astroalign Documentation

Release 2.4

Martin Beroiz

Mar 25, 2021

Contents

1 Citation **3**
1.1 Guide: 4

2 Indices and tables **11**

Python Module Index **13**

Index **15**

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://quatrope.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

Citation

If you use astroalign in a scientific publication, we would appreciate citations to the following paper:

> Astroalign: A Python module for astronomical image registration. Beroiz, M., Cabral, J. B., & Sanchez, B. Astronomy and Computing, Volume 32, July 2020, 100384.

Bibtex entry:

```
@article{BEROIZ2020100384,
title = "Astroalign: A Python module for astronomical image registration",
journal = "Astronomy and Computing",
volume = "32",
pages = "100384",
year = "2020",
issn = "2213-1337",
doi = "https://doi.org/10.1016/j.ascom.2020.100384",
url = "http://www.sciencedirect.com/science/article/pii/S221313372030038X",
author = "M. Beroiz and J.B. Cabral and B. Sanchez",
keywords = "Astronomy, Image registration, Python package",
abstract = "We present an algorithm implemented in the Astroalign Python module for␣
↪image registration     in astronomy. Our module does not rely on WCS information␣
↪and instead matches three-point asterisms (    triangles) on the images to find the␣
↪most accurate linear transformation between them. It is especially     useful in␣
↪the context of aligning images prior to stacking or performing difference image␣
↪analysis.     Astroalign can match images of different point-spread functions,␣
↪seeing, and atmospheric conditions."
}
```

Full Publication: https://www.sciencedirect.com/science/article/pii/S221313372030038X

Or cite the project itself from ASCL:

> Beroiz, M. I. (2019). Astroalign: Asterism-matching alignment of astronomical images. Astrophysics Source Code Library.

Bibtex:

```
@article{beroiz2019astroalign,
    title={Astroalign: Asterism-matching alignment of astronomical images},
    author={Beroiz, Martin I},
    journal={Astrophysics Source Code Library},
    year={2019}
}
```

1.1 Guide:

1.1.1 Installation

The easiest way to install is using pip:

```
pip install astroalign
```

This will install the latest stable version on PyPI.

If you want to use the latest development code from GitHub, clone the repo on your local machine and install:

```
git clone https://github.com/quatrope/astroalign
cd astroalign
pip install -e .
```

1.1.2 Tutorial

A simple usage 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. 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.

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.

Note:

- If instead of images, you have lists of bright, reference star positions on each image, see *Finding the transformation*.
- `astroalign.register` will also accept as input, data objects with `data` and `mask` properties, like `NDData`, `CCDData` and `Numpy` masked arrays. See *Objects with data and mask property*.

- Check this Jupyter notebook for a more complete example.

Warning: Flux may not be conserved after the transformation.

Note: If your image requires special care see *Examples*.

Images with RGB channels

Astroalign can work with color images provided the channel index be the last axis in the array. Adding the channel dimension in the last axis of the array is the default behavior for pillow and scikit-image. The transformation is found on the `mean` average of all the channels. PNG images with RGBA channels work similarly.

Example:

```
from PIL import Image
import astroalign as aa
source = Image.open("source.jpg")
target = Image.open("target.jpg")
registered, footprint = aa.register(source, target)
# Convert back to pillow image if necessary:
registered = Image.fromarray(registered.astype("unit8"))
```

Pillow may require array to be unsigned 8-bit integer format.

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.

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`:

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

The inputs `source` and `target` can be either numpy arrays of the image pixels, **or any iterable of (x, y) pairs**, corresponding to star positions.

Having an iterable of (x, y) pairs is especially useful in situations where source detection requires special care. In situations like that, source detection can be done separately and the resulting catalogs fed to `find_transform`.

`find_transform` returns 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.

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)
```

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`.

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.

Objects with `data` and `mask` property

If your image is stored in objects with `data` and `mask` properties, such as ccdproc's CCDData or astropy's NDData or a NumPy masked array you can use them as input for `register`, `find_transform` and `apply_transform`.

In general in these cases 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.

See *Module API* for the API specification.

1.1.3 Examples

Very few stars on the field

Note: The minimum number of stars necessary to find a transformation is 3

If your field has few stars on the field, of about 3 to 6, you may want to restrict astroalign to only pick that number of stars, to prevent catching noisy structures as sources.

Use `max_control_points` keyword argument to do so:

```
>>> import astroalign as aa
>>> registered_image, footprint = aa.register(source, target, max_control_points=3)
```

This keyword will also work in `find_transform`.

Faint stars

If your stars are faint, they may not be bright enough to pass the 5σ threshold. If you need to lower the detection σ used in the source detection process, adjust the `detection_sigma` keyword argument:

```
>>> import astroalign as aa
>>> registered_image, footprint = aa.register(source, target, detection_sigma=2)
```

This keyword will also work in `find_transform`.

Avoiding hot pixels and other CCD artifacts

If your CCD is dominated by persistent defects like hot or dead pixels, they may be taken as legitimate sources and output the identity transformation.

We suggest cleaning the image first using CCDProc's cosmicray_lacosmic to clean the image before trying registration:

```
>>> from ccdproc import cosmicray_lacosmic as lacosmic
>>> clean_source, mask = lacosmic(myimage)
>>> registered_image, footprint = aa.register(clean_source, clean_target, min_area=9)
```

Another quick fix can be increasing the expected connected pixels in order to be considered a source. Increment `min_area` from default value of 5:

```
>>> import astroalign as aa
>>> registered_image, footprint = aa.register(source, target, min_area=9)
```

1.1.4 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.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, CCData or NDData array of the source image to be transformed.
- **target** (*numpy array*) – A 2D NumPy, CCData or NDData 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*, *max_control_points=50*, *detection_sigma=5*, *min_area=5*)
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, CCData or NDData 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, CCData or NDData array of the target (destination) image or an interable of (x, y) coordinates of the target control points.
- **max_control_points** – The maximum number of control point-sources to find the transformation.
- **detection_sigma** – Factor of background std-dev above which is considered a detection. This value is ignored if input are not images.
- **min_area** – Minimum number of connected pixels to be considered a source. This value is ignored if input are not images.

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.
- `ValueError` – If it cannot find more than 3 stars on any input.

`astroalign.`**`register`**(*source*, *target*, *fill_value=None*, *propagate_mask=False*, *max_control_points=50*, *detection_sigma=5*, *min_area=5*)

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

Parameters

- **`source`** (*`numpy array`*) – A 2D NumPy, CCData or NDData array of the source image to be transformed.
- **`target`** (*`numpy array`*) – A 2D NumPy, CCData or NDData array of the target image. Used to set the output image shape as well.
- **`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.
- **`max_control_points`** – The maximum number of control point-sources to find the transformation.
- **`detection_sigma`** – Factor of background std-dev above which is considered a detection.
- **`min_area`** – Minimum number of connected pixels to be considered a source.

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

astroalign, 8

Index

A

apply_transform() (*in module astroalign*), 9
astroalign (*module*), 8

F

find_transform() (*in module astroalign*), 9

M

MaxIterError, 8
MIN_MATCHES_FRACTION (*in module astroalign*), 8

N

NUM_NEAREST_NEIGHBORS (*in module astroalign*), 9

P

PIXEL_TOL (*in module astroalign*), 9

R

register() (*in module astroalign*), 10