PyNTA Documentation

Release 0.1

Aquiles Carattino

Jul 30, 2019

CONTENTS

Nanoparticle tracking analysis refers to a technique used for characterizing small objects optically. The base principle is that by following the movement of nanoparticles over time, it is possible to calculate their diffusion properties and thus derive their size.

PyNTA aims at bridging the gap between experiments and results by combining data acquisition and analysis in one simple to use program.

PyNTA is shipped as a package that can be installed into a virtual environment with the use of pip. It can be both triggered with a built in function or can be included into larger projects.

CHAPTER

ONE

INSTALLING

PyNTA can be easily installed by running:

```
pip install pynta
```

However, it is also possible to install the latest development version. The source code of the program is hosted at https://github.com/nanoepics/pynta. If you want to install the development version of PyNTA you can run the following command:

```
pip install git+https://github.com/nanoepics/pynta
```

If you need further assistance with the installation of the code, please check *Installing*.

CHAPTER

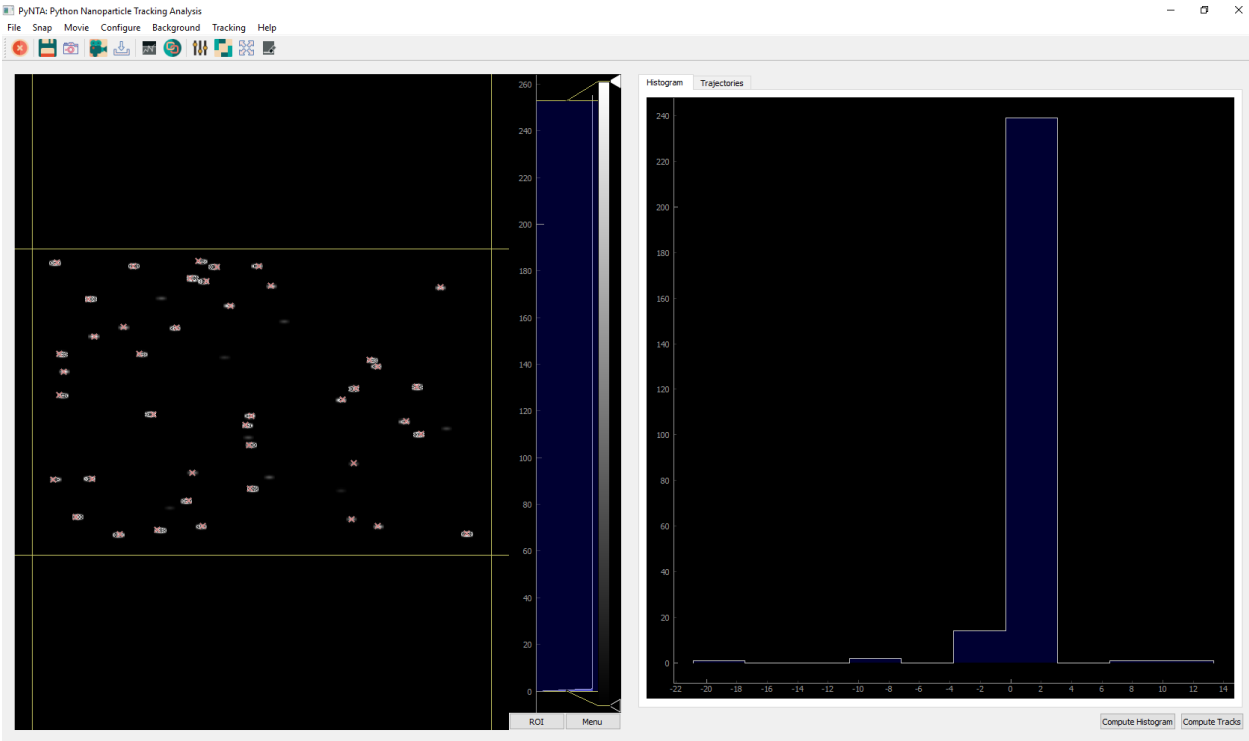
TWO

START THE PROGRAM

After installing, the program can be started from the command line by running the following:

```
python -m pynta -c config.yml
```

Remember that config.yml needs to exist. To create your own configuration file, you can start with the example provided in the examples folder. Once the program starts, it will look like the following:



CHAPTER

THREE

CONTRIBUTING TO THE PROGRAM

The program is open source and therefore you can modify it in any way that you see fit. You have to remember that the code was written with a specific experiment in mind and therefore it may not fulfill or the requirements of more advanced imaging software.

However the design of the program is such that would allow its expansion to meet future needs. In case you are wondering how the code can be improved you can start by reading *How to Contribute Code*, or directly submerge yourself in the documentation of the different classes *PyNTA API*.

If you want to start right away to improve the code, you can always look at the *List of Todo's*.

CHAPTER

FOUR

ACKNOWLEDGEMENTS

This program was developed by Aquiles Carattino with the support of funding from NWO, The Netherlands Scientific Organization, under VICI grant (PI: Prof. Allard Mosk) and Projectruimte FOM.PR1.005 grant (PI: Dr. Sanli Faez) . This work was carried on at Utrecht University in the months between June 2018 and June 2019.

4.1 Installing

Pynta can be installed directly with pip. The first step is to create a virtual environment on your machine in order to avoid clashes with the versions of the dependencies. Virtual Environments are must-have tool, regardless of what you are doing. You can read a discussion about them directly on Python For The Lab. You can also see how to create a virtual environment *Setting up a Python Virtual Environment*.

To install PyNTA you can run the following command:

```
pip install git+https://github.com/nanoepics/pynta
```

This will get you the latest stable version of Pynta. If you, however, would like to test new features, you can download the development branch of the program:

```
pip install git+https://github.com/nanoepics/pynta@develop
```

Especially if you want to try the development version, you should install it in a virtual environment. We can't guarantee that the development will be future compatible, i.e. that the program stays compatible with itself over time. One of the highest risks is that an upgrade on the development branch may brake your config files or customizations you have done.

Moreover there is the risk of requiring dependencies that are not fully supported or that are later dropped.

4.1.1 Dependencies

By default, when you install PyNTA, the following dependencies will be installed on your computer:

- trackpy
- pyqt5<5.11
- numpy
- pyqtgraph
- pint
- h5py
- pandas

- pyyaml
- pyzmq
- numba

However, not all dependencies are mandatory for the program to work. For instance, if you are not interested in the GUI but are planning to run the program from the command line, you are free to skip PyQt5, Pyqtgraph.

Numba is used because it accelerates the tracking of particles with **trackpy**. But if it is not available on the computer, the program will run anyways.

Trackpy

Trackpy is instrumental for the program to work correctly. This package is able to detect particles on an image based on few parameters but also to link the particles together, building up single-particle traces. PyNTA uses trackpy to perform all the detection and analysis in real time.

One of the constrains of trackpy is that it depends heavily on Pandas, which is a great tool while working in combination with Jupyter notebooks, but is not that great for user interfaces. Which require to transform from Pandas Data Frames to numpy arrays all the time.

PyQt5 and Pyqtgraph

The user interface is built on PyQt5 in combination with PyQtGraph. PyQt5 versions newer than 5.11 fail at installing through the setup process (but they do work if installed directly with pip). If this bug is resolved, the constrain on the version of PyQt to use should be lifted. Moreover, it is desirable to switch to PySide2 as soon as the project is mature.

4.1.2 Operating System Support

PyNTA was tested both on Linux and Windows machines. However, the main environment for PyNTA to run is Windows 10. There are some very fundamental differences on how processes are started between Linux and Windows that have mutual drawbacks. For example, on Windows processes are spawned, meaning that classes are re-imported and not instantiated. Therefore, processes don't start with a shared state. This prevents, for example, to start a new process for a method of a class.

This forced the architecture of the program to rely heavily on functions and not methods, making the code slightly more convoluted than what was desirable. The approach works on Linux also, but the performance may not be optimal.

4.2 The Config File

To start the program, it is necessary to define a configuration file. You can `get a config file here`. However, the best place to find the latest examples of config files is on the Github Repository. The idea behind the config file is that it makes it transparent both to the end user and to the developer the different settings available throughout the program.

The example config files only show the minimum possible contents. You are free to add as many entries as you would like. However, they are not going to be displayed in the GUI, nor will be used automatically. They will, however, be stored as metadata together with all the files. You could use the config file in order to annotate your experiments, for example.

4.2.1 The Format

The config file is formatted as a YAML file. These files are very easy to transform into python dictionaries and are very easy to type. So, for example, to change how the tracking algorithm works, one would change the following lines:

```
tracking:
  locate:
    diameter: 11   # Diameter of the particles (in pixels) to track, has to be an odd
↪number
    invert: False
    minmass: 100
```

Note that for the file to make sense, it has to be indexed with 2 spaces. When reading it, it automatically converts some data types. For example, diameter will be available as `config['tracking']['locate']['diameter']` and will be of type integer. `invert` will be a boolean, etc. If the data type is not clear, the default is a string. So, for example:

```
camera:
  exposure_time: 30ms # Initial exposure time (in ms)
```

Will generate a `config['camera']['exposure_time']` of type string, that will need to be transformed to a quantity later on. Note also that comments are ignored (after the # nothing is read).

4.2.2 Real Cameras

Currently Pynta supports a handful of cameras. If you would like to load a hamamatsu camera, you should change the following line:

```
camera:
  model: dummy_camera
```

with the following:

```
camera:
  model: hamamatsu
```

If you would like to understand how the loading of the camera works, in order to add your own, you can check `initialize_camera()`.

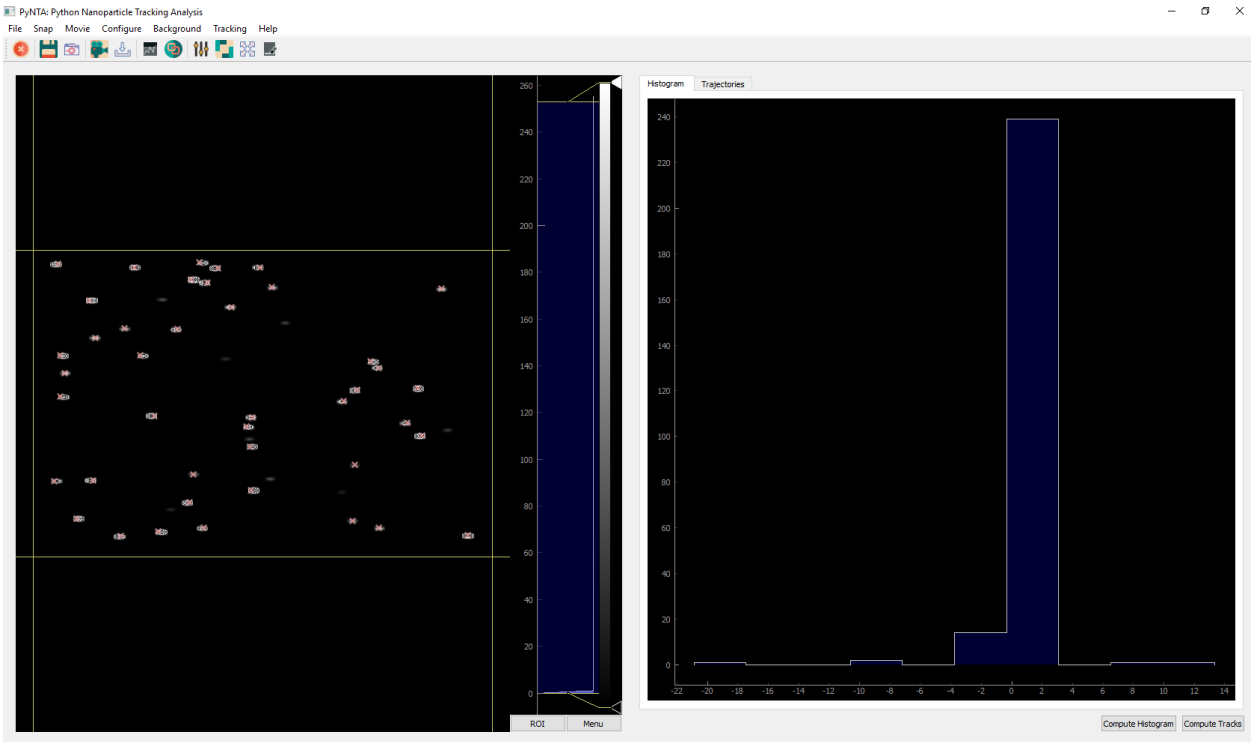
4.3 Getting Started

In order to familiarize yourself with the program, the best idea is to start with simulated data. In this way you avoid all the problems arising from interfacing with a real instrument and you can see the limitations of the program. The example config available on the repository is already configured to work with a simulated camera. It is recommended that you copy the contents of the file into a folder on your own computer.

4.3.1 Opening the Program

After *Installing* PyNTA, you can trigger it from the command line. You can simply run the following command:

```
pynta
```



After a few moments, a screen like the one below will welcome you to the program.

This will use synthetic data by default, i.e. you can snap or acquire a movie and test the capabilities of the program without the need of connecting with a real camera. Once you are confident with the program and you would like to start using real hardware, you need to develop *a proper config file*. Once you have it, you can run the following command:

```
python -m pynta -c config.yml
```

You can reproduce the config file that gives you the synthetic data and then move to real devices.

4.3.2 The Tools

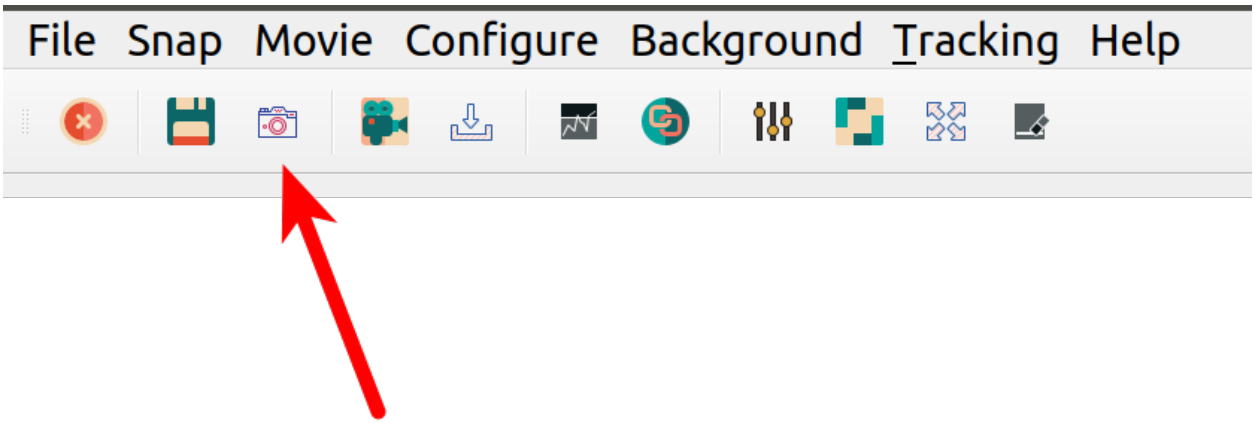
Most of the options were designed to be self-explanatory. However it is important to give a short discussion in order to speed the introduction to the tool. After initializing, normally one would like to snap a photo in order to see what is being recorded by the camera. You can achieve it by clicking the button as shown in the image below:

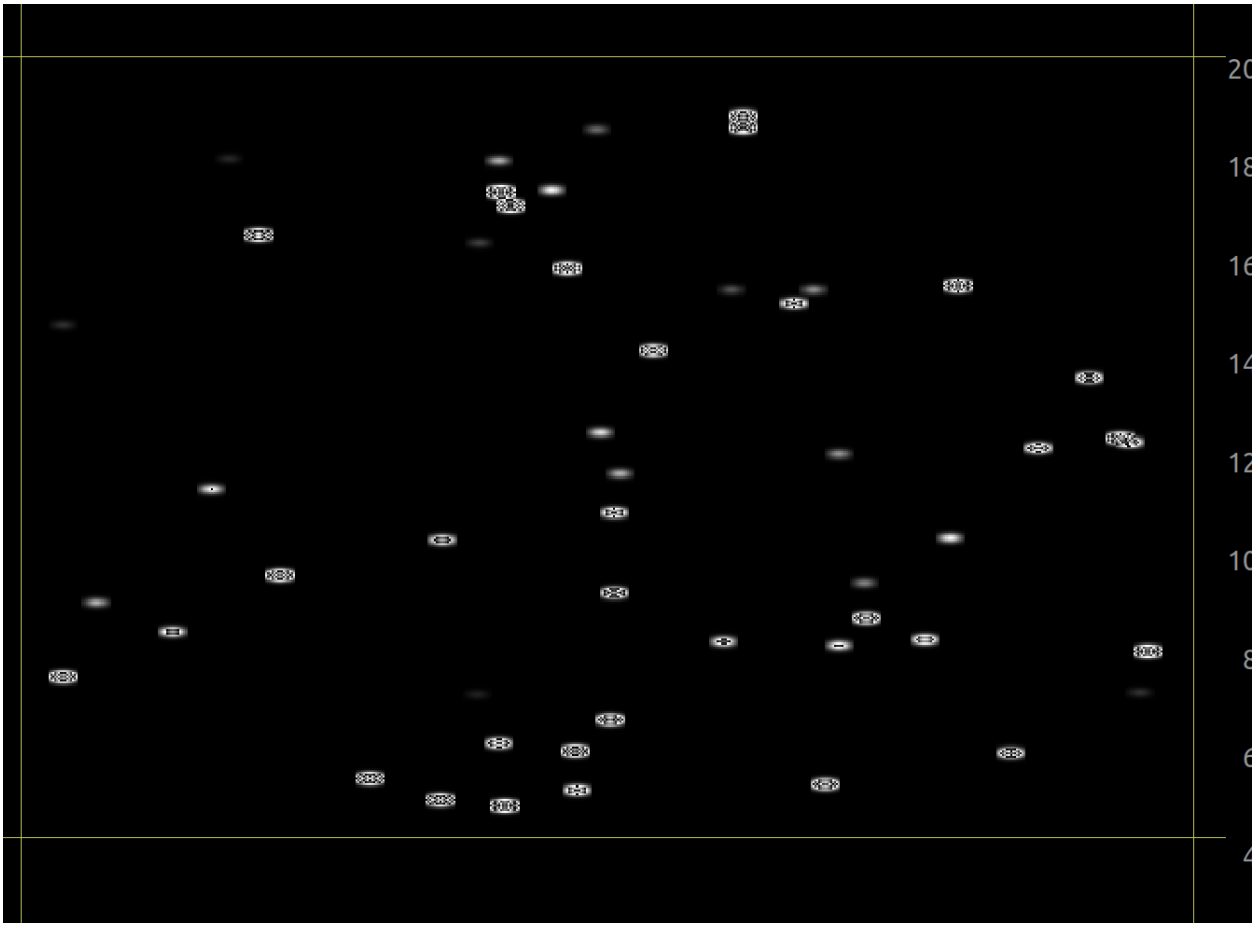
This will record a single image from the camera and will be displayed on the space right below:

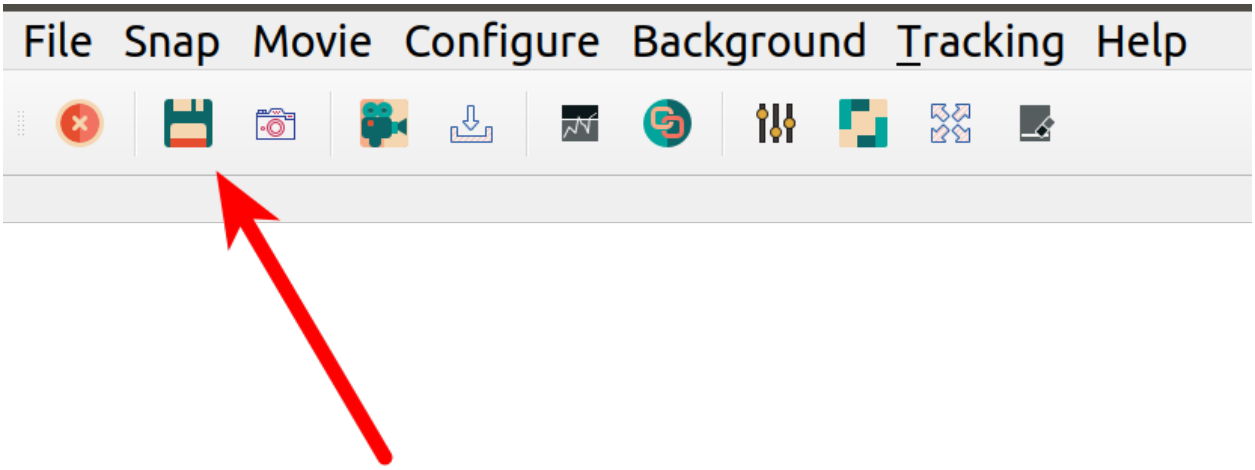
The image can be zoomed-in and out by scrolling with the central wheel of the mouse. Dragging allows to move around the image. In order to return to the full view, it is possible to right-click on the image and select `View All`. The histogram on the right of the image shows the levels for displaying. You can adjust the minimum and maximum as well as the color scale. Right clicking on the image allows you to do an `Auto Range`, i.e. to adjust the levels such that the maximum and minimum correspond to those of the data being displayed.

If you want to save the image you can click on the icon for saving, as shown below:

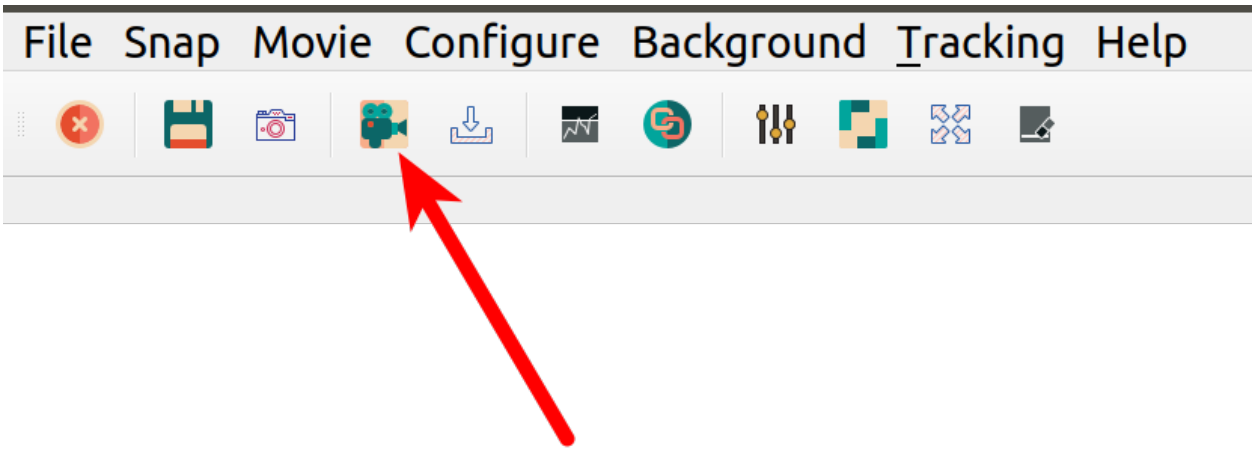
PyNTA also allows you to acquire continuous images, by clicking on the icon highlighted below. The exact behaviour will depend on the camera employed. For example, if a frame-grabber is available, the exact timing between frames can be guaranteed. Cameras without a buffer, however, will have a timing that depends on the computer ability to


File
Snap
Movie
Configure
Background
Tracking
Help





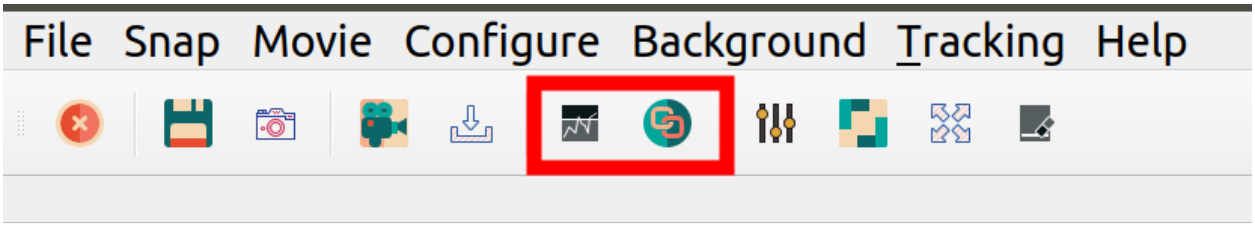
read from them. The communication with the camera happens in a separate thread, trying to guarantee the maximum reliability of the timing.



Another feature is the continuous saves option, which is right next to the start movie button. The continuous saves streams all the available frames to a file on the hard rive. The location of the file is determined in the config file or, as we will see later, can be set in the configuration on the User Interface. In case of acquiring at high frame rates, not all frames are displayed to the user, but all of them will be saved.

4.3.3 Tracking and Linking

The feature that really makes PyNTA unique is the ability to identify and track nanoparticles on a video in real time. The procedure for tracking and analysis requires of two steps. First, you have to start identifying the particles, with the button called `start tracking`. You will see red crosses appearing on the particles in the image. It takes a few instants to setup the linking procedure, during which the movie may seem to freeze.



If you are satisfied with how the identification of particles works, you can start linking the positions. Linking is a procedure that identifies whether locations in consecutive frames belong to the same particle or not. This procedure can be computationally expensive and requires fine tuning of the parameters. Linking also happens in a separate process, and in parallel to the acquisition and identification of particles.

4.4 How to Contribute Code

If you want to collaborate with PyNTA, you can start by reading the *Information for Developers*.

4.5 Setting up a Python Virtual Environment

This guide is thought for users on Windows that want to use virtual environments on their machines.

1. Run:

   ```
   pip.exe install virtualenv
   ```

   At this point you have a working installation of virtual environment that will allow you to isolate your development from your computer, ensuring no mistakes on versions will happen. Let's create a new working environment called Testing

7. Run:

   ```
   virtualenv.exe Testing
   ```

   This command will create a folder called Testing, in which all the packages you are going to install are going to be kept.

8. To activate the Virtual Environment, run:

   ```
   .\Testing\Scripts\activate
   ```

(The `.` at the beginning is very important). If an error happens (most likely) follow the instructions below. Windows has a weird way of handling execution policies and we are going to change that.

Open PowerShell with administrator rights (normally, just right click on it and select run as administrator) Run the following command:

```
Set-ExecutionPolicy RemoteSigned
```

This will allow to run local scripts. Go back to the PowerShell without administrative rights and run again the script activate.

9. When you are inside a virtual environment, you should see the name between `()` appearing at the beginning of the command line.

   Now you are working on a safe development environment. If you run:

   ```
   pip freeze
   ```

   You will see a list of all the packages currently installed in your environment. The list should be empty.

10. To deactivate the virtual environment just run:

    ```
    deactivate
    ```

11. If you run `freeze` again, you will see all the packages installed in the computer:

    ```
    pip freeze
    ```

4.6 Information for Developers

In these pages you will find the information needed for expanding the codebase of PyNTA. You can go through the documentation of every module by following the links at the bottom of the page. However, there are some general guidelines regarding how to understand the code that can help you pin point the specific parts of the program that you would like to modify.

The design pattern chosen for the program is called MVC, that stands for Model-View-Controller. This pattern splits the different attributions of the code in order to make it more reusable. You can find a discussion about the MVC for lab applications on this website. The general principle is that by generating clear differences between parts of the program, you can obtain a highly sustainable program. Below you can see a description of what every module is and how you can help expanding it.

Every module will have its own particularities, which are explained briefly below. For more information, you should see the individual documentation and go through the code to understand better how the program's architecture was developed. It is important to note that PyNTA is a highly parallelized program. It relies on **zeromq** to pass messages between different processes. We have implemented a publisher/subscriber architecture that allows one central process to broadcast messages which can be captured by any listening procedure. This enables to spin on or off processes without altering the code. Moreover, it would make a clear path for remote control, data storage, etc.

The decentralized architecture of PyNTA makes it very flexible and powerful, but it also makes the code base much harder to understand for novice developers. Asynchronous programming is not a trivial task and can confuse people who are new to programming complex solutions (the code is no longer read from top to bottom). **Zeromq** was easy to implement, but may not be the best performing library for heavy numpy applications. Better approaches may be found in the future without compromising the existing code.

Below you will find an introduction to the code architecture. Read through to understand where to find answers and where is the proper place to implement updates and solutions.

4.6.1 Controller

In our definition of the MVC pattern, *controller* are the drivers of the devices. Controllers for cameras, for example, rely on library files (.dll files on Windows, .so in Linux) that can be more or less documented. For example `hamamatsu` uses the *DCAM-API*, while `photonicscience` uses *scmoscam.dll*. Controllers should follow the specifications of each device as closely as possible. For instance, the units each devices uses for setting exposure times or framerates, etc. Moreover, they shouldn't implement anything that the device is not capable of doing. A clear example would be to set a region of interest. Some cameras support it but some don't. You could be tempted to apply a ROI in the software directly, but you shouldn't do it in the Controller, there is a better place as we will see later.

Controllers are not always plain python files, sometimes are entire packages on themselves. For instance, the controller for Basler cameras (PyPylons) or National Instruments cards (pyDAQmx) are packages on their own. Sometimes the controllers where developed by other people, like the case of the Hamamatsu code, which was borrowed from Zhuang's lab. If you explore the controllers folder, you will notice that there are some available, like the `keysight` ,that holds the drivers for an oscilloscope and function generator. Those controllers are remnants of a different incarnation of PyNTA, but that may still be valuable for the future.

4.6.2 Model

In the MVC pattern that we have defined earlier, *model* is the place to define all the logic on how to use the devices is located. The core principle of splitting Controllers and Models is to separate what the device can do by design and what the user imposes to the devices. For example, imagine a camera which can acquire frames one by one. The controller would provide a way to snap a single-frame. However, a user may want to acquire a movie by developing a for-loop that acquires a series of frames. The controller should reflect the capacities of the device, while the model will include all the user-imposed logic.

A clear advantage of having models is that in the case where controllers are not part of the package, for example if the driver is provided by the manufacturer itself, a model will make explicit how the driver is used. For example, NI-cards are very complex devices, and if one wants to monitor a signal, there are several required steps, which can be written down as a method of a class and re-used as much as wanted. Models can make use of explicit units (through Pint), for example.

Models are not limited to devices, but they also make explicit how an experiment is performed. In PyNTA, there are available at least two different experimental models, one for performing nanoparticle tracking analysis in 2D images (the classical NTA) and one to perform tracking analysis in hollow optical fibers, i.e., 1-D nanoparticle tracking. In these experiment models, you can find all the steps and how they relate to each other. For example, you can't subtract background if you haven't acquired the background first, etc.

Models for the Cameras

Imagine you have a camera that doesn't support setting a region of interest (ROI), you can still crop the resulting images in the software, giving the same result as what a camera that does support ROI does. By having an intermediate layer between the controllers and the users, it is possible to detach the specific logic of an experiment and the specifications of the devices. Moreover, it makes very straightforward to add new devices to the experiment, exchanging cameras, etc.

Therefore in *cameras* is where we will develop classes that have always the same methods and outputs defined, but that behave completely different when communicating with the devices. The starting point is the `skeleton`, where the `cameraBase` class is defined. In this class all the methods and variables needed by the rest of the program are defined. This strategy not only allows to keep track of the functions, it also enables the subclassing, which will be discussed later.

Having models also allow to quickly change from one camera to another. For example, if one desires to switch from a *Hamamatsu* to a *PSI*, the only needed thing to do is to replace:

```
from pynta.model.cameras.hamamatsu import camera
```

With:

```
from pynta.model.cameras.psi import camera
```

As you see, both modules `Hamamatsu` and `PSI` define a class called camera. And this classes will have the same methods defined, therefore whatever code relies on camera will be working just fine. One of the obvious advantages of having a Model is that we can define a `Dummy Camera` that allows to test the code without being connected to any real device.

If you go through the code, you'll notice that the classes defined in Models inherit `cameraBase` from the `skeleton`. The quick advantage of this is that any function defined in the skeleton will be already available in the child objects. Therefore, if you want to add a new function, let's say `set_gain`, you will have to start by adding that method to the `skeleton`. This will make the function readily available to all the models, even if just as a mockup or to `raise NotImplementedError`. Then we can overload the method by defining it again in the class we are working on. It may be that not all the cameras are able to set a gain, and we can just leave a function that `return True`. If it is a functionality that you expect any camera to have, for example triggering an image acquisition, you can set the skeleton function to `raise NotImplementedError`. This will give a very descriptive error of what went wrong if you haven't implemented the function in your model class.

Todo: It is also possible to define the methods as `@abstractmethods` which will automatically raise an exception. It may be worth exploring this possibility if there are several developers involved.

Model for the Experiment

The way a user interacts with a camera is only part of the logic of an experiment. There are a lot of different steps and conditions that a person needs in order to obtain data. In the case of PyNTA, one of the requirements would be to analyse the images being acquired by a camera in real time and track the particles on them. Therefore, the best solution in order to develop clear code is to generate a new class for the experiment itself. You can see it in *experiment* (the general, base one) or on `nanoparticle_tracking`, a more elaborated one.

The idea behind the experiment model is that it makes it very clear what the logic of the experiment is. You need to load a camera, you need to acquire a movie, you need to analyse the frames, etc. It is very easy to add new steps, change the order in which different things happen, etc. Depending on the developer, starting with the experiment model may be the best place to implement changes, since it makes it very clear what is happening and what can be improved. A good idea would be to start developing your own experiment as a command-line based tool.

4.6.3 View

In the MVC patter, *view* is, as the name suggests, where the GUI is defined. Since we defined all the logic of the experiment in a separate class, the View takes care of only triggering specific steps of the experiment and displaying the results. In principle, the only logic present in the view modules is only for aesthetic reasons. For instance, disable a specific button while a measurement is going on, etc. However, it should be the experiment model the one that avoids triggering two measurements if it is not possible, etc. That guarantees a lower-level safety.

view is where the GUI lives. Within the module, you will also find *pynta.view.GUI*, in which the different widgets that make up the window are defined and another folder called designer, where the Qt Designer files are located. Adapting the looks of a program should start by looking into the `.ui` files, then checking the associated widgets in the `GUI` module, in order to connect the proper signals, etc. and finally modifying the main view class.

PyNTA API

Pynta Controllers

Devices

Collection of drivers for different devices which are useful to perform measurements. Drivers may depend on external libraries. Thus, be aware that you may need to install some dependencies in order to make the controllers work correctly. A common example is the requirement of pyvisa to communicate with serial devices, or the DLL's from Hamamatsu in order to use their cameras.

For several devices, drivers in Python are provided by the manufacturers and thus they won't be found in this module. A clear example is the Python wrapper of Pylon to work with Basler cameras, known as PyPylon. In that case, PyNTA only provides a model which relies on that package. For DAQ devices such as those from National Instruments, PyNTA depends on PyDAQmx, although NI has released its own wrapper, NIDAQmx-Python which may be worth exploring.

Hamamatsu Driver

Original file taken from ZhuangLab

A ctypes based interface to Hamamatsu cameras. (tested on a sCMOS Flash 4.0).

The documentation is a little confusing to me on this subject.. I used c_int32 when this is explicitly specified, otherwise I use c_int.

Todo: I'm using the "old" functions because these are documented. Switch to the "new" functions at some point.

Todo: How to stream 2048 x 2048 at max frame rate to the flash disk? The Hamamatsu software can do this.

This file was adapted to Python 3 and documented in the numpy style by Aquiles Carattino <aquiles@uetke.com>

copyright Hazen Babcock

license The MIT License

exception pynta.controller.devices.hamamatsu.hamamatsu_camera.**DCAMException**(*message*)
Bases: Exception

Monitor exceptions.

class pynta.controller.devices.hamamatsu.hamamatsu_camera.**DCAM_PARAM_PROPERTYATTR**
Bases: _ctypes.Structure

The dcam property attribute structure.

attribute
Structure/Union member

attribute2
Structure/Union member

cbSize
Structure/Union member

iGroup
Structure/Union member

iProp
Structure/Union member

iPropStep_Element
Structure/Union member

iProp_ArrayBase
Structure/Union member

iProp_NumberOfElement
Structure/Union member

iReserved1
Structure/Union member

iReserved3
Structure/Union member

iUnit
Structure/Union member

nMaxChannel
Structure/Union member

nMaxView
Structure/Union member

option
Structure/Union member

valuedefault
Structure/Union member

valuemax
Structure/Union member

valuemin
Structure/Union member

valuestep
Structure/Union member

class pynta.controller.devices.hamamatsu.hamamatsu_camera.**DCAM_PARAM_PROPERTYVALUETEXT**
Bases: _ctypes.Structure

The dcam text property structure.

cbSize
Structure/Union member

iProp
Structure/Union member

text
Structure/Union member

textbytes
Structure/Union member

value
Structure/Union member

class pynta.controller.devices.hamamatsu.hamamatsu_camera.**HCamData**(*size*)
Bases: object

Hamamatsu camera data object. Initially I tried to use create_string_buffer() to allocate storage for the data from the camera but this turned out to be too slow. The software kept falling behind the camera and create_string_buffer() seemed to be the bottleneck.

copyData(*address*)

getData()

getDataPtr()

class pynta.controller.devices.hamamatsu.hamamatsu_camera.**HamamatsuCamera**(*camera_id*)
Bases: object

CAPTUREMODE_SEQUENCE = 1
Basic camera interface class. This version uses the Hamamatsu library to allocate camera buffers. Storage for the data from the camera is allocated dynamically and copied out of the camera buffers.

CAPTUREMODE_SNAP = 0

captureSetup()
Capture setup (internal use only). This is called at the start of new acquisition sequence to determine the current ROI and get the camera configured properly.

checkStatus(*fn_return*, *fn_name='unknown'*)
Check return value of the dcam function call. Throw an error if not as expected? @return The return value of the function.

fireTrigger()
Triggers the camera when in software mode.

getCameraProperties()
Return the ids & names of all the properties that the camera supports. This is used at initialization to populate the self.properties attribute. @return A python dictionary of camera properties.

getFrames()
Gets all of the available frames. This will block waiting for new frames even if there new frames available when it is called. @return [frames, [frame x size, frame y size]].

getModelInfo(*camera_id*)
Returns the model of the camera @param camera_id The (integer) camera id number. @return A string containing the camera name.

getProperties()
Return the list of camera properties. This is the one to call if you want to know the camera properties. @return A dictionary of camera properties.

getPropertyAttribute(*property_name*)
Return the attribute structure of a particular property. FIXME (OPTIMIZATION): Keep track of known attributes? @param property_name The name of the property to get the attributes of. @return A DCAM_PARAM_PROPERTYATTR object.

getPropertyRW(*property_name*)
Return if a property is readable / writeable. @return [True/False (readable), True/False (writeable)].

getPropertyRange(*property_name*)
Return the range for an attribute. @param property_name The name of the property (as a string). @return [minimum value, maximum value].

getPropertyText(*property_name*)
Return the text options of a property (if any). @param property_name The name of the property to get the text values of. @return A dictionary of text properties (which may be empty).

getPropertyValue(*property_name*)
Return the current setting of a particular property. @param property_name The name of the property. @return [the property value, the property type].

initCamera()

isCameraProperty(*property_name*)
Check if a property name is supported by the camera. @param property_name The name of the property. return True/False if property_name is a supported camera property.

newFrames()
Return a list of the ids of all the new frames since the last check. This will block waiting for at least one new frame. @return [id of the first frame, .. , id of the last frame]

setPropertyValue(*property_name*, *property_value*)
Set the value of a property. @param property_name The name of the property. @param property_value The value to set the property to.

setSubArrayMode()
This sets the sub-array mode as appropriate based on the current ROI.

setmode(*mode*)
Sets the acquisition mode of the camera.

settrigger(*mode*)

shutdown()
Close down the connection to the camera.

startAcquisition()
Start data acquisition.

stopAcquisition()
Stop data acquisition.

class pynta.controller.devices.hamamatsu.hamamatsu_camera.**HamamatsuCameraMR**(*camera_id*)
Bases: *pynta.controller.devices.hamamatsu.hamamatsu_camera.HamamatsuCamera*

Memory recycling camera class.

This version allocates "user memory" for the Hamamatsu camera buffers. This memory is also the location of the storage for the np_array element of a HCamData() class. The memory is allocated once at the beginning, then recycled. This means that there is a lot less memory allocation & shuffling compared to the basic class, which performs one allocation and (I believe) two copies for each frame that is acquired.

Warning: There is the potential here for chaos. Since the memory is now shared there is the possibility that downstream code will try and access the same bit of memory at the same time as the camera and this could end badly.

Todo: Use lockbits (and unlockbits) to avoid memory clashes? This would probably also involve some kind of reference counting scheme.

getFrames()
Gets all of the available frames. This will block waiting for new frames even if there new frames available when it is called.

Todo: It does not always seem to block? The length of frames can be zero. Are frames getting dropped? Some sort of race condition?

Return list [frames, [frame x size, frame y size]]

startAcquisition()
Allocate as many frames as will fit in 2GB of memory and start data acquisition.

stopAcquisition()
Stops the acquisition and releases the memory associated with the frames.

pynta.controller.devices.hamamatsu.hamamatsu_camera.**convertPropertyName**(*p_name*)
"Regularizes" a property name. We are using all lowercase names with the spaces replaced by underscores. @param p_name The property name string to regularize. @return The regularized property name.

Keysight

Photonic Science GEVSCMOS

Driver for Photonic Science Cameras (Pleora GEV). This driver was originally written by Perceval Guillou <perceval@photonic-science.com>, in Py2 and has been successfully tested with scmoscontrol.dll SCMOS Pleora (GEV) control dll (x86)v5.6.0.0 (date modified 10/2/2013)

This version copyright: Sanli Faez <s.faez@uu.nl>

Todo: The coding style is not in line with the rest of PyNTA. The GEVSMOS class can be cleaned up and documented

---

class pynta.controller.devices.photonicscience.scmoscam.**GEVSCMOS**(*cwd_path*, *name*)

Bases: object

AbortSnap()

AutoBinningFilter(*enable*)

Close()

Demangle(*image_pointer*, *Nx*, *Ny*)

EnableAutoLevel(*enable*)

EnableBestFit(*enable*)

EnableBinningFilter(*enable*)

EnableBrightPixel(*enable*)

EnableClip(*enable*)

EnableFlatField(*enable*)

EnableGamma(*enable*)

EnableOffset(*enable*)

EnableRemapping(*enable*)

EnableSharpening(*enable*)

EnableSmooth(*enable*)

EnableStreaming(*enable*)

FreeSequence()

GetDLL()

GetDLLName()

GetImage(*imp=None*)

GetImagePointer()

GetMode()

GetName()

GetOptions()

GetPedestal()

GetRawImage()

GetRemapSize()

GetSequencePointer(*id*)

GetSize()

GetSizeMax()

GetState()

GetStatus()

GetTemperature()

Has8bitGainModes()

HasBinning()

HasClockSpeedLimit()

HasHPMapping()

HasIntensifier()

HasTemperature()

InitFunctions()

InitSequence(*imnum*)

IsFlipped()

IsInCamCor()

IsIntensifier()

LoadCamDLL()

MakeFlatField()

Open()

OpenMap(*file_name='distortion.map'*)

Remap(*image_pointer*, *Nx*, *Ny*)

ResetOptions()

SaveSequence()

SelectIportDevice()

SetALCMaxExp(*maxexp*)

SetALCWin(*l*, *t*, *r*, *b*)

SetBFPeek(*peek*)

SetChipGain(*gain*)

SetClockSpeed(*mode*)

SetExposure(*expo*, *unit*)

SetFlatAverage(*average_number*)

SetFlickerMode(*value*)

SetGainMode(*mode*)

SetGammaBright(*value*)

SetGammaPeak(*value*)

SetIFDelay(*delay*)

SetIntensifierGain(*gain*)

SetPowerSavingMode(*mode*)

SetSoftBin(*Sx*, *Sy*)

SetSubArea(*left*, *top*, *right*, *bottom*)

SetTemperature(*temp*)

SetTrigger(*mode*)

SetVideoGain(*gain*)

Snap()

SnapAndReturn()

SnapSequence()

SoftBinImage(*image_pointer*, *Nx*, *Ny*)

UnloadCamDLL()

UpdateSize()

UpdateSizeMax()

PyNTA Models

Models are the place to develop the logic of how the devices are used and how the experiment is performed. You will find models for cameras and daqs. Cameras are heavily used in PyNTA, while DAQs where inherited from a previous incarnation and were left here as a reference and to speed up future developments in which not only a camera has to be controlled.

The model for the experiment is the easiest place where the developer can have a sense of what is going on under-the-hood. You can check, for example, *NPTracking* in order to see the steps that make a tracking measurement. Remember that the program runs with different threads and processes in parallel, and therefore the behavior may not be trivial to understand.

Camera Models

Base Camera Model

Camera class with the base methods. Having a base class exposes the general API for working with cameras. This file is important to keep track of the methods which are exposed to the View. The class BaseCamera should be subclassed when developing new Models for other cameras. This ensures that all the methods are automatically inherited and there are no breaks downstream.

Conventions

Images are 0-indexed. Therefore, a camera with 1024pxx1024px will be used as img[0:1024, 0:1024] (remember Python leaves out the last value in the slice.

Region of Interest is specified with the coordinates of the corners. A full-frame with the example above would be given by X=[0,1023], Y=[0,1023]. Be careful, since the maximum width (or height) of the camera is 1024.

The camera keeps track of the coordinates of the initial pixel. For full-frame, this will always be [0,0]. When this is very important for the GUI, since after the first crop, if the user wants to crop even further, the information has to be referenced to the already cropped area.

Note: **IMPORTANT** Whatever new function is implemented in a specific model, it should be first declared in the BaseCamera class. In this way the other models will have access to the method and the program will keep running (perhaps with non intended behavior though).

---

copyright Aquiles Carattino <aquiles@uetke.com>

license GPLv3, see LICENSE for more details

class pynta.model.cameras.base_camera.**BaseCamera**(*camera*)

Bases: object

ACQUISITION_MODE = {0: 'Single', 1: 'Continuous'}

GetCCDHeight()

Returns: the CCD height in pixels

GetCCDWidth()

Returns the CCD width in pixels

MODE_CONTINUOUS = 1

MODE_SINGLE_SHOT = 0

acquisition_ready()

Checks if the acquisition in the camera is over.

clear_ROI()

Clears the ROI from the camera.

clear_binning()

Clears the binning of the camera to its default value.

configure(*properties: dict*)

getSerialNumber()
Returns the serial number of the camera.

get_acquisition_mode()
Returns the acquisition mode, either continuous or single shot.

get_exposure()
Gets the exposure time of the camera.

get_size()
Returns the size in pixels of the image being acquired. This is useful for checking the ROI settings.

initialize()
Initializes the camera.

read_camera()
Reads the camera

set_ROI(*X*, *Y*)
Sets up the ROI. Not all cameras are 0-indexed, so this is an important place to define the proper ROI.

Parameters

- **X** (*list*) – array type with the coordinates for the ROI X[0], X[1]
- **Y** (*list*) – array type with the coordinates for the ROI Y[0], Y[1]

Returns X, Y lists with the current ROI information

set_acquisition_mode(*mode*)
Set the readout mode of the camera: Single or continuous. :param int mode: One of self.MODE_CONTINUOUS, self.MODE_SINGLE_SHOT :return:

set_binning(*xbin*, *ybin*)
Sets the binning of the camera if supported. Has to check if binning in X/Y can be different or not, etc.

Parameters

- **xbin** –
- **ybin** –

Returns

set_exposure(*exposure*)
Sets the exposure of the camera.

stopAcq()
Stops the acquisition without closing the connection to the camera.

stop_camera()
Stops the acquisition and closes the connection with the camera.

trigger_camera()
Triggers the camera.

Simulated Brownian Diffusion

The SimBrownian class generates synthetic images corresponding to particles performing a thermal Brownian motion. This class was designed in order to make explicit the real parameters of the particles (i.e. the diffusion coefficient in real-space) with the magnification of the microscope. The PSF on the camera is derived from the pixel size, the

wavelength and the numerical aperture of the objective. If the parameter frames_to_accumulate is set to a positive value, memory will be allocated in order to store the first N simulated frames. Once that number is achieved, the frames will keep looping between the already generated data. The decision not to simulate the frames a-priori was to give a better feeling regarding the user-interface. It is also assumed that frames are of datatype np.int16, which is a sensible default for most cameras, although int8 can also be useful for speeding up the trackpy algorithm.

Warning: There is no memory check regarding the accumulated frames.

Todo: Think how to add noise, background, and intensity fluuctuations to the particles.

copyright Aquiles Carattino <aquiles@uetke.com>

license GPLv3, see LICENSE for more details

class pynta.model.cameras.simulate_brownian.**SimBrownian**(*camera_size: tuple = (500, 500)*)

Bases: object

Parameters **camera_size** (*tuple*) – number of pixels in the x and y direction

Returns generated an image with specified noise and particles displaced accordingly

NA = 1
Numerical aperture of the objective, used to estimate PSF

dif_coef = 2
Diffusion coefficient um^2/s

dif_coef_2 = 0

frames_to_accumulate = 0
Number of frames will be accumulated in order to speed up simulations (they will be an infinite loop). Set to 0 in order to avoid accumulating frames

gen_image()

Returns generated image with specified noise and particles position

kernel_size = 5
Number of pixels used to calculate the PSF of the particle

magnification = 30
Magnification of the microscope

next_random_step()

noise = 0
Background noise TODO: Needs to be implemented

num_particles = 100
Number of particles per frame

pixel_size = 5
In real space, um

resize_view(*camera_size*)
SimulateBrownian.resizeView() adjusts the coordinates of the moving particles such that they fit into the desired framesize of the simulated dummycamera

signal = 300
Peak intensity for a particle

time_step = 0.03
Time step in the simulation. Should be set to the acquisition rate if used for a camera, seconds

wavelength = 0.5
um, used to estimate PSF

Dummy Camera Model

Dummy camera class for testing GUI and other functionality. This specific version generates randomly diffusing particles. However, the settings are controlled in a different class, *SimBrownian*.

Todo: The camera defines plenty of parameters that are not used or that they are confusing later on. Rasing exceptions does not happen even if trying to extend beyond the maximum dimensions of the CCD.

Todo: The parameters for the simulation of the brownian motion should be made explicitly here, in such a way that can be used from within the config file as well.

Todo: Some of the methods do not return the same datatype as the real models

---

copyright Aquiles Carattino <aquiles@uetke.com>

license GPLv3, see LICENSE for more details

class pynta.model.cameras.dummy_camera.**Camera**(*camera*)
Bases: *pynta.model.cameras.base_camera.BaseCamera*

GetCCDHeight()

Returns The CCD height in pixels

GetCCDWidth()

Returns The CCD width in pixels

MODE_CONTINUOUS = 1

MODE_SINGLE_SHOT = 0

acquisition_ready()
Checks if the acquisition in the camera is over.

getSerialNumber()
Returns the serial number of the camera.

get_acquisition_mode()
Returns the acquisition mode, either continuous or single shot.

get_exposure()
Gets the exposure time of the camera.

get_size()

Returns Returns the size in pixels of the image being acquired. This is useful for checking the ROI settings.

initialize()
Initializes the camera.

read_camera()
Reads the camera

set_ROI(*X*, *Y*)
Sets up the ROI. Not all cameras are 0-indexed, so this is an important place to define the proper ROI.

Parameters

- **X** – array type with the coordinates for the ROI X[0], X[1]
- **Y** – array type with the coordinates for the ROI Y[0], Y[1]

Returns

set_acquisition_mode(*mode*)
Set the readout mode of the camera: Single or continuous.

Param int mode: One of self.MODE_CONTINUOUS, self.MODE_SINGLE_SHOT

set_binning(*xbin*, *ybin*)
Sets the binning of the camera if supported. Has to check if binning in X/Y can be different or not, etc.

Param xbin: binning in x

Param ybin: binning in y

set_exposure(*exposure*)
Sets the exposure of the camera.

stopAcq()
Stops the acquisition without closing the connection to the camera.

stop_camera()
Stops the acquisition and closes the connection with the camera.

trigger_camera()
Triggers the camera.

Basler Camera Model

Model to adapt PyPylon to the needs of PyNTA. PyPylon is only a wrapper for Pylon, thus the documentation has to be found in the folder where Pylon was installed. It refers only to the C++ documentation, which is very extensive, but not necessarily clear.

Some assumptions

The program forces software trigger during *initialize()*.

class pynta.model.cameras.basler.**Camera**(*camera*)
Bases: *pynta.model.cameras.base_camera.BaseCamera*

GetCCDHeight() → int
Get the full height (in pixels) of the camera sensor.

Return int Maximum height

Deprecated since version 0.1.3: Use self.max_height instead

GetCCDWidth() → int
Get the full width of the camera sensor.

Return int Maximum width

Deprecated since version 0.1.3: Use self.max_width instead

clear_ROI()
Resets the ROI to the maximum area of the camera

get_exposure() → pint.quantity.build_quantity_class.<locals>.Quantity
Gets the exposure time of the camera.

get_size() → Tuple[int, int]
Get the size of the current Region of Interest (ROI). Remember that the actual size may be different from the size that the user requests, given that not all cameras accept any pixel. For example, Basler has some restrictions regarding corner pixels and possible widths.

Return tuple (Width, Height)

initialize()
Initializes the communication with the camera. Get's the maximum and minimum width. It also forces the camera to work on Software Trigger.

> **Warning:** It may be useful to integrate other types of triggers in applications that need to synchronize with other hardware.

read_camera()
Reads the camera

set_ROI(*X: Tuple[int, int], Y: Tuple[int, int]*) → Tuple[int, int]
Set up the region of interest of the camera. Basler calls this the Area of Interest (AOI) in their manuals. Beware that not all cameras allow to set the ROI (especially if they are not area sensors). Both the corner positions and the width/height need to be multiple of 4. Compared to Hamamatsu, Baslers provides a very descriptive error warning.

Parameters

- **X** (*tuple*) – Horizontal limits for the pixels, 0-indexed and including the extremes. You can also check *Base Camera* To select, for example, the first 100 horizontal pixels, you would supply the following: (0, 99)
- **Y** (*tuple*) – Vertical limits for the pixels.

set_acquisition_mode(*mode*)
Set the readout mode of the camera: Single or continuous. :param int mode: One of self.MODE_CONTINUOUS, self.MODE_SINGLE_SHOT :return:

set_exposure(*exposure: pint.quantity.build_quantity_class.<locals>.Quantity*) → pint.quantity.build_quantity_class.<locals>.Quantity
Sets the exposure of the camera.

stop_camera()
Stops the acquisition and closes the connection with the camera.

trigger_camera()
Triggers the camera.

Hamamatsu Model

Model class for controlling Hamamatsu cameras via de DCAM-API. At the time of writing this class, little documentation on the DCAM-API was available. Hamamatsu has a different time schedule regarding support of their own API. However, Zhuang's lab Github repository had a python driver for the Orca camera and with a bit of tinkering things worked out.

DCAM-API relies mostly on setting parameters into the camera. The correct data type of each parameter is not well documented; however it is possible to print all the available properties and work from there. The properties are stored in a filed named params.txt next to the *Hamamatsu Driver*

Note: When setting the ROI, Hamamatsu only allows to set multiples of 4 for every setting (X,Y and vsize, hsize). This is checked in the function. Changing the ROI cannot be done directly, one first needs to disable it and then re-enable.

---

copyright Aquiles Carattino <aquiles@uetke.com>

license GPLv3, see LICENSE for more details

class pynta.model.cameras.hamamatsu.**Camera**(*camera*)
Bases: *pynta.model.cameras.base_camera.BaseCamera*

GetCCDHeight()
Returns The CCD height in pixels

GetCCDWidth()
Returns The CCD width in pixels

MODE_CONTINUOUS = 1

MODE_EXTERNAL = 2

MODE_SINGLE_SHOT = 0

acquisition_ready()
Checks if the acquisition in the camera is over.

getSerialNumber()
Returns the serial number of the camera.

get_acquisition_mode()
Returns the acquisition mode, either continuous or single shot.

get_exposure()
Gets the exposure time of the camera.

get_size()
Returns the size in pixels of the image being acquired. This is useful for checking the ROI settings.

initialize()
Initializes the camera.

Returns

read_camera()
Reads the camera

set_ROI(*X*, *Y*)
Sets up the ROI. Not all cameras are 0-indexed, so this is an important place to define the proper ROI. X – array type with the coordinates for the ROI X[0], X[1] Y – array type with the coordinates for the ROI Y[0], Y[1]

set_acquisition_mode(*mode*)
Set the readout mode of the camera: Single or continuous. Parameters mode : int One of self.MODE_CONTINUOUS, self.MODE_SINGLE_SHOT

set_exposure(*exposure*)
Sets the exposure of the camera.

stopAcq()
Stops the acquisition without closing the connection to the camera.

stop_camera()
Stops the acquisition and closes the connection with the camera.

trigger_camera()
Triggers the camera.

Photonic Science GEV Model

Model for Photonic Science GEV Cameras. The model just implements the basic methods defined in the *BaseCamera()* using a Photonic Sicence camera. The controller for this camera is `photonicscience`

copyright Aquiles Carattino <aquiles@uetke.com>

license GPLv3, see LICENSE for more details

class pynta.model.cameras.psi.**Camera**(*camera*)
Bases: *pynta.model.cameras.base_camera.BaseCamera*

GetCCDHeight()
Gets the CCD height.

GetCCDWidth()
Gets the CCD width.

getParameters()
Returns all the parameters passed to the camera, such as exposure time, ROI, etc. Not necessarily the parameters go to the hardware, it may be that some are just software related.

Return dict keyword => value.

Todo: Implement this method

---

get_size()
Returns the size in pixels of the image being acquired.

initialize()
Initializes the camera.

Todo: *pynta.controller.devices.photonicscience.scmoscam.GEVSCMOS.SetGainMode()* behaves unexpectedly. One is forced to set the gain mode twice to have it right. So far, this is the only way to prevent the *weird lines* from appearing. Checking the meaning of the gains is a **must**.

---

read_camera()
Reads the camera

set_ROI(*X*, *Y*)
Sets up the ROI.

set_exposure(*exposure*)
Sets the exposure of the camera.

Todo: Include units for ensuring the proper exposure time is being set.

---

setupCamera(*params*)
Setups the camera with the given parameters.

- params['exposureTime']
- params['binning']
- params['gain']
- params['frequency']
- params['ROI']

Todo: not implemented

---

stopAcq()
Stop the acquisition even if ongoing.

stop_camera()
Stops the acquisition and closes the camera. This has to be called before quitting the program.

trigger_camera()
Triggers the camera.

Data Acquisition Cards Models

Available DAQS

Dummy DAQ

DAQ model for testing GUI and other functionalities. Based on the skeleton. It does not interact with any real device, it just generates random data and accepts inputs which have no effect.

copyright Aquiles Carattino <aquiles@uetke.com>

license GPLv3, see LICENSE for more details

class pynta.model.daqs.daq_dummy.**DAQDummy**(*dev_number=None*)
Bases: *pynta.model.daqs.skeleton.DaqBase*

fastTimetrace(*conditions*)
Acquires a fast timetrace of the selected devices. conditions['devs'] – list of devices to monitor conditions['accuracy'] – accuracy in milliseconds. conditions['time'] – total time of acquisition for each channel in seconds.

getAnalog(*conditions*)
Gets the analog values acquired with the triggerAnalog function. conditions – dictionary with the number of points ot be read

readMonitor()
Reads the monitor values of all the channels specified.

startMonitor(*conditions*)
Starts continuous acquisition of the specified channels with the specified timing interval. conditions['devs'] – list of devices to monitor conditions['accuracy'] – accuracy for the monitor. If not defined defaults to 0.1s

stopMonitor()
Stops all the tasks related to the monitor.

triggerAnalog(*conditions*)
Triggers an analog measurement. It does not read the value. conditions – a dictionary with the needed parameters for an analog acquisition.

Base Model for DAQs

Base model that makes explicit the API for working with DAQ cards. Every new DAQ card should inherit this model. This allows to check the argument type, for example, but they also guarantee forward-compatibility in case new methods are developed.

Note: **IMPORTANT** Whatever new function is implemented in a specific model, it should be first declared in the laserBase class. In this way the other models will have access to the method and the program will keep running (perhaps with non intended behavior though).

---

copyright Aquiles Carattino <aquiles@uetke.com>

license AGPLv3, see LICENSE for more details

class pynta.model.daqs.skeleton.**DaqBase**(*dev_number*)
Bases: object

analog_input_setup(*conditions*)
conditions – a dictionary with the needed parameters for an analog acquisition.

analog_output_setup(*conditions*)
Sets up an analog output task.

Parameters **conditions** –

Returns

read_analog(*task_number*, *conditions*)
Gets the analog values acquired with the triggerAnalog function.

trigger_analog(*task_number*)
Triggers an analog measurement. It does not read the value.

conditions – dictionary with the number of points ot be read

Model for Experiments

Tracking in Hollow Optical Fibers

This experiment is very similar to the nanoparticle tracking experiment, but everything happens inside a hollow optical fiber. Some steps are different; for example, the user first has to focus the image of the camera on the top part of the

setup in order to couple the laser to the optical fiber. Then, the user needs to maximise the background signal on the microscope's camera in order to fine-tune the coupling.

The measurement is essentially a 1-D measurement of diffusion, in which equations need to be adapted for including diffusion in a cylinder.

copyright Aquiles Carattino <aquiles@uetke.com>

license GPLv3, see LICENSE for more details

class pynta.model.experiment.dispertech.fiber_tracking.**FiberTracking**(*filename=None*)
Bases: *pynta.model.experiment.base_experiment.BaseExperiment*

Experiment class for performing nanoparticle tracking analysis inside a hollow optical fiber.

BACKGROUND_CORRECTION = (('single_snap', 0), ('rolling_avg', 1))

ROLLING_AVERAGE = 1
Uses a window of averages to correct the background

SINGLE_SNAP_BKG = 0
Uses only one image to correct the background

initialize()
Initializes all the devices at the same time using threads.

initialize_cameras()
The experiment requires two cameras, and they need to be initialized before we can proceed with the measurement. This requires two entries in the config file with names `camera_fiber`, which refers to the camera which monitors the end of the fiber and `camera_microscope`, which is the one that is used to do the real measurement.

initialize_electronics()
Routine to initialize the rest of the electronics. For example, the LED's can be set to a default on/off state. This is also used to measure the temperature.

initialize_mirror()
Routine to initialize the movable mirror. The steps in this method should be those needed for having the mirror in a known position (i.e. the homing procedure).

Nanoparticle Tracking

Nanoparticle tracking is a technique that allows to measure the size of very small objects. The core idea is that by locating objects subject to brownian motion, it is possible to reconstruct their movement, which in turn can be fitted to a model which depends on properties of the medium (i.e. viscosity) and on the diameter of the particles.

Nanoparticle tracking analysis (NTA) is a common name used to the entire cycle of data acquisition, localization, and analysis. Commercial devices such as NanoSight and ZetaView provide a closed-solution to the problem. PyNTA aims at providing a superior approach, allowing researchers to have real-time information on the sample studied and a completely transparent approach regarding algorithms used.

copyright Aquiles Carattino <aquiles@uetke.com>

license GPLv3, see LICENSE for more details

class pynta.model.experiment.nanoparticle_tracking.np_tracking.**NPTracking**(*filename=None*)
Bases: *pynta.model.experiment.base_experiment.BaseExperiment*

Experiment class for performing a nanoCET measurement.

BACKGROUND_NO_CORRECTION = 0

BACKGROUND_SINGLE_SNAP = 1

calculate_waterfall(*image*)
A waterfall is the product of summing together all the vertical values of an image and displaying them as lines on a 2D image. It is how spectrometers normally work. A waterfall can be produced either by binning the image in the vertical direction directly at the camera, or by doing it in software. The first has the advantage of speeding up the readout process. The latter has the advantage of working with any camera. This method will work either with 1D arrays or with 2D arrays and will generate a stack of lines.

check_background()
Checks whether the background is set.

clear_roi()
Clears the region of interest and returns to the full frame of the camera.

empty_locations_queue()
Empties the queue with location data.

empty_saver_queue()
Empties the queue where the data from the movie is being stored.

finalize()
Needs to be overridden by child classes.

initialize_camera()
Initializes the camera to be used to acquire data. The information on the camera should be provided in the configuration file and loaded with `load_configuration()`. It will load the camera assuming it is located in nanoparticle_tracking/model/cameras/[model].

Todo: Define how to load models from outside of PyNTA. E.g. from a user-specified folder.

---

link_particles()
Starts linking the particles while the acquisition is in progress.

property link_particles_running

localize_particles_image(*image=None*)
Localizes particles based on trackpy. It is a convenience function in order to use the configuration parameters instead of manually passing them to trackpy.

save_image()
Saves the last acquired image. The file to which it is going to be saved is defined in the config.

save_stream()
Saves the queue to a file continuously. This is an async function, that can be triggered before starting the stream. It relies on the multiprocess library. It uses a queue in order to get the data to be saved. In normal operation, it should be used together with `add_to_stream_queue`.

property save_stream_running

set_roi(*X*, *Y*)
Sets the region of interest of the camera, provided that the camera supports cropping. All the technicalities should be addressed on the camera model, not in this method.

Parameters

- **X** (*list*) – horizontal position for the start and end of the cropping
- **Y** (*list*) – vertical position for the start and end of the cropping

Raises `ValueError` – if either dimension of the cropping goes out of the camera total amount of pixels

Returns The final cropping dimensions, it may be that the camera limits the user desires

snap()
Snap a single frame.

snap_background()
Snaps an image that will be stored as background.

start_free_run()
Starts continuous acquisition from the camera, but it is not being saved. This method is the workhorse of the program. While this method runs on its own thread, it will broadcast the images to be consumed by other methods. In this way it is possible to continuously save to hard drive, track particles, etc.

start_linking_locations()

start_saving_location()

start_tracking()
Starts the tracking of the particles

stop_free_run()
Stops the free run by setting the `_stop_event`. It is basically a convenience method to avoid having users dealing with somewhat lower level threading options.

stop_link_particles()
Stops the linking process.

stop_linking_locations()

stop_save_stream()
Stops saving the stream.

stop_saving_location()

stop_tracking()

sysexcept(*exc_type*, *exc_value*, *exc_traceback*)

property temp_locations

Experiment Models

Experiment models make explicit the different steps needed to perform a measurement. The Base Experiment class defines some methods that are transversal to all experiments (such as loading a configuration file) but individual experiment models can overwrite this methods to develop custom solutions.

Moreover, PyNTA introduces the PUB/SUB pattern in order to exchange information between different parts of the program in a flexible and efficient way. You can find more information on *publisher* and *subscriber*.

copyright Aquiles Carattino <aquiles@uetke.com>

license GPLv3, see LICENSE for more details

base_experiment.py

Base class for the experiments. `BaseExperiment` defines the common patterns that every experiment should have. Importantly, it starts an independent process called publisher, that will be responsible for broadcasting messages that

are appended to a queue. The messages rely on the pyZMQ library and should be tested further in order to assess their limitations. The general pattern is that of the PUB/SUB, with one publisher and several subscribers.

The messages should include a *topic* and data. For this, the elements in the queue should be dictionaries with two keywords: **data** and **topic**. `data['data']` will be serialized through the use of cPickle, and is handled automatically by pyZQM through the use of `send_pyobj`. The subscribers should be aware of this and use either unpickle or `recv_pyobj`.

In order to stop the publisher process, the string `'stop'` should be placed in `data['data']`. The message will be broadcast and can be used to stop other processes, such as subscribers.

Todo: Check whether the serialization of objects with cPickle may be a bottleneck for performance.

---

copyright Aquiles Carattino <aquiles@uetke.com>

license GPLv3, see LICENSE for more details

class pynta.model.experiment.base_experiment.**BaseExperiment**(*filename=None*)
Bases: object

Base class to define experiments. Should keep track of the basic methods needed regardless of the experiment to be performed. For instance, a way to start and a way to finalize a measurement.

property alive_threads

clear_threads()
Keep only the threads that are alive.

connect(*method*, *topic*, **args*, ***kwargs*)
Async method that connects the running publisher to the given method on a specific topic.

Parameters

- **method** – method that will be connected on a given topic
- **topic** (*str*) – the topic that will be used by the subscriber to discriminate what information to collect.
- **args** – extra arguments will be passed to the subscriber, which in turn will pass them to the function
- **kwargs** – extra keyword arguments will be passed to the subscriber, which in turn will pass them to the function

property connections

finalize()
Needs to be overridden by child classes.

property list_alive_threads

load_configuration(*filename*)
Loads the configuration file in YAML format.

Parameters **filename** (*str*) – full path to where the configuration file is located.

Raises **FileNotFoundError** – if the file does not exist.

property num_threads

set_up()
Needs to be overridden by child classes.

stop_publisher()
Puts the proper data to the queue in order to stop the running publisher process

stop_subscribers()
Puts the proper data into every alive subscriber in order to stop it.

update_config(***kwargs*)

Publisher

Publishers are responsible for broadcasting the message over the ZMQ PUB/SUB architecture. The publisher runs continuously on a separated process and grabs elements from a queue, which in turn are sent through a socket to any other processes listening.

Todo: In the current implementation, data is serialized for being added to a Queue, then deserialized by the publisher and serialized again to be sent. These three steps could be simplify into one if, for example, one assumes that objects where pickled. There is also a possibility of assuming numpy arrays and using a zero-copy strategy.

---

copyright Aquiles Carattino <aquiles@uetke.com>

license GPLv3, see LICENSE for more details

class pynta.model.experiment.publisher.**Publisher**(*port=None*)
Bases: object

Publisher class in which the queue for publishing messages is defined and also a separated process is started. It is important to have a new process, since the serialization/deserialization of messages from the QUEUE may be a bottleneck for performance.

empty_queue()
If the publisher stops before broadcasting all the messages, the Queue may still be using some memory. This method is simply getting all the elements in order to free memory. Can be useful for garbage collection or better control of the downstream program.

join(*timeout=0*)

property port

publish(*topic*, *data*)
Adapts the data to make it faster to broadcast

Parameters

- **topic** (*str*) – Topic in which to publish the data
- **data** – Data to be published

Returns None

start()
Start a new process that will be responsible for broadcasting the messages.

Todo: Find a way to start the publisher on a different port if the one specified is in use.

---

stop()

pynta.model.experiment.publisher.**publisher**(*queue*, *event*, *port*)
Simple method that starts a publisher on the port 5555.

Parameters

- **queue** (*multiprocessing.Queue*) – Queue of messages to be broadcasted
- **event** (*multiprocessing.Event*) – Event to stop the publisher
- **port** (*int*) – port in which to broadcast data

Todo: The publisher's port should be determined in a configuration file.

Deprecated since version 0.1.0.

Subscriber

Example script on how to run separate processes to process the data coming from a publisher like the one on `publisher.py`. The first process just grabs the frame and puts it in a Queue. The Queue is then used by another process in order to analyse, process, save, etc. It has to be noted that on UNIX systems, getting from a queue with `Queue.get()` is particularly slow, much slower than serializing a numpy array with cPickle.

`pynta.model.experiment.subscriber.`**`subscribe`**(*port*, *topic*)

`pynta.model.experiment.subscriber.`**`subscriber`**(*func*, *topic*, *event*, **args*, ***kwargs*)

Experiment Configuration

Class which holds some parameters that need to be used throughout the lifetime of a program. Keeping them in a separate class gives great flexibility, because it allows to overwrite values at run time.

Todo: Changes to config at runtime will have no effect on other processes. Find a way in which config can broadcast itself to all the instances of the class

---

copyright Aquiles Carattino <aquiles@uetke.com>

license GPLv3, see LICENSE for more details

class `pynta.model.experiment.config.`**`Config`**
Bases: `object`

View

View

In the broad definition, the View should provide all the tools for the interaction between the user and the computer. These could be both through the command line or through a Graphical User Interface. In practice, however, command-line interfaces appear naturally once the models and controllers are properly developed. It is also possible to use Jupyter notebooks to interface with devices and perform experiments. Therefore, the View will be fundamentally responsible for generating graphical applications.

copyright Aquiles Carattino <aquiles@uetke.com>

license GPLv3, see LICENSE for more details

GUI modules

GUI

Developing a GUI for an application is a delicated task. On the one hand, developers want to deliver quick solutions. On the other, users are exposed to programs developed over decades, by large teams, including designers. Therefore, it is very hard for a single developer to obtain user interfaces as beautiful as the ones you can get from commercial suppliers.

However, it is possible to build a collection of tools that can be reused and that can generate a much more consistent result throughout different applications. The approach PyNTA follows is to develop GUI's using Qt5. There are different ports of Qt for Python, such as PySide and PyQt. For the time being, PyNTA is based on PyQt5, but this can change without previous notice.

To develop a Graphical User Interface (GUI), we have opted to use Qt Designer, and the files are loaded directly to the class through `uic.loadUi` instead of compiling the files into a Python class. On the one hand, this simplifies the cycle for updating the interface, on the other it does not make explicit which methods are available. Compiling the files is up to the developer, but the *official* approach is to use only the UI files generated by Qt Designer.

Todo: Different wrappers of Qt for Python expose the same API in different ways. We should explore using an intermediate package to unify the use of Qt.

---

copyright Aquiles Carattino <aquiles@uetke.com>

license GPLv3, see LICENSE for more details

class pynta.view.GUI.camera_focusing.**CameraFocusing**(*experiment=None*, *parent=None*)

Bases: `PyQt5.QtWidgets.QMainWindow`

class pynta.view.GUI.camera_viewer_widget.**CameraViewerWidget**(*parent=None*)

Bases: `PyQt5.QtWidgets.QWidget`

Widget for holding the images generated by the camera.

do_auto_scale()

draw_target_pointer(*locations*)

gets an image and draws a circle around the target locations.

Parameters **locations** (*DataFrame*) – DataFrame generated by trackpy's locate method. It only requires columns *x* and *y* with coordinates.

get_roi_values()

Get's the ROI values in camera-space. It keeps track of the top left corner in order to update the values before returning. :return: Position of the corners of the ROI region assuming 0-indexed cameras.

keyPressEvent(*key*)

Triggered when there is a key press with some modifier. Shift+C: Removes the cross hair from the screen Ctrl+C: Emits a specialTask signal Ctrl+V: Emits a stopSpecialTask signal These last two events have to be handeled in the mainWindow that implemented this widget.

mouseMoved(*arg*)

Updates the position of the cross hair. The mouse has to be moved while pressing down the Ctrl button.

set_roi_lines(*X*, *Y*)

setup_cross_cut(*max_size*)

Set ups the horizontal line for the cross cut.

setup_cross_hair(*max_size*)
Sets up a cross hair.

setup_mouse_tracking()

setup_roi_lines(*max_size*)
Sets up the ROI lines surrounding the image.

Parameters **max_size** (*list*) – List containing the maximum size of the image to avoid ROIs bigger than the CCD.

specialTask

stopSpecialTask

update_image(*image*)

class pynta.view.GUI.config_tracking_widget.**ConfigTrackingWidget**(*parent=None*)
Bases: PyQt5.QtWidgets.QWidget

apply_config

flags = 262144

get_config()

print_config(*config*)

revert_changes()

update_config(*config*)

Parameters **config** (*dict*) – Dictionary with the new values

Configuration Widget

Simple widget for storing the parameters of the UUTrack.Model._session. It creates and populates tree thanks to the UUTrack.Model._session._session.getParams(). The widget has two buttons, one that updates the session by emitting a *signal* to the main thread and another the repopulates the tree whith the available parameters.

Todo: Remove the printing to screen of the parameters once the debugging is done.

class pynta.view.GUI.config_widget.**ConfigWidget**(*parent=None*)
Bases: PyQt5.QtWidgets.QWidget

Widget for configuring the main parameters of the camera.

apply_config

flags = 262144

get_config()

revert_changes()

update_config(*config*)

Parameters **config** (*dict*) – Dictionary with the new values

class pynta.view.GUI.histogram_tracks_widget.**HistogramTracksWidget**(*parent=None*)
Bases: PyQt5.QtWidgets.QWidget

In this example we draw two different kinds of histogram.

class pynta.view.GUI.histogram_widget.**HistogramWidget**(*parent=None*)
Bases: PyQt5.QtWidgets.QWidget

update_distribution(*values*)

class pynta.view.GUI.main_window.**MainWindowGUI**(*refresh_time=30*)
Bases: PyQt5.QtWidgets.QMainWindow

background_reduction()

calculate_histogram()

clear_roi()

closeEvent(*self*, *QCloseEvent*)

configure()

connect_actions()

connect_buttons()

connect_signals()

initialize_camera()

load_config()

load_data()

safe_close()

save_image()

set_roi()

show_about()

show_cheat_sheet()

snap()

start_continuous_saves()

start_linking()

start_movie()

start_saving_tracks()

start_tracking()

stop_continuous_saves()

stop_linking()

stop_movie()

stop_saving_tracks()

stop_tracking()

update_config(*config*)

update_gui()

update_tracking_config(*config*)

update_tracks()

In this example we draw two different kinds of histogram.

class pynta.view.GUI.tracks_widget.**TracksWidget**(*parent=None*)
Bases: PyQt5.QtWidgets.QWidget

plot_trajectories(*locations*)

Parameters **locations** – Dataframe of locations as given by trackpy

class pynta.view.main.**MainWindow**(*experiment*)
Bases: *pynta.view.GUI.main_window.MainWindowGUI*

calculate_histogram()

clear_roi()

closeEvent(*self*, *QCloseEvent*)

initialize_camera()

save_image()

set_roi()

snap()

start_continuous_saves()

start_linking()

start_movie()

start_saving_tracks()

start_tracking()

stop_continuous_saves()

stop_linking()

stop_movie()

stop_saving_tracks()

update_config(*config*)

update_gui()

update_histogram(*values*)

update_tracking_config(*config*)

update_tracks()

Subscriber Thread

Allows to transform topics coming from a socket (ZMQ) to Qt signals that can be connected to any method, etc.

copyright Aquiles Carattino <aquiles@uetke.com>

license GPLv3, see LICENSE for more details

class pynta.view.subscriber_thread.**SubscriberThread**(*port*, *topic*)
Bases: PyQt5.QtCore.QThread

data_received

run(*self*)

pynta.tests package

Submodules

pynta.tests.test_controllers module

Test the available controllers for the proper structure. Since controllers are inherently impossible to test on all systems, the only reasonable thing to do is to check for consistency in methods.

Todo: Define the minimum structure that makes a controller

Todo: Implement tests to assert whether the controllers have the needed methods

copyright Aquiles Carattino <aquiles@uetke.com>

license AGPLv3, see LICENSE for more details

pynta.tests.test_examples module

Check whether the examples are able to perform an experiment with dummy devices.

Todo: Find a way of testing the UI, is it possible within Travis? may be useful to check what PyQtGraph is doing for the tests.

copyright Aquiles Carattino <aquiles@uetke.com>

license AGPLv3, see LICENSE for more details

pynta.tests.test_models module

Models can be checked for structure, but also models which depend on dummy devices can be tested thoroughly.

Todo: Define minimum structure for each kind of model

Todo: Define tests for dummy-based models.

copyright Aquiles Carattino <aquiles@uetke.com>

license AGPLv3, see LICENSE for more details.

pynta.tests.test_zmq module

Test the limits of the pyZMQ library

Example on how to test the bandwidth limits for pyZMQ. We create a Queue with a given number of frames and an empty queue. The first is fed to the publisher, while the second is used by the subscriber to append the data. In this way we can check whether all the frames arrived in order and the bandwidth limitation for ZMQ.

If any attempts at improve the architecture of publisher/subscriber are tried, this test is going to be useful to determine whether there is a real improvement in performance.

Note: Replacing send_pyobj by the nocopy option is faster, however it also requires to reshape the data, which adds an overhead and eventually reaches equivalent bandwidths.

Warning: This script consumes a lot of memory because it allocates a queue with thousands of elements in it. If you have a more limited system, consider lowering the amount of elements or changing the data type.

pynta.tests.test_zmq.**main**(*num_data_frames=1000*, *dtype=<class 'numpy.uint16'>*)

pynta.tests.test_zmq.**test_func**(*data*, *queue*)

Module contents

pynta.tools package

Submodules

pynta.tools.worker_thread module

work_thread.py

Running the acquisition on a separate thread gives a lot of flexibility when designing the program. It comes, however with some potential risks. First, threads are still running on the same Python interpreter. Therefore they do not overcome the GIL limitations. They are able to share memory, which makes them transparent to less experienced users. Potentially, different threads will access the same resources (i.e. devices) creating a clash. It is hard to implement semaphores or locks for every possible scenario, especially with devices such as cameras which can run without user intervention for long periods of time.

copyright Aquiles Carattino <aquiles@uetke.com>

license AGPLv3, see LICENSE for more details

class pynta.tools.worker_thread.**WorkerThread**(*camera*, *keep_alive=False*)
Bases: threading.Thread

Thread for acquiring from the camera. If the exposure time is long, running on a separate thread will enable to perform other tasks. It also allows to acquire continuously without freezing the rest of the program.

Todo: QThreads are much handier than Python threads. Should we put Qt as a requirement regardless of whether the program runs on CLI or UI mode?

---

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

Module contents

pynta.util package

Submodules

pynta.util.circular_buffer module

class pynta.util.circular_buffer.**circularlist**(*size*)
Bases: object

append(*value*)
Append an element

pynta.util.importer module

pynta.util.importer.**from_here**(**args*)

pynta.util.log module

nanoparticle_tracking.util.log.py

Adding log capacities to PyNTA

copyright Aquiles Carattino <aquiles@uetke.com>

license AGPLv3, see LICENSE for more details

pynta.util.log.**get_logger**(*name='nanoparticle_tracking', add_null_handler=True*)

pynta.util.log.**log_to_file**(*filename, level=20, fmt=None*)

pynta.util.log.**log_to_screen**(*level=20, fmt=None*)

Module contents

pynta.util.**get_logger**(*name='nanoparticle_tracking', add_null_handler=True*)

pynta.util.**log_to_file**(*filename, level=20, fmt=None*)

pynta.util.**log_to_screen**(*level=20, fmt=None*)

Exceptions

exception pynta.exceptions.exceptions.**PublisherNotStarted**
Bases: Exception

4.7 List of Todo's

Todo: I'm using the "old" functions because these are documented. Switch to the "new" functions at some point.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/controller/devices/hamamatsu/hamamatsu_camera.py:docstring of pynta.controller.devices.hamamatsu.hamamatsu_camera, line 10.)

Todo: How to stream 2048 x 2048 at max frame rate to the flash disk? The Hamamatsu software can do this.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/controller/devices/hamamatsu/hamamatsu_camera.py:docstring of pynta.controller.devices.hamamatsu.hamamatsu_camera, line 12.)

Todo: Use lockbits (and unlockbits) to avoid memory clashes? This would probably also involve some kind of reference counting scheme.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/controller/devices/hamamatsu/hamamatsu_camera.py:docstring of pynta.controller.devices.hamamatsu.hamamatsu_camera.HamamatsuCameraMR, line 12.)

Todo: It does not always seem to block? The length of frames can be zero. Are frames getting dropped? Some sort of race condition?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/controller/devices/hamamatsu/hamamatsu_camera.py:docstring of pynta.controller.devices.hamamatsu.hamamatsu_camera.HamamatsuCameraMR.getFrames, line 4.)

Todo: The coding style is not in line with the rest of PyNTA. The GEVSMOS class can be cleaned up and documented

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/controller/devices/photonicscience/scmoscam.py:docstring of pynta.controller.devices.photonicscience.scmoscam, line 9.)

Todo: It is also possible to define the methods as `@abstractmethods` which will automatically raise an exception. It may be worth exploring this possibility if there are several developers involved.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/docs/developers/index.rst, line 49.)

Todo: The camera defines plenty of parameters that are not used or that they are confusing later on. Rasing exceptions does not happen even if trying to extend beyond the maximum dimensions of the CCD.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/model/cameras/dummy_camera.py:docstring of pynta.model.cameras.dummy_camera, line 6.)

Todo: The parameters for the simulation of the brownian motion should be made explicitly here, in such a way that can be used from within the config file as well.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/model/cameras/dummy_camera.py:docstring of pynta.model.cameras.dummy_camera, line 9.)

Todo: Some of the methods do not return the same datatype as the real models

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/model/cameras/dummy_camera.py:docstring of pynta.model.cameras.dummy_camera, line 12.)

Todo: Implement this method

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/model/cameras/psi.py:docstring of pynta.model.cameras.psi.Camera.getParameters, line 7.)

Todo: *pynta.controller.devices.photonicscience.scmoscam.GEVSCMOS.SetGainMode()* behaves unexpectedly. One is forced to set the gain mode twice to have it right. So far, this is the only way to prevent the *weird lines* from appearing. Checking the meaning of the gains is a **must**.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/model/cameras/psi.py:docstring of pynta.model.cameras.psi.Camera.initialize, line 3.)

Todo: Include units for ensuring the proper exposure time is being set.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/model/cameras/psi.py:docstring of pynta.model.cameras.psi.Camera.set_exposure, line 3.)

Todo: not implemented

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/model/cameras/psi.py:docstring of pynta.model.cameras.psi.Camera.setupCamera, line 9.)

Todo: Think how to add noise, background, and intensity fluuctuations to the particles.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/model/cameras/simulate_brownian.py:docstring of pynta.model.cameras.simulate_brownian, line 15.)

Todo: Check whether the serialization of objects with cPickle may be a bottleneck for performance.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/model/experiment/base_experiment.py:docstring of pynta.model.experiment.base_experiment, line 16.)

Todo: In the current implementation, data is serialized for being added to a Queue, then deserialized by the publisher and serialized again to be sent. These three steps could be simplify into one if, for example, one assumes that objects where pickled. There is also a possibility of assuming numpy arrays and using a zero-copy strategy.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/model/experiment/publisher.py:docstring of pynta.model.experiment.publisher, line 8.)

Todo: Find a way to start the publisher on a different port if the one specified is in use.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/model/experiment/publisher.py:docstring of pynta.model.experiment.publisher.Publisher.start, line 3.)

Todo: The publisher's port should be determined in a configuration file.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/model/experiment/publisher.py:docstring of pynta.model.experiment.publisher.publisher, line 6.)

Todo: Changes to config at runtime will have no effect on other processes. Find a way in which config can broadcast itself to all the instances of the class

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/model/experiment/config.py:docstring of pynta.model.experiment.config, line 6.)

Todo: Define how to load models from outside of PyNTA. E.g. from a user-specified folder.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/model/experiment/nanoparticle_tracking/np_tracking.py:docstring of pynta.model.experiment.nanoparticle_tracking.np_tracking.NPTracking.initialize_camera, line 5.)

Todo: Define the minimum structure that makes a controller

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/tests/test_controllers.py:docstring of pynta.tests.test_controllers, line 5.)

Todo: Implement tests to assert whether the controllers have the needed methods

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/tests/test_controllers.py:docstring of pynta.tests.test_controllers, line 6.)

Todo: Find a way of testing the UI, is it possible within Travis? may be useful to check what PyQtGraph is doing for the tests.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/tests/test_examples.py:docstring of pynta.tests.test_examples, line 3.)

Todo: Define minimum structure for each kind of model

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/tests/test_models.py:docstring of pynta.tests.test_models, line 4.)

Todo: Define tests for dummy-based models.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/tests/test_models.py:docstring of pynta.tests.test_models, line 5.)

Todo: QThreads are much handier than Python threads. Should we put Qt as a requirement regardless of whether the program runs on CLI or UI mode?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/tools/worker_thread.py:docstring of pynta.tools.worker_thread.WorkerThread, line 4.)

Todo: Different wrappers of Qt for Python expose the same API in different ways. We should explore using an intermediate package to unify the use of Qt.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/view/GUI/__init__.py:docstring of pynta.view.GUI, line 19.)

Todo: Remove the printing to screen of the parameters once the debugging is done.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/python-nta/checkouts/latest/pynta/view/GUI/config_widget.py:docstring of pynta.view.GUI.config_widget, line 6.)

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

D

E

F

G

H

I

J

K

L

M

R

S

T

U

V

W