pystackreg

Release 0.2.5

Gregor Lichtner, Philippe Thevenaz

May 23, 2021

CONTENTS:

1 pyStackReg **1**
- 1.1 Summary . . . 1
- 1.2 Description . . . 1
- 1.3 Installation . . . 2
- 1.4 Documentation . . . 2
- 1.5 Tutorial . . . 2
- 1.6 Usage . . . 2
- 1.7 Author information . . . 4
- 1.8 License . . . 5

2 pystackreg tutorial **7**
- 2.1 Import pystackreg . . . 7
- 2.2 Perform single image registration . . . 8
- 2.3 Perform stack transformations . . . 11
- 2.4 Use saved transformation matrices . . . 14
- 2.5 Custom callback for registration progress . . . 15
- 2.6 Clipping of negative values . . . 16
- 2.7 Registration of stacks having the frames/("time") not as the first axis . . . 16

3 API **19**
- 3.1 Overview . . . 19
- 3.2 Registration & Transformation . . . 19
- 3.3 Utility functions . . . 19

4 Authors **21**

5 Changelog **23**
- 5.1 0.2.5 . . . 23
- 5.2 0.2.3 . . . 23
- 5.3 0.2.2 . . . 24
- 5.4 0.2.1 . . . 24
- 5.5 0.2.0 . . . 24

6 Indices and tables **25**

CHAPTER

ONE

PYSTACKREG

1.1 Summary

Python/C++ port of the ImageJ extension TurboReg/StackReg written by Philippe Thevenaz/EPFL.

A python extension for the automatic alignment of a source image or a stack (movie) to a target image/reference frame.

1.2 Description

pyStackReg is used to align (register) one or more images to a common reference image, as is required usually in time-resolved fluorescence or wide-field microscopy. It is directly ported from the source code of the ImageJ plugin `TurboReg` and provides additionally the functionality of the ImageJ plugin `StackReg`, both of which were written by Philippe Thevenaz/EPFL (available at http://bigwww.epfl.ch/thevenaz/turboreg/).

pyStackReg provides the following five types of distortion:

- translation
- rigid body (translation + rotation)
- scaled rotation (translation + rotation + scaling)
- affine (translation + rotation + scaling + shearing)
- bilinear (non-linear transformation; does not preserve straight lines)

pyStackReg supports the full functionality of StackReg plus some additional options, e.g., using different reference images and having access to the actual transformation matrices (please see the examples below). Note that pyStackReg uses the high quality (i.e. high accuracy) mode of TurboReg that uses cubic spline interpolation for transformation.

Please note: The bilinear transformation cannot be propagated, as a combination of bilinear transformations does not generally result in a bilinear transformation. Therefore, stack registration/transform functions won't work with bilinear transformation when using "previous" image as reference image. You can either use another reference ("first" or "mean" for first or mean image, respectively), or try to register/transform each image of the stack separately to its respective previous image (and use the already transformed previous image as reference for the next image).

1.3 Installation

The package is available on conda forge and on PyPi.

- Install using **conda**

```
conda install pystackreg -c conda-forge
```

- Install using **pip**

```
pip install pystackreg
```

1.4 Documentation

The documentation can be found on readthedocs:

https://pystackreg.readthedocs.io/

1.5 Tutorial

- A tutorial notebook can be found in the *examples/notebooks* folder or statically here: https://pystackreg.readthedocs.io/en/latest/tutorial.html

1.6 Usage

The following example opens two different files and registers them using all different possible transformations

```
from pystackreg import StackReg
from skimage import io

#load reference and "moved" image
ref = io.imread('some_original_image.tif')
mov = io.imread('some_changed_image.tif')

#Translational transformation
sr = StackReg(StackReg.TRANSLATION)
out_tra = sr.register_transform(ref, mov)

#Rigid Body transformation
sr = StackReg(StackReg.RIGID_BODY)
out_rot = sr.register_transform(ref, mov)

#Scaled Rotation transformation
sr = StackReg(StackReg.SCALED_ROTATION)
out_sca = sr.register_transform(ref, mov)

#Affine transformation
sr = StackReg(StackReg.AFFINE)
out_aff = sr.register_transform(ref, mov)
```

(continues on next page)

(continued from previous page)

```
#Bilinear transformation
sr = StackReg(StackReg.BILINEAR)
out_bil = sr.register_transform(ref, mov)
```

The next example shows how to separate registration from transformation (e.g., to register in one color channel and then use that information to transform another color channel):

```
from pystackreg import StackReg
from skimage import io

img0 = io.imread('some_multiframe_image.tif')
img1 = io.imread('another_multiframe_image.tif')
# img0.shape: frames x width x height (3D)

sr = StackReg(StackReg.RIGID_BODY)

# register 2nd image to 1st
sr.register(img0[0, :, :], img0[1,:,:])

# use the transformation from the above registration to register another frame
out = sr.transform(img1[1,:,:])
```

The next examples shows how to register and transform a whole stack:

```
from pystackreg import StackReg
from skimage import io

img0 = io.imread('some_multiframe_image.tif') # 3 dimensions : frames x width x height

sr = StackReg(StackReg.RIGID_BODY)

# register each frame to the previous (already registered) one
# this is what the original StackReg ImageJ plugin uses
out_previous = sr.register_transform_stack(img0, reference='previous')

# register to first image
out_first = sr.register_transform_stack(img0, reference='first')

# register to mean image
out_mean = sr.register_transform_stack(img0, reference='mean')

# register to mean of first 10 images
out_first10 = sr.register_transform_stack(img0, reference='first', n_frames=10)

# calculate a moving average of 10 images, then register the moving average to the mean␣
↪of
# the first 10 images and transform the original image (not the moving average)
out_moving10 = sr.register_transform_stack(img0, reference='first', n_frames=10, moving_
↪average = 10)
```

The next example shows how to separate registration from transformation for a stack (e.g., to register in one color channel and then use that information to transform another color channel):

```
from pystackreg import StackReg
from skimage import io

img0 = io.imread('some_multiframe_image.tif') # 3 dimensions : frames x width x height
img1 = io.imread('another_multiframe_image.tif') # same shape as img0

# both stacks must have the same shape
assert img0.shape == img1.shape

sr = StackReg(StackReg.RIGID_BODY)

# register each frame to the previous (already registered) one
# this is what the original StackReg ImageJ plugin uses
tmats = sr.register_stack(img0, reference='previous')
out = sr.transform_stack(img1)

# tmats contains the transformation matrices -> they can be saved
# and loaded at another time
import numpy as np
np.save('transformation_matrices.npy', tmats)

tmats_loaded = np.load('transformation_matrices.npy')

# make sure you use the correct transformation here!
sr = StackReg(StackReg.RIGID_BODY)

# transform stack using the tmats loaded from file
sr.transform_stack(img1, tmats=tmats_loaded)

# with the transformation matrices at hand you can also
# use the transformation algorithms from other packages:
from skimage import transform as tf

out = np.zeros(img0.shape).astype(np.float)

for i in range(tmats.shape[0]):
    out[i, :, :] = tf.warp(img1[i, :, :], tmats[i, :, :], order=3)
```

1.7 Author information

This is a port of the original Java code by Philippe Thevenaz to C++ with a Python wrapper around it. All credit goes to the original author:

```
/*====================================================================
| Philippe Thevenaz
| EPFL/STI/IMT/LIB/BM.4.137
| Station 17
| CH-1015 Lausanne VD
| Switzerland
|
| phone (CET): +41(21)693.51.61
```

(continues on next page)

(continued from previous page)

```
| fax: +41(21)693.37.01
| RFC-822: philippe.thevenaz@epfl.ch
| X-400: /C=ch/A=400net/P=switch/O=epfl/S=thevenaz/G=philippe/
| URL: http://bigwww.epfl.ch/
\===================================================================*/

/*====================================================================
| This work is based on the following paper:
|
| P. Thevenaz, U.E. Ruttimann, M. Unser
| A Pyramid Approach to Subpixel Registration Based on Intensity
| IEEE Transactions on Image Processing
| vol. 7, no. 1, pp. 27-41, January 1998.
|
| This paper is available on-line at
| http://bigwww.epfl.ch/publications/thevenaz9801.html
|
| Other relevant on-line publications are available at
| http://bigwww.epfl.ch/publications/
\===================================================================*/
```

1.8 License

```
You are free to use this software for commercial and non-commercial
purposes. However, we expect you to include a citation or acknowledgement
whenever you present or publish research results that are based
on this software. You are free to modify this software or derive
works from it, but you are only allowed to distribute it under the
same terms as this license specifies. Additionally, you must include
a reference to the research paper above in all software and works
derived from this software.
```

CHAPTER

TWO

PYSTACKREG TUTORIAL

This notebook demonstrates the uses of pystackreg on an example dataset of a live neuronal structure.

2.1 Import pystackreg

```
import os
import sys
import numpy as np
from matplotlib import pyplot as plt
from skimage import transform, io, exposure

from pystackreg import StackReg
import pystackreg
```

```
print(pystackreg.__version__)
```

```
0.2.3rc1
```

2.1.1 Define helper functions

```
def overlay_images(imgs, equalize=False, aggregator=np.mean):

    if equalize:
        imgs = [exposure.equalize_hist(img) for img in imgs]

    imgs = np.stack(imgs, axis=0)

    return aggregator(imgs, axis=0)
```

```
def composite_images(imgs, equalize=False, aggregator=np.mean):

    if equalize:
        imgs = [exposure.equalize_hist(img) for img in imgs]

    imgs = [img / img.max() for img in imgs]

    if len(imgs) < 3:
```

(continues on next page)

(continued from previous page)

```
        imgs += [np.zeros(shape=imgs[0].shape)] * (3-len(imgs))

    imgs = np.dstack(imgs)

    return imgs
```

2.1.2 Load sample data

```
data_path = os.path.join(os.getcwd(), '..', 'data')
unreg = io.imread(os.path.join(data_path, 'pc12-unreg.tif'))

# convert to float to prevent overflows when calculating differences images
unreg = unreg.astype(np.float)
```

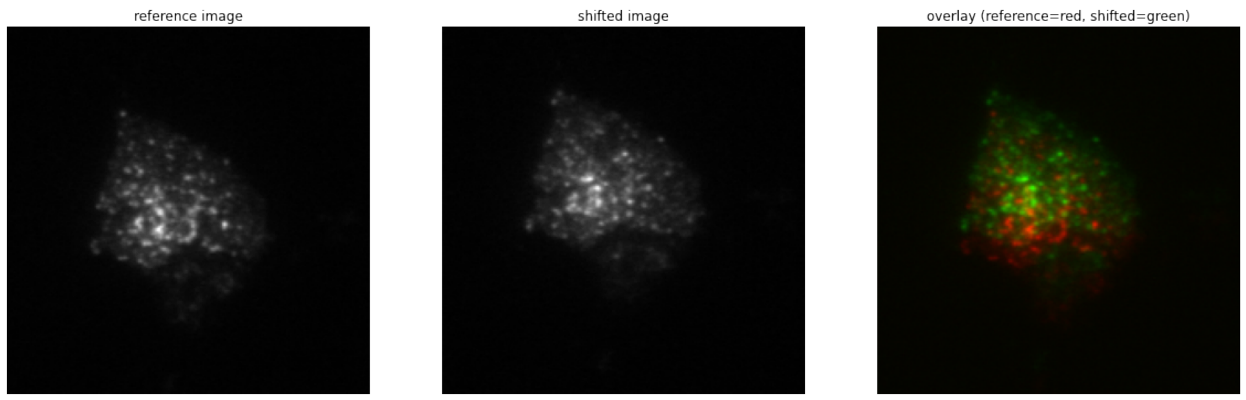
2.2 Perform single image registration

```
transformations = {
    'TRANSLATION': StackReg.TRANSLATION,
    'RIGID_BODY': StackReg.RIGID_BODY,
    'SCALED_ROTATION': StackReg.SCALED_ROTATION,
    'AFFINE': StackReg.AFFINE,
    'BILINEAR': StackReg.BILINEAR
}
```

```
#load reference and "moved" image
ref = unreg[0, :, :]
mov = unreg[4, :, :]
```

2.2.1 Show original images

```
f, ax = plt.subplots(1, 3, figsize=(20, 9))

before_reg = composite_images([ref, mov])

ax[0].imshow(ref, cmap='gray')
ax[0].set_title('reference image')
ax[0].axis('off')

ax[1].imshow(mov, cmap='gray')
ax[1].set_title('shifted image')
ax[1].axis('off')

ax[2].imshow(before_reg)
ax[2].set_title('overlay (reference=red, shifted=green)')
ax[2].axis('off');
```



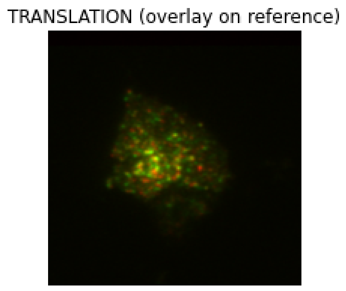
As can be seen from the overlay image (right), the second image (middle) is shifted considerably with respect to the reference image (left).

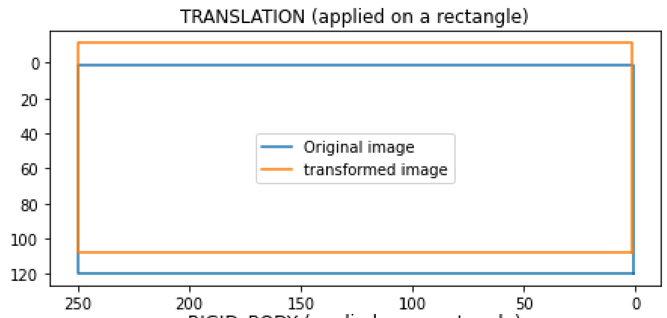
2.2.2 Transform shifted image

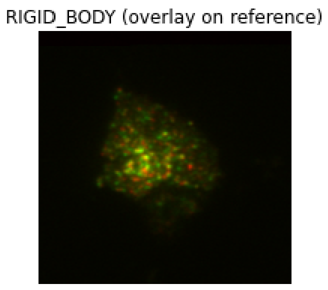
We will next use all available transformation modes from pystackreg to register the shifted image to the reference image.

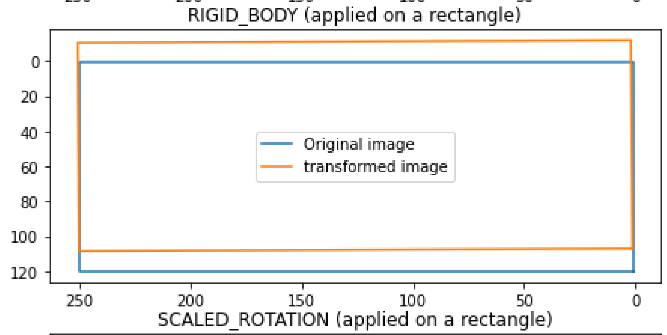
```
def show_transformation(tmat, ax=None):
    if ax is None:
        _, ax = plt.subplots()
    p = np.array([[1,120,1], [1,1,1], [250, 1, 1], [250,120,1], [1,120,1]])
    ax.plot(p[:, 0], p[:,1])
    q=np.dot(p, tmat.T)
    ax.plot(q[:, 0], q[:,1])
    ax.invert_xaxis()
    ax.invert_yaxis()
    ax.legend(['Original image', 'transformed image'])
```

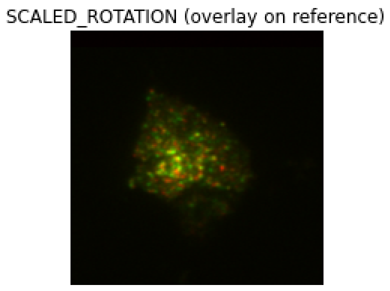
```
f, ax = plt.subplots(5, 2, figsize=(16, 18))

for i, (name, tf) in enumerate(transformations.items()):
    sr = StackReg(tf)
    reg = sr.register_transform(ref, mov)
    reg = reg.clip(min=0)

    after_reg =  composite_images([ref, reg])

    ax[i][0].imshow(after_reg, cmap='gray', vmin=0, vmax=1)
    ax[i][0].set_title(name + ' (overlay on reference)')
    ax[i][0].axis('off')

    if name != 'BILINEAR':
        show_transformation(sr.get_matrix(), ax[i][1])
        ax[i][1].set_title(name + ' (applied on a rectangle)')
    else:
        ax[i][1].axis('off')
```

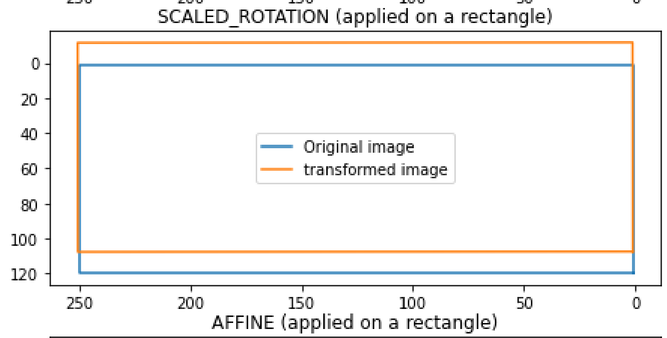


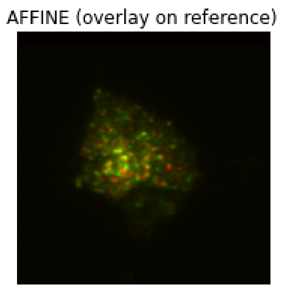


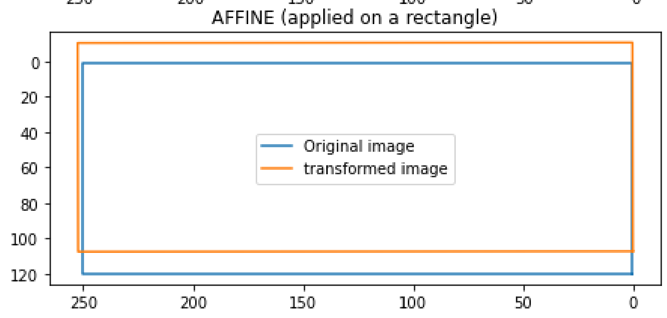


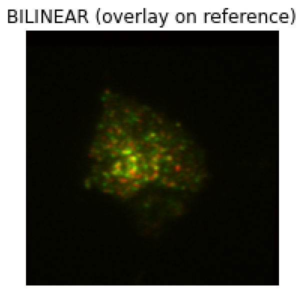












Translation already performs a quite good job at registering the images.

In the right column, the action of the transformation matrix on a rectangle is shown.

2.3 Perform stack transformations

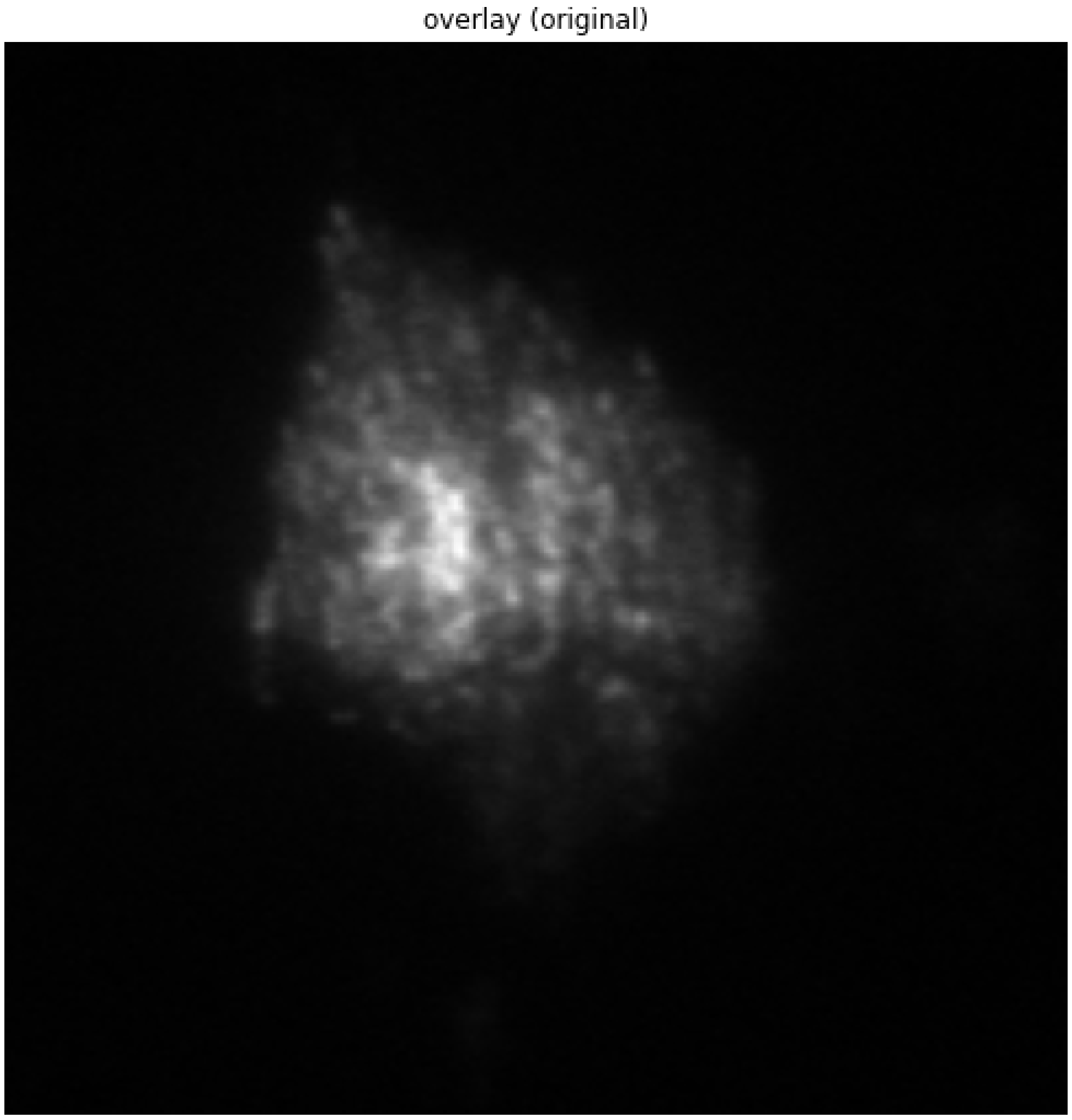
```
print(f"Number of images in the stack: {len(unreg)}")
```

```
Number of images in the stack: 5
```

The original stack contains 5 images, we have so far only looked at the registration of two images out of that stack. We will next see how to use pystackreg to register the whole stack of 5 images.

2.3.1 Show original images (overlay)

```
f, ax = plt.subplots(1, 1, figsize=(10, 9))

ax.imshow(overlay_images(unreg), cmap='gray')
ax.set_title('overlay (original)')
ax.axis('off');
```



The above image is the mean of each of the 5 images of the stack. As can bee seen, it is quite blurry due to the misalignment.

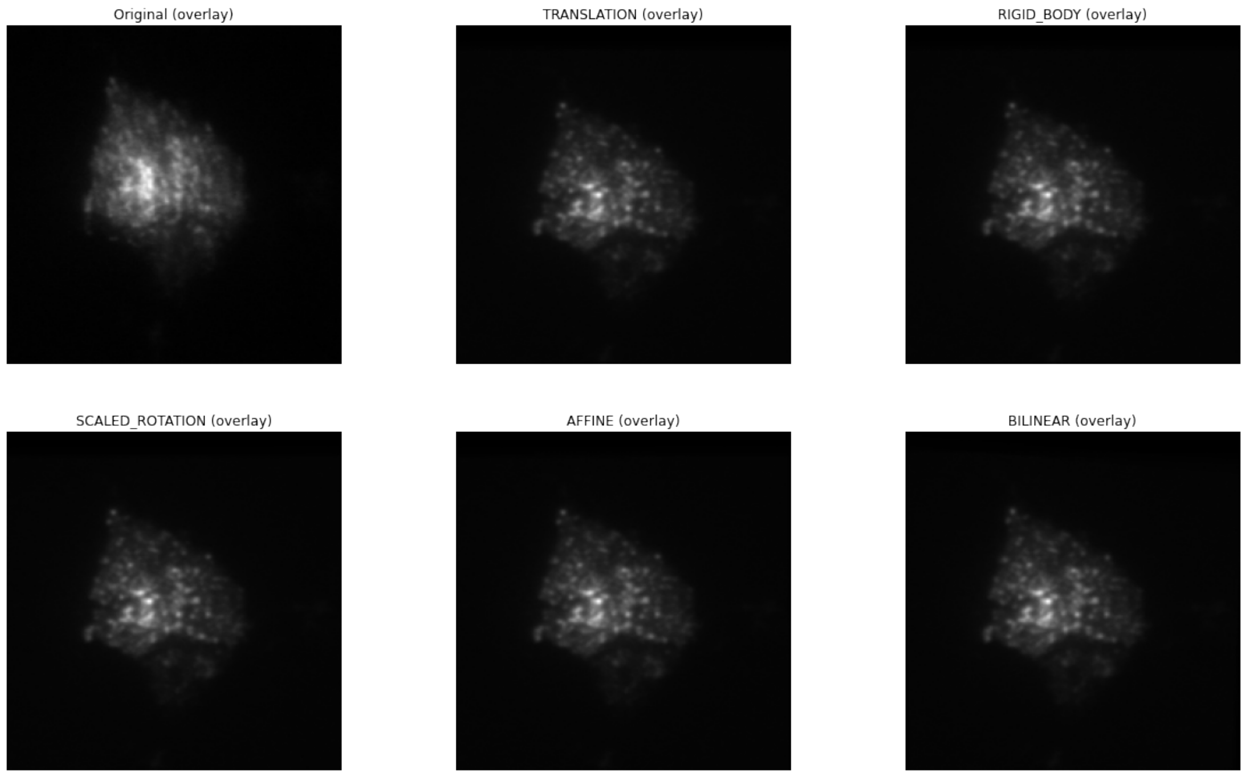
2.3.2 Register and transform the whole stack

We will next use all 5 kinds of transformation offered by pystackreg to register the stack's images. Afterwards the average of the registered images is shown for each of the different types of transformations.

Note that we will save the transformation matrices of each transformation in a list `tmats` for later use.

```
f, ax = plt.subplots(2, int(np.ceil((len(transformations)+1)/2)), figsize=(20, 12))
ax = ax.ravel()

ax[0].imshow(overlay_images(unreg, aggregator=np.mean), cmap='gray')
ax[0].set_title('Original (overlay)')
ax[0].axis('off')

# store transformation matrices for later use in this variable
tmats = []

for i, (name, tf) in enumerate(transformations.items()):
    sr = StackReg(tf)

    reference = 'first' if name == 'BILINEAR' else 'previous'

    tmat = sr.register_stack(unreg, axis=0, reference=reference, verbose=True)
    reg = sr.transform_stack(unreg)

    tmats.append(tmat)

    ax[i+1].imshow(overlay_images(reg, aggregator=np.mean), cmap='gray')
    ax[i+1].set_title(name + ' (overlay)')
    ax[i+1].axis('off')
```

```
100%|| 4/4 [00:00<00:00, 59.11it/s]
100%|| 4/4 [00:00<00:00, 32.64it/s]
100%|| 4/4 [00:00<00:00, 30.12it/s]
100%|| 4/4 [00:00<00:00, 26.77it/s]
100%|| 4/4 [00:00<00:00, 18.93it/s]
```



As can bee seen, all types of transformation lead to a much clearer average image of the stack, equivalent to nicely registered images.

2.4 Use saved transformation matrices

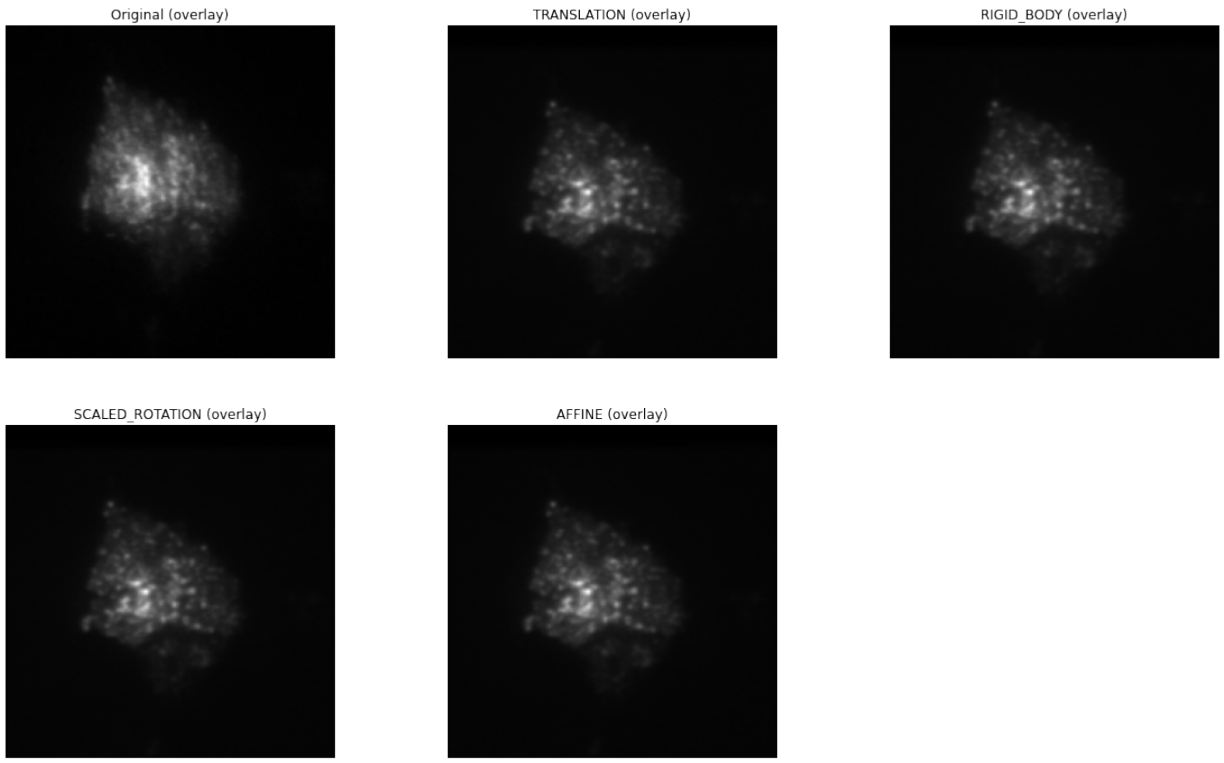
Note that in the above operation we have saved the transformation matrices generated from pystackreg in the variable `tmats`. We will now use the affine transformation implemented in skimage to perform what pystackreg does in its `transform` functions using a different package. This will demonstrate that the transformation matrices supplied by pystackreg can be stored and used in any desired context even without having to use pystackreg.

```
f, ax = plt.subplots(2, int(np.ceil((len(transformations)+1)/2)), figsize=(20, 12))
ax = ax.ravel()

ax[0].imshow(overlay_images(unreg, aggregator=np.mean), cmap='gray')
ax[0].set_title('Original (overlay)')
ax[0].axis('off')

for i, (name, tf) in enumerate(transformations.items()):

    if name == 'BILINEAR':
        # Bilinear transformation is not an affine transformation, we can't use the␣
↪transformation matrix here
        continue

    # copy the unregistered image
    reg = unreg.copy()
```

(continues on next page)

(continued from previous page)

```
    for i_img in range(unreg.shape[0]):
        # get skimage's AffineTransform object
        tform = transform.AffineTransform(matrix=tmats[i][i_img, :, :])

        # transform image using the saved transformation matrix
        reg[i_img, :, :] = transform.warp(reg[i_img, :, :], tform)


    ax[i+1].imshow(overlay_images(reg, aggregator=np.mean), cmap='gray')
    ax[i+1].set_title(name + ' (overlay)')
    ax[i+1].axis('off')

# turn off axis in remaining plots
for i in range(len(transformations), len(ax)):
    ax[i].axis('off')
```



2.5 Custom callback for registration progress

The `register()` and `register_stack()` function can be called with a custom callback function that can be used to display progress information to the user.

The custom callback needs to accept the two parameters `current_iteration` and `end_iteration`:

```
def show_progress(current_iteration, end_iteration):
    print(f"Registering {current_iteration} of {end_iteration} images")

```

(continues on next page)

(continued from previous page)

```
sr = StackReg(StackReg.RIGID_BODY)
reg = sr.register_transform_stack(unreg, axis=0, progress_callback=show_progress)
```

```
Registering 1 of 4 images
Registering 2 of 4 images
Registering 3 of 4 images
Registering 4 of 4 images
```

2.6 Clipping of negative values

The transformation output of pystackreg is exactly equivalent to that of the ImageJ plugins TurboReg/StackReg on which it is based. The output of the transform function therefore has a `float` datatype and may contain negative values. To again create an image with integer values, the utility function `pystackreg.util.to_uint16()` can be used.

```
from pystackreg.util import to_uint16
```

```
sr = StackReg(StackReg.RIGID_BODY)
reg = sr.register_transform_stack(unreg - unreg.min())

print(f"The output of the transform function has the datatype {reg.dtype} with a minimal␣
↪value of {reg.min()}")
```

```
The output of the transform function has the datatype float64 with a minimal value of -
↪29.488252639770508
```

```
reg_int = to_uint16(reg)

print(f"After using the to_uint16 function, the output has a datatype {reg_int.dtype}␣
↪with a minimal value of {reg_int.min()}")
```

```
After using the to_uint16 function, the output has a datatype uint16 with a minimal␣
↪value of 0
```

2.7 Registration of stacks having the frames/("time") not as the first axis

pystackreg expects the frames ("time") axis in a stacked image as the first axis, i.e. for a 3D numpy array, pystackreg expects the dimensions to be `frames x width x height`. pystackreg automatically tries to find out which axis is the frames axis and raises a warning if that axis is not the axis that the user supplied.

2.7.1 Create test data

We will first create a stack that has the frames axis at axis 2 (instead of the default axis 0) by moving the axis from our sample data:

```
unreg.shape
```

```
(5, 201, 199)
```

```
unreg_axis = np.moveaxis(unreg, 0, 2)
unreg_axis.shape
```

```
(201, 199, 5)
```

2.7.2 Register & transform

Next we register the stack using pystackreg's default parameters

```
sr = StackReg(StackReg.RIGID_BODY)
sr.register_stack(unreg_axis);
```

```
/home/lichtneg/anaconda3/envs/py36/lib/python3.6/site-packages/pystackreg-0.2.3rc1-py3.6-
↪linux-x86_64.egg/pystackreg/pystackreg.py:383: UserWarning: Detected axis 2 as the␣
↪possible time axis for the stack due to its low variability, but axis 0 was supplied␣
↪for registration. Are you sure you supplied the correct axis?
  lowest_var_axis, axis
```

Notice that this raises a warning as pystackreg correctly identified axis 2 as the frames axis. So let's use that axis for registration:

```
sr.register_stack(unreg_axis, axis=2);
```

Next transform the stack. Note that we need to supply the axis parameter again here - otherwise an error will be raised.

```
reg_axis = sr.transform_stack(unreg_axis, axis=2)
```

CHAPTER THREE

API

3.1 Overview

This section describes pystackreg's API.

3.2 Registration & Transformation

3.3 Utility functions

CHAPTER FOUR

AUTHORS

- Developer of the original ImageJ plugins TurboReg/StackReg: Philippe Thévenaz (http://bigwww.epfl.ch/thevenaz/)
- Developer of this package (Port of the Java code from P. Thévenaz to C++ and Python wrapper): Gregor Lichtner @glichtner

CHAPTER

FIVE

CHANGELOG

5.1 0.2.5

5.1.1 fixed

- Compilation in environment without numpy

5.2 0.2.3

5.2.1 added

- Added example data and tutorial notebook
- Added unit tests
- Additional documentation
- Detection of time series axis in stacks - will raise a warning if supplied axis in stack registration does not correspond to the detected axis

5.2.2 changed

- *progress_callback* function now gets called with the iteration number, not iteration index (iteration number = iteration index + 1)

5.2.3 fixed

- Fixed exception when using a different axis than 0 for registering stacks

5.3 0.2.2

5.3.1 changed

- License changed to allow distribution on python package repositories

5.4 0.2.1

5.4.1 added

- progress callback function can be supplied to *register_stack()* and *register_transform_stack()* functions via the *progress_callback* that is then called after every iteration (i.e. after each image registration)

5.4.2 changed

- progress bar output not shown by default, has to be enabled by using the *verbose=True* parameter in the *register_stack()* and *register_transform_stack()* functions

5.5 0.2.0

5.5.1 added

- bilinear transformation

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search