astroalign Documentation

Release 1.0.3

Martin Beroiz

May 18, 2018

Contents

ASTROALIGN is a simple package 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.

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.

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

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 -e .
```

1.2 Tutorial

1.2.1 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 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 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 = aa.register(source, target)
```

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

If `source` is a masked array, `registered_image` will have a mask transformed like `source` with pixels outside the boundary masked with True (read more in *Working with masks*).

1.2.2 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)
```

1.2.3 If you know the star-to-star correspondence

As a convenience, estimate_transform from *scikit-image* is imported to astroalign.

If for any reason you know which star corresponds to what 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`.

1.2.4 Applying a transformation to a set of points

As a convenience, matrix_transform from *scikit-image* is imported to astroalign.

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.

See *Module Methods* for more information.

1.3 Module Methods

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.

3. 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

`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.`**`align_image`**(*ref_image*, *img2transf*, *n_ref_src=50*, *n_img_src=70*, *px_tol=2.0*)
Deprecated: Alias for `register` for backwards compatibility.

`astroalign.`**`apply_transform`**(*transform*, *source*, *target*)
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.

Returns A numpy 2D array of the transformed source. If source is a masked array the returned image will also be a masked array with outside pixels set to True.

`astroalign.`**`find_affine_transform`**(*test_srcs*, *ref_srcs*, *max_pix_tol=2.0*, *min_matches_fraction=0.8*, *invariant_map=None*)

Deprecated: Alias for `find_transform` for backwards compatibility.

`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*)

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.

Returns A numpy 2D array of the transformed source. If source is a masked array the returned image will also be a masked array with outside pixels set to True.

1.4 Working with masks

Sometimes, CCD defects can confuse the alignment algorithm because of misplaced star centroids, or fake point-like sources on the image. In those cases, you may want to mask those artifacts so they are not counted as control points.

The way to do so is to wrap your image in a numpy masked array:

```
>>> myarray = np.ma.array(myarray, mask=badpixelmask)
```

and mask bad pixels with True, following the numpy masked array convention.

You can now call astroalign methods in the usual way:

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

The type of the returned `registered_image` wil be the same type as the input image, regardless of the type of `target`.

That is, if the source image is a masked array, the output will also be a masked array, with the masked transformed in the same way as the source image and filled with True for pixels outside the boundary.

CHAPTER 2

Indices and tables

- genindex
- search

Python Module Index

Index