**g_filter**(*img*, *filterSize*, *kernelSize=None*)

Applies 2D Gaussian filter to image. By default the kernel size is 4 times the filter_size (sigma).

Returns filtered image as numpy array.

Parameters

- **img** – input image as 2D/3D numpy array
- **filterSize** – float, sigma of Gaussian filter

Keyword Arguments
kernelSize – int, size of convolution kernal

pybundle.**crop_filter_mask**(*img*, *loc*, *mask*, *filterSize*, *resize=False*, ***kwargs*)

For convenient quick processing of images. Sequentially crops image to bundle, applies Gaussian filter and then sets pixels outside bundle to 0. Set loc to None to automatically locate bundle. Optional parameter 'resize' allows the output images to be rescaled. If using auto-locate, can optionally specify find_bundle() options as additional keyword arguments.

Returns output image as 2D/3D numpy array

Parameters

- **img** – input image, 2D/3D numpy array

- **loc** – location of bundle as tuple of (centreX, centreY, radius), set to None to determining this using find_bundle
- **mask** – 2D numpy array with value of 1 inside bundle and 0 outside bundle
- **filterSize** – sigma of Gaussian filter

Keyword Arguments

- **resize** – size to rescale output image to, default is no resize
- **Others** – if loc is not specified, other optional keyword arguments will be passed to find_bundle.

pybundle.**edge_filter**(*imgSize*, *edgePos*, *skinThickness*)

Creates a 2D edge filter with cosine smoothing.

Returns the filter in spatial frequency domain filter as a 2D numpy array.

Parameters

- **imgSize** – size of (square) images which will be processed, and size of filter output
- **edgePos** – spatial frequency of cut-off
- **skinThickness** – slope of edge, the distance, in spatial frequency, over which it goes from 90% to 10%

pybundle.**filter_image**(*img*, *filt*)

Applies a Fourier domain filter to an image, such as created by edge_filter(). Filter must be same size as image (x and y) but not multi-channel (i.e. a 2D array).

Returns filtered image as 2D/3D numpy array.

Parameters

- **img** – input image (spatial domain), 2D/3D numpy array
- **filt** – spatial frequency domain representation of filter, 2D numpy array

pybundle.**median_filter**(*img*, *filterSize*)

Applies 2D median filter to an image.

Returns the filtered image as a 2D numpy array.

Parameters

- **img** – input image as 2D/3D numpy array
- **filterSize** – float, sigma of Gaussian filter

1.8.4 Functions for Triangular Linear Interpolation

High-level functions

pybundle.**calib_tri_interp**(*img*, *coreSize*, *gridSize*, ***kwargs*)

Performs calibration to allow subsequent core removal by triangular linear interpolation. Reconstructed images will be of size (gridSize, gridSize). 'coreSize' is used by the core finding routine, and should be an estimate of the core spacing. This function returns the entire calibration as an instance of BundleCalibration which can subsequently by used by recon_tri_interp. If background and/or normalisation images are specified, subsequent reconstructions will have background subtraction and/or normalisation respectively.

Returns an instance of BundleCalibration

Thanks to Cheng Yong Xin, Joseph, who collaborated in implementation of this function.

Parameters

- **img** – calibration image of bundle as 2D (mono) or 3D (colour) numpy array
- **coreSize** – float, estimate of average spacing between cores
- **gridSize** – int, output size of image, supply a single value, image will be square

Keyword Arguments

- **centreX** – int, optional, x centre location of bundle, if not specified will be determined automatically
- **centreY** – int, optional, y centre location of bundle, if not specified will be determined automatically
- **radius** – int, optional, radius of bundle, if not specified will be determined automatically
- **filterSize** – float, optional, sigma of Gaussian filter applied, defaults to 0 (no filter)
- **background** – optional, image used for background subtraction as 2D numpy array
- **normalise** – optional, image used for normalisation, as 2D numpy array. Can be same as calibration image, defaults to no normalisation
- **autoMask** – optional, boolean, if True the calibration image will be masked to prevent spurious core detections outside of bundle, defualts to True
- **mask** – optional, boolean, when reconstructing output image will be masked outside of bundle, defaults to True
- **whiteBalance** – optional, boolean, if True then each colour channel is normalised individually, defaults to False.

pybundle.**recon_tri_interp**(*img*, *calib*, ***kwargs*)

Removes core pattern using triangular linear interpolation. Requires an initial calibration using calib_tri_interp.

Returns reconstructed image as 2D/3D numpy array.

Parameters

- **img** – raw image to be reconstructed as 2D (mono) or 3D (colour) numpy array
- **calib** – bundle calibration as instance of BundleCalibration

Keyword Arguments

numba – optional, if true use JIT acceleration using Numba, default is False

Low-level functions

pybundle.**find_cores**(*img*, *coreSpacing*)

Find cores in bundle image using regional maxima. Generally fast and accurate.

Returns tuple of (x_pos, y_pos) where x_pos and y_pos are 1D numpy arrays.

Parameters

- **img** – 2D/3D numpy array
- **coreSpacing** – float, estimate of the separation between cores in pixels.

pybundle.**core_values**(*img*, *coreX*, *coreY*, *filterSize*, ***kwargs*)

Extract intensity of each core in fibre bundle image. First applies a Gaussian filter unless filterSize is None. Supports JIT acceleration if numba is installed.

Parameters

- **coreX** – 1D numpy array giving x co-ordinates of core centres
- **coreY** – 1D numpy array giving y co-ordinates of core centres
- **filterSize** – float, sigma of Gaussian filter

Keyword Arguments

numba – optional, if true numba JIT used for faster execution, defaults to False.

pybundle.**init_tri_interp**(*img*, *coreX*, *coreY*, *centreX*, *centreY*, *radius*, *gridSize*, ***kwargs*)

Used by calib_tri_interp to perform Delaunay triangulation of core positions, and find each pixel of reconstruction grid in barycentric co-ordinates w.r.t. enclosing triangle.

Returns instance of BundleCalibration.

Parameters

- **img** – calibration image as 2D (mono) or 3D (colour) numpy array
- **coreX** – x centre of each core as 1D numpy array
- **coreY** – y centre of each core as 1D numpy array
- **centreX** – x centre location of bundle (reconstruction will be centred on this)
- **centreY** – y centre location of bundle (reconstruction will be centred on this)
- **radius** – radius of bundle (reconstruction will cover a square out to this radius)

Keyword Arguments

- **gridSize** – output size of image, supply a single value, image will be square
- **filterSize** – optional, sigma of Gaussian filter applied, defaults to no filter
- **background** – optional, image used for background subtractionn as 2D numpy array
- **normalise** – optional, image used for normalisation, as 2D numpy array. Can be same as calibration image, defaults to no normalisation
- **mask** – optional, boolean, when reconstructing output image will be masked outside of bundle, defaults to True
- **whiteBalance** – optional, boolean, if True then each colour channel is normalised individually, defaults to False.

1.8.5 Utility Functions

pybundle.**average_channels**(*img*)

Returns an image which is the the average pixel value across all channels of a colour image. It is safe to pass a 2D array which will be returned unchanged.

Parameters

img – image as 2D/3D numpy array

pybundle.**extract_central**(*img*, *boxSize=None*)

Extract a central square from an image. The extracted square is centred on the input image, with size 2 * boxSize if possible, otherwise the largest square that can be extracted.

Returns cropped image as 2D numpy array.

Parameters
img – input image as 2D numpy array

Keyword Arguments
boxSize – size of cropping square, default is largest possible

pybundle.**max_channels**(*img*)

Returns an image which is the the maximum pixel value across all channels of a colour image. It is safe to pass a 2D array which will be returned unchanged.

Parameters
img – image as 2D/3D numpy array

pybundle.**radial_profile**(*img*, *centre*)

Produce angular averaged radial profile through image img centred on centre, a tuple of (x_centre, y_centre)

Returns radial profile as 1D numpy array

Parameters

- **img** – input image as 2D numpy array
- **centre** – centre point for radial profile, tuple of (x,y)

pybundle.**save_image8**(*img*, *filename*)

Saves image as 8 bit tif without scaling

pybundle.**save_image8_scaled**(*img*, *filename*)

Saves image as 8 bit tif with scaling to use full dynamic range.

Parameters

- **img** – image as 2D/3D numpy array
- **filname** – str, path to file. Folder must exist.

pybundle.**save_image16**(*img*, *filename*)

Saves image as 16 bit tif without scaling.

Parameters

- **img** – image as 2D/3D numpy array
- **filname** – str, path to file. Folder must exist.

pybundle.**save_image16_scaled**(*img*, *filename*)

Saves image as 16 bit tif with scaling to use full dynamic range.

Parameters

- **img** – image as 2D/3D numpy array
- **filname** – str, path to file. Folder must exist.

pybundle.**to8bit**(*img*, ***kwargs*)

Returns an 8 bit representation of image. If min and max are specified,
these pixel values in the original image are mapped to 0 and 255 respectively, otherwise the smallest and largest values in the whole image are mapped to 0 and 255, respectively.

Arguments:
img : input image as 2D numpy array

Keyword Arguments

- **minVal** – optional, pixel value to scale to 0
- **maxVal** – optional, pixel value to scale to 255

pybundle.**to16bit**(*img*, ***kwargs*)

Returns an 16 bit representation of image. If min and max are specified, these pixel values in the original image are mapped to 0 and 2^16 respectively, otherwise the smallest and largest values in the whole image are mapped to 0 and 2^16 - 1, respectively.

Parameters
img – input image as 2D numpy array

Keyword Arguments

- **minVal** – optional, pixel value to scale to 0
- **maxVal** – optional, pixel value to scale to 2^16 - 1

- genindex

Acknowledgements: Cheng Yong Xin, Joseph, contributed to triangular linear interpolation; Petros Giataganas who developed some of the Matlab code that parts of this library were ported from. Funding to Mike Hughes's lab from EPSRC (Ultrathin fluorescence microscope in a needle, EP/R019274/1), Royal Society (Ultrathin Inline Holographic Microscopy).

INDEX

T