tetra3

Release 0.1

Aug 21, 2019

Contents:

tetra3 is a fast lost-in-space plate solver for star trackers written in Python.

Use it to identify stars in images and get the corresponding direction (i.e. right ascension and declination) in the sky which the camera points to. The only thing tetra3 needs to know is the approximate field of view of your camera.

The software is available in the tetra3 GitHub repository. General instructions are available at the tetra3 ReadTheDocs website. tetra3 is Free and Open Source Software released by the European Space Agency under the Apache License 2.0. See NOTICE.txt in the repository for full licensing details.

Performance will vary, but in general solutions will take 10 milliseconds (excluding time to extract star positions from images) with 10 arcsecond (50 microradian) accuracy.

A camera with a field of view of at least 10 degrees and 512 by 512 pixels is a good starting point. It is important that the distortion of the lens is low to preserve the true shape of the star patterns. Your camera should be able to acquire stars down to magnitude 6.5 for best results (for a narrow field of view camera this becomes very important as there are few bright stars).

A real-world set of images acquired with a FLIR Blackfly S BFS-U3-31S4M-C (Sony IMX265 sensor; binned 2x2) camera and a Fujifilm HF35XA-5M 35mm f/1.9 lens are included as test data (11.4 degrees field of view).

To effectively use tetra3 with your camera you may need to build a database optimised for your use case. See the API documentation for details on this. A database built for the test data is included as default_database.npz and may suit your needs if you have a similar camera setup.

CHAPTER 1

Installation

1.1 Getting Python

tetra3 is written for Python 3.7 (and therefore runs on almost any platform) and should work with most modern Python 3 installations. There are many ways to get Python on your system. The preferred method is by downloading Miniconda for Python 3+. If you are on Windows you will be given the option to *add conda to PATH* during installation. If you do not select this option, the instructions (including running python and tetra3) which refer to the terminal/CMD window will need to be carried out in the *Anaconda Prompt* instead.

Now, open a terminal/CMD window and test that you have conda and python by typing:

```
conda
python
exit()
```

(On Windows you may be sent to the Windows Store to download Python on the second line. Do not do this, instead go to *Manage app execution aliases* in the Windows settings and disable the python and python3 aliases to use the version installed with miniconda.)

Now ensure you are up to date:

```
conda update conda
```

You can now run any python script by typing `python script_name.py`.

1.1.1 Optional: Environment for tetra3

To run tetra3 in a python environment that is separated from your other projects (therefore avoiding requirements clashes and other issues) you may elect to create an environment. To do so:

```
conda create --name tetra3_env python=3.7 pip
conda activate tetra3_env
```

and then proceed the same. (To go to your base environment type `conda deactivate`.)

1.2 Getting tetra3

tetra3 is not available on PIP/PyPI (the Python Package Index). Instead you need to get the software repository from GitHub and place it in the directory where you wish to use it. (I.e. as a folder next to the python project you are writing.)

1.2.1 The quick way

Go to the GitHub repository, click *Clone or Download* and *Download ZIP* and extract the tetra3 directory to where you want to use it.

1.2.2 The good way

To be able to easily download and contribute updates to tetra3 you should install Git. Follow the instructions for your platform over here.

Now open a terminal/CMD window in the directory where you wish to use tetra3 and clone the GitHib repository:

```
git clone "https://github.com/esa/tetra3.git"
```

You should see the tetra3 directory created for you with all neccessary files. Check the status of your repository by typing:

```
cd tetra3
git status
```

which should tell you that you are on the branch "master" and are up to date with the origin (which is the GitHub version of tetra3). If a new update has come to GitHub you can update yourself by typing:

```
git pull
```

If you wish to contribute (please do!) and are not familiar with Git and GitHub, start by creating a user on GitHub and setting you username and email:

```
git config --global user.name "your_username_here"
git config --global user.email "email@domain.com"
```

You will now also be able to push proposed changes to the software. There are many good resources for learning about Git, the documentation which includes the reference, a free book on Git, and introductory videos is a good place to start.

1.3 Installing tetra3

To install the requirements open a terminal/CMD window in the tetra3 directory and run:

```
python setup.py install
```

to install all requirements. Test that everything works by running an example:

```
cd examples
python test_tetra3.py
```

which should print out the solutions for the included test images.

1.4 If problems arise

Please get in touch by filing an issue.

CHAPTER 2

API reference

This reference is auto-generated from the source code in the tetra3 GitHub repository. General instructions are avilable at the tetra3 ReadTheDocs website.

2.1 tetra3: A fast lost-in-space plate solver for star trackers.

Use it to identify stars in images and get the corresponding direction (i.e. right ascension and declination) in the sky which the camera points to. The only thing tetra3 needs to know is the approximate field of view of your camera. tetra3 also includes a versatile function to find spot centroids and statistics.

Included in the package:

- *tetra3.Tetra3*: Class to solve images and load/create databases.
- *tetra3.get_centroids_from_image()*: Extract spot centroids from an image.
- *tetra3.crop_and_downsample_image()*: Crop and/or downsample an image.

A default database (named *default_database*) is included in the repo, it is built for a maximum field of view of 12 degrees and and the default settings.

Note: If you wish to build you own database (e.g. for different field of view) you must download the Yale Bright Star Catalog 'BCS5' from <http://tdc-www.harvard.edu/catalogs/bsc5.html> and place in the tetra3 directory. (Direct download link: <http://tdc-www.harvard.edu/catalogs/BSC5>.)

It is critical to set up the centroid extraction parameters (see *get_centroids_from_image()* to reliably return star centroids from a given image. After this is done, pass the same keyword arguments to *Tetra3.solve_from_image()* to use them when solving your images.

This is Free and Open-Source Software based on *Tetra* rewritten by Gustav Pettersson at ESA.

The original software is due to: J. Brown, K. Stubis, and K. Cahoy, "TETRA: Star Identification with Hash Tables", Proceedings of the AIAA/USU Conference on Small Satellites, 2017. <https://digitalcommons.usu.edu/smallsat/2017/all2017/124/> <github.com/brownj4/Tetra>

2.1.1 tetra3.Tetra3

class tetra3.**Tetra3**(*load_database=None*, *debug_folder=None*)

Solve star patterns and manage databases.

To find the direction in the sky an image is showing this class calculates a "fingerprint" of the stars seen in the image and looks for matching fingerprints in a pattern catalogue loaded into memory. Subsequently, all stars that should be visible in the image (based on the fingerprint's location) are looked for and the match is confirmed or rejected based on the probability that the found number of matches happens by chance.

Each pattern is made up of four stars, and the fingerprint is created by calculating the distances between every pair of stars in the pattern and normalising by the longest to create a set of five numbers between zero and one. This information, and the desired tolerance, is used to find the indices in the database where the match may reside by a hashing function.

A database needs to be generated with patterns which are of appropriate scale for the field of view (FOV) of your camera. Therefore, generate a database using *generate_database()* with a *max_fov* which is the FOV of your camera (or slightly larger). A database with *max_fov=12* (degrees) is included as *default_database.npz*.

Star locations (centroids) are found using *tetra3.get_centroids_from_image()*, use one of your images to find settings which work well for your images. Then pass those settings as keyword arguments to *solve_from_image()*.

Example 1: Load database and solve image

```
import tetra3
# Create instance
t3 = tetra3.Tetra3()
# Load a database
t3.load_database('default_database')
# Create dictionary with desired extraction settings
```

(continues on next page)

(continued from previous page)

```
extract_dict = {'min_sum': 250, 'max_axis_ratio': 1.5}
# Solve for image, optionally passing known FOV estimate and error range
result = t3.solve_from_image(image, fov_estimate=11, fov_max_error=.5,␣
↪**extract_dict)
```

Example 2: Generate and save database

```
import tetra3
# Create instance
t3 = tetra3.Tetra3()
# Generate and save database
t3.generate_database(max_fov=20, save_as='my_database_name')
```

Parameters

- **load_database** (*str or pathlib.Path, optional*) – Database to load. Will call *load_database()* with the provided argument after creating instance.
- **debug_folder** (*pathlib.Path, optional*) – The folder for debug logging. If None (the default) the folder tetra3/debug will be used/created.

generate_database(*max_fov*, *save_as=None*, *pattern_stars_per_fov=10*, *catalog_stars_per_fov=20*, *star_min_magnitude=6.5*, *star_min_separation=0.05*, *pattern_max_error=0.005*)

Create a database and optionally save to file. Typically takes 5 to 30 minutes.

Note: If you wish to build you own database (e.g. for different field of view) you must download the Yale Bright Star Catalog 'BCS5' from <http://tdc-www.harvard.edu/catalogs/bsc5.html> and place in the tetra3 directory. (Direct download link: <http://tdc-www.harvard.edu/catalogs/BSC5>.)

Parameters

- **max_fov** (*float*) – Maximum angle (in degrees) between stars in the same pattern.
- **save_as** (*str or pathlib.Path, optional*) – Save catalog here when finished. Calls *save_database()*.
- **pattern_stars_per_fov** (*int, optional*) – Number of stars used for patterns in each region of size 'max_fov'.
- **catalog_stars_per_fov** (*int, optional*) – Number of stars in catalog in each region of size 'max_fov'.
- **star_min_magnitude** (*float, optional*) – Dimmest apparent magnitude of stars in database.
- **star_min_separation** (*float, optional*) – Smallest separation (in degrees) allowed between stars (to remove doubles).
- **pattern_max_error** (*float, optional*) – Maximum difference allowed in pattern for a match.

Example

```
# Create instance
t3 = tetra3.Tetra3()
# Generate and save database
t3.generate_database(max_fov=20, save_as='my_database_name')
```

load_database(*path='default_database'*)
Load database from file.

Parameters **path** (*str or pathlib.Path*) – The file to load. If given a str, the file will be looked for in the tetra3 directory. If given a pathlib.Path, this path will be used unmodified. The suffix .npz will be added.

save_database(*path*)
Save database to file.

Parameters **path** (*str or pathlib.Path*) – The file to save to. If given a str, the file will be saved in the tetra3 directory. If given a pathlib.Path, this path will be used unmodified. The suffix .npz will be added.

solve_from_image(*image*, *fov_estimate=None*, *fov_max_error=None*, *pattern_checking_stars=6*, *match_radius=0.01*, *match_threshold=1e-09*, ***kwargs*)
Solve for the sky location of an image.

Star locations (centroids) are found using *tetra3.get_centroids_from_image()* and keyword arguments are passed along to this method. Every combination of the *pattern_checking_stars* (default 6) brightest stars found is checked against the database before giving up.

Example

```
# Create dictionary with desired extraction settings
extract_dict = {'min_sum': 250, 'max_axis_ratio': 1.5}
# Solve for image
result = t3.solve_from_image(image, **extract_dict)
```

Parameters

- **image** (*numpy.ndarray*) – The image to solve for, must be convertible to numpy array.
- **fov_estimate** (*float, optional*) – Estimated field of view of the image in degrees.
- **fov_max_error** (*float, optional*) – Maximum difference in field of view from the estimate allowed for a match in degrees.
- **pattern_checking_stars** (*int, optional*) – Number of stars used to create possible patterns to look up in database.
- **match_radius** (*float, optional*) – Maximum distance to a star to be considered a match as a fraction of the image field of view.
- **match_threshold** (*float, optional*) – Maximum allowed mismatch probability to consider a tested pattern a valid match.
- ****kwargs** (*optional*) – Other keyword arguments passed to *tetra3.get_centroids_from_image()*.

Returns

A dictionary with the following keys is returned:

- 'RA': Right ascension of centre of image in degrees.
- 'Dec': Declination of centre of image in degrees.
- 'Roll': Rotation of image relative to north celestial pole.
- 'FOV': Calculated field of view of the provided image.
- 'RMSE': RMS residual of matched stars in arcseconds.
- 'Matches': Number of stars in the image matched to the database.
- 'Prob': Probability that the solution is a mismatch.
- 'T_solve': Time spent searching for a match in milliseconds.
- 'T_extract': Time spent exctracting star centroids in milliseconds.

If unsuccsessful in finding a match, None is returned for all keys of the dictionary except 'T_solve' and 'T_exctract'.

Return type dict

database_properties
Dictionary of database properties.

Keys:

- 'pattern_mode': Method used to identify star patterns.
- 'pattern_size': Number of stars in each pattern.
- 'pattern_bins': Number of bins per dimension in pattern catalog.
- 'pattern_max_error' Maximum difference allowed in pattern for a match.
- 'max_fov': Maximum angle between stars in the same pattern (Field of View; degrees).
- 'pattern_stars_per_fov': Number of stars used for patterns in each region of size 'max_fov'.
- 'catalog_stars_per_fov': Number of stars in catalog in each region of size 'max_fov'.
- 'star_min_magnitude': Dimmest apparent magnitude of stars in database.
- 'star_min_separation': Smallest separation allowed between stars (to remove doubles; degrees).

Type dict

debug_folder
Get or set the path for debug logging. Will create folder if not existing.

Type pathlib.Path

has_database
True if a database is loaded.

Type bool

pattern_catalog
Catalog of patterns in the database.

Type numpy.ndarray

star_table
Table of stars in the database.

The table is an array with six columns:

- Right ascension (radians)
- Declination (radians)
- x = cos(ra) * cos(dec)
- y = sin(ra) * cos(dec)
- z = sin(dec)
- Apparent magnitude

Type numpy.ndarray

2.1.2 tetra3.get_centroids_from_image

tetra3.**get_centroids_from_image**(*sigma=3*, *image_th=None*, *crop=None*, *downsample=None*, *filtsize=7*, *bg_sub_mode='local_median'*, *sigma_mode='local_median_abs'*, *binary_open=True*, *centroid_window=None*, *max_area=None*, *min_area=3*, *max_sum=None*, *min_sum=None*, *max_axis_ratio=None*, *max_returned=None*, *return_moments=False*, *return_images=False*)

Extract spot centroids from an image and calculate statistics.

This is a versatile function for finding spots (e.g. stars or satellites) in an image and calculating/filtering their positions (centroids) and statistics (e.g. sum, area, shape).

The coordinates start at the top/left edge of the pixel, i.e. x=y=0.5 is the centre of the top-left pixel. To convert the results to integer pixel indices use the floor operator.

To aid in finding optimal settings pass *return_images=True* to get back a dictionary with partial extraction results and tweak the parameters accordingly.

In general, the best extraction is attained with *bg_sub_mode='local_median'* and *sigma_mode='local_median_abs'* with a reasonable (e.g. 7 to 15) size filter. However, this may be slow (especially for larger filter sizes). A recommendable and much faster alternative is *bg_sub_mode='local_mean'* and *sigma_mode='global_root_square'* with a large (e.g. 15 to 25) sized filter. You may also elect to do background subtraction and finding the threshold by your own methods, then pass *bg_sub_mode=None* and your threshold as *image_th* to bypass these extraction steps.

The algorithm proceeds as follows:

1. Convert image to 2D numpy.ndarray with type float32.
2. Call `tetra3.crop_and_downsample_image()` with the image and supplied arguments *crop* and *downsample*.
3. Subtract the background if *bg_sub_mode* is not None. Four methods are available:
   - 'local_median' (the default): Create the background image using a median filter of size *filtsize* and subtract pixelwise.
   - 'local_mean': Create the background image using a mean filter of size *filtsize* and subtract pixelwise.
   - 'global_median': Subtract the median value of all pixels from each pixel.

- 'global_mean': Subtract the mean value of all pixels from each pixel.

4. Calculate the image threshold if image_th is None. If image_th is defined this value will be used to threshold the image. The threshold is determined by calculating the noise standard deviation with the metod selected as *sigma_mode* and then scaling it by *sigma* (default 3). The available methods are:
   - 'local_median_abs' (the default): For each pixel, calculate the standard deviation as the median of the absolute values in a region of size *filtsize* and scale by 1.48.
   - 'local_root_square': For each pixel, calculate the standard deviation as the square root of the mean of the square values in a region of size *filtsize*.
   - 'global_median_abs': Use the median of the absolute value of all pixels scaled by 1.48 as the standard deviation.
   - 'global_root_square': Use the square root of the mean of the square of all pixels as the standard deviation.
5. Create a binary mask using the image threshold. If *binary_open=True* (the default) apply a binary opening operation with a 3x3 cross as structuring element to clean up the mask.
6. Label all regions (spots) in the binary mask.
7. Calculate statistics on each region and reject it if it fails any of the max or min values passed. Calculated statistics are: area, sum, centroid (first moments) in x and y, second moments in xx, yy, and xy, major over minor axis ratio.
8. Sort the regions, largest sum first, and keep at most *max_returned* if not None.
9. If *centroid_window* is not None, recalculate the statistics using a square region of the supplied width (instead of the region from the binary mask).
10. Undo the effects of cropping and downsampling by adding offsets/scaling the centroid positions to correspond to pixels in the original image.

Parameters

- **image** (*numpy.ndarray*) – Image to find centroids in.
- **sigma** (*float, optional*) – The number of noise standard deviations to threshold at. Default 3.
- **image_th** (*float, optional*) – The value to threshold the image at. If supplied *sigma* and *simga_mode* will have no effect.
- **crop** (*tuple, optional*) – Cropping to apply, see *tetra3.crop_and_downsample_image()*.
- **downsample** (*int, optinal*) – Downsampling to apply, see *tetra3.crop_and_downsample_image()*.
- **filtsize** (*int, optional*) – Size of filter to use in local operations. Must be odd. Default 7.
- **bg_sub_mode** (*str, optional*) – Background subtraction mode. Must be one of 'local_median' (the default), 'local_mean', 'global_median', 'global_mean'.
- **sigma_mode** (*str, optinal*) – Mode used to calculate noise standard deviation. Must be one of 'local_median_abs' (the default), 'local_root_square', 'global_median_abs', or 'global_root_square'.
- **binary_open** (*bool, optional*) – If True (the default), apply binary opening with 3x3 cross to thresholded binary mask.

- **centroid_window** (*int, optional*) – If supplied, recalculate statistics using a square window of the supplied size.
- **max_area** (*int, optional*) – Reject spots larger than this.
- **min_area** (*int, optional*) – Reject spots smaller than this. Default 3.
- **max_sum** (*float, optional*) – Reject spots with a sum larger than this.
- **min_sum** (*float, optional*) – Reject spots with a sum smaller than this.
- **max_axis_ratio** (*float, optional*) – Reject spots with a ratio of major over minor axis larger than this.
- **max_returned** (*int, optional*) – Return at most this many spots.
- **return_moments** (*bool, optional*) – If set to True, return the calculated statistics (e.g. higher order moments, sum, area) together with the spot positions.
- **return_images** (*bool, optional*) – If set to True, return a dictionary with partial results from the steps in the algorithm.

Returns

If *return_moments=False* and *return_images=False* (the defaults) an array of shape (N,2) is returned with centroid positions (y down, x right) of the found spots in order of brightness. If *return_moments=True* a tuple of numpy arrays is returned with: (N,2) centroid positions, N sum, N area, (N,3) xx yy and xy second moments, N major over minor axis ratio. If *return_images=True* a tuple is returned with the results as defined previously and a dictionary with images and data of partial results.

Return type numpy.ndarray or tuple

2.1.3 tetra3.crop_and_downsample_image

tetra3.**crop_and_downsample_image**(*crop=None*, *downsample=None*, *sum_when_downsample=True*, *return_offsets=False*)

Crop and/or downsample an image. Cropping is applied before downsampling.

Parameters

- **image** (*numpy.ndarray*) – The image to crop and downsample. Must be 2D.
- **crop** (*int or tuple, optional*) – Desired cropping of the image. May be defined in three ways:
  - Scalar: Image is cropped to given fraction (e.g. crop=2 gives 1/2 size image out).
  - 2-tuple: Image is cropped to centered region with size crop = (height, width).
  - 4-tuple: Image is cropped to region with size crop[0:2] = (height, width), offset from the centre by crop[2:4] = (offset_down, offset_right).
- **downsample** (*int, optional*) – Downsampling factor, e.g. downsample=2 will combine 2x2 pixel regions into one. The image width and height must be divisible by this factor.
- **sum_when_downsample** (*bool, optional*) – If True (the default) downsampled pixels are calculated by summing the original pixel values. If False the mean is used.
- **return_offsets** (*bool, optional*) – If True, return the applied cropping offset.

Returns If *return_offsets=False* (the default) a 2D array with the cropped and dowsampled image is returned. If *return_offsets=True* is passed a tuple containing the image and a tuple with the cropping offsets (top, left) is returned.

Return type numpy.ndarray or tuple

Python Module Index

t

Index