astroalign Documentation

Release 2.0

Martin Beroiz

Jul 26, 2019

Contents

ASTROALIGN is a python module that will try to register (align) two stellar astronomical images, especially when there is no WCS information available.

It does so by finding similar 3-point asterisms (triangles) in both images and estimating the affine transformation between them.

Generic registration routines try to match point features, using corner detection routines to make the point correspondence. These generally fail for stellar astronomical images, since stars have very little stable structure and so, in general, indistinguishable from each other. Asterism matching is more robust, and closer to the human way of matching stellar images.

Astroalign can match images of very different fields of view, point-spread functions, seeing and atmospheric conditions.

You can find a Jupyter notebook example with the main features at http://toros-astro.github.io/astroalign.

Note: It may not work, or work with special care, on images of extended objects with few point-like sources or in very crowded fields.

Note: If your images contain a large number of hot pixels, this may result in an incorrect registration. Please refer to the tutorial for how to solve this problem using CCDProc's cosmic-ray remover.

CHAPTER 1

Guide:

1.1 Installation

The easiest way to install is using pip:

```
pip install astroalign
```

This will install the latest stable version on PIPy.

If you want to use the latest development version from github, unpack or clone the repo on your local machine, change the directory to where setup.py is, and install using setuptools:

```
python setup.py install
```

or pip:

```
pip install .
```

1.2 Tutorial

1.2.1 A simple usage example

Note: Check this Jupyter notebook for a more complete example.

Suppose we have two images of about the same portion of the sky, and we would like to transform one of them to fit on top of the other one. Suppose we do not have WCS information, but we are confident that we could do it by eye, by matching some obvious asterisms on the two images.

In this particular use case, astroalign can be of great help to automatize the process.

After we load our images into numpy arrays, we simple choose one to be the source image to be transformed and the other to be the target.

Note: astroalign will also accept as input, data objects with *data* and *mask* properties, like NDData, CCDData and Numpy masked arrays. For more information, see *Dealing with Data Objects with data and mask properties (NDData, CCDData, Numpy masked arrays)*

The usage for this simple most common case would be as follows:

```
>>> import astroalign as aa
>>> registered_image, footprint = aa.register(source, target)
```

`registered_image` is now a transformed (numpy array) image of `source` that will match pixel to pixel to `target`.

`footprint` is a boolean numpy array, *True* for masked pixels with no information.

Warning: Flux may not be conserved after the transformation.

1.2.2 Mask Fill Value

If you need to mask the aligned image with a special value over the region where transformation had no pixel information, you can use the *footprint* mask to do so:

```
>>> registered_image, footprint = aa.register(source, target)
>>> registered_image[footprint] = -99999.99
```

Or you can pass the value to the *fill_value* argument:

```
>>> registered_image, footprint = aa.register(source, target, fill_value=-99999.99)
```

Both will yield the same result.

1.2.3 Finding the transformation

In some cases it may be necessary to inspect first the transformation parameters before applying it, or we may be interested only in a star to star correspondence between the images. For those cases, we can use `find_transform`.

`find_transform` will return a scikit-image SimilarityTransform object that encapsulates the matrix transformation, and the transformation parameters. It will also return a tuple with two lists of star positions of `source` and its corresponding ordered star postions on the `target` image.:

```
>>> transf, (source_list, target_list) = aa.find_transform(source, target)
```

source and target here can be either numpy arrays of the image pixels, or any iterable (x, y) pair, corresponding to a star position.

The transformation parameters can be found in `transf.rotation`, `transf.traslation`, `transf.scale` and the transformation matrix in `transf.params`.

If the transformation is satisfactory, we can apply it to the image with `apply_transform`. Continuing our example:

```
>>> if transf.rotation > MIN_ROT:
...     registered_image = aa.apply_transform(transf, source, target)
```

1.2.4 If you know the star-to-star correspondence

Note: estimate_transform from *scikit-image* is imported into astroalign as a convenience.

If for any reason you know which star corresponds to which other, you can call `estimate_transform`.

Let us suppose we know the correspondence:

- (127.03, 85.98) in source –> (175.13, 111.36) in target
- (23.11, 31.87) in source –> (0.58, 119.04) in target
- (98.84, 142.99) in source –> (181.55, 206.49) in target
- (150.93, 85.02) in source –> (205.60, 91.89) in target
- (137.99, 12.88) in source –> (134.61, 7.94) in target

Then we can estimate the transform:

```
>>> src = np.array([(127.03, 85.98), (23.11, 31.87), (98.84, 142.99),
...                 (150.93, 85.02), (137.99, 12.88)])
>>> dst = np.array([(175.13, 111.36), (0.58, 119.04), (181.55, 206.49),
...                 (205.60, 91.89), (134.61, 7.94)])
>>> tform = aa.estimate_transform('affine', src, dst)
```

And apply it to an image with `apply_transform` or to a set of points with `matrix_transform`.

1.2.5 Applying a transformation to a set of points

Note: matrix_transform from *scikit-image* is imported into astroalign as a convenience.

To apply a known transform to a set of points, we use *matrix_transform*. Following the example in the previous section:

```
>>> dst_calc = aa.matrix_transform(src, tform.params)
```

`dst_calc` should be a 5 by 2 array similar to the `dst` array.

1.2.6 Dealing with Data Objects with data and mask properties (NDData, CCDData, Numpy masked arrays)

If your input data comes in the form of ccdproc's CCDData or astropy's NDData or a numpy masked array, there are a few ways to interact with astroalign.

In general, for objects with *data* and *mask* properties, it is convenient to transform their masks along with the data and to add the footprint onto the mask.

Astroalign provides this functionality with the *propagate_mask* argument to *register* and *apply_transform*.

For example:

```
>>> from astropy.nddata import NDData
>>> nd = NDData([[0, 1], [2, 3]], [[True, False], [False, False]])
```

and we want to apply a clockwise 90 degree rotation:

```
>>> import numpy as np
>>> from skimage.transform import SimilarityTransform
>>> transf = SimilarityTransform(rotation=np.pi/2., translation=(1, 0))
```

Then we can call astroalign as usual, but with the *propagate_mask* set to True:

```
>>> aligned_image, footprint = aa.apply_transform(transf, nd, nd, propagate_mask=True)
```

This will transform *nd.data* and *nd.mask* simultaneously and add the *footprint* mask from the transformation onto *nd.mask*:

```
>>> aligned_image
array([[2., 0.],
   [3., 1.]])
>>> footprint
array([[False,  True],
   [False, False]])
```

Creating a new object of the same input type is now easier:

```
>>> new_nd = NDData(aligned_image, mask=footprint)
```

The same will apply for CCDData objects and Numpy masked arrays.

1.2.7 Dealing with hot pixels

Hot pixels always appear on the same position of the CCD. If your image is dominated by hot pixels, the source detection algorithm may pick those up and output the identity tranformation.

To avoid this, you can use CCDProc's cosmicray_lacosmic to clean the image before trying registration:

```
from ccdproc import cosmicray_lacosmic as lacosmic
clean_image = lacosmic(myimage)
```

See *Module API* for the API specification.

1.3 Module API

ASTROALIGN is a simple package that will try to align two stellar astronomical images, especially when there is no WCS information available.

It does so by finding similar 3-point asterisms (triangles) in both images and deducing the affine transformation between them.

General registration routines try to match feature points, using corner detection routines to make the point correspondence. These generally fail for stellar astronomical images, since stars have very little stable structure and so, in general, indistinguishable from each other.

Asterism matching is more robust, and closer to the human way of matching stellar images.

Astroalign can match images of very different field of view, point-spread functions, seeing and atmospheric conditions.

(c) Martin Beroiz

astroalign.**MAX_CONTROL_POINTS = 50**
The maximum control points (stars) to use to build the invariants.

Default: 50

astroalign.**MIN_MATCHES_FRACTION = 0.8**
The minimum fraction of triangle matches to accept a transformation.

If the minimum fraction yields more than 10 triangles, 10 is used instead.

Default: 0.8

exception astroalign.**MaxIterError**

astroalign.**NUM_NEAREST_NEIGHBORS = 5**
The number of nearest neighbors of a given star (including itself) to construct the triangle invariants.

Default: 5

astroalign.**PIXEL_TOL = 2**
The pixel distance tolerance to assume two invariant points are the same.

Default: 2

astroalign.**apply_transform**(*transform*, *source*, *target*, *fill_value=None*, *propagate_mask=False*)
Applies the transformation `transform` to `source`.

The output image will have the same shape as `target`.

Parameters

- **transform** – A scikit-image `SimilarityTransform` object.
- **source** (*numpy array*) – A 2D numpy array of the source image to be transformed.
- **target** (*numpy array*) – A 2D numpy array of the target image. Only used to set the output image shape.
- **fill_value** (*float*) – A value to fill in the areas of aligned_image where footprint == True.
- **propagate_mask** (*bool*) – Wether to propagate the mask in source.mask onto footprint.

Returns A tuple (aligned_image, footprint). aligned_image is a numpy 2D array of the transformed source footprint is a mask 2D array with True on the regions with no pixel information.

astroalign.**find_transform**(*source*, *target*)
Estimate the transform between `source` and `target`.

Return a SimilarityTransform object `T` that maps pixel x, y indices from the source image s = (x, y) into the target (destination) image t = (x, y). T contains parameters of the tranformation: `T.rotation`, `T.translation`, `T.scale`, `T.params`.

Parameters

- **source** (*array-like*) – Either a numpy array of the source image to be transformed or an interable of (x, y) coordinates of the target control points.
- **target** (*array-like*) – Either a numpy array of the target (destination) image or an interable of (x, y) coordinates of the target control points.

Returns

The transformation object and a tuple of corresponding star positions in source and target.:

```
T, (source_pos_array, target_pos_array)
```

Raises

- `TypeError` – If input type of `source` or `target` is not supported.
- `Exception` – If it cannot find more than 3 stars on any input.

`astroalign.`**`register`**(*source*, *target*, *fill_value=None*, *propagate_mask=False*)

Transform `source` to coincide pixel to pixel with `target`.

Parameters

- **`source`** (*`numpy array`*) – A 2D numpy array of the source image to be transformed.
- **`target`** (*`numpy array`*) – A 2D numpy array of the target image. Only used to set the output image shape.
- **`fill_value`** (*`float`*) – A value to fill in the areas of aligned_image where footprint == True.
- **`propagate_mask`** (*`bool`*) – Wether to propagate the mask in source.mask onto footprint.

Returns A tuple (aligned_image, footprint). aligned_image is a numpy 2D array of the transformed source footprint is a mask 2D array with True on the regions with no pixel information.

CHAPTER 2

Indices and tables

- genindex
- search

Python Module Index

a

Index