7-2020

Astroalign: A Python Module for Astronomical Image Registration

Martin Beroiz
The University of Texas Rio Grande Valley

Juan B. Cabral

Bruno Sanchez

Recommended Citation

Astroalign: A Python Module for Astronomical Image Registration

Martin Beroiz[a], Juan B. Cabral[b,c], Bruno Sanchez[d]

[a] University of Texas Rio Grande Valley (UTRGV), Texas, USA
[b] Centro Internacional Franco Argentino de Ciencias de la Información y de Sistemas (CIFASIS, CONICET–UNR), Ocampo y Esmeralda, S2000EZP, Rosario, Argentina.
[c] Instituto de Astronomía Teórica y Experimental - Observatorio Astronómico Córdoba (IATE, UNC–CONICET), Córdoba, Argentina.
[d] Department of Physics, Duke University, 120 Science Drive, Durham, NC, 27708, USA

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.

Keywords: Astronomy, Image Registration, Python Package

1. Introduction

Image registration is the process of transforming images that use different coordinate systems, so that after the transformation they share a common frame.

Image registration is widely used in the medical fields (Fischer & Modersitzki 2008), including nuclear imaging (Hutton et al. 2002) and radiology (Hill et al. 2001). It is also widely used in Computer Vision (Zitov & Flusser 2003) for object recognition (Zhengwei Yang & Cohen 1999), image stitching (Brown & Lowe 2007) and visual mapping (Konolige & Agrawal 2008). Image registration is used in virtually any field that needs to reconcile images taken from different points of view. In astronomy, image registration is used for tasks such as stacking images, difference image analysis, and creating mosaics.

In the particular field of astronomy, all images have a natural common reference frame, given by the spherical coordinates of position on the celestial sphere. Registration between two images usually amounts to finding the mathematical transformation between each coordinate frame.

The astrometric information of an astronomical image is typically stored in the header of a FITS file using a World Coordinate System (WCS) (Greisen & Calabretta 2002). Solving for the WCS is only one step away from registering any two images, since the last remaining step is to remap one or both images into the desired destination reference frame. The Python packages 'reproject' (Robitaille 2018) and 'MontagePy' (Jacob et al. 2009; Berriman & Good 2017) work in this way.

When no WCS information is available, and the pointing of the image is uncertain or unknown, a program like astrometry.net (Lang et al. 2010) will identify the portion of the sky and generate appropriate WCS information.

Generating WCS from a completely unknown image is a daunting challenge, one that astrometry.net handles handsomely well. But despite its undisputed usefulness, it may be an overkill for some cases. Roughly speaking, the task is to spot a particular section of the sky on 40K square degrees for images that typically span less than 1 square degree. This exhaustive search can be computationally expensive and it requires the download of several hundreds of megabytes of index files[1].

Unlike Astrometry.net, Astroalign[2] does not try to solve for WCS coordinates. Astroalign is a fast and lightweight Python module that registers astronomical images bringing the images to a common frame via the best linear transformation between them. For this reason it is particularly suited for situations where we want to correct for drift or displacements over an observation night, or if we want to reconcile two images taken by different observatories, but where no astrometry is needed.

Astroalign assumes little about the origin and format of the image, making it suitable for astrophotographers and amateur astronomers, who do not usually deal with FITS-formatted images. Regardless, Astroalign is compatible with modern astronomy data objects such as astropy's NDData, ccdproc's CCD-Data, and Numpy arrays (with or without masks), so that astronomers can easily integrate with other Python packages. Astroalign can match images of different point-spread functions, seeing and atmospheric conditions. Astroalign is tested under continuous integration and has documentation available online[3]. It may not work, or work with special care, on images of extended objects with few point-like sources or in crowded fields.

2. The Algorithm

In the field of Computer Vision (CV), generic feature-based registration routines like SIFT (Lowe 2004), SURF (Bay et al.

---

[1] astrometry.net has an online tool that does not require downloading files.
[2] http://github.com/toros-astro/astroalign
[3] https://astroalign.readthedocs.io

2008), or ORB (Rublee et al. 2011), try to identify and match "interesting points" using corner detection routines to estimate a point correspondence between the images. These generally fail for stellar astronomical images, since stars are effectively point sources that have very little distinct structure and hence, indistinguishable from each other. Asterism matching is more robust and closer to the way humans match images.

The idea of using star patterns to match stellar images goes back to an early publication by Groth (1986). The author deals with the very similar problem of finding the correspondence between two lists of coordinates from star catalogs. His method already contains many of the core ideas used in our method, but it is prone to generate a large number of outliers, difficult to remove. It also scales poorly with the fourth power of the number of reference points.

Pál & Bakos (2006) and Tabur (2007) extend the ideas in Groth (1986) to large catalogs of sources and include possible non-linear distortions of the field. Astrometry.net (Lang et al. 2010) also improves on many of the shortcomings from Groth (1986). Astrometry.net uses quadrilaterals instead of triangles and a Bayesian decision criteria to pick the correct image transformation. We discuss comparison with previous methods in Section 3.

Similar to previous work, the core idea of Astroalign's algorithm consists of characterizing asterisms by using "geometric hashes" that are invariant to translation, rotation, and scaling. Similar asterisms will have similar invariant tuples in both images so a correspondence between them can be made.

As an example, the lengths of the sides of a triangle are invariant to translation and rotation. They remain the same whatever position or orientation the triangle might have. Additionally, any function of the *ratio* of the sides will be invariant to scaling. We explain the invariant tuple that we use in Section 2.2.

The idea for the Astroalign algorithm can be summarized in a few steps that we enumerate below.

1. Do the following for both images

    Make a catalog of a few brightest sources and push them to a 2D tree (a *k-d tree* data structure for k=2) to quickly query for close neighbors.

    For each star, select its four nearest neighbors.

    Create all the $\binom{5}{3} = 10$ possible triangles from that set of stars.

    For each triangle in that set, calculate a tuple of invariants that fully characterize the triangle and push the invariant tuple into another k-d tree.

    Remove duplicate invariant tuples that come from considering repeated triangles.

2. Identify and match all possible invariants from both lists by querying the nearest neighbor in the k-d trees, thus creating a correspondence between triangles on the images.

3. For each triangle match, make a correspondence between the vertices by considering the lengths of the sides to which they belong. This way, we establish a point-to-point correspondence for three points.

4. Propose a transformation derived from one point-to-point triangle correspondence. If the proposed transformation fits over 80% (or 10, whichever is smaller) of the rest of the points, accept the transformation and return the transformed image.

5. If the transformation cannot fit other triangles, or falls below the acceptance threshold, proceed with the next invariant match until one acceptable transformation is found.

6. If the number of unsuccessful tries exceeds a maximum limit, an exception is raised.

In the following sections, we explain in more detail the steps above.

2.1. The Image Transformation

At the crux of the registration problem is the determination of the geometric pixel transformation that carries one image into another. In Computer Vision these transformations are called *homographies*. A homography is a linear transformation between images on projective spaces. The most general homography can relate images of a planar surface with different perspective points of view. In our particular case where, for all practical purposes, the sources are located at infinity, we do not use perspective deformations parameters, but we still adopt the language of CV, due to the widespread use of this formalism in the available software packages. The most notable difference with regular 2D matrix operators is the extension of (x, y) coordinates to *homogenous* coordinates $(x, y, 1)$ to include rotations and translations in the same operation.

The parameters for the most general transformation that we consider consists of a clockwise rotation angle α, a uniform scaling parameter λ, and a 2D translation vector (t_x, t_y). They are summarized in the following coordinate transformation matrix:

$$\begin{pmatrix} \lambda\cos\alpha & \lambda\sin\alpha & \lambda t_x \\ -\lambda\sin\alpha & \lambda\cos\alpha & \lambda t_y \\ 0 & 0 & 1 \end{pmatrix} \equiv \begin{pmatrix} a_0 & b_0 & c_0 \\ -b_0 & a_0 & c_1 \\ 0 & 0 & 1 \end{pmatrix} \quad (1)$$

To linearize our problem, instead of considering $\{\lambda, \alpha, t_x, t_y\}$ as free parameters, we consider $\{a_0, b_0, c_0, c_1\}$ instead, as if they were independent.

Each correspondence of points $(x_1, y_1) \rightarrow (x_2, y_2)$ in both images is related by equation (2),

$$\begin{pmatrix} a_0 & b_0 & c_0 \\ -b_0 & a_0 & c_1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix} = \begin{pmatrix} x_2 \\ y_2 \\ 1 \end{pmatrix}. \quad (2)$$

With three of these point-correspondences, we can solve for the parameters $\{a_0, b_0, c_0, c_1\}$ and $\{\lambda, \alpha, t_x, t_y\}$.

2.2. Asterisms and Invariant Features

The search for the point correspondences is not done on the set of points itself, but rather uses triangles, or point triplets as
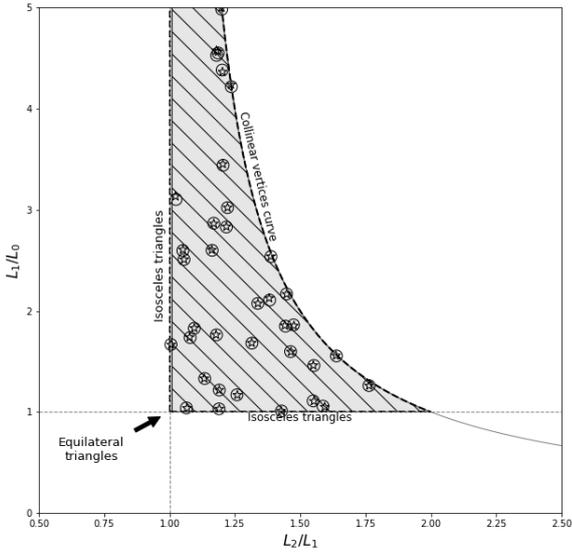
2

Figure 1: **Region for the Astroalign invariant mapping**. In the region, we graph the invariant tuples from the example in Section 4.1. Star-shaped markers represent invariant tuples from the source image and circles represent invariant tuples from the target image.

proxies. Unlike single points, triangles can be uniquely characterized.

For a triangle, full knowledge of all three side lengths $\{L_i\}$ is enough to fully characterize it, irrespective of orientation or position. If we want to characterize it up to a global scaling, then knowing 2 inner angles is sufficient. Equivalently, knowing two independent ratios of the side lengths is also sufficient. In fact, any function of two independent length ratios will suffice to fully characterize the triangle.

For Astroalign we chose the mapping defined by:

$$\mathscr{M}(\{L_i\}) = \left(\frac{L_2}{L_1}, \frac{L_1}{L_0} \right) \tag{3}$$

$$\text{where } L_2 \geq L_1 \geq L_0. \tag{4}$$

$\mathscr{M}$ maps into an invariant tuple that fully describes a triangle up to translation, rotation, scaling, and coordinate flipping. The invariant map $\mathscr{M}$ maps the positive octant of $\mathbb{R}^3$ of all possible side lengths of a triangle onto a region of the positive quadrant of $\mathbb{R}^2$ in the invariant-feature space.

To find out what this region is, we note that, by definition, $L_2 \geq L_1 \geq L_0$,

$$x = \frac{L_2}{L_1} \geq 1$$

$$y = \frac{L_1}{L_0} \geq 1$$

Furthermore, from the triangle inequality:

$$L_2 \leq L_1 + L_0 \implies x \leq 1 + \frac{1}{y}$$

$$y \leq \frac{1}{x - 1}$$

The curve $y = (x - 1)^{-1}$ corresponds to collinear vertices. Equilateral triangles will map to the point $(1, 1)$ and an isosceles triangle will map either to the $x = 1$ or $y = 1$ line depending on whether the unequal side is the largest or smallest side. Very peaky triangles will tend to accumulate between the collinear vertices curve and the $x = 1$ line for large values of y. These observations can be summarized in Figure 1, where we also show some example invariant tuples from Section 4.1.

2.3. Selection of Sources

The selection of stars from Step 1 is an important part of the algorithm. To identify the positions and fluxes of sources, Astroalign relies on the external package *Source Extraction and Photometry (SEP)* (Barbary 2016), a project that extracts the core functionality of the program SEextractor (Bertin & Arnouts 1996) into a C library and adds a Python API on top.

Mimicking the way humans match images, we pick for the transform estimation, those point sources that have higher chances to be persistent across the images (i.e. the brightest). Selecting too few sources would make the matching algorithm very sensitive to outliers and missing stars and, thus, unstable. Selecting too many sources will make computations unnecessarily complicated and prone to fail because of spurious mismatches. We found that capping the number of sources at the brightest 50 on both images is a good empirical compromise between robustness and efficiency.

However, the cap at 50 sources does not restrict its use for fields with fewer sources in it. Astroalign is designed to work with as few as three sources—the minimum required to derive a transformation.

In the cases where the source selection requires special care, like in the case of faint sources or very crowded fields, the source selection can be done separately by the user and the resulting catalogs fed to Astroalign in place of the array images.

2.4. Triangulation

Once the catalog of reference sources is made on each image, we push all entries to a k-d tree. A k-d tree is a data structure optimized to efficiently query for nearest neighbors in $O(\log n)$ operations[4].

As mentioned before, we use triangles as a proxy for point-to-point correspondence (see Section 2.2). Given N stars, there are $N(N - 1)(N - 2)/6$ possible triangles in a fully-connected triangulation. However, not all of these are needed in practice.

Instead of working with a fully-connected triangulation we adopt a *nearest neighbor triangulation*. With each star and its four nearest neighbors, we calculate the invariant tuples for the

---

[4]On average.

3

10 triangles that can be made from those five sources. Each invariant tuple is then pushed to another k-d tree.

At the end of this step we have two k-d trees from each image. The triangles that correspond to the same triplet of stars in the two images, will lie very close on the invariant space. Consequently, we can query the triangles on one of the k-d trees for its closest partner on the other image. This is also done efficiently in $O(n \log n)$ operations.

Every triangle that has a partner in the other image less than 0.1 units away (about 5% error) is considered a matched triangle.

One triangle match is enough to completely characterize the four parameters of the transformation (1)–(2), and this proposed transformation is then tested using the rest of the triangle correspondences.

2.5. Testing the Transformation

To estimate how well a transformation fits, we make use of the formalism commonly used in *stereovision* (Zhang 1998) to estimate the error of a reprojection.

The reprojection error (RE) is the geometric distance between a projected point and a measured point. Given two points in homogenous coordinates x and x', and a transformation H, the reprojection error is defined as the the distance d from a point x'_i to its corresponding epipolar line $l'_i = Hx_i$.

$$d(x'_i, l'_i) = \frac{x'^T_i l'_i}{\sqrt{l'^2_1 + l'^2_2}} \tag{5}$$

For a set of correspondence points $\{(x_i, x'_i)\}$ we define a global error (*linear criterion*) as:

$$RE = \sum_i \left(l'^2_1 + l'^2_2 \right) d^2(x'_i, l'_i) \tag{6}$$

The full formalism of epipolar geometry is quite extensive (see Zhang (1998) for a detailed review of this topic). We only use it to have a convenient way to estimate the error for a given transformation and set of correspondences.

The actual implementation is taken from the *scikit-image* Python package, that gives an unbiased and fast approximation to the linear criterion called the "Sampson distance" (Luong et al. 1993; Fathy et al. 2011). For each triangle correspondence we calculate the residuals of the transformation with this distance, using the three correspondence points. If the Sampson distance is smaller than 3, the correspondence is accepted, and rejected otherwise. A transformation that matches 80% of the triangle matches or 10 (whichever is lower) is accepted and returned.

This selection process is carried out within a RANSAC process. The Random Sample Consensus or RANSAC (Fischler & Bolles 1987), is an iterative method to estimate fit parameters of a model when the data to fit contains a significant fraction of outliers. It was first introduced as a solution to the "Location Determination Problem" to determine the 3D location from where an image was taken. RANSAC is also distributed as part of the popular CV library *OpenCV* (Bradski 2000).

As is common practice in CV, Astroalign uses a modified implementation of RANSAC[5] to estimate the parameters of the homography, while simultaneously ignoring outliers caused by orphan asterisms.

2.6. Image Reprojection

Once a transformation is accepted, the RANSAC algorithm re-calculates the transformation using all of the in-lier triangle vertices identified in the previous step.

Astroalign then uses this re-calculated transformation to perform a bi-cubic polynomial interpolation of the source image using the `warp` function from Scikit-Image's `transformation` module. A footprint mask of the transformation is also returned along with the transformed image granting the user control on the un-transformed, out-of-boundary pixels.

We tested flux conservation for this type of reprojection, using a simple aperture photometry calculation before and after the image had been transformed. We found that the ratio of flux per unit area variability is consistent with a Gaussian distribution centered at one, with a variance of $\sim 5 \times 10^{-3}$. This means that rigid transformations can introduce errors at the centi-magnitude level, or less than half a percent in flux in the worst-case scenario. We consider that this quantifiable error is acceptable for the purpose of Astroalign.

3. Comparison with Other Methods

Astroalign follows the main ideas developed in Lang et al. (2010) and incorporates several of the existing concepts found in the field of CV. Astroalign can also be compared to the ideas introduced in other implementations like Pál & Bakos (2006) and Tabur (2007).

One main difference between our algorithm and Astrometry.net is that Astroalign is not doing a 'blind astrometric calibration' nor returns WCS information. Unlike Groth (1986), Pál & Bakos (2006) and Tabur (2007), our main goal is not to cross-match lists of stars, but to find a geometric transformation between a pair of images. In fact, by design, we restrict the number of sources in our list of guiding stars to a minimum. We also only consider rigid (linear) transformations between images and we do not consider non-linear deformations of wide fields like in Pál & Bakos (2006).

Another difference with previous methods is our triangulation method. Astrometry.net deals with catalogs of the full sky and uses quadrilaterals at different scales instead of triangles. The quadrilaterals are chosen to fit in a subdivision of the total sky into Healpix grid cells. Tabur (2007) prioritizes its search to very acute-angle triangles that are easier to identify in the invariant space. Pál & Bakos (2006) uses a Delaunay triangulation but adding more redundancy in close-by subgroups of stars to make the triangulation robust to missing or new sources on the field. Astroalign follows a similar approach than that of Pál & Bakos (2006) by calculating an interlaced full triangulation in small subgroups instead of storing the full triangulation

---

[5]Initially developed by Andrew Straw.

of the field of view. Our choice to store the triangles in a k-d tree data structure allows us to retrieve similar matches in a fast $O(n \log n)$ number of operations.

Our system to test a transformation is similar to that of Lang et al. (2010) and the *Optimistic Pattern Matching* described in Tabur (2007), in that we test each hypothetized transformation immediately in the belief that the transformation will most likely be correct. If the transformation accurately describes other triangles on the mesh, the transformation is immediately accepted and returned. But unlike Astrometry.net, which uses a Bayesian decision process to verify the prediction power of the position of other stars in the field, we use the reprojection error described in Section 2.5. In our case, the quick estimation and early exit are actually a consequence of using a RANSAC algorithm implementation to search for the transformation. RANSAC also deals with the problem of missing stars or transient objects in the field.

This short comparison does not pretend to be an exhaustive review of the literature. For a review on earlier work, please refer to Murtagh (1992). These differences are summarized in Appendix A.

4. Technical details about the *Astroalign* package

4.1. Public API and application example

The Astroalign Python package splits the registration functionality into a few main functions. The most important ones are:

`find_transform(...)` Estimate the transform between source and target images and also return a tuple with two ordered lists of star correspondences.

`apply_transform(...)` Apply a given transformation (normally found with the `find_transform` function) to some source image.

`register(...)` Find and apply the transformation to *source* to make it coincide pixel-to-pixel with *target*. This is a convenience wrapper around the two previous functions. The main difference is that `apply_transform()` accepts a transformation to apply, but `register()` always uses the best one found by `find_transform()`. This make this function easier to use, but less flexible than the other two combined.

As an application example showing the functionality of the implementation, let us consider two sample simulated images as shown in Figure 2.

The source image on the left panel is 200×200 pixels, with 10 stars of Gaussian profile uniformly distributed across the image. The target image on the right panel simulates an image of the same region of the sky, but rotated 30° clockwise with respect to the source image, 300×300 pixels (dilation factor of 1.5 with respect to source), and containing the same 10 stars

Table 1: Image simulation parameters

	Source	Target
Number of point sources	10	10
FWHM [pixels]	4.71	3.53
Background level [counts]	0.5	0.5
Background std-dev [counts]	0.71	0.71
Flux of stars range [counts]	200–500	200–500

of Gaussian profile but with a different full width at half maximum (FWHM). Notice that the target image has better resolution (and seeing) than the source image. The parameters of the simulation are summarized in Table 1.

On the third panel of Figure 2, we can see the registered image after `register()` is applied to the source. The registered image is now 300×300 pixels in shape but otherwise aligned with target.

The color circles are some of the stars detected by the function `find_transform()`, where the same color identifies the same star on each image.

The invariant tuples internally calculated in this example for both images are graphed in Figure 1. These invariants are not accessible through the public API but are shown to illustrate the method.

One thing to note is that some invariants contain collinear vertices and the distribution over the region is fairly sparse. Sparseness is a desired quality that helps in the identification phase.

4.2. Benchmarks

To evaluate the performance of Astroalign, we create a collection of random *source* and *target* images on which we vary three different parameters:

1. **Size:** The image size in pixels. We tested on square images with sizes 256×256, 512×512, 768×768, and 1024×1024 pixels.
2. **Stars:** The total number of stars in the image. We tested fields with 300, 500, 1,000, and 10,000 stars.
3. **Background Noise:** We vary the λ parameter for the simulated Poisson noise of the CCD. We create images with λ equal to 100, 500, 1,000, and 5,000 counts.

With these parameters, we simulated 10 pairs of *source* and *target* images for each one of the 64 possible combinations, giving a total of 640 test cases. The benchmark was then executed on a computer with the following specifications:

CPU – 2.4GHz Intel Core i7-4700HQ (Quad-core, 6MB cache, up to 3.4GHz with Turbo Boost)

RAM – 16GB DDR3L (2x 8GB, 1,600MHz)

OS – Ubuntu Linux 18.04 64bits.

Software – *Python* 3.7.3, *NumPy* 1.16.4, *SciPy* 1.3.0, *Scikit-Image* 0.15.0 and *Sep* 1.0.3
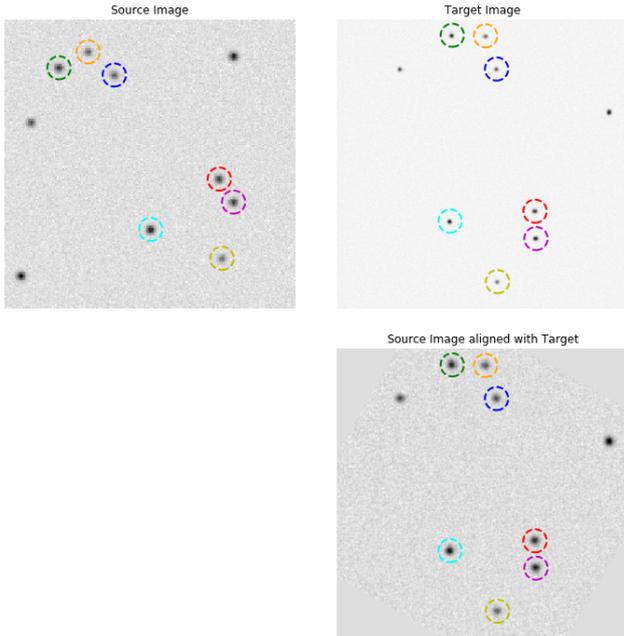
5

Figure 2: **Simulated source and target images**. On the left panel is the source image with 10 simulated stars. The right upper panel is the target image with greater resolution, rotated 30° clockwise with respect to the source. On the bottom panel is the source image aligned to target. The star correspondence is marked on a few stars with color-coded circles across all figures and was done using `find_transform()`.

The results show that the critical factor in determining the execution speed of a project is dependent on the size of the image to be registered (see Figure 3).

To achieve more accurate results in the special case of time vs. size graph, we performed a second benchmark with $2,560$ pairs of random square images of sizes between 256×256 and 1024×1024. We kept the amount of stars and background noise fixed at $10,000$ sources and $\lambda = 1,000$ counts respectively. We can see in Figure 4 that a linear regression fit adjusts well the data points. The regression has a mean squared error *MRE* ~ 1% and a coefficient of determination $R^2 \sim 0.861$. The result of this second experiment seems to indicate that the algorithm has a linear growth in execution time as the size of the image increases.

The project running time is thus bounded and predictable, since it is usually the case that images in a processing pipeline are equal in size.

4.3. Quality assurance

Software quality assurance deals with the measurement of qualitative and quantitative metrics. The most common techniques to extract this information are *unit-testing* and *code-coverage*.

The objective of *unit-testing* is to isolate and show that each part of the program is correct (Jazayeri 2007). *Code-coverage* is a measure of how much code is executed by the unit tests, expressed as a percentage (Miller & Maloney 1963). Having an extensive code-coverage prevents restricting tests to only a
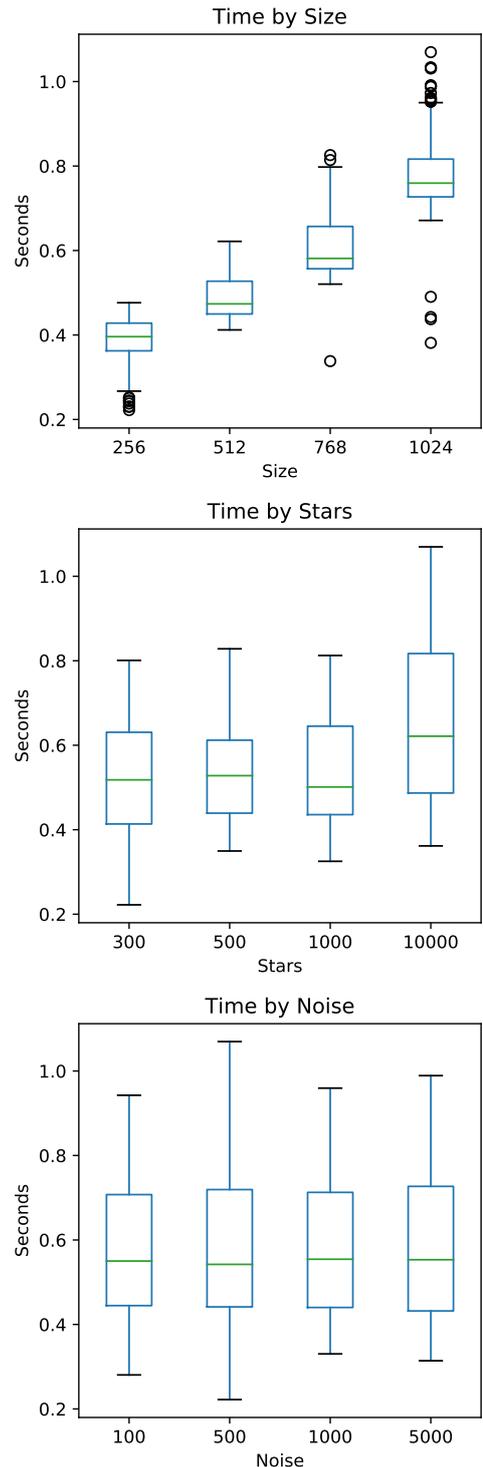

Figure 3: Results of the benchmark on 640 test cases varying the values of size, stars and background noise. In all cases, the vertical axis represents the execution time in seconds, while the horizontal axis represents the different parameter values. We can see that the algorithm increases execution time as the size of the images increases, while in all other cases the times remain relatively unchanged. This entire benchmark dataset can be found at: `https://github.com/toros-astro/astroalign/blob/master/benchmarks/time_benchmarks/20191023.csv`
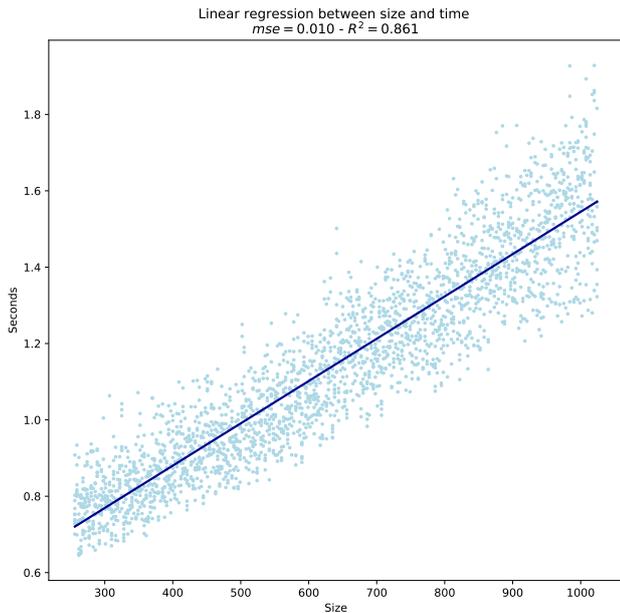
Figure 4: Measured registration times for $2,500$ pairs of random square images of sizes between 256×256 and 1024×1024 pixels. The images keep the amount of stars and background noise fixed at $10,000$ sources and $\lambda = 1,000$ counts respectively. The horizontal axis represents the size of the image in pixels, and the vertical axis represents the registration time in seconds. Each dot is a measurements, and the blue line is the ordinary least squares regression line. This entire benchmark dataset can be found at: `https://github.com/toros-astro/astroalign/blob/master/benchmarks/time_regressions/20200220.csv`

subset of the entire project. In the Astroalign project we provide a collection of nine unit-tests that complete 97% of code-coverage using Python versions 2.7, 3.6 and 3.7 [6].

We are interested in the maintainability of the project and, for this reason, we endorse the PEP 8 – Style Guide for Python Code (Van Rossum et al. 2001), using the *flake8* [7] tool, which automatically checks for any deviation in style making code easier to understand (Latte et al. 2019).

Finally, the entire source code is MIT-licensed (Initiative et al. 2006), and available in a public repository[8]. All versions committed to this code are automatically tested with a continuous-integration service [9], and documentation is automatically built from the repository and made public in the *read-the-docs* service[10] (Holscher 2016).

With these techniques and tools we try to provide high quality for the Astroalign project and its development process.

4.4. *Integration with the Python scientific–stack*

Python is a very accessible language for casual programmers, including astronomers, and has become the tool of choice for almost every new astronomical project (Greenfield 2011).

---

[6]These metrics are calculated as of June 4th, 2019.

[7]`http://flake8.pycqa.org`

[8]`https://github.com/toros-astro/astroalign`

[9]`https://travis-ci.org/toros-astro/astroalign`

[10]`https://astroalign.readthedocs.io/en/latest`

When the *astropy* (Robitaille et al. 2013) library was released, the entire astronomical community was able to build an entire astronomy-related ecosystem upon this same foundation (Price-Whelan et al. 2018). Therefore it is natural for Astroalign to also try to take advantage of this library. The most relevant of these projects are available as astropy-affiliated projects[11].

Astroalign is built on top of the Python scientific stack *Numpy* (Van Der Walt et al. 2011) for modeling the image as an efficient multidimensional array; *Scipy* (Jones et al. 2014) for an implementation of k-d trees; *Scikit-Image* (Van der Walt et al. 2014) for the basic image processing; and *Source Extraction and Photometry (SEP)* (Barbary 2016; Bertin & Arnouts 1996) for segmentation and analysis of astronomical images from a photometric point of view.

Finally, Astroalign is available for installation on the *Python-Package-Index* (PyPI)[12] and can be installed using the *pip* program with the command `pip install astroalign`. This makes our project easy to integrate with any other tool in the Python ecosystem (Valiev et al. 2018).

Astroalign is registered with *The Astrophysics Source Code Library (ASCL)* (Allen & Schmidt 2014) and has been accepted as an Astropy affiliated project.

5. Conclusion

In this paper we presented Astroalign, a Python module to register stellar images, especially suited for the case where there is no WCS information available. Astroalign is well suited to use in astrophotography, coaddition, and image subtraction, and integrates well with other existing astronomy Python packages.

Image registration is done by identifying corresponding three-point asterisms (triangles) on both images and estimating the affine transformation between them. Asterism matching is a more robust and closer to the human way of matching images than generic image registration algorithms based on features.

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

References

References

Allen, A. & Schmidt, J. 2014, arXiv preprint arXiv:1407.5378

Barbary, K. 2016, The Journal of Open Source Software, 1, 59

Bay, H., Ess., A., Tuytelaars, T., & Goolab, L. 2008, Computer Vision and Image Understanding, 110, 346

Beroiz, M. & BC, J. 2019, toros-astro/astroalign: Version 2.0.1

Beroiz, M. I. 2019, Astrophysics Source Code Library

Berriman, G. B. & Good, J. C. 2017, Publications of the Astronomical Society of the Pacific, 129, 058006

---

[11]`https://www.astropy.org/affiliated/`

[12]`https://pypi.org/project/astroalign/`

Bertin, E. & Arnouts, S. 1996, Astronomy and Astrophysics Supplement Series, 117, 393

Bradski, G. 2000, Dr. Dobb's Journal of Software Tools

Brown, M. & Lowe, D. G. 2007, International Journal of Computer Vision, 74, 59

Delaunay, B. et al. 1934, Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk, 7, 1

Fathy, M. E., Hussein, A. S., & Tolba, M. F. 2011, Pattern Recognition Letters, 32, 383

Fischer, B. & Modersitzki, J. 2008, Inverse Problems, 24, 034008

Fischler, M. A. & Bolles, R. C. 1987, in Readings in Computer Vision, ed. M. A. Fischler & O. Firschein (San Francisco (CA): Morgan Kaufmann), 726 – 740

Greenfield, P. 2011, in Astronomical Data Analysis Software and Systems XX, Vol. 442, 425

Greisen, E. W. & Calabretta, M. R. 2002, A&A, 395, 1061

Groth, E. J. 1986, AJ, 91, 1244

Hill, D. L. G., Batchelor, P. G., Holden, M., & Hawkes, D. J. 2001, Physics in Medicine and Biology, 46, R1

Holscher, E. 2016, An introduction to Sphinx and Read the Docs for Technical Writers

Hutton, B. F., Braun, M., Thurfjell, L., & Lau, D. Y. 2002, European Journal of Nuclear Medicine and Molecular Imaging, 29, 559

Initiative, O. S. et al. 2006, 2015b.[Online]. Available: https://opensource.org/licenses/MIT.[Accessed 27 March 2017]

Jacob, J. C., Katz, D. S., Berriman, G. B., et al. 2009, International Journal of Computational Science and Engineering, 4

Jazayeri, M. 2007, in Future of Software Engineering, 2007. FOSE'07 (IEEE), 199–213

Jones, E., Oliphant, T., & Peterson, P. 2014, {SciPy}: Open source scientific tools for {Python}

Konolige, K. & Agrawal, M. 2008, IEEE Transactions on Robotics, 24, 1066

Lang, D., Hogg, D. W., Mierle, K., Blanton, M., & Roweis, S. 2010, The astronomical journal, 139, 1782

Lang, D., Hogg, D. W., Mierle, K., Blanton, M., & Roweis, S. 2010, AJ, 139, 1782

Lang, D., Hogg, D. W., Mierle, K., Blanton, M., & Roweis, S. 2012, Astrophysics Source Code Library

Latte, B., Henning, S., & Wojcieszak, M. 2019, in Proceedings of the Workshops of the Software Engineering Conference 2019, Vol. Vol-2308, 96–99

Lowe, D. G. 2004, International Journal of Computer Vision, 60, 91

Luong, Q.-T., Deriche, R., Faugeras, O., & Papadopoulo, T. 1993, On determining the fundamental matrix : analysis of different methods and experimental results, Research Report RR-1894, INRIA

Miller, J. C. & Maloney, C. J. 1963, Communications of the ACM, 6, 58

Murtagh, F. 1992, Publications of the Astronomical Society of the Pacific, 104, 301

Pál, A. 2012, Monthly Notices of the Royal Astronomical Society, 421, 1825

Pál, A. & Bakos, G. Á. 2006, PASP, 118, 1474

Price-Whelan, A., Sipőcz, B., Günther, H., et al. 2018, The Astronomical Journal, 156, 123

Robitaille, T. 2018, reproject: astronomical image reprojection in Python

Robitaille, T. P., Tollerud, E. J., Greenfield, P., et al. 2013, Astronomy & Astrophysics, 558, A33

Rublee, E., Rabaud, V., Konolige, K., & Bradski, G. 2011, in 2011 International Conference on Computer Vision (IEEE)

Tabur, V. 2007, PASA, 24, 189

Valiev, M., Vasilescu, B., & Herbsleb, J. 2018, in Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, 644–655

Van Der Walt, S., Colbert, S. C., & Varoquaux, G. 2011, Computing in Science & Engineering, 13, 22

Van der Walt, S., Schönberger, J. L., Nunez-Iglesias, J., et al. 2014, PeerJ, 2, e453

Van Rossum, G., Warsaw, B., & Coghlan, N. 2001, Python. org, 1565

Zhang, Z. 1998, International Journal of Computer Vision, 27, 161

Zhengwei Yang & Cohen, F. S. 1999, IEEE Transactions on Image Processing, 8, 934

Zitov, B. & Flusser, J. 2003, Image and Vision Computing, 21, 977

Appendix A. Comparative Table for Registration Software Packages

In the following lines we summarize some of the most recognized registration implementations. See Section 3 for a full discussion.

- Astroalign

  Language: Python

  Input type: Images

  Tessellation: Nearest-neighbor triangulation

  License: MIT

  Homepage: https://astroalign.readthedocs.io/

  Reference: Beroiz & BC (2019); Beroiz (2019)

- Astromety.net

  Language: ANSI-C

  Input type: Images

  Tessellation: Quads on Healpix cells

  License: BSD 3-Clause

  Homepage: http://astrometry.net/

  Reference: Lang et al. (2010, 2012)

- OPM_A

  Language: ANSI-C

  Input type: Large lists of star positions with non-linear distortion

  Tessellation: Triangulation with priority assigned to acute-angle triangles

  License: Custom

  Homepage: http://stella.astron.s.u-tokyo.ac.jp/nmatsuna/Japanese/software/OPM.html (Japanese)

  Reference: Tabur (2007)

- FITSH (grmatch and grtrans)

  Language: ANSI-C

  Input type: Large lists of star positions

  Tessellation: Delaunay Triangulation (Delaunay et al. 1934)

  License: GPL-3

  Homepage: https://fitsh.net/

  Reference: Pál & Bakos (2006); Pál (2012)