pypogs

Release 0.1

Apr 26, 2022

Contents:

1 Installation **3**
- 1.1 Getting Python . . . 3
  - 1.1.1 Create environment for pypogs . . . 3
- 1.2 Getting pypogs . . . 4
  - 1.2.1 The quick way . . . 4
  - 1.2.2 The good way . . . 4
- 1.3 Installing pypogs . . . 5
- 1.4 Adding hardware support . . . 5
  - 1.4.1 Mount: Celestron, Orion, or SkyWatcher . . . 5
  - 1.4.2 Camera: FLIR/PointGrey . . . 5
  - 1.4.3 Receiver: National Instruments DAQ . . . 6
- 1.5 If problems arise . . . 6

2 Focal Plane Assembly Build **7**
- 2.1 Parts needed . . . 8
  - 2.1.1 Base items . . . 8
  - 2.1.2 Optics and mounts . . . 9
  - 2.1.3 Extras . . . 9
- 2.2 Constuction . . . 10

3 Graphical User Interface Guide **11**
- 3.1 Hardware Setup and Properties . . . 12
- 3.2 Manual Control . . . 13
- 3.3 Location and Alignment . . . 13
- 3.4 Target . . . 13
- 3.5 Controller Properties . . . 14
- 3.6 Tracking . . . 14
- 3.7 Status . . . 14
- 3.8 Live View and Interactive Control . . . 14

4 API reference **17**
- 4.1 High-level control of pypogs core . . . 17
- 4.2 Feedback and tracking algorithms . . . 29
- 4.3 Camera hardware interface . . . 44
- 4.4 Mount hardware interfaces . . . 49
- 4.5 Receiver hardware interfaces . . . 53

Python Module Index **57**

Index **59**

pypogs is an automated closed-loop satellite tracker for portable telescopes written in Python.

Use it to control your optical ground station, auto-align it to the stars, and automatically acquire and track satellites with closed-loop camera feedback. Additionally we include instructions for how to build a fibre-coupling Focal Plane Assembly (FPA) replacing the eyepiece in any unmodified portable telescope.

pypogs includes a platform independent Graphical User Interface (GUI) to manage alignment, tracking feedback, and hardware settings. The GUI controls the pypogs core through a public API (see documentation); pypogs may be controlled fully from the command line as well.

The software is available in the pypogs GitHub repository. All documentation is hosted at the pypogs ReadTheDocs website. pypogs 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. Our testing shows approximately 1 arcsecond RMS tracking of stars and MEO/GEO satellites, with 4 arcseconds RMS tracking of LEO satellites. With this performance you can launch the received signal into a 50µm and 150µm core diameter multimode fibre respectively with the proposed FPA. We require no modifications to the telescope nor a fine steering mirror for these results; pypogs will enable the lowest cost high-performance optical ground stations yet.

An article describing the system was presented at IEEE ICSOS in 2019; the paper is available here. The GitHub respository includes a preprint. If you find pypogs useful in your work, please cite:

G. M. Pettersson, J. Perdigues, and Z. Sodnik, "Unmodified Portable Telescope for Space-to-Ground Optical Links," in *Proc. IEEE International Conference on Space Optical Systems and Applications (ICSOS)*, 2019.

CHAPTER 1

Installation

1.1 Getting Python

pypogs is written for Python 3.7 (and therefore runs on almost any platform) and should work with most modern Python 3 installations. However, it is strongly recommended to use a separate Python environment for pypogs to avoid issues with clashing or out-of-date requirments as well as keeping your environment clean. Therefore the preferred method of getting Python 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 pypogs) 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 Create environment for pypogs

Now create an environment specific for pypogs by:

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

You will need to activate the environment each time you restart the terminal/CMD window. (To go to your base environment type `conda deactivate`.)

1.2 Getting pypogs

pypogs 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, or alone if you will just use the provided Graphical User Interface).

1.2.1 The quick way

Go to the GitHub repository, click *Clone or Download* and *Download ZIP* and extract the pypogs directory to where you want to use it. You will notice that the *pypogs/tetra3* directory is empty (thus pypogs will not work). Therefore also follow the link to tetra3 in the GitHub repository, download it the same way and place the contents in the *pypogs/tetra3* directory.

1.2.2 The good way

To be able to easily download and contribute updates to pypogs 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 pypogs and clone the GitHib repository:

```
git clone --recurse-submodules "https://github.com/esa/pypogs.git"
```

You should see the *pypogs* directory created for you with all neccessary files, including that the *pypogs/tetra3* directory Check the status of your repository by typing:

```
cd pypogs
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 pypogs).

If you find that *pypogs/tetra3* is empty (due to cloning without recursion), get it by:

```
git submodule update --init
```

If a new update has come to GitHub you can update yourself by typing:

```
git pull --recurse-submodules
```

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. Note that if you contribute to tetra3 you should push those changes to <https://github.com/esa/tetra3.git>.

1.3 Installing pypogs

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

```
pip install -r requirements.txt
```

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

```
cd examples
python run_pypogsGUI.py
```

which should create an instace of pypogs and open the Graphical User Interface for you.

1.4 Adding hardware support

To connect hardware devices you need to download and install both drivers and Python packages for the respective device.

Missing support for your hardware? File an issue and we will work together to expand pypogs hardware support!

1.4.1 Mount: Celestron, Orion, or SkyWatcher

Use *model='celestron'* in pypogs. No extra drivers or packages are neccessary.

1.4.2 Camera: FLIR/PointGrey

Use *model='ptgrey'* in pypogs. You will need both the *FLIR Spinnaker* drivers and the *pyspin* Python package. How to install:

1. Go to FLIR's Spinnaker page, and click *Download Now* to go to their "download box". Download *Latest Spinnaker Full SDK/SpinnakerSDK_FULL_{. . . }.exe* for your system (most likely Windows x64), and *Latest Python Spinnaker/spinnaker_python-{. . . .}.zip*. Take note that the python .zip should say *cp37* (for Python 3.7) and *amd64* for x64 system or *win32* for x86 system (matching the SpinnakerSDK you downloaded previously).
2. Run the Spinnaker SDK installer, "Camera Evaluation" mode is sufficient for our needs.
3. Run the newly installed SpinView program and make sure your camera(s) work as expected.
4. Extract the contents of the spinnaker_python .zip archive.
5. Open a terminal/CMD window (or Anaconda Prompt); activate your pypogs_env environment. Go to the extracted .zip archive, where the file *spinnaker_python-{. . . }.whl* is located.
6. Install the package by:

   ```
   pip install spinnaker_python-{...}.whl
   ```

You should now be ready to load and use your FLIR/PointGrey camera! Try running the GUI and add the camera with model *ptgrey* and the serial number printed on the camera (and shown in SpinView).

1.4.3 Receiver: National Instruments DAQ

Use *model='ni_daq'* in pypogs. Yoy will need both the *NI DAQmx* drivers and the *nidaqmx* Python package.

TODO: Detailed steps.

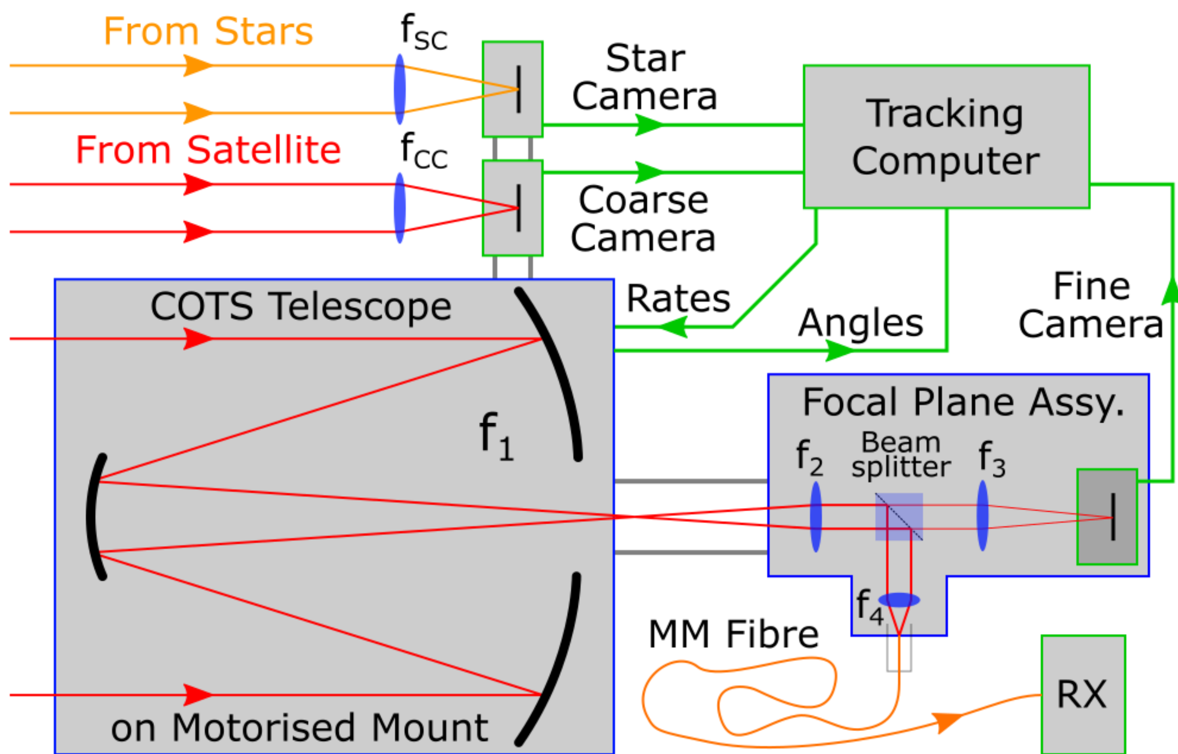
1.5 If problems arise

Please get in touch by filing an issue.

CHAPTER 2

Focal Plane Assembly Build

You may build a ground station for use with pypogs, with the general architecture shown below:



This can be implemented without any modifications to your telescope by replacing the eyepiece of your telescope with a focal plane assembly (FPA) show below besides an eyepiece:

The FPA costs less than 1000 euros (or dollars) to build (excl. the camera).

2.1 Parts needed

All optomechanics and optics may be ordered from Thorlabs except for the adapter to a telescope eyepice barrel. Your telescope vendor should have such an item. An example can be seen from Teleskop Service. You may adapt this FPA to use T2-mount (instead of C-mount) adapters and to use 2” instead of 1.25” eyepiece barrels.

2.1.1 Base items

- SC6W; 16 mm Cage Cube; 1pc
- SM05CP2; SM05 End Cap; 1pc
- SPM2; Cage Prism Mount; 1pc
- SB6C; Cube Clamp; 1pc
- SR1-P4; Cage Rod, 1” Long, 4 Pack; 1pc
- SSP05; XY Slip Plate; 1pc
- SM05FC; FC/PC SM05 Fibre Adapter; 1pc *
- SM05A3; SM05 to SM1 Adapter; 2pc

- SM1L10; SM1 Lens Tube, 1” Long; 1pc **
- SM1M10; SM1 Lens Tube w/o External Thread, 1” Long; 1pc ***
- SM1A9; C-Mount to SM1 Adapter; 1pc
- SM1A10; SM1 to C-Mount Adapter; 1pc

* May change for fibre connector of your choosing.
** May need to adjust length based on focal length *f3*.
*** May need to adjust length based on focal length *f2*.

2.1.2 Optics and mounts

For lenses *f2* and *f3* a lens with diameter 1” or less (1/2” recommended) may be used. For your chosen lens diameter, get an appropriate *SM1A{. . . }* adapter from the category “Mounting Adapters with Internal and External Threads”. You may also use pre-mounted lenses and an adapter to SM1 thread in these positions.

For the lens *f4* a lens with diameter 1/2” or less may be used (8mm recommended). For your chosen lens diameter, get an appropriate *SP{. . . }* adapter from the category “16 mm Cage Plates for Unmounted Optics” (or the SP02 for 1/2” lens diameter). You may also use pre-mounted lenses and an adapter to SM05 thread in the SP02 mount if preferred.

The beam-splitter must be a 10mm cube. Thorlabs has a range of non-polarising beam-splitters with reflection:transmission of 10:90, 30:70, 50:50, 70:30, 90:10. Typically 10% to the tracking camera is used. The maximum clear aperture of the FPA is limited by the beamsplitter’s clear aperture, which for Thorlabs’ offerings is 8mm.

There is space in the FPA to mount filters (1” or 1/2”) after *f2* or before *f3* inside each lens tube, or a filter (1/2”) in an SP02 cage plate before *f4* to suit your application.

A recommended setup would be:

- SM1A6T; SM1 Adapter for 1/2” (SM05) Optic; 2pc (*f2* and *f3* mount)
- AC127-025-B; f=25mm, 1/2” Achromatic Doublet; 1pc (*f2*)
- AC127-030-B; f=30mm, 1/2” Achromatic Doublet; 1pc (*f3*)
- SP10; 16 mm Cage Plate for 8mm Optic; 1pc (*f4* mount)
- AC080-010-B; f=10mm, 8mm Achromatic Doublet; 1pc (*f4*)
- BS071; 90:10 (R:T) beam-splitter 10mm cube; 1pc

2.1.3 Extras

It is recommended to get some glue (e.g. cyanoacrylate) to permanently bond the beam-splitter to the cage prism mount for stability, and matte-black aluminium foil to shield the FPA from stray light.

Useful to rotate the camera:

- CMSP025; C-Mount Spacer Ring, .25mm Thick; 1pc
- CMSP050; C-Mount Spacer Ring, .5mm Thick; 1pc
- CMSP100; C-Mount Spacer Ring, 1mm Thick; 1pc

Useful for building/aligning:

- SPW602; Spanner Wrench for SM1; 1pc
- SPW603; Spanner Wrench for SM05; 1pc

- SR05-P4; Cage Rod, 1/2” Long, 4 Pack; 1pc
- SCPA1; 16 mm Cage Alignment Plate; 1pc
- CPS635R; Collimated Laser Module, 635 nm, 1.2 mW; 1pc
- CPS11K(-EC); Ø11 mm Laser Diode Module Mounting Kit; 1pc
- SP03; 16mm Cage Plate Clear Aperture; 1pc
- MSP2(/M); Mini Pedestal Pillar Post; 1pc
- MSC2; Mini Clamping Fork; 1pc

2.2 Constuction

1. Add two SM05A3 adapters to opposite sides of your SC6W cage cube.
2. Add four SR1 rods to one side of your cage cube.
3. Glue your beam-splitter to the SPM2 mount and insert into cage cube (take care of orientation!).
4. Mount your camera to the SM1L10 lens tube with the SM1A9 adapter.
5. Mount your *f3* lens in its SM1A6T adapter (take care of orientation!) and place in lens tube.

    Optional for setting camera rotation:

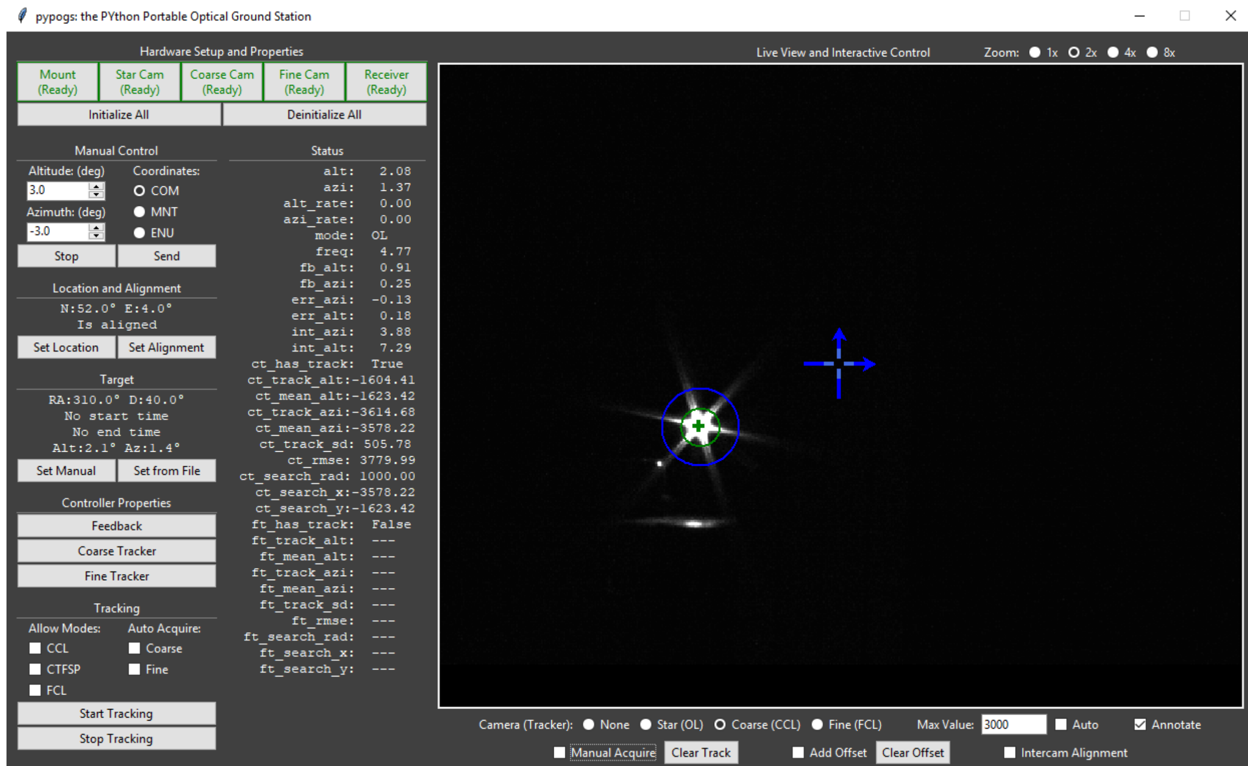
    a. Attach the lens tube with the camera to the cage cube.
    b. Add CMSPxxx spacers as desired to rotate the camera.
    c. Remove the camera again and continue the steps.

6. Lock your *f3* lens in infinity focus (e.g. focus on something far away).
7. Mount the lens tube to the cage cube SM05A3 adapter.
8. Mount your *f4* lens in its SP10 cage plate adapter (take care of orientation!).
9. Lock the *f4* cage plate on the SR1 rods, flush with the cage cube.
10. Attach SM05FC fibre adapter in SSP05 slip plate and mount on the SR1 rods.
11. Attach a fibre and illuminate from the opposite end. You will see reflections from the beam-splitter on your camera. Rotate the beam-splitter until the reflections overlap and adjust the slip plate position until a centered sharp image of the fibre face is seen.
12. Lock your beam-splitter rotation and add SB6C clamp, lighly pressing on the beam-splitter.
13. Mount your *f2* lens in its SM1A6T adapter (take care of orientation!) and place in SM1M10 lens tube. Attach to cage cube’s SM05A3 adapter.
14. Fix position of *f2* such that primary focus is a few mm outside the lens tube (typically this lens sits as close to the cage cube as possible).
15. Add SM1A10 adapter and your C-mount to telescope eyepiece barrel (e.g. 1.25”).
16. Finish by adding SM05CP2 end cap in the unused hole.

CHAPTER 3

Graphical User Interface Guide

pypogs includes a simple yet powerful Graphical User Interface (GUI) to control a tracking session. It includes for example hardware setup, alignment, target selection and tracking, automatic or manual closed-loop acquisition, and controlling all system parameters. A typical tracking scenario may look as below:



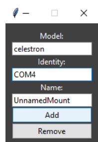
Below, an explanation of the GUI features and functionality is presented. You may open an empty GUI by going to the *examples* folder in a terminal/CMD window and typing:

```
python run_pypogsGUI.py
```

Note that it is entirely possible (even recommended) to first load and set up pypogs using the API before loading the GUI, to configure the system to your desires on startup. See the example file *setupCLI_runGUI.py* for how this may be done (this is in particular the setup for ESA's test system). To find appropriate settings to change for your system, however, it's a good idea to load everything through the GUI and set it up as you like, then edit the mentioned example to those settings.

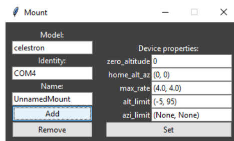
3.1 Hardware Setup and Properties

Hardware is loaded and settings are controlled here. Clicking *Mount* will open this window:



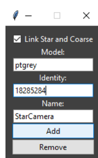
Here, type the *Model* and *Identity* appropriate for the hardware device you want to add. See the API documentation for specific allowed values. Clicking *Add* will connect to the device.

You should now see two things: 1) the *Mount* button in the main window says *Ready* and gets a green border; 2) the window changes to this:

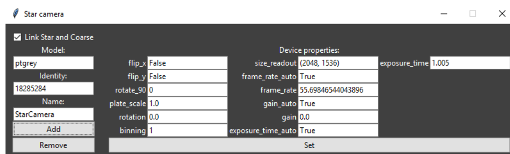


Where you can now change the hardware settings. For the mount these are typically just left to default.

The cameras are added similarily, with one important distinction. When adding either coarse or star camera a checkbox labelled *Link Star and Coarse* appears. Check this box if you are using the same physical device as star and coarse cameras, see below:



After clicking add, it should change to:



Where now many important settings are available. In particular, *the camera plate scale must be set correctly.*

Loading the other cameras and a receiver is straightforward, following the same procedure as above.

When you are done, all the hardware devices you wish to use should say *Ready* with green borders.

3.2 Manual Control

Here you may send move commands to the telescope mount. There are three coordinate systems available to choose from. They are defined more specifically in the ICSOS paper.

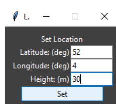
1. COM (commanded): Send the entered values directly to the mount.
2. MNT (mount): Go to this poisiton in the mount-local corrected coordinate system. Requires system to be aligned.
3. ENU (east north up): Go to this position in the traditional altitude-azimuth angles relative to the horizon and north direction. Requires system to be aligned and located.

Pressing *Send* commands the move. Pressing *Stop* aborts the move if still in progress.

3.3 Location and Alignment

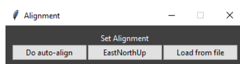
Here you may set the telescopes location on Earth, and the alignment.

Clicking *Set Location* opens this window:



Where you should enter the coordinates of your telescope. Note that north and east are positive directions and that the height is the *distance above the reference ellipsoid* (not above sea level). After pressing *Set* the location should be listed in the main window.

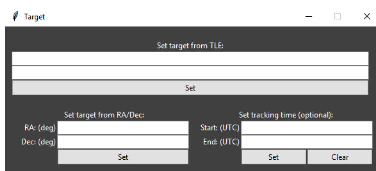
Clicking *Set Alignment* opens this window:



Here, you may click *Do auto-align* to do the star-camera based auto alignment procedure (takes c. 3min 15s). If you have physically aligned your telescope to the traditional ENU coordinates you may instead click *EastNorthUp*. The text in the main window should change to read “Is Aligned” when finished. (*Load from file* is not yet implemented.)

3.4 Target

Here you may set the tracking target. All tracking is calculated in real-time, so there are no options for pre-planning. Clicking *Set Manual* brings up this window:

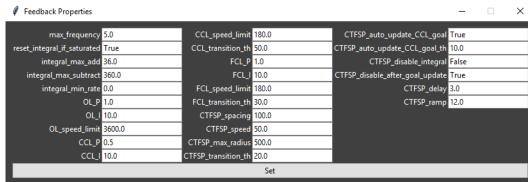


Here you may enter either a satellite’s orbit as a two-line element (TLE) or a deep-sky object by its right ascension (RA) and declination (Dec). Clicking *Set* should list the target in the main window. After setting the target you may optionally enter times (as an ISO UTC timestamp e.g. *2019-09-16 13:14:15*) for when tracking should start and stop. If you do this, pressing *Start Tracking* will slew to the start position target and then wait for the start time before tracking.

(*Set from File* is not yet implemented.)

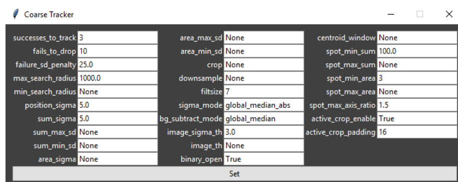
3.5 Controller Properties

Here you can control the nitty gritty of the feedback system as well as the fine and coarse trackers. See the API reference for what each of these items do. Pressing *Feedback* opens this window:



where you can change each item and then press *Set*.

Clicking either of *Coarse Tracker* and *Fine Tracker* similarily opens the respective windows:



3.6 Tracking

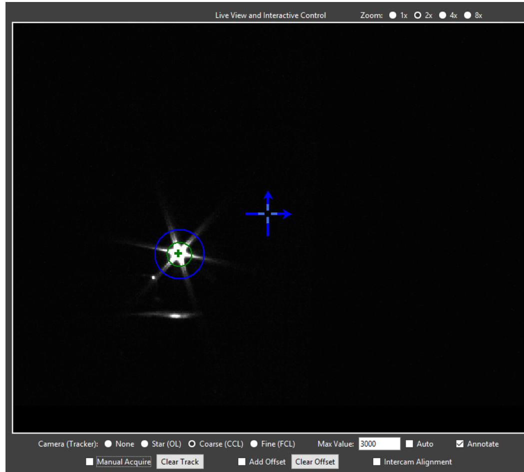
Here you control the general tracking functionality. Allowing the three modes CCL (coarse closed loop), CTFSP (coarse to fine spiral), and FCL (fine closed loop) is decided here. For pypogs to switch into a mode the available track needs to fulfill the transition conditions (see Controller Properties) and the mode must be allowed. You can also allow auto-acquisition of satellites (i.e. pypogs will attempt to track bright spots that show up on the cameras and determine if they are good or not).

3.7 Status

This lists (in real-time) the tracking data pypogs is using/calculating.

3.8 Live View and Interactive Control

The majority of the GUI is dedicated to user interaction with the trackers. Typically it looks like this:



The top right corner allows the live view to be zoomed in.

The first row below the live view controls what is displayed. The checkboxes *None*, *Star*, *Coarse*, and *Fine* show the corresponding camera. *The chosen camera also determines which tracker is controlled when interacting.* *Max Value* sets the scaling of the live view (the value which is mapped to white), it may also be set to *Auto*. *Annotate* selects if the tracking overlay is shown on the live view.

Several overlays are shown if you are viewing the coarse or fine camera:

- Large blue cross-hair with arrows: The position of this crosshair shows the *intercamera alignment*. (I.e. for the coarse camera this is where the target needs to be to appear on the fine camera.) The arrows show the horizontal (azimuth) and vertical (altitude) direction (i.e. if camera rotation is nonzero the crosshair will be rotated equivalently.)
- Small light blue cross-hair: This shows the offset relative to the intercamera alignment, which most of the time is zero. (For details see offset description below.)
- Green circle: This shows the current tracking estimate. The circle is centered at the tracking mean and has radius of the tracking standard deviation.
- Green cross: This is the instant track position.
- Blue circle: This shows the tracking search region. Only spots appearing inside this circle are allowed for tracking.

The second row below the live view interacts with the selected tracker. There are three functions:

- Manual acquisition: If you see your target satellite on the live view, check the *Manual Acquire* box and then click on the satellite. This will set the search region (blue circle) to where you clicked and the satellite will be acquired. You can also clear the current track (if it's stuck on the wrong target).
- Offsets: Often there is significant error in the predicted position of the satellite. Of course, you can have pypogs closed-loop track the satellite (using acquisition as above). However, you may also offset the tracker to manually point towards a specific target without enabling the tracker. Consider, e.g., you are in OL (open loop) tracking mode, and are viewing the coarse (CCL) camera. If you check *Add Offset* and then click a point in the live view, an offset will be added to the OL tracker such that this point ends up at the large blue cross-hair.
- Intercamera alignment: This is typically only used when setting up the system at the start of an observing session. Checking *Intercam Alignment* and clicking in the live view will move the large blue cross-hair to this position. Use this to align the cameras by: 1) point the telescope to something recognisable in the fine camera; 2) select the coarse camera, move the cross-hair to the same object as is shown in the fine camera; 3) do the same for the star camera.

CHAPTER 4

API reference

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

4.1 High-level control of pypogs core

- *pypogs.System* is the main instance for interfacing with and controlling the pypogs core. It allows the user to create and manage the hardware, tracking threads, targets etc.
- *pypogs.Alignment* manages the alignment and location of the system, including coordinate transformations.
- *pypogs.Target* manages the target and start/end times.

This is Free and Open-Source Software originally written by Gustav Pettersson at ESA.

License: Copyright 2019 the European Space Agency

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

https://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

class pypogs.**System**(*data_folder=None*, *debug_folder=None*)

Instantiate and control pypogs functionality and hardware.

The first step is always to create a System instance. The hardware and all other classes are accessible as properties of the System (e.g. *coarse_camera*, *control_loop_thread*) and should be created by asking the System instance to do so.

Note: Tables of Earth's rotation must be downloaded to do coordinate transforms. Calling *update_databases()* will attempt to download these from the internet (if expired). To facilitate offline

use of pypogs, these will otherwise not be automatically downloaded unless strictly necessary. If you use pypogs offline do this update every few months to keep the coordinate transforms accurate.

Example

```
import pypogs
sys = pypogs.System()
```

Add and control hardware: There are five possible hardware instances:

- *mount* (*pypogs.Mount*)
- *star_camera* (*pypogs.Camera*)
- *coarse_camera* (*pypogs.Camera*)
- *fine_camera* (*pypogs.Camera*)
- *receiver* (*pypogs.Receiver*)

They should be added to the System by using an add method, e.g. *add_mount()* for *mount*, and may be removed by the respective *clear_mount()*. One special case is if the star and coarse cameras are the same physical camera, then use *add_star_camera_from_coarse()* to copy the *coarse_camera* object to *star_camera*. After adding, see the respective class documentation for controlling the settings.

Example:

```
sys.add_coarse_camera(model='ptgrey', identity='18285284')
sys.add_star_camera_from_coarse()
sys.add_fine_camera(model='ptgrey', identity='18285254')
sys.coarse_camera.exposure_time = 100  # milliseconds
sys.coarse_camera.frame_rate = 2  # hertz
sys.coarse_camera is sys.star_camera  # returns True
```

Settings for closed loop tracking: When a coarse or fine Camera object is added, a corresponding tracking thread (e.g. *coarse_track_thread* for *coarse_camera*) is also created. This is a *pypogs.TrackingThread* object which also holds a *pypogs.SpotTracker* object (accessible via `coarse_track_thread.spot_tracker`). The parameters for these (see respective documentation) are used to set up the detection for tracking. For best performance it is critical to set up these well.

Further, a *pypogs.ControlLoopThread* object is available as *control_loop_thread* and defines the feedback parameters used for closed loop tracking (e.g. gains, modes, switching logic).

Example:

```
sys.coarse_track_thread.feedforward_threshold = 10
sys.coarse_track_thread.spot_tracker.bg_subtract_mode = 'global_median'
sys.control_loop_thread.CCL_P = 1
```

Set location and alignment: *alignment* references the *pypogs.Alignment* instance (auto created) used to determine and calibrate the location, alignment, and mount corrections. See the class documentation for how to set location and alignments. To do auto-alignment, use the *do_auto_star_alignment()* method (requires a star camera). Plate solving for the auto alignment is performed via the tetra3 package. You can set a custom instance via *tetra3*, otherwise a default instance will be created for you.

If your mount has built in alignment (and/or is physically aligned to the earth) you may call `alignment.set_alignment_enu()` to set the telescope alignment to East, North, Up (ENU) coordinates, which will also disable the corrections done in pypogs. ENU is the traditional astronomical coordinate system for altitude (elevation) and azimuth telescopes, measured as degrees above the horizon and degrees away from north (towards east) respectively.

Example:

```
sys.alignment.set_location_lat_lon(lat=52.2155, lon=4.4194, height=45)
sys.do_auto_star_alignment()
```

Set target: *`target`* references the *`pypogs.Target`* instance (auto created) which holds the target and (optional) tracking start and end times. The target may be set directly to an *astropy SkyCoord* or a *skyfield EarthSatellite* by `target.set_target()` or these can be created by pypogs by e.g. calling `target.set_target_from_tle()` with a Two Line Element (TLE) for a satellite or `target.set_target_from_ra_dec()` with right ascension and declination (in decimal degrees) for a star.

Example satellite:

```
tle = ['1 28647U 05016B   19180.65078896  .00000014  00000-0  19384-4 0 ␣
↪9991',\
       '2 28647  56.9987 238.9694 0122260 223.0550 136.0876 15.05723818 ␣
↪6663']
sys.target.set_target_from_tle(tle)
```

Example star:

```
sys.target.set_target_from_ra_dec(37.95456067, 89.26410897)  # Polaris
```

Control tracking: You are now ready! Just call *`start_tracking()`* and then `stop_tracking()` when you are finished.

Release hardware: Before exiting, it's strongly recommended to call *`deinitialize()`* to release the hardware resources.

Parameters

- **`data_folder`** (*`pathlib.Path, optional`*) – The folder for data saving. If None (the default) the folder *pypogs*/data will be used/created.
- **`debug_folder`** (*`pathlib.Path, optional`*) – The folder for debug logging. If None (the default) the folder *pypogs*/debug will be used/created.

`add_coarse_camera`(*model=None*, *identity=None*, *name='CoarseCamera'*, *auto_init=True*)

Create and set the coarse camera. Calls pypogs.Camera constructor with name='CoarseCamera' and the given arguments.

Parameters

- **`model`** (*`str, optional`*) – The model used to determine the correct hardware API. Supported: 'ptgrey' for PointGrey/FLIR Machine Vision cameras (using Spinnaker/PySpin APIs).
- **`identity`** (*`str, optional`*) – String identifying the device. For *ptgrey* this is 'serial number' *as a string*.
- **`name`** (*`str, optional`*) – Name for the device, defaults to 'CoarseCamera'.

- **auto_init** (*bool, optional*) – If both model and identity are given when creating the Camera and auto_init is True (the default), Camera.initialize() will be called after creation.

add_coarse_camera_from_star()
Set the coarse camera to be the current star camera object.

add_fine_camera(*model=None*, *identity=None*, *name='FineCamera'*, *auto_init=True*)
Create and set the fine camera. Calls pypogs.Camera constructor with name='FineCamera' and the given arguments.

Parameters

- **model** (*str, optional*) – The model used to determine the correct hardware API. Supported: 'ptgrey' for PointGrey/FLIR Machine Vision cameras (using Spinnaker/PySpin APIs).
- **identity** (*str, optional*) – String identifying the device. For *ptgrey* this is 'serial number' *as a string*. name (str, optional): Name for the device, defaults to 'FineCamera'.
- **auto_init** (*bool, optional*) – If both model and identity are given when creating the Camera and auto_init is True (the default), Camera.initialize() will be called after creation.

add_mount(**args*, ***kwargs*)
Create and set a pypogs.Mount for System.mount. Arguments passed to constructor.

Parameters

- **model** (*str, optional*) – The model used to determine the the hardware control interface. Supported: 'celestron' for Celestron NexStar and Orion/SkyWatcher SynScan (all the same) hand controller communication over serial.
- **identity** (*str or int, optional*) – String or int identifying the device. For model *celestron* this can either be a string with the serial port (e.g. 'COM3' on Windows or '/dev/ttyUSB0' on Linux) or an int with the index in the list of available ports to use (e.g. identity=0 i if only one serial device is connected.)
- **name** (*str, optional*) – Name for the device.
- **auto_init** (*bool, optional*) – If both model and identity are given when creating the Mount and auto_init is True (the default), Mount.initialize() will be called after creation.

add_receiver(**args*, ***kwargs*)
Create and set a pypogs.Receiver for the system. Arguments passed to constructor.

Parameters

- **model** (*str, optional*) – The model used to determine the correct hardware API. Supported: 'ni_daq' for National Instruments DAQ cards (tested on USB-6211).
- **identity** (*str, optional*) – String identifying the device and input. For *ni_daq* this is 'device/input' eg. 'Dev1/ai1' for device 'Dev1' and analog input 1; only differential input is supported for *ni_daq*.
- **name** (*str, optional*) – Name for the device.
- **save_path** (*pathlib.Path, optional*) – Save path to set. See Receiver.save_path for details.

- **auto_init** (*bool, optional*) – If both model and identity are given when creating the Receiver and auto_init is True (the default), Receiver.initialize() will be called after creation.

add_star_camera (*model=None*, *identity=None*, *name='StarCamera'*, *auto_init=True*)
Create and set the star camera. Calls pypogs.Camera constructor with name='StarCamera' and the given arguments.

Parameters

- **model** (*str, optional*) – The model used to determine the correct hardware API. Supported: 'ptgrey' for PointGrey/FLIR Machine Vision cameras (using Spinnaker/PySpin APIs).
- **identity** (*str, optional*) – String identifying the device. For *ptgrey* this is 'serial number' *as a string*.
- **name** (*str, optional*) – Name for the device, defaults to 'StarCamera'.
- **auto_init** (*bool, optional*) – If both model and identity are given when creating the Camera and auto_init is True (the default), Camera.initialize() will be called after creation.

add_star_camera_from_coarse ()
Set the star camera to be the current coarse camera object.

clear_coarse_camera ()
Set the coarse camera to None.

clear_fine_camera ()
Set the fine camera to None.

clear_mount ()
Set the mount to None.

clear_receiver ()
Set the receiver to None.

clear_star_camera ()
Set the star camera to None.

deinitialize ()
Deinitialise camera, mount, and receiver if they are initialised.

do_alignment_test (*max_trials=2*, *rate_control=True*)
Move to 40 positions spread across the sky and measure the alignment errors.

Data is saved to CSV file named with the current time and '_System_align_test_hemisp.csv'.

Parameters

- **max_trials** (*int, optional*) – Maximum attempts to take each image and solve the position. Default 2.
- **rate_control** (*bool, optional*) – If True (the default) rate control (see pypogs.Mount) is used.

do_auto_star_alignment (*max_trials=1*, *rate_control=True*)
Do the auto star alignment procedure by taking eight star images across the sky.

Will call System.Alignment.set_alignment_from_observations() with the captured images.

Parameters

- **max_trials** (*int, optional*) – Maximum attempts to take each image and solve the position. Default 1.
- **rate_control** (*bool, optional*) – If True (the default) rate control (see pypogs.Mount) is used.

get_alt_az_of_target(*times=None*, *time_step=0.1*)
Get the corrected altitude and azimuth angles and rates of the target from the current alignment.

Parameters

- **times** (*astropy.time.Time, optional*) – The time(s) to calculate for. If None (the default) the current time is used. If time array the calculation is done for each time in the array.
- **time_step** (*float, optional*) – The time step in seconds used to calculate the angular rates (default .1).

Returns Nx2 array with altitude and azimuth angles in degrees. numpy.ndarray: Nx2 array with altitude and azimuth rates in degrees per second.

Return type numpy.ndarray

get_itrf_direction_of_target(*times=None*)
Get direction (unit vector) in ITRF from the telescope position to the target.

Parameters **times** (*astropy.time.Time, optional*) – The time(s) to calculate for. If None (the default) the current time is used. If time array the calculation is done for each time in the array.

Returns Shape (3,) if single time, shape (3,N) if array time.

Return type numpy.ndarray

initialize()
Initialise cameras, mount, and receiver if they are not already.

slew_to_target(*time=None*, *block=True*, *rate_control=True*)
Slew the mount to the defined target.

Parameters

- **time** (*astropy.time.Time, optional*) – The time to calculate target position. If None (the default) the current time is used.
- **block** (*bool, optional*) – If True (the default), execution is blocked until move finishes.
- **rate_control** (*bool, optional*) – If True (the default) rate control (see pypogs.Mount) is used.

start_tracking()
Track the target, using closed loop feedback if defined.

The target will be tracked between the start and end times defined in the target. To stop manually call System.stop_tracking().

stop()
Stop all tasks.

static update_databases()
Download and update Skyfield and Astropy databases (of earth rotation).

alignment
Get the system alignment object.

Type *pypogs.Alignment*

coarse_camera
Get or set the coarse camera. Will create System.coarse_track_thread if not existing.

Type *pypogs.Camera*

coarse_track_thread
Get the coarse tracking thread.

Type *pypogs.TrackingThread*

control_loop_thread
Get the system control loop thread.

Type System.ControlLoopThread

data_folder
Get or set the path for data saving. Will create folder if not existing.

Type pathlib.Path

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

Type pathlib.Path

fine_camera
Get or set the fine camera. Will create System.fine_track_thread if not existing.

Type *pypogs.Camera*

fine_track_thread
Get the fine tracking thread.

Type *pypogs.TrackingThread*

is_busy
Returns True if system is doing some control task.

Type bool

is_init
Returns True if all attached hardware (cameras, mount, and receiver) are initialised.

Type bool

mount
Get or set the mount.

Type *pypogs.Mount*

receiver
Get or set a pypogs.Receiver for the telescope.

star_camera
Get or set the star camera.

Type *pypogs.Camera*

target
Get the system target object.

Type *pypogs.Alignment*

tetra3
Get or set the tetra3 instance used for plate solving star images. Will create an instance with 'default_database' if none is set.

Type tetra3.Tetra3

class pypogs.**Alignment**(*data_folder=None*, *debug_folder=None*)
Alignment and location of a telescope and coordinate transforms.

Location is the position of the telescope on Earth. It is provided as latitude, longitude, height relative to a reference ellipsoid. WGS84 (the GPS ellipsoid) is used in pypogs (and almost everywhere else).

Alignment refers to the different coordinate frames in use to get the telescope to point in the correct direction. A direction in some coordinate frame can either be represented as a cartesian unit vector or as two angles: altitude and azimuth. In the former case they are appended by _xyz and in the latter by _altaz. There are four coordinate frames to consider, ITRF, ENU, MNT, and COM, see below for descriptions.

The fundamental coordinates used are ITRF_xyz unit vectors. They give direction in an earth fixed cartesian frame. This can often be called the ECEF (Earth-Centered Earth-Fixed) or ECR (Earth-Centered Rotating) frame. pypogs uses external packages (Astropy and Skyfield) to get directions in ITRF_xyz, but the other three are managed here.

If using auto-align, all you really need to know is the ITRF or ENU direction you want the telescope to point towards, use this class to get COM_altaz, and send that to the mount.

Note: If you have a ITRF position vector (e.g. from centre of earth to a satellite) use Alignment.get_itrf_relative_from_position() to get the unit vector direction from the telescope location or pass the optional argument *position=True* when converting from ITRF_xyz.

If you have a telescope with internal alignment and corrective terms such that the communicated values with the mount are in traditional ENU_altaz coordinates, set the location and then call Alignment.set_alignment_enu(). This will make MNT coincide with the present ENU frame and disable corrections.

To understand this class deeply, one needs to understand the different coordinate systems that are used. They are:

1. ITRF: *International Terrestrial Reference Frame* is the common ECEF (Earth-Centered Earth-Fixed) cartesian coordinate frame. This gives a position in (x, y, z) where the coordinates rotate along with Earth, so a point on Earth always has the same coordinates. (0, 0, 0) is the centre of Earth. This is the "base" coordinate frame in this class which everything else is referenced to.
2. ENU: *East North Up* is the local tangent plane coordinate frame and depends on the location of the telescope. The coordinates may be described in their cartesian components (e, n, u) but are more commonly referenced by the two angles ENU altitude (degrees above the horizon) and ENU azimuth (degrees rotation east from north). ENU_altaz is the traditional astronomical coordinate frame seen e.g. in star charts. To transform into this frame the location must be known.
3. MNT: *Mount* is the *local* coordinate system of the telescope mount. This is another cartesian coordinate where the coordinate axes coincide with the telescope mount axes. In particular, MNT c is the azimuth rotation axis of the gimbal, MNT a is the altitude rotation axis of the gimbal (when the gibals are at the zero position), and MNT b completes the right hand system (pointing along the telescope boresight). The coordinates are commonly expressed as MNT altitude (degrees above the plane normal to the azimuth rotation axis) and MNT azimuth (degrees rotation around the azimuth axis). Traditionally, telescopes are physically aligned such that MNT and ENU coincide, but we relax this harsh constraint via software!
4. COM: *Commanded* is the angles which must be send to the mount to actually end up at the desired MNT_altaz. The nonperpendicularity (Cnp), vertical deflection (Cvd), and altitude zero offset (Alt0) of

the physical mount must be corrected for to get better than about 1000 arcseconds pointing. In an ideal mount COM and MNT coincide.

Parameters

- **data_folder** (*pathlib.Path, optional*) – The folder for data saving. If None (the default) the folder *pypogs*/data will be used/created.
- **debug_folder** (*pathlib.Path, optional*) – The folder for debug logging. If None (the default) the folder *pypogs*/debug will be used/created.

get_com_altaz_from_enu_altaz(*enu_altaz*)
Transform the given ENU AltAz coordinate to COM AltAz.

Parameters **enu_altaz** (*numpy array-like*) – Size 2 or shape (2,N).

Returns Shape (2,) if single input, shape (2,N) if array input.

Return type numpy.ndarray

get_com_altaz_from_itrf_xyz(*itrf_xyz*, *position=False*)
Transform the given ITRF xyz coordinates to COM AltAz.

Parameters

- **itrf_xyz** (*numpy array-like*) – Size 3 or shape (3,N).
- **position** (*bool, optional*) – If True, itrf_xyz is treated as the position in ITRF we want to observe, and the ENU AltAz required to see it from our location is returned. If False (the default), itrf_xyz is treated as a vector with the desired direction to point in ITRF.

Returns Shape (2,) if single input, shape (2,N) if array input.

Return type numpy.ndarray

get_com_altaz_from_mnt_altaz(*mnt_altaz*)
Transform the given MNT AltAz coordinate to COM AltAz.

Parameters **mnt_altaz** (*numpy array-like*) – Size 2 or shape (2,N).

Returns Shape (2,) if single input, shape (2,N) if array input.

Return type numpy.ndarray

get_enu_altaz_from_com_altaz(*com_altaz*)
Transform the given COM AltAz coordinate to ENU AltAz.

Parameters **mnt_altaz** (*numpy array-like*) – Size 2 or shape (2,N).

Returns Shape (2,) if single input, shape (2,N) if array input.

Return type numpy.ndarray

get_enu_altaz_from_itrf_xyz(*itrf_xyz*, *position=False*)
Transform the given ITRF xyz coordinates to ENU AltAz.

Parameters

- **itrf_xyz** (*numpy array-like*) – Size 3 or shape (3,N).
- **position** (*bool, optional*) – If True, itrf_xyz is treated as the position in ITRF we want to observe, and the ENU AltAz required to see it from our location is returned. If False (the default), itrf_xyz is treated as a vector with the desired direction to point in ITRF.

Returns Shape (2,) if single input, shape (2,N) if array input.

Return type numpy.ndarray

get_enu_altaz_from_mnt_altaz(*mnt_altaz*)
Transform the given MNT AltAz coordinate to ENU AltAz.

Parameters **mnt_altaz** (*numpy array-like*) – Size 2 or shape (2,N).

Returns Shape (2,) if single input, shape (2,N) if array input.

Return type numpy.ndarray

get_itrf_relative_from_position(*itrf_pos*)
Get the vector in ITRF pointing from the telescope to the given position. Not normalised.

get_itrf_xyz_from_enu_altaz(*enu_altaz*)
Transform the given ENU AltAz coordinate to ITRF xyz. May be numpy array-like.

Parameters **enu_altaz** (*numpy array-like*) – Size 2 or shape (2,N).

Returns Shape (3,) if single input, shape (3,N) if array input.

Return type numpy.ndarray

get_itrf_xyz_from_mnt_altaz(*mnt_altaz*)
Transform the given MNT AltAz coordinate to ITRF xyz. May be numpy array-like.

Parameters **mnt_altaz** (*numpy array-like*) – Size 2 or shape (2,N).

Returns Shape (3,) if single input, shape (3,N) if array input.

Return type numpy.ndarray

get_location_itrf()
numpy.ndarray: Get the location in ITRF x (m), y (m), z (m) as shape (3,) array.

get_location_lat_lon_height()
tuple of float: Get the location in latitude (deg), longitude (deg), height (m) above ellipsoid.

get_mnt_altaz_from_com_altaz(*com_altaz*)
Transform the given COM AltAz coordinate to MNT AltAz.

Parameters **mnt_altaz** (*numpy array-like*) – Size 2 or shape (2,N).

Returns Shape (2,) if single input, shape (2,N) if array input.

Return type numpy.ndarray

get_mnt_altaz_from_enu_altaz(*enu_altaz*)
Transform the given ENU AltAz coordinate to MNT AltAz.

Parameters **enu_altaz** (*numpy array-like*) – Size 2 or shape (2,N).

Returns Shape (2,) if single input, shape (2,N) if array input.

Return type numpy.ndarray

get_mnt_altaz_from_itrf_xyz(*itrf_xyz*, *position=False*)
Transform the given ITRF xyz coordinates to MNT AltAz.

Parameters

- **itrf_xyz** (*numpy array-like*) – Size 3 or shape (3,N).
- **position** (*bool, optional*) – If True, itrf_xyz is treated as the position in ITRF we want to observe, and the ENU AltAz required to see it from our location is returned.

If False (the default), itrf_xyz is treated as a vector with the desired direction to point in ITRF.

Returns Shape (2,) if single input, shape (2,N) if array input.

Return type numpy.ndarray

set_alignment_enu()

Set the MNT frame equal to the current ENU frame and zero all correction terms.

Note: If the location is changed after set_alignment_enu() the MNT frame will not be updated automatically to coincide with the new ENU.

set_alignment_from_observations(*obs_data*, *alt0=None*, *Cvd=None*, *Cnp=None*)

Use star camera/plate solving observations to set the alignment and mount correction terms.

The observation data must be a list containing eight tuples, each from observing these specific COM AltAz coordinates in order: (40,-135), (60,-135), (60,-45), (40,-45), (40,45), (60,45), (60,135), (40,135). Each tuple in the list must be a 5-tuple with: (Right Ascension (deg), Declination (deg), timestamp (astropy Time), COM Alt (deg), COM Az (deg)) for the measurement. Set Right Ascension and Declination to None if the measurement failed.

This method also solves for the mount correction terms alt0, Cvd, Cnp. If some of these are well known pass the argument to use that value instead of solving for it. You may also pass zeros to disable the corrections.

Parameters

- **obs_data** (*list of tuple*) – See specifics above. The correct sequence is generated from System.do_auto_star_alignment().
- **alt0** (*float, optional*) – Altitude offset (deg). If None (default) it will be solved for.
- **Cvd** (*float, optional*) – Vertical deflecion coefficient. If None (default) it will be solved for.
- **Cnp** (*float, optional*) – Axes nonperpendicularity (deg). If None (default) it will be solved for.

set_location_itrf(*x*, *y*, *z*)

Set the location via ITRF position x (m), y (m), z (m).

set_location_lat_lon(*lat*, *lon*, *height=0*)

Set the location via latitude (deg), longitude (deg), height (m, default 0) above ellipsoid.

data_folder

Get or set the path for data saving. Will create folder if not existing.

Type pathlib.Path

debug_folder

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

Type pathlib.Path

is_aligned

Returns True if telescope has location.

Type bool

is_located
Returns True if telescope has location.

Type bool

class pypogs.**Target**
Target to track and start and end times.

Two types of targets are supported, Astropy *SkyCoord* (any deep space object) and Skyfield *EarthSatellite* (an Earth orbiting satellite).

The target can be set by the property Target.target_object to one of the supported types. It can also be created from the right ascension and declination (for deep sky) or the Two Line Element (TLE) for a satellite. See Target.set_target_from_ra_dec() and Target.set_target_from_tle() respectively.

You may also give a start and end time (e.g. useful for satellite rise and set times) when creating the target or by the method Target.set_start_end_time().

With a target set, get the ITRF_xyz coordinates at your preferred times with Target.get_target_itrf_xyz().

Note:

- If the target is a SkyCoord, the ITRF coordinates will be a unit vector in the direction of the target.
- If the target is an EarthSatellite, the ITRF coordinates will be an absolute position (in metres) from the centre of Earth to the satellite.

clear_start_end_time()
Set both start and end time to None.

get_short_string()
Get a short descriptive string of the target.

Returns str

get_target_itrf_xyz(*times=None*)
Get the ITRF_xyz position vector from the centre of Earth to the EarthSatellite (in metres) or the ITRF_xyz unit vector to the SkyCoord.

Parameters **times** (Astropy *Time*) – Single or array Time for when to calculate the vector.

Returns Shape (3,) if single input, shape (3,N) if array input.

Return type numpy.ndarray

get_tle_raw()
Get the TLE of the target.

Returns tuple of str

Raises AssertionError – If the current target is not a Skyfield *EarthSatellite*.

set_start_end_time(*start_time=None*, *end_time=None*)
Set the start and end times.

Parameters

- **start_time** (astropy *Time*, optional) – The start time to set.
- **end_time** (astropy *Time*, optional) – The end time to set.

set_target_deep_by_name(*name*, *start_time=None*, *end_time=None*)
Use Astropy name lookup for setting a SkyCoord deep sky target.

Parameters

- **name** (*str*) – Name to search for.
- **start_time** (astropy *Time*, optional) – The start time to set.
- **end_time** (astropy *Time*, optional) – The end time to set.

set_target_from_ra_dec(*ra*, *dec*, *start_time=None*, *end_time=None*)
Create an Astropy *SkyCoord* and set as the target.

Parameters

- **ra** (*float*) – Right ascension in decimal degrees.
- **dec** (*float*) – Declination in decimal degrees.
- **start_time** (astropy *Time*, optional) – The start time to set.
- **end_time** (astropy *Time*, optional) – The end time to set.

set_target_from_tle(*tle*, *start_time=None*, *end_time=None*)
Create a Skyfield *EarthSatellite* and set as the target.

Parameters

- **tle** (*tuple or str*) – 2-tuple with the two TLE rows or a string with newline character between lines.
- **start_time** (astropy *Time*, optional) – The start time to set.
- **end_time** (astropy *Time*, optional) – The end time to set.

end_time
Get or set the tracking end time.

Type astropy *Time* or None

has_target
Returns True if a target is set.

Type bool

start_time
Get or set the tracking start time.

Type astropy *Time* or None

target_object
Get or set the target object.

Type Astropy *SkyCoord* or Skyfield *EarthSatellite* or None

4.2 Feedback and tracking algorithms

- ***pypogs.ControlLoopThread*** **is the main control loop for pypyogs. It manages the transition** logic and feedback loop.
- *pypogs.TrackingThread* is the target tracking loop. One is created for each camera to read arriving images. It provides the current error etc. to the pypogs.ControlLoopThread.
- *pypogs.SpotTracker* is attached to each pypogs.TrackingThread and implements the spot detection and tracking.

This is Free and Open-Source Software originally written by Gustav Pettersson at ESA.

License: Copyright 2019 the European Space Agency

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

https://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

class pypogs.**ControlLoopThread**(*parent*, *data_folder=None*, *debug_folder=None*)

Run a control loop for satellite tracking.

This thread holds a reference to the *parent* pypogs.System instance and takes control of all devices. Set up this thread with your desired tracking parameters (such as gains, frequency, feedback maximum rates, etc.) and then call ControlLoopThread.start(). The thread runs with a maximum frequency defined by max_frequency, it defaults to 5 Hz. The target (and start and end times if desired) must be defined in the *parent*.target object.

Upon reaching a transition criteria, the tracking mode will be changed (e.g. open-loop to closed-loop) automatically either to a "better" or "worse" mode. The available modes and their parameters are defined below:

- OL: Open-loop control. The mount angles are read out and the control signal is based on the difference between the mount angles and the target angles. Gains OL_P (proportional, default 1.0), OL_I (integral, default 10 seconds), and rate limit OL_speed_limit (default 3600 arcsec/s) apply.
- CCL: Coarse closed-loop control. The measured position from the coarse camera, returned by *parent*.coarse_track_thread.track_x_y, is used (scaled to degrees) for the control signal. Gains CCL_P (proportional, default .5), CCL_I (integral, default 10 s), and rate limit CCL_speed_limit (default 180 arcsec/s) apply. The transition OL > CCL requires the following:
  - *parent*.coarse_track_thread exists and has a track
  - aforementioned track standard deviation is below CCL_transition_th (default 100 arcsec)
- FCL: Fine closed-loop control. The measured position from the fine camera, returned by *parent*.fine_track_thread.track_x_y, is used (scaled to degrees) for the control signal. Gains FCL_P (proportional, default 1.0), FCL_I (integral, default 10 s), and rate limit FCL_speed_limit (default 180 arcsec/s) apply. The transition CCL > FCL requires the following:
  - *parent*.fine_track_thread exists and has a track
  - aforementioned track standard deviation is below FCL_transition_th (default 30 arcsec)
- CTFSP: Coarse-to-fine spiral acquisition. Only available if CTFSP_enable is set to True. (defaults to False). It is recommended to use this only to automatically find the intercamera alignment, and then disable again for fastest possible acquisition. Enabling CTFSP has several effects:
  - When in CCL, the transition CCL > FCL will not occur as described above. Instead, when in CCL and the *parent*.coarse_track_thread.rms_error falls below CTFSP_transition_th (default 20 arcsec), the transition CCL > CTFSP occurs.
  - When in CTFSP, CCL tracking parameters are used. If CTFSP_zero_integral = True is set (defaults to False) the integral term is set to zero.
  - When in CTFSP, the *parent*.coarse_track_thread.spot_tracker goal is supplemented by the goal offset with a spiral which has arms spaced by CTFSP_spacing (default 100 arcsec), at a rate of CTFSP_speed (default 50 arcsec/s), until a radius CTFSP_max_radius (default 500 arcsec) is reached, upon which the mode is changed back to CCL and the spiral offset is deleted.
  - When in CTFSP, the transition CTFSP > FCL happens *immediately* when *parent*.fine_track_thread exists and has a track.

- When in FCL, if CTFSP_auto_update_CCL_goal = True, the *parent*.coarse_track_thread.spot_tracker goal is updated with the current track position, effectively calibrating the intercamera alignment. This only happens when *parent*.fine_track_thread.rms_error is below CTFSP_auto_update_CCL_goal_th (default 10 arcsec). If CTFSP_disable_after_goal_update is True (the default), CTFSP mode will be disabled automatically after successful alignment.

The availability of each mode can be controlled by setting CCL_enable, FCL_enable, CTFSP_enable to True or False, OL mode may not be disabled.

Note: Integral windup protection is achieved by two means:

1. The integral term is reset if the control signal desired is greater than the speed limit (disable by reset_integral_if_saturated=False).
2. The integral term may grow (in magnitude) by a maximum value of integral_max_add (default 36 arcsec) per loop. (It may shrink by a maximum value of integral_max_subtract, default 360 arcsec).

Tracking data is saved to a .csv file in the data_folder. The filenames are auto-generated with the start time (as an ISO timestamp) and _ControlLoopThread.csv.

Parameters

- **parent** (`pypogs.System`) – The System to control.
- **data_folder** (*pathlib.Path, optional*) – The folder for data saving. If None (the default) the folder *pypogs*/data will be used/created.
- **debug_folder** (*pathlib.Path, optional*) – The folder for debug logging. If None (the default) the folder *pypogs*/debug will be used/created.

start()
Starts the control loop in a background thread. Must have target and be aligned, located, and initialised.

stop()
Stop the thread.

CCL_I
Get or set Coarse Closed-Loop integral time (1/gain) in seconds.

Type float

CCL_P
Get or set Coarse Closed-Loop proportional gain.

Type float

CCL_enable
Get or set if Coarse Closed-Loop is allowed.

Type bool

CCL_speed_limit
Get or set Coarse Closed-Loop speed limit (arcsec per second).

Type float

CCL_transition_th
Get or set the coarse track SD required to transition OL to CCL (arcsec).

Type float

CTFSP_auto_update_CCL_goal
Get or set if CTFSP should update the CCL goal (i.e. save the alignment).

Type bool

CTFSP_auto_update_CCL_goal_th
Get or set the required FCL RMSE to auto update the CCL goal in arcseconds.

Type float

CTFSP_delay
Get or set the delay from starting CTFSP mode until spiraling begins in seconds.

Type float

CTFSP_disable_after_goal_update
Get or set if CTFSP should be automatically disabled when the alignment is finished.

Type bool

CTFSP_disable_integral
Get or set if CTFSP should disable (zero) the integral term.

Type bool

CTFSP_enable
Get or set if Coarse-to-Fine Spiral auto-alignment is allowed. Disabled by default.

It is recommended to only enable this when necessary for alignment, and then disable for faster acquisitions.

Type bool

CTFSP_max_radius
Get or set the maximum (reset) radius for Coarse-to-Fine Spiral auto-alignment in arcseconds.

Type float

CTFSP_ramp
Get or set the CTFSP speed ramp up in seconds.

Type float

CTFSP_spacing
Get or set the spacing between arms in Coarse-to-Fine Spiral auto-alignment in arcseconds.

Type float

CTFSP_speed
Get or set the speed in Coarse-to-Fine Spiral auto-alignment in arcsec per second.

Type float

CTFSP_transition_th
Get or set the CCL RMSE required to start Coarse-to-Fine Spiral auto-alignment in arcseconds.

Type float

FCL_I
Get or set Fine Closed-Loop integral time (1/gain) in seconds.

Type float

FCL_P
Get or set Fine Closed-Loop proportional gain.

Type float

FCL_enable
Get or set if Fine Closed-Loop is allowed.

Type bool

FCL_speed_limit
Get or set Fine Closed-Loop speed limit (arcsec per second).

Type float

FCL_transition_th
Get or set the fine track SD required to transition OL to CCL (arcsec).

Type float

OL_I
Get or set Open-Loop integral time (1/gain) in seconds.

Type float

OL_P
Get or set Open-Loop proportional gain.

Type float

OL_goal_offset_x_y
Get or set the open-loop goal offset (arcseconds).

Type tuple of float

OL_goal_x_y
Get or set the open-loop goal (arcseconds).

Type tuple of float

OL_speed_limit
Get or set Open-Loop speed limit (arcsec per second).

Type float

available_properties
Get the available tracking parameters (e.g. gains).

Type tuple of str

data_folder
Get or set the path for data saving. Will create folder if not existing.

Type pathlib.Path

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

Type pathlib.Path

integral_max_add
Get or set the maximum error (arcseconds) allowed to be added (in magnitude) to the integral.

Type float

integral_max_subtract
Get or set the maximum error (arcseconds) allowed to be subtracted (in magnitude) to the integral.

Type float

integral_min_rate
Get or set the minimum target rate (arcsec per second) required to enable the integral term.

Type float

is_running
Returns True if control thread is running.

Type bool

max_frequency
Get or set the maximum frequency in hertz.

Type float or None

name
Get or set a name for the thread. Default 'ControlLoopThread'.

Type str

reset_integral_if_saturated
Get or set if the integral term should be reset (zeroed) if control output is at the speed limit.

Type bool

state_cache
Get dictionary with last cached state.

Type dict

class pypogs.**TrackingThread**(*camera=None*, *spot_tracker=None*, *name=None*, *image_folder=None*, *img_save_frequency=1.0*, *data_folder=None*, *debug_folder=None*)

Run a thread to track a point on a camera.

This thread should be given a pypogs.Camera object and a pypogs.SpotTracker object. An empty SpotTracker will be created and attached if not given as input. The camera acquisition properties are set directly in the Camera and the spot detection parameters are set directly in the SpotTracker. They may be accessed (and set) by TrackingThread.camera and TrackingThread.spot_tracker respectively. If an image_folder is given to the TrackingThread the images read are saved (as .tiff) to the path, the maximum saving frequency can be limited by setting img_save_frequency (default 1 Hz) to avoid saving massive amounts of data.

The TrackingThread will update on every image event received from the Camera, so the frequency is controlled by the Camera.frame_rate. If the thread is already busy, the image will be ignored and a frame drop warning logged.

After setting up and starting the thread, read TrackingThread.track_alt_az to get the last measured position (error from the goal) of the satellite being tracked. See note below for coordinate definition. The tracker keeps an estimate of the mean and standard deviation of position, signal sum, and signal area which may be used to keep the track. Please read the SpotTracker (in this module) documentation carefully to set up the spot detection and tracking parameters.

The TrackingThread can also pass feed-forward information to the SpotTracker to compensate for the movement of the telescope on the position of the target. The SpotTracker mean position will be moved at a speed of TrackingThread.feedforward_rate (tuple of floats; arcsec per second) but only if the step change is greater than TrackingThread.feedforward_threshold (default 10 arcsec). It is also possible to move the SpotTracker goal offset by a speed defined by TrackingThread.goal_offset_rate (arcsec per second) to follow a moving goal offset (i.e. during spiralling).

Note: The TrackingThread has two distinct coordinate systems. The alt_az system measures positions relative to the defined goal and is (1) derotated with the Camera.rotation parameter and (2) switched (i.e. x and y becomes az and alt respectively) in order to coincide with the control loop's alt_az system compared to the SpotTracker x and y positions. However, many parameters (e.g. the goal and anything appended by _absolute) are in the same x_y coordinates of the SpotTracker *without any derotation*. All distance measurements are scaled by the Camera.plate_scale such that the outputs are directly in arcseconds.

Parameters

- **camera** (pypogs.Camera, *optional*) – The Camera object to read from.
- **spot_tracker** (pypogs.SpotTracker, *optional*) – The SpotTracker used to detect and track from the images. If this is not supplied an empty SpotTracker will be created.
- **name** (*str, optional*) – A name for the TrackingThread.
- **image_folder** (*pathlib.Path, optional*) – The folder for data saving. If None (the default) images will not be saved.
- **img_save_frequency** (*float, optional*) – Maximum frequency for saving images. If None every image will be saved (this may use a lot of disk space and computer resources).
- **data_folder** (*pathlib.Path, optional*) – The folder for data saving. If None (the default) the folder *pypogs*/data will be used/created.
- **debug_folder** (*pathlib.Path, optional*) – The folder for debug logging. If None (the default) the folder *pypogs*/debug will be used/created.

Example

```
# Create a camera instance (see pypogs.camera.Camera)
cam = pypogs.Camera(model='ptgrey', identity='18285284', name='CoarseCam')
# Create a TrackingThread instance
tt = pypogs.TrackingThread(camera=cam, name='CoarseTrackThread')
# Set up tracking parameters (see SpotTracker in this module for details)
tt.spot_tracker.max_search_radius = 500
tt.spot_tracker.min_search_radius = 100
tt.spot_tracker.position_sigma = 5
# (Optional) set up a directory for image saving at .5 Hz
tt.image_folder = Path('./tracking_images')
tt.img_save_frequency = .5
# Start the tracker
tt.start()
# Wait for a while
time.sleep(2)
# Read the position
print(tt.track_alt_az)
# Stop the tracker
tt.stop()
# Deinitialise the camera
cam.deinitialize()
```

start()
Starts the tracking in a background thread. Must have Camera and SpotTracker.

stop()
Stop the thread.

auto_acquire_track
Get or set if tracks should be automatically acquired.

Type bool

camera
Get or set the Camera object for the tracking thread.

Type *pypogs.Camera*

data_folder
Get or set the path for data saving. Will create folder if not existing.

Type pathlib.Path

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

Type pathlib.Path

feedforward_rate
Get or set the speed (arcsec/sec) with which the SpotTracker mean position should be moved.

Type “tuple of float

feedforward_threshold
Get or set the minimum feedforward step (arcsec).

Type float

frequency
Get or set the target loop frequency in Hz. Must be greater than zero.

Type float

frequency_actual
Get the actual loop frequency (for the last update) in Hz.

Type float

goal_offset_rate
Get or set the speed (arcsec/sec) with which the SpotTracker goal offset position should be moved.

Type “tuple of float

goal_offset_x_y
Set temporary (cleared on lost track) offset from the defined goal.

Type tuple of float

goal_x_y
Get or set the goal (in absolute x and y relative to image centre) the track is measured against.

Defaults to (0,0)

Type tuple of float

has_track
Returns true if there is a track. Returns None if not running.

Type bool

image_folder
Get or set the path for saving images. Will create folder if not existing.

The saving frequency can be limited by setting TrackingThread.img_save_frequency

Type pathlib.Path

img_save_frequency
Get or set the maximum frequency to save images. If None (the default) all images will be saved.

Type float

is_running
Returns True if the thread is running.

Type bool

mean_alt_az
Get the mean (smoothed) alt and az position relative to the goal (derotated). None if no track.

Smoothing determined by SpotTracker.smoothing_parameter

Type tuple of float

mean_x_y_absolute
Get the absolute mean x and y position (relative to image centre). None if no track.

Smoothing determined by SpotTracker.smoothing_parameter

Type tuple of float

name
Get or set a name for the thread

Type str

pos_search_rad
Get or set the current search radius.

Type float

pos_search_x_y
Get or set the current search position. Will also set search radius to max_search_radius.

Type tuple of float

rms_error
Get the total smoothed RMS error relative to the goal. None if no track.

Smoothing determined by SpotTracker.rmse_smoothing_parameter

Type float

spot_tracker
Get or set the SpotTracker object.

Type *SpotTracker*

track_alt_az
Get the latest alt and az position relative to the goal (derotated). None if no track or latest update failed.

Type tuple of float

track_sd
Get the total standard deviation of the track. None if no track.

Type float

track_x_y_absolute
Get the latest absolute x and y position (relative to image centre). None if no track or latest update failed.

Type tuple of float

class pypogs.**SpotTracker**(*name=None*, *data_folder=None*, *debug_folder=None*)
Class for detecting and tracking spots (i.e. satellites) from a stream or series of images.

Typically this class is not used directly, but attached to a TrackingThread (in this module).

The tracker works by keeping an estimate of the mean and standard deviation (SD) of position, signal sum, and signal area and searching for spots which fall within *sigma* SDs away from the mean. By default the position and signal sum are used with 5-sigma limits to keep the track. To change the limits see e.g SpotTracker.sigma_position; setting any of these to None will disable using that parameter to discern tracks. If more than one spot falls within the track limits the spot closest in position to the mean position will be used. *It is mandatory to set a reasonable maximum search radius (default 1000 arcsec).* This will also be used to initialise the position SD estimate. *It is recommended to set a reasonable minimum search radius (default None)* to at least one pixel's width. It is also possible to set maximum and minimum limits on the signal sum and area SD estimates, otherwise it will be limited to between zero and the mean value of the respective signal.

You may define a goal position against which the output should be measured. By default this is (0,0) which is the centre of the image. The units for all positions will be in the units you provide via plate_scale (default 1 arcsec/pixel) to SpotTracker.update_from_image(). The goal and properties ending in _absolute are always measured relative to image centre.

The estimators use exponentially weighted moving mean and variance. SpotTracker.smoothing_parameter (default 10) defines how samples are weighted and roughly corresponds to the number of samples to average.

If SpotTracker.success_to_start_track (default 3) spots are found in a row a new track will be established. If SpotTracker.fails_to_drop_track (default 10) failures occur in a row the track is dropped. If an update fails during tracking the tracking parameters will be penalised by SpotTracker.failure_sd_penalty (default 25%) to account for drift in the mean.

A performance metric, the root-mean-squared error (RMSE), is provided which measures the position error relative to the goal (instead of the mean, as the SD does). By default this uses SpotTracker.smoothing_parameter with exponential averaging, but SpotTracker.rmse_smoothing_parameter may be defined to control this individually.

You may elect to use your own spot detection system and directly update the tracker with position with SpotTracker.update_from_observation(). If this is done, the tracker will accept the updated position without checking if it falls within the defined tracking range.

Note: The SpotTracker uses position information in (x,y) coordinates where (0,0) is the centre of the image, x increases to the right and y increases upwards. When images are used to update the tracker (via SpotTracker.update_from_image()) a plate scale should be supplied such that the output is in useful units, i.e. arcseconds.

Parameters

- **name** (*str, optional*) – A name for the SpotTracker.
- **data_folder** (*pathlib.Path, optional*) – The folder for data saving. If None (the default) the folder *pypogs*/data will be used/created. *TODO! Currently does nothing*
- **debug_folder** (*pathlib.Path, optional*) – The folder for debug logging. If None (the default) the folder *pypogs*/debug will be used/created.

change_mean_relative(*dx*, *dy*)

Add an offset to SpotTracker.mean_x_y positions. Useful for feed-forward of control signals.

Parameters

- **dx** (*float*) – Amount to add in x.
- **dy** (*float*) – Amount to add in y.

penalize_track(*percentage=25*)

Scale the search radius, sum SD, and areaSD used for tracking by the given percentage.

update_from_image (*img*, *plate_scale=1*)
Update the SpotTracker from a new image.

Will use tetra3.extract_star_positions() to find spots in the image with the extraction parameters which have been set in the SpotTracker. A spot falling within the current tracking range will be searched for and, if found, the tracker will be updated. If not found, the estimator SD will be penalised by SpotTracker.failure_penalty (default 10%).

Parameters

- **img** (*numpy.ndarray*) – The image to update with.
- **plate_scale** (*float, optional*) – Image plate scale (typically arcseconds per pixel).
- **to 1** (*Defaults*) –

Returns

True if has_track *and* the provided image was successfully used to update the tracker.

Return type bool

update_from_observation (*x*, *y*, *summ*, *area*)
Update with new observed data for the track. Will not check any tracking validity (see update_from_image()).

If any of x, y, summ, area are None it will be interpreted as a failed track cycle and nothing updated.

active_crop_enable
Get or set if active cropping should be used.

If enabled and the SpotTracker has a track, the incoming image will be automatically cropped to only contain the search region (to speed up computation). This may have a slight effect on background subtraction and SD estimators.

Type bool

active_crop_padding
Get or set the number of pixels to pad on each side of the search region for active cropping.

Type int

area_max_sd
Get or set the maximum (and initialisation) for signal area standard deviation estimator.

Type float

area_min_sd
Get or set the minimum for signal area standard deviation estimator.

Type float

area_sigma
Get or set the signal area standard deviation threshold.

Set None to disable tracking in spot area.

Type float or None

auto_acquire_track
Get or set if tracks should be automatically acquired.

Type bool

available_properties
Get the available tracking parameters (e.g. gains).

Type tuple of str

bg_subtract_mode
Get or set the background subtraction mode.

Allowed values:

- global_median: Subtract the global median value.
- local_median: Subtract the median in a SpotTracker.filtsize wide window around each pixel.
- global_mean: Subtract the global mean value.
- local_mean: Subtract the median in a SpotTracker.filtsize wide window around each pixel.

Type str

binary_open
Get or set if binary opening (with 1-connected structure element) should be used to clean the thresholded image.

Type bool

centroid_window
Get or set the second round centroiding window size.

If None, the centroid will be calculated from the spot region in the thresholded image. If set, a square window of the given size will be used to recalculate the centroid.

Type int or None

crop
Get or set the cropping of incoming images. If None no cropping is applied.

Can be set in two ways:

- 2-tuple: The image is cropped to this (width, height) number of pixels in the centre.
- 4-tuple: The image is cropped to this (width, height, offset_right, offset_down)

Type tuple of int or or None

data_folder
Get or set the path for data saving. Will create folder if not existing.

Type pathlib.Path

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

Type pathlib.Path

downsample
Get or set downsampling (binning) of incoming images.

Note: If a received image's shape can not be divided by this factor an error will be thrown!

Type int or None

fails_to_drop
Get or set the number of successive failed extractions after which a track is dropped.

Type int

failure_sd_penalty
Get or set the penalty in percent to give to the standard deviation on a failed extraction.

Type float or None

filtsize
Get or set the filter size (width and height) used for local background subtraction and SD estimate.

Type int

goal_offset_x_y
Get or set the tracking offset from the goal.

Type tuple of float

goal_x_y
Get or set the tracking goal in absolute x and y.

Type tuple of float

has_track
Returns True if there currently is a track.

Type bool

image_sigma_th
Get or set the number of standard deviations of image noise used for image thresholding.

If SpotTracker.image_th is set (not None), this value will be ignored.

Type float

image_th
Get or set the image thresholding value.

If this value is set (not None), the image standard deviation estimator and image_sigma_th will be ignored and the provided value used directly.

Type int or None

max_search_radius
Get or set the maximum search radius. Also sets initial standard deviation. Must be set; default 1000.

Type float

mean_area
Get the mean (smoothed) signal area. None if no track.

Smoothing determined by SpotTracker.smoothing_parameter

Type float

mean_sum
Get the mean (smoothed) signal sum. None if no track.

Smoothing determined by SpotTracker.smoothing_parameter

Type float

mean_x_y
Get the mean (smoothed) x and y position relative to the goal. None if no track.

Smoothing determined by SpotTracker.smoothing_parameter

Type tuple of float

mean_x_y_absolute

Get the absolute mean x and y position (relative to image centre). None if no track.

Smoothing determined by SpotTracker.smoothing_parameter

Type tuple of float

min_search_radius

Get or set the minimum search radius. Also sets initial standard deviation.

Type float or None

name

Get or set a name for the SpotTracker

Type str

pos_search_rad

Get or set the current search radius.

Type float

pos_search_x_y

Get or set the x y position (absolute) where we search for track.

Will clear tracker if set.

Type tuple of float or None

position_sigma

Get or set the number of standard deviations the search radius should be. Set None to disable tracking in position.

Type float or None

rms_error

Get the total smoothed RMS error *relative to the goal*. None if no track.

Smoothing determined by SpotTracker.rmse_smoothing_parameter

Type float

rmse_smoothing_parameter

Get or set the smoothing parameter. It roughly corresponds to the number of samples to average.

- This parameter is used for estimating the RMS Error relative to the goal position.
- If not set, SpotTracker.smoothing_parameter is used
- Exponential smoothing is used. rmse_smoothing_parameter is the *inverse* of 'alpha'. Each smoothed value s is defined from the measurements x by:

  `rmse[n]**2 = (1-alpha) * (rmse[n-1]**2 + alpha*((x[n]-goal_x)**2 + (y[n]-goal_y)**2))`

  `rmse[0]**2 = (x[0]-goal_x)**2 + (y[0]-goal_y)**2`

Type float

sd_area

Get the signal area standard deviation. None if no track.

Smoothing determined by SpotTracker.smoothing_parameter

Type float

sd_sum

Get the signal sum standard deviation. None if no track.

Smoothing determined by SpotTracker.smoothing_parameter

Type float

sigma_mode

Get or set the mode used to calculate the image sigma (standard deviation) after background subtraction.

This value multiplied with SpotTracker.image_sigma_th will be used to threshold the image. If SpotTracker.image_th is not None (default is None), that will be used as the threshold instead, and sigma_mode and image_sigma_th have no effect.

Allowed values:

- global_median_abs: Use the median of the absolute value of the image and multiply by 1.48.
- local_median_abs: As above, but in a SpotTracker.filtsize wide window around each pixel.
- global_root_square: Use the root mean square of the image.
- local_root_square: As above, but in a SpotTracker.filtsize wide window around each pixel.

Type str

smoothing_parameter

Get or set the smoothing parameter. It roughly corresponds to the number of samples to average.

- This parameter is used for estimating the mean and standard deviation of the position, sum, and area.
- The total RMS Error smoothing can be set independently by SpotTracker.rmse_smoothing_parameter. If not, this value is used there as well.
- Exponential smoothing is used. smoothing_parameter is the *inverse* of 'alpha'. Each smoothed value s is defined from the measurements x by:

  `mean[n] = alpha*x[n] + (1-alpha)*mean[n-1]` `mean[0] = x[0]`

  `sd[n]**2 = (1-alpha)*(sd[n-1]**2 + alpha*(x[n]-mean[n-1])**2)`

The SD estimator is initialised to given maximum values max_search_radius, sum_max_sd, area_max_sd if available, otherwise it is set to the magnitude of the first given value.

Type int or float

spot_max_area

Get or set the maximum number of pixels in the thresholded image which constitute a star.

Type int or None

spot_max_axis_ratio

Get or set the maximum ratio of major to minor axes for a region to constitute a star.

The major and minor axes are calculated from second moments.

Type float or None

spot_max_sum

Get or set the maximum sum of values in a region to constitute a star.

Type float or None

spot_min_area
Get or set the minimum number of pixels in the thresholded image which constitute a star.

Type int or None

spot_min_sum
Get or set the minimum sum of values in a region to constitute a star.

Type float or None

successes_to_track
Get or set the number of successive successful extractions to start a track.

Type int

sum_max_sd
Get or set the maximum (and initialisation) for signal sum standard deviation estimator.

Type float or None

sum_min_sd
Get or set the minimum for signal sum standard deviation estimator.

Type float or None

sum_sigma
Get or set the sigmal sum standard deviation threshold. Set None to disable tracking in sum.

Type float or None

track_area
Get the latest signal area. None if no track or latest update failed.

Type float

track_sd
Get the total standard deviation of the track. None if no track.

Type float

track_sum
Get the latest signal sum. None if no track or latest update failed.

Type float

track_x_y
Get the latest x and y position relative to the goal. None if no track or latest update failed.

Type tuple of float

track_x_y_absolute
Get the latest absolute x and y position (relative to image centre). None if no track or latest update failed.

Type tuple of float

4.3 Camera hardware interface

Current hardware support:

- 'ptgrey' for FLIR (formerly Point Grey) machine vision cameras. Requires Spinnaker API and PySpin, see the installation instructions. Tested with Blackfly S USB3 model BFS-U3-31S4M.

This is Free and Open-Source Software originally written by Gustav Pettersson at ESA.

License: Copyright 2019 the European Space Agency

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

https://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

class pypogs.**Camera**(*model=None*, *identity=None*, *name=None*, *auto_init=True*, *debug_folder=None*)

Control acquisition and receive images from a camera.

To initialise a Camera a *model* (determines hardware interface) and *identity* (identifying the specific device) must be given. If both are given to the constructor the Camera will be initialised immediately (unless auto_init=False is passed). Manually initialise with a call to Camera.initialize(); release hardware with a call to Camera.deinitialize().

After the Camera is initialised, acquisition properties (e.g. exposure_time and frame_rate) may be set and images received. The Camera also supports event-driven acquisition, see Camera.add_event_callback(), where new images are automatically passed on to the desired functions.

Parameters

- **model** (*str, optional*) – The model used to determine the correct hardware API. Supported: 'ptgrey' for PointGrey/FLIR Machine Vision cameras (using Spinnaker and PySpin).
- **identity** (*str, optional*) – String identifying the device. For model *ptgrey* this is 'serial number' *as a string*.
- **name** (*str, optional*) – Name for the device.
- **auto_init** (*bool, optional*) – If both model and identity are given when creating the Camera and auto_init is True (the default), Camera.initialize() will be called after creation.
- **debug_folder** (*pathlib.Path, optional*) – The folder for debug logging. If None (the default) the folder *pypogs*/debug will be used/created.

Example

```
# Create instance and set parameters (will auto initialise)
cam = pypogs.Camera(model='ptgrey', identity='18285284', name='CoarseCam')
cam.gain = 0 #decibel
cam.exposure_time = 100 #milliseconds
cam.frame_rate_auto = True
# Start acquisition
cam.start()
# Wait for a while
time.sleep(2)
# Read the latest image
img = cam.get_latest_image()
# Stop the acquisition
cam.stop()
# Release the hardware
cam.deinitialize()
```

add_event_callback(*method*)
Add a method to be called when a new image shows up.

The method should have the signature (image, timestamp, *args, **kwargs) where:

- image (numpy.ndarray): The image data as a 2D numpy array.
- timestamp (datetime.datetime): UTC timestamp when the image event occurred (i.e. when the capture finished).
- *args, **kwargs should be allowed for forward compatibility.

The callback should *not* be used for computations, make sure the method returns as fast as possible.

Parameters **method** – The method to be called, with signature (image, timestamp, *args, **kwargs).

deinitialize()
De-initialise the device and release hardware resources. Will stop the acquisition if it is running.

get_latest_image()
Get latest image in the cache immediately. Camera must be running.

Returns 2d array with image data.

Return type numpy.ndarray

get_new_image(*timeout=10*)
Get an image guaranteed to be started *after* calling this method. Camera does not have to be running.

Parameters **timeout** (`float`) – Maximum time (seconds) to wait for the image before raising TimeoutError.

Returns 2d array with image data.

Return type numpy.ndarray

get_next_image(*timeout=10*)
Get the next image to be completed. Camera does not have to be running.

Parameters **timeout** (`float`) – Maximum time (seconds) to wait for the image before raising TimeoutError.

Returns 2d array with image data.

Return type numpy.ndarray

initialize()
Initialise (make ready to start) the device. The model and identity must be defined.

remove_event_callback(*method*)
Remove method from event callbacks.

start()
Start the acquisition. Device must be initialised.

stop()
Stop the acquisition.

available_properties
Get all the available properties (settings) supported by this device.

Type tuple of str

binning
Number of pixels to bin in each dimension (e.g. 2 gives 2x2 binning).

ptgrey cameras bin by summing, *zwoasi* cameras bin by averaging.

Setting will stop and restart camera if running. Will scale size_readout to show the same sensor area.

Type int

color_bin
Get or set if colour binning is active. Defaults to True for colour cameras. Is always False for mono cameras.

When colour binning is True, each 2x2 Bayer group on the image sensor will form one RGB pixel in the output. If set to False, interpolation will be used to create an RGB image at full resolution. Interpolation may slow down the image processing significantly.

Type bool

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

Type pathlib.Path

exposure_time
Get or set the camera expsure time in ms. Will set auto exposure time to False.

Type float

exposure_time_auto
Get or set automatic exposure time. If True the exposure time will be continuously updated.

Type bool

exposure_time_limit
Get the minimum and maximum expsure time in ms supported.

Type tuple of float

flip_x
Get or set if the image X-axis should be flipped. Default is False.

Type bool

flip_y
Get or set if the image Y-axis should be flipped. Default is False.

Type bool

frame_rate
Get or set the camera frame rate in Hz. Will set auto frame rate to False.

Type float

frame_rate_actual
Get the actual image frame rate in Hz. Returns None if not running.

Type float

frame_rate_auto
Get or set automatic frame rate. If True camera will run as fast as possible.

Type bool

frame_rate_limit
Get the minimum and maximum frame rate in Hz supported.

Type tuple of float

gain
Get or set the camera gain in the camera's native unit.

Type float

gain_auto
Get or set automatic gain. If True the gain will be continuously updated.

Type bool

gain_limit
Get the minimum and maximum gain supported in the camera's native unit.

Type tuple of float

identity
Get or set the device and/or input. Model must be defined first.

- For model *ptgrey* this is the serial number *as a string*
- For model *zwoasi* this is the index (starting at zero)
- Must set before initialising the device and may not be changed for an initialised device.

Type str

is_init
True if the device is initialised (and therefore ready to start).

Type bool

is_running
True if device is currently acquiring data.

Type bool

model
Get or set the device model.

Supported:

- 'ptgrey' for FLIR/Point Grey cameras (using Spinnaker/PySpin SDKs).

- This will determine which hardware API that is used.
- Must set before initialising the device and may not be changed for an initialised device.

Type str

name
Get or set the name.

Type str

plate_scale
Get or set the plate scale of the Camera in arcsec per pixel.

This will not affect anything in this class but is used elsewhere. Set this to the physical pixel plate scale *before* any binning. When getting the plate scale it will be scaled by the binning factor.

Type float

rotate_90
Get or set how many times the image should be rotated by 90 degrees. Applied *after* flip_x and flip_y.

Type int

rotation
Get or set the camera rotation relative to the horizon in degrees.

This does not affect the received images, but is used elsewhere. Use rotate_90 first to keep this rotation small.

Type float

size_max
Get the maximum allowed readout size (width, height) in pixels.

Type tuple of int

size_readout
Get or set the number of pixels read out (width, height). Will automatically center.

This applies after binning, i.e. this is the size the output image will be.

For model *zwoasi* the set size will be rounded down to the nearest multiple of 8 in width and 2 in height.

Setting will stop and restart camera if running.

Type tuple of int

4.4 Mount hardware interfaces

Current hardware support:

- 'celestron' for Celestron, Orion and SkyWatcher telescopes (using NexStar serial protocol). No additional packages required. Tested with Celestron model CPC800.

This is Free and Open-Source Software originally written by Gustav Pettersson at ESA.

License: Copyright 2019 the European Space Agency

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

https://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

class pypogs.**Mount** (*model=None*, *identity=None*, *name=None*, *auto_init=True*, *debug_folder=None*)
Control a telescope gimbal mount.

To initialise a Mount a *model* (determines hardware interface) and *identity* (identifying the specific device) must be given. If both are given to the constructor the Mount will be initialised immediately (unless auto_init=False is passed). Manually initialise with a call to Mount.initialize(); release hardware with a call to Mount.deinitialize().

After the Mount is initialised, the gimbal angles and rates may be read and commanded. Several properties (e.g maximum angles and rates) may be set.

Parameters

- **model** (*str, optional*) – The model used to determine the the hardware control interface. Supported: 'celestron' for Celestron NexStar and Orion/SkyWatcher SynScan (all the same) hand controller communication over serial.
- **identity** (*str or int, optional*) – String or int identifying the device. For model *celestron* this can either be a string with the serial port (e.g. 'COM3' on Windows or '/dev/ttyUSB0' on Linux) or an int with the index in the list of available ports to use (e.g. identity=0 i if only one serial device is connected.)
- **name** (*str, optional*) – Name for the device.
- **auto_init** (*bool, optional*) – If both model and identity are given when creating the Mount and auto_init is True (the default), Mount.initialize() will be called after creation.
- **debug_folder** (*pathlib.Path, optional*) – The folder for debug logging. If None (the default) the folder *pypogs*/logs will be used/created.

Example

```
# Create instance (will auto initialise)
mount = pypogs.Mount(model='celestron', identity='COM3', name='CPC800')
# Move to position
mount.move_to_alt_az(30, 10) #degrees; by default blocks until finished
# Set gimbal rates
mount.set_rate_alt_az(0, -1.5) #degrees per second
# Wait for a while
time.sleep(2)
# Stop moving
mount.stop()
# Disconnect from the mount
mount.deinitialize()
```

Note: The Mount class allows two modes of control for moving to positions. The default is rate_control=True, where this class will continuously send rate commands until the desired position is reached. It is possible to use the internal motion controller in the mount by passing rate_control=False. However, it is slow and implements backlash compensation. In our testing the accuracy difference is negligible so the default is recommended.

static degrees_to_0_360(*number*)
float: Convert angle (degrees) to range [0, 360).

static degrees_to_n180_180(*number*)
float: Convert angle (degrees) to range (-180, 180]

deinitialize()
De-initialise the device and release hardware (serial port). Will stop the mount if it is moving.

get_alt_az()
Get the current alt and azi angles of the mount.

Returns the (altitude, azimuth) angles of the mount in degrees (-180, 180].

Return type tuple of float

initialize()
Initialise (make ready to start) the device. The model and identity must be defined.

static list_available_ports()
List the available serial port names and descriptions.

Returns (device, description) for each available serial port (see serial.tools.list_ports).

Return type list of tuple

move_home(*block=True*, *rate_control=True*)
Move to the position defined by Mount.home_alt_az.

Parameters

- **block** (*bool, optional*) – If True (the default) the call to this method will block until the move is finished.
- **rate_control** (*bool, optional*) – If True (the default) the rate of the mount will be controlled until position is reached, if False the position command will be sent to the mount for execution.

move_to_alt_az(*alt*, *azi*, *block=True*, *rate_control=True*)
Move the mount to the given position. Must be initialised.

Parameters

- **alt** (*float*) – Altitude angle (degrees).
- **azi** (*float*) – Azimuth angle (degrees).
- **block** (*bool, optional*) – If True (the default) the call to this method will block until the move is finished.
- **rate_control** (*bool, optional*) – If True (the default) the rate of the mount will be controlled until position is reached, if False the position command will be sent to the mount for execution.

set_rate_alt_az(*alt*, *azi*)
Set the mount slew rate. Must be initialised.

Parameters

- **alt** (*float*) – Altitude rate (degrees per second).
- **azi** (*float*) – Azimuth rate (degrees per second).

stop()
Stop moving.

wait_for_move_to(*timeout=120*)
Wait for mount to finish move.

Parameters timeout (*int, optional*) – Maximum time (seconds) to wait before raising TimeoutError. Default 120.

alt_limit
Get or set the altitude limits (degrees) where the mount can safely move. May be set to None. Default (-5, 95). Not enforced when slewing (set_rate) the mount.

Type tuple of float

available_properties
Get all the available properties (settings) supported by this device.

Type tuple of str

azi_limit
Get or set the azimuth limits (degrees) where the mount can safely move. May be set to None. Default (None, None). Not enforced when slewing (set_rate) the mount.

Type tuple of float

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

Type pathlib.Path

home_alt_az
Get or set the home position (altitude, azimuth) in degrees. Default (0, 0)

Type tuple of float

identity
Get or set the device and/or input. Model must be defined first.

- For model *celestron* or *iptron azmp* this can either be a string with the serial port (e.g. 'COM3' on Windows or '/dev/ttyUSB0' on Linux) or an int with the index in the list of available ports to use (e.g. identity=0 i if only one serial device is connected.)
- Must set before initialising the device and may not be changed for an initialised device.

Raises `AssertionError` – if unable to connect to and verify identity of the mount.

Type str

is_init
True if the device is initialised (and therefore ready to control).

Type bool

is_moving
Returns True if the mount is currently moving.

max_rate
Get or set the max slew rate (degrees per second) for the axes (altitude, azimith). Default (4.0, 4.0).

If a scalar is set, both axes' rates will be set to this value.

Type tuple of float

model
Get or set the device model.

Supported:

- 'celestron' for Celestron NexStar and Orion/SkyWatcher SynScan hand controllers over serial.

- This will determine which hardware interface is used.
- Must set before initialising the device and may not be changed for an initialised device.

Type str

name
Get or set the name.

Type str

state_cache
Get cache with the current state of the Mount. Updates on calls to get_alt_az() and set_rate_alt_az().

Keys: azi: float, alt: float, azi_rate: float, alt_rate: float

Type dict

zero_altitude
Get or set the zero altitude angle (degrees). Default 0.

Normally the mount is initialised with the telescope level. In this case zero_altitude is 0. However, if the mount is e.g. initialised with the telescope pointing straight up, zero_altitude must be set to +90.

Type float

4.5 Receiver hardware interfaces

Current hardware support:

- 'ni_daq' for National Instruments DAQ data acquisition cards. Requires NI-DAQmx API and nidaqmx, see the installation instructions. Tested with NI DAQ model USB-6211.

This is Free and Open-Source Software originally written by Gustav Pettersson at ESA.

License: Copyright 2019 the European Space Agency

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

https://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

class pypogs.**Receiver**(*model=None*, *identity=None*, *name=None*, *auto_init=True*, *data_folder=None*, *debug_folder=None*)
Control acquisition and read received power from a photodetector.

To initialise a Receiver a *model* (determines hardware interface) and *identity* (identifying the specific device) must be given. If both are given to the constructor the Receiver will be initialised immediately (unless auto_init=False is passed). Manually initialise with a call to Receiver.initialize(); release hardware with a call to Receiver.deinitialize().

The raw data can be saved to a file by specifying data_folder (filenames are auto-generated). While the acquisition is running the instantaneous (last measurement) and (exponentially) smoothed power can be read.

Parameters

- **model** (*str, optional*) – The model used to determine the correct hardware API. Supported: 'ni_daq' for National Instruments DAQ cards (tested on USB-6211).
- **identity** (*str, optional*) – String identifying the device and input. For *ni_daq* this is 'device/input' eg. 'Dev1/ai1' for device 'Dev1' and analog input 1; only differential input is supported for *ni_daq*.
- **name** (*str, optional*) – Name for the device.
- **auto_init** (*bool, optional*) – If both model and identity are given when creating the Receiver and auto_init is True (the default), Receiver.initialize() will be called after creation.
- **data_folder** (*pathlib.Path, optional*) – The folder for data saving. If None (the default) no data will be saved.
- **debug_folder** (*pathlib.Path, optional*) – The folder for debug logging. If None (the default) the folder *pypogs*/debug will be used/created.

Example

```
# Create instance and set parameters (will auto initialise)
rec = pypogs.Receiver(model='ni_daq', identity='Dev1/ai1', name='PhotoDiode')
rec.sample_rate = 1000 #Samples per second
rec.smoothing_parameter = 100 #number of samples to smooth over
rec.measurement_range = (-10, 10) #Volts for ni_daq
# Add a save path (filenames are auto-generated)
rec.data_folder = pathlib.Path('./datafolder')
# Start acquisition
rec.start()
# Wait for a while
time.sleep(2)
# Read the smooth and instantaneous powers
print('Smoothed power is: ' + str(rec.smooth_power))
print('Instant power is: ' + str(rec.instant_power))
# Stop the acquisition
rec.stop()
```

deinitialize()
De-initialise the device. Will stop the acquisition if it is running.

initialize()
Initialise (make ready to start) the device. The model and identity must be defined.

start()
Start the acquisition. Device must be initialised. Data will only be saved if data_folder is set.

stop()
Stop the acquisition. Will ensure all data in the buffer is read before stopping.

available_properties
Get all the available properties (settings) supported by this device.

Type tuple of str

data_folder
Get or set the path for data saving. Will create folder if not existing.

Type pathlib.Path

identity
Get or set the device and/or input. Model must be defined first.

- For model *ni_daq* this is 'device/input' eg. 'Dev1/ai1' for device 'Dev1' and analog input 1. Only differential input is supported for NI DAQ.
- Must set before initialising the device and may not be changed for an initialised device.

Type str

instant_power
Get the latest raw measurement.

Type float

is_init
True if the device is initialised (and therefore ready to start).

Type bool

is_running
True if device is currently acquiring data.

Type bool

measurement_range
Get or set the measurement range (lower_limit, upper_limit).

- If given as a scalar the range will be set to +- the supplied value.

Type tuple, int or float

model
Get or set the device model.

Supported:

- 'ni_daq' for National Instruments DAQ devices (e.g. USB-6211).

- This will determine which hardware API that is used.
- Must set before initialising the device and may not be changed for an initialised device.

Type str

name
Get or set the name.

Type str

sample_rate
Get or set the sample rate (in Hz) of the device. Must initialise the device first.

Type int or float

smooth_power
Get the current smoothed measurement (see smoothing_parameter).

Type float

smoothing_parameter
Get or set the smoothing parameter. It roughly corresponds to the number of samples to average.

- Exponential smoothing is used. smoothing_parameter is the *inverse* of 'alpha'. Each smoothed value s is defined from the measurements x by:

```
s[n] = alpha*x[n] + (1-alpha)*s[n-1]; s[0] = x[0]
```

Type int or float

Python Module Index

p

pypogs.hardware.hardware_camera, 44
pypogs.hardware.hardware_mount, 49
pypogs.hardware.hardware_receiver, 53
pypogs.system, 17
pypogs.tracking, 29

Index

A

active_crop_enable (*pypogs.SpotTracker attribute*), 39
active_crop_padding (*pypogs.SpotTracker attribute*), 39
add_coarse_camera() (*pypogs.System method*), 19
add_coarse_camera_from_star() (*pypogs.System method*), 20
add_event_callback() (*pypogs.Camera method*), 45
add_fine_camera() (*pypogs.System method*), 20
add_mount() (*pypogs.System method*), 20
add_receiver() (*pypogs.System method*), 20
add_star_camera() (*pypogs.System method*), 21
add_star_camera_from_coarse() (*pypogs.System method*), 21
Alignment (*class in pypogs*), 24
alignment (*pypogs.System attribute*), 22
alt_limit (*pypogs.Mount attribute*), 51
area_max_sd (*pypogs.SpotTracker attribute*), 39
area_min_sd (*pypogs.SpotTracker attribute*), 39
area_sigma (*pypogs.SpotTracker attribute*), 39
auto_acquire_track (*pypogs.SpotTracker attribute*), 39
auto_acquire_track (*pypogs.TrackingThread attribute*), 35
available_properties (*pypogs.Camera attribute*), 46
available_properties (*pypogs.ControlLoopThread attribute*), 33
available_properties (*pypogs.Mount attribute*), 51
available_properties (*pypogs.Receiver attribute*), 54
available_properties (*pypogs.SpotTracker attribute*), 39
azi_limit (*pypogs.Mount attribute*), 51

B

bg_subtract_mode (*pypogs.SpotTracker attribute*), 40
binary_open (*pypogs.SpotTracker attribute*), 40
binning (*pypogs.Camera attribute*), 46

C

Camera (*class in pypogs*), 45
camera (*pypogs.TrackingThread attribute*), 35
CCL_enable (*pypogs.ControlLoopThread attribute*), 31
CCL_I (*pypogs.ControlLoopThread attribute*), 31
CCL_P (*pypogs.ControlLoopThread attribute*), 31
CCL_speed_limit (*pypogs.ControlLoopThread attribute*), 31
CCL_transition_th (*pypogs.ControlLoopThread attribute*), 31
centroid_window (*pypogs.SpotTracker attribute*), 40
change_mean_relative() (*pypogs.SpotTracker method*), 38
clear_coarse_camera() (*pypogs.System method*), 21
clear_fine_camera() (*pypogs.System method*), 21
clear_mount() (*pypogs.System method*), 21
clear_receiver() (*pypogs.System method*), 21
clear_star_camera() (*pypogs.System method*), 21
clear_start_end_time() (*pypogs.Target method*), 28
coarse_camera (*pypogs.System attribute*), 23
coarse_track_thread (*pypogs.System attribute*), 23
color_bin (*pypogs.Camera attribute*), 47
control_loop_thread (*pypogs.System attribute*), 23
ControlLoopThread (*class in pypogs*), 30
crop (*pypogs.SpotTracker attribute*), 40
CTFSP_auto_update_CCL_goal (*pypogs.ControlLoopThread attribute*), 31
CTFSP_auto_update_CCL_goal_th (*pypogs.ControlLoopThread attribute*), 32

CTFSP_delay (*pypogs.ControlLoopThread attribute*), 32
CTFSP_disable_after_goal_update (*pypogs.ControlLoopThread attribute*), 32
CTFSP_disable_integral (*pypogs.ControlLoopThread attribute*), 32
CTFSP_enable (*pypogs.ControlLoopThread attribute*), 32
CTFSP_max_radius (*pypogs.ControlLoopThread attribute*), 32
CTFSP_ramp (*pypogs.ControlLoopThread attribute*), 32
CTFSP_spacing (*pypogs.ControlLoopThread attribute*), 32
CTFSP_speed (*pypogs.ControlLoopThread attribute*), 32
CTFSP_transition_th (*pypogs.ControlLoopThread attribute*), 32

D

data_folder (*pypogs.Alignment attribute*), 27
data_folder (*pypogs.ControlLoopThread attribute*), 33
data_folder (*pypogs.Receiver attribute*), 54
data_folder (*pypogs.SpotTracker attribute*), 40
data_folder (*pypogs.System attribute*), 23
data_folder (*pypogs.TrackingThread attribute*), 36
debug_folder (*pypogs.Alignment attribute*), 27
debug_folder (*pypogs.Camera attribute*), 47
debug_folder (*pypogs.ControlLoopThread attribute*), 33
debug_folder (*pypogs.Mount attribute*), 51
debug_folder (*pypogs.SpotTracker attribute*), 40
debug_folder (*pypogs.System attribute*), 23
debug_folder (*pypogs.TrackingThread attribute*), 36
degrees_to_0_360() (*pypogs.Mount static method*), 50
degrees_to_n180_180() (*pypogs.Mount static method*), 50
deinitialize() (*pypogs.Camera method*), 46
deinitialize() (*pypogs.Mount method*), 50
deinitialize() (*pypogs.Receiver method*), 54
deinitialize() (*pypogs.System method*), 21
do_alignment_test() (*pypogs.System method*), 21
do_auto_star_alignment() (*pypogs.System method*), 21
downsample (*pypogs.SpotTracker attribute*), 40

E

end_time (*pypogs.Target attribute*), 29
exposure_time (*pypogs.Camera attribute*), 47
exposure_time_auto (*pypogs.Camera attribute*), 47
exposure_time_limit (*pypogs.Camera attribute*), 47

F

fails_to_drop (*pypogs.SpotTracker attribute*), 40
failure_sd_penalty (*pypogs.SpotTracker attribute*), 41
FCL_enable (*pypogs.ControlLoopThread attribute*), 32
FCL_I (*pypogs.ControlLoopThread attribute*), 32
FCL_P (*pypogs.ControlLoopThread attribute*), 32
FCL_speed_limit (*pypogs.ControlLoopThread attribute*), 33
FCL_transition_th (*pypogs.ControlLoopThread attribute*), 33
feedforward_rate (*pypogs.TrackingThread attribute*), 36
feedforward_threshold (*pypogs.TrackingThread attribute*), 36
filtsize (*pypogs.SpotTracker attribute*), 41
fine_camera (*pypogs.System attribute*), 23
fine_track_thread (*pypogs.System attribute*), 23
flip_x (*pypogs.Camera attribute*), 47
flip_y (*pypogs.Camera attribute*), 47
frame_rate (*pypogs.Camera attribute*), 47
frame_rate_actual (*pypogs.Camera attribute*), 47
frame_rate_auto (*pypogs.Camera attribute*), 47
frame_rate_limit (*pypogs.Camera attribute*), 47
frequency (*pypogs.TrackingThread attribute*), 36
frequency_actual (*pypogs.TrackingThread attribute*), 36

G

gain (*pypogs.Camera attribute*), 48
gain_auto (*pypogs.Camera attribute*), 48
gain_limit (*pypogs.Camera attribute*), 48
get_alt_az() (*pypogs.Mount method*), 50
get_alt_az_of_target() (*pypogs.System method*), 22
get_com_altaz_from_enu_altaz() (*pypogs.Alignment method*), 25
get_com_altaz_from_itrf_xyz() (*pypogs.Alignment method*), 25
get_com_altaz_from_mnt_altaz() (*pypogs.Alignment method*), 25
get_enu_altaz_from_com_altaz() (*pypogs.Alignment method*), 25
get_enu_altaz_from_itrf_xyz() (*pypogs.Alignment method*), 25
get_enu_altaz_from_mnt_altaz() (*pypogs.Alignment method*), 26
get_itrf_direction_of_target() (*pypogs.System method*), 22
get_itrf_relative_from_position() (*pypogs.Alignment method*), 26
get_itrf_xyz_from_enu_altaz() (*pypogs.Alignment method*), 26

get_itrf_xyz_from_mnt_altaz() (*pypogs.Alignment method*), 26
get_latest_image() (*pypogs.Camera method*), 46
get_location_itrf() (*pypogs.Alignment method*), 26
get_location_lat_lon_height() (*pypogs.Alignment method*), 26
get_mnt_altaz_from_com_altaz() (*pypogs.Alignment method*), 26
get_mnt_altaz_from_enu_altaz() (*pypogs.Alignment method*), 26
get_mnt_altaz_from_itrf_xyz() (*pypogs.Alignment method*), 26
get_new_image() (*pypogs.Camera method*), 46
get_next_image() (*pypogs.Camera method*), 46
get_short_string() (*pypogs.Target method*), 28
get_target_itrf_xyz() (*pypogs.Target method*), 28
get_tle_raw() (*pypogs.Target method*), 28
goal_offset_rate (*pypogs.TrackingThread attribute*), 36
goal_offset_x_y (*pypogs.SpotTracker attribute*), 41
goal_offset_x_y (*pypogs.TrackingThread attribute*), 36
goal_x_y (*pypogs.SpotTracker attribute*), 41
goal_x_y (*pypogs.TrackingThread attribute*), 36

H

has_target (*pypogs.Target attribute*), 29
has_track (*pypogs.SpotTracker attribute*), 41
has_track (*pypogs.TrackingThread attribute*), 36
home_alt_az (*pypogs.Mount attribute*), 52

I

identity (*pypogs.Camera attribute*), 48
identity (*pypogs.Mount attribute*), 52
identity (*pypogs.Receiver attribute*), 54
image_folder (*pypogs.TrackingThread attribute*), 36
image_sigma_th (*pypogs.SpotTracker attribute*), 41
image_th (*pypogs.SpotTracker attribute*), 41
img_save_frequency (*pypogs.TrackingThread attribute*), 36
initialize() (*pypogs.Camera method*), 46
initialize() (*pypogs.Mount method*), 50
initialize() (*pypogs.Receiver method*), 54
initialize() (*pypogs.System method*), 22
instant_power (*pypogs.Receiver attribute*), 54
integral_max_add (*pypogs.ControlLoopThread attribute*), 33
integral_max_subtract (*pypogs.ControlLoopThread attribute*), 33
integral_min_rate (*pypogs.ControlLoopThread attribute*), 33
is_aligned (*pypogs.Alignment attribute*), 27
is_busy (*pypogs.System attribute*), 23
is_init (*pypogs.Camera attribute*), 48
is_init (*pypogs.Mount attribute*), 52
is_init (*pypogs.Receiver attribute*), 54
is_init (*pypogs.System attribute*), 23
is_located (*pypogs.Alignment attribute*), 27
is_moving (*pypogs.Mount attribute*), 52
is_running (*pypogs.Camera attribute*), 48
is_running (*pypogs.ControlLoopThread attribute*), 33
is_running (*pypogs.Receiver attribute*), 54
is_running (*pypogs.TrackingThread attribute*), 36

L

list_available_ports() (*pypogs.Mount static method*), 50

M

max_frequency (*pypogs.ControlLoopThread attribute*), 34
max_rate (*pypogs.Mount attribute*), 52
max_search_radius (*pypogs.SpotTracker attribute*), 41
mean_alt_az (*pypogs.TrackingThread attribute*), 37
mean_area (*pypogs.SpotTracker attribute*), 41
mean_sum (*pypogs.SpotTracker attribute*), 41
mean_x_y (*pypogs.SpotTracker attribute*), 41
mean_x_y_absolute (*pypogs.SpotTracker attribute*), 42
mean_x_y_absolute (*pypogs.TrackingThread attribute*), 37
measurement_range (*pypogs.Receiver attribute*), 55
min_search_radius (*pypogs.SpotTracker attribute*), 42
model (*pypogs.Camera attribute*), 48
model (*pypogs.Mount attribute*), 52
model (*pypogs.Receiver attribute*), 55
Mount (*class in pypogs*), 49
mount (*pypogs.System attribute*), 23
move_home() (*pypogs.Mount method*), 51
move_to_alt_az() (*pypogs.Mount method*), 51

N

name (*pypogs.Camera attribute*), 48
name (*pypogs.ControlLoopThread attribute*), 34
name (*pypogs.Mount attribute*), 52
name (*pypogs.Receiver attribute*), 55
name (*pypogs.SpotTracker attribute*), 42
name (*pypogs.TrackingThread attribute*), 37

O

OL_goal_offset_x_y (*pypogs.ControlLoopThread attribute*), 33
OL_goal_x_y (*pypogs.ControlLoopThread attribute*), 33

OL_I (*pypogs.ControlLoopThread attribute*), 33
OL_P (*pypogs.ControlLoopThread attribute*), 33
OL_speed_limit (*pypogs.ControlLoopThread attribute*), 33

P

penalize_track() (*pypogs.SpotTracker method*), 38
plate_scale (*pypogs.Camera attribute*), 48
pos_search_rad (*pypogs.SpotTracker attribute*), 42
pos_search_rad (*pypogs.TrackingThread attribute*), 37
pos_search_x_y (*pypogs.SpotTracker attribute*), 42
pos_search_x_y (*pypogs.TrackingThread attribute*), 37
position_sigma (*pypogs.SpotTracker attribute*), 42
pypogs.hardware.hardware_camera (*module*), 44
pypogs.hardware.hardware_mount (*module*), 49
pypogs.hardware.hardware_receiver (*module*), 53
pypogs.system (*module*), 17
pypogs.tracking (*module*), 29

R

Receiver (*class in pypogs*), 53
receiver (*pypogs.System attribute*), 23
remove_event_callback() (*pypogs.Camera method*), 46
reset_integral_if_saturated (*pypogs.ControlLoopThread attribute*), 34
rms_error (*pypogs.SpotTracker attribute*), 42
rms_error (*pypogs.TrackingThread attribute*), 37
rmse_smoothing_parameter (*pypogs.SpotTracker attribute*), 42
rotate_90 (*pypogs.Camera attribute*), 48
rotation (*pypogs.Camera attribute*), 49

S

sample_rate (*pypogs.Receiver attribute*), 55
sd_area (*pypogs.SpotTracker attribute*), 42
sd_sum (*pypogs.SpotTracker attribute*), 43
set_alignment_enu() (*pypogs.Alignment method*), 27
set_alignment_from_observations() (*pypogs.Alignment method*), 27
set_location_itrf() (*pypogs.Alignment method*), 27
set_location_lat_lon() (*pypogs.Alignment method*), 27
set_rate_alt_az() (*pypogs.Mount method*), 51
set_start_end_time() (*pypogs.Target method*), 28
set_target_deep_by_name() (*pypogs.Target method*), 28
set_target_from_ra_dec() (*pypogs.Target method*), 29
set_target_from_tle() (*pypogs.Target method*), 29
sigma_mode (*pypogs.SpotTracker attribute*), 43
size_max (*pypogs.Camera attribute*), 49
size_readout (*pypogs.Camera attribute*), 49
slew_to_target() (*pypogs.System method*), 22
smooth_power (*pypogs.Receiver attribute*), 55
smoothing_parameter (*pypogs.Receiver attribute*), 55
smoothing_parameter (*pypogs.SpotTracker attribute*), 43
spot_max_area (*pypogs.SpotTracker attribute*), 43
spot_max_axis_ratio (*pypogs.SpotTracker attribute*), 43
spot_max_sum (*pypogs.SpotTracker attribute*), 43
spot_min_area (*pypogs.SpotTracker attribute*), 43
spot_min_sum (*pypogs.SpotTracker attribute*), 44
spot_tracker (*pypogs.TrackingThread attribute*), 37
SpotTracker (*class in pypogs*), 37
star_camera (*pypogs.System attribute*), 23
start() (*pypogs.Camera method*), 46
start() (*pypogs.ControlLoopThread method*), 31
start() (*pypogs.Receiver method*), 54
start() (*pypogs.TrackingThread method*), 35
start_time (*pypogs.Target attribute*), 29
start_tracking() (*pypogs.System method*), 22
state_cache (*pypogs.ControlLoopThread attribute*), 34
state_cache (*pypogs.Mount attribute*), 52
stop() (*pypogs.Camera method*), 46
stop() (*pypogs.ControlLoopThread method*), 31
stop() (*pypogs.Mount method*), 51
stop() (*pypogs.Receiver method*), 54
stop() (*pypogs.System method*), 22
stop() (*pypogs.TrackingThread method*), 35
successes_to_track (*pypogs.SpotTracker attribute*), 44
sum_max_sd (*pypogs.SpotTracker attribute*), 44
sum_min_sd (*pypogs.SpotTracker attribute*), 44
sum_sigma (*pypogs.SpotTracker attribute*), 44
System (*class in pypogs*), 17

T

Target (*class in pypogs*), 28
target (*pypogs.System attribute*), 23
target_object (*pypogs.Target attribute*), 29
tetra3 (*pypogs.System attribute*), 23
track_alt_az (*pypogs.TrackingThread attribute*), 37
track_area (*pypogs.SpotTracker attribute*), 44
track_sd (*pypogs.SpotTracker attribute*), 44

track_sd (*pypogs.TrackingThread attribute*), 37
track_sum (*pypogs.SpotTracker attribute*), 44
track_x_y (*pypogs.SpotTracker attribute*), 44
track_x_y_absolute (*pypogs.SpotTracker attribute*), 44
track_x_y_absolute (*pypogs.TrackingThread attribute*), 37
TrackingThread (*class in pypogs*), 34

U

update_databases() (*pypogs.System static method*), 22
update_from_image() (*pypogs.SpotTracker method*), 38
update_from_observation() (*pypogs.SpotTracker method*), 39

W

wait_for_move_to() (*pypogs.Mount method*), 51

Z

zero_altitude (*pypogs.Mount attribute*), 52