UUTracking Documentation

Release 0.1

Aquiles Carattino

Apr 13, 2017

Contents:

A powerful interface for scientific cameras and instruments

UUtracking 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 1

Installing

The best place to look for the code of the program is the repository at `https://github.com/aquilesC/UUTrack`. If you need further assistance with the installation of the code, please check *Installation instructions*

CHAPTER 2

Start the program

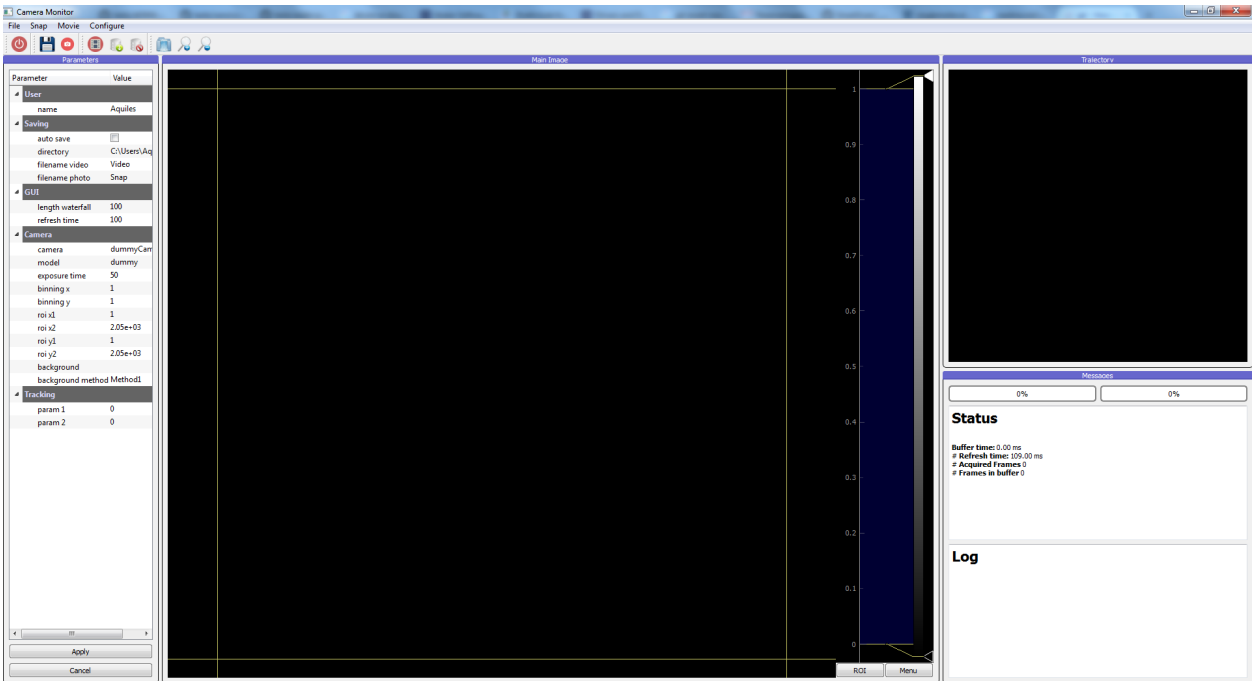
To run the program you can just import the startCamera module from the root of UUTrack and trigger it with a config file:

```
from UUTrack import startCamera

ConfigDir = 'Path/to/config'
ConfigFile = 'config.yml'
startCamera.start(ConfigDir, ConfigFile)
```

Note that the splitting between the config directory and the config file is done to allow users to have different config files in the same directory, for example for different configurations of the setup. It also allows to include a pre-window to select with a GUI the desired configuration and then trigger the `startCamera.start` method.

It is important to have a Config file ready for the program to run properly. You can check the example *Config File*


Camera Monitor
File
Snap
Movie
Configure
Parameters
Parameter
Value
User
name
Aquiles
Saving
auto save
directory
filename video
Video
filename photo
Snap
GUI
length waterfall
100
refresh time
100
Camera
camera
model
dummy
exposure time
50
binning x
1
binning y
1
roi x1
1
roi x2
2.05e+03
roi y1
1
roi y2
2.05e+03
background
background method
Method1
Tracking
param 1
0
param 2
0
Apply
Cancel
Main Image
ROI
Menu
Trajectory
Messages
0%
0%
Status
Buffer time: 0.00 ms
Refresh time: 109.00 ms
Acquired Frames 0
Frames in buffer 0
Log

CHAPTER 3

Changing the code

The program is open source and therefore you can modify all what you want. 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 *Improving the code*, or directly submerge yourself in the documentation of the different classes *UUTrack package*.

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

Note: The naming convention through the code is not very uniform. I've tried to use CamelCase for classes, and underscores for variables, but this has to be sanitized for making it more consistent. Sometimes classes and variables are called the same, making everything very confusing.

UUTrack

Here you find all the documentation.

UUTrack package

Here you find all the documentation of the program. If you are trying to add a new camera, you should start by looking at the *Model package*.

Subpackages

Controller package

The last part of the program are the *controllers* for different devices. The focus of the entire UUTrack program are cameras. Controllers for cameras normally rely on library files (.dll files on Windows) that can be more or less documented. For example `hamamatsu` uses the *DCAM-API*, while `PhotonicScience` uses *scmoscam.dll*. The idea of having a Controller module separated from the Model module is the ability to copy pasting code from other sources. For example the Hamamatsu code is available on Zhuangs lab github repository, while the Photonic Science code was sent by the company itself.

Having separate modules for the controller and the model allows to share code between different setups making it more transparent for the users. For example, one may not need to set the ROI of the camera, therefore should not worry about implementing it. However learning from the *Models* of others can be extremely useful; for instance, Hamamatsu only allows to set ROI parameters that are multiple of 4. Moreover if you don't reset the ROI before changing it, the dll crashes. Photonic Science has its own share with setting the gain.

Between the controllers there is a module named `keysight` that holds the drivers for an oscilloscope and function generator. It works, but was never implemented into the main window. The idea is to use it in the `specialTaskWorker` for generating signals or acquiring fast timetraces.

Subpackages

UUTrack.Controller.devices.hamamatsu.hamamatsu_camera.py

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.

Section author: Hazen Babcock 10/13

exception `UUTrack.Controller.devices.hamamatsu.hamamatsu_camera.`**`DCAMException`**(*message*)
Bases: `Exception`

Camera exceptions.

class `UUTrack.Controller.devices.hamamatsu.hamamatsu_camera.`**`DCAM_PARAM_PROPERTYATTR`**
Bases: `_ctypes.Structure`

The dcam property attribute structure.

class `UUTrack.Controller.devices.hamamatsu.hamamatsu_camera.`**`DCAM_PARAM_PROPERTYVALUETEXT`**
Bases: `_ctypes.Structure`

The dcam text property structure.

class UUTrack.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.

class UUTrack.Controller.devices.hamamatsu.hamamatsu_camera.**HamamatsuCameraMR**(*camera_id*)
Bases: UUTrack.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.

FIXME: 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. FIXME: It does not always seem to block? The length of frames can

be zero. Are frames getting dropped? Some sort of race condition?

return [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.

UUTrack.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.

UUTrack.Controller.devices.PhotonicScience.scmoscam.py

A wrapper class originally written by Perceval Guillou, perceval@photonic-science.com in Py2 and has been tested successfully with scmoscontrol.dll SCMOS Pleora (GEV) control dll (x86)v5.6.0.0 (date modified 10/2/2013)

SaFa @nanoLINX has adapted the wrapper class for a camera control program.

v1.0, 24 feb. 2015

Section author: SaFa <S.Faez@uu.nl>

Module contents

Model package

Model is a subpackage of the UUTrack program. Models define the way the user will interact with the devices. For example when dealing with a camera, one of the most likely actions is to set the exposure time, trigger an acquisition and read the image. How this is achieved is dependent on every camera.

Therefore in *Model package* we will place 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 UUTrack.Model.Cameras.Hamamatsu import camera
```

With:

```
from UUTrack.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`, one has 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, 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.

Subpackages

Inside Model there are subpackages, one per device type (i.e.: Cameras, Oscilloscopes, etc.). And inside each of them, there is a file per brand (i.e. hamamatsu.py, etc.). Importantly, every device type should define a base class that the rest of the classes will inherit. The base class can even be a mockup that only generates random data, but that is enough for testing the rest of the application.

Model.Cameras package

Submodules

UUTrack.Model.Cameras._skeleton.py

Camera class with the skeleton functions. Important to keep track of the methods that are exposed to the View. The class cameraBase should be subclassed when developing new Models. This ensures that all the methods are automatically inherited and there is no breaks downstream.

Note: **IMPORTANT** Whatever new function is implemented in a specific model, it should be first declared in the cameraBase 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).

Section author: Aquiles Carattino <aquiles@aquicarattino.com>

class UUTrack.Model.Cameras._skeleton.**cameraBase**(*camera*)

Bases: object

GetCCDHeight()

Returns: the CCD height in pixels

GetCCDWidth()

Returns the CCD width in pixels

acquisitionReady()

Checks if the acquisition in the camera is over.

clearROI()

Clears the ROI from the camera.

getAcquisitionMode()

Returns the acquisition mode, either continuous or single shot.

getExposure()

Gets the exposure time of the camera.

getSerialNumber()

Returns the serial number of the camera.

getSize()

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

initializeCamera()

Initializes the camera.

readCamera()

Reads the camera

setAcquisitionMode(*mode*)

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

setBinning(*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

setExposure(*exposure*)

Sets the exposure of the camera.

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

Returns

stopAcq()

Stops the acquisition without closing the connection to the camera.

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

triggerCamera()
Triggers the camera.

UUTrack.Model.Cameras.Hamamatsu.py

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 regardin 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.

Section author: Aquiles Carattino <aquiles@aquicarattino.com>

class UUTrack.Model.Cameras.Hamamatsu.**camera**(*camera*)
Bases: *UUTrack.Model.Cameras._skeleton.cameraBase*

GetCCDHeight()
Returns The CCD height in pixels

GetCCDWidth()
Returns The CCD width in pixels

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

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

getExposure()
Gets the exposure time of the camera.

getSerialNumber()
Returns the serial number of the camera.

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

initializeCamera()
Initializes the camera.

Returns

readCamera()
Reads the camera

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

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

setROI(*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]

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

triggerCamera()
Triggers the camera.

UUTrack.Model.Cameras.PSI

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

copyright 2017

Section author: Aquiles Carattino <aquiles@aquicarattino.com>

class `UUTrack.Model.Cameras.PSI.`**camera**(*camera*)
Bases: *UUTrack.Model.Cameras._skeleton.cameraBase*

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

---

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

initializeCamera()
Initializes the camera.

Todo

`UUTrack.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**.

---

readCamera()
Reads the camera

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

Todo

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

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

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.

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

triggerCamera()
Triggers the camera.

Submodules

The Model also defines some important functionalities for the program, like the `_session` and the *workerSaver*. This modules are here because they are general to different implementations of the code and not just to the GUI. For example, if one desires to acquire at very high frame rates, regardless of using a GUI or a CLI, the saving to disk should happen in a parallel fashion.

UUTrack.Model.workerSaver

When working with multi threading in Python it is important to define the function that will be run in a separate thread. workerSaver is just a function that will be moved to a separate, parallel thread to save data to disk without interrupting the acquisition.

Since the workerSaver function will be passed to a different Process (via the *multiprocessing* package) the only way for it to receive data from other threads is via a Queue. The workerSaver will run continuously until it finds a string as the next item.

To understand how the separate process is created, please refer to `movieSave()`

The general principle is

```
>>> filename = 'name.hdf5'
>>> q = Queue()
>>> metadata = _session.serialize() # This prints a YAML-ready version of the session.
>>> p = Process(target=workerSaver, args=(filename, metaData, q,))
>>> p.start()
>>> q.put([1, 2, 3])
>>> q.put('Stop')
>>> p.join()
```

copyright 2017

Section author: Aquiles Carattino <aquiles@aquicarattino.com>

`UUTrack.Model.workerSaver.`**`clearQueue`**`(q)`

Clears the queue by reading it.

Params q Queue to be cleaned.

`UUTrack.Model.workerSaver.`**`workerSaver`**`(fileData, meta, q)`

Function that can be run in a separate thread for continuously save data to disk.

Parameters

- **`fileData`** (*str*) – the path to the file to use.
- **`meta`** (*str*) – Metadata. It is kept as a string in order to provide flexibility for other programs.
- **`q`** (*Queue*) – Queue that will store all the images to be saved to disk.

UUTrack.Model.config

Deprecated since version 0.1.

loads configuration files. It is a relic file and not used anymore.

View Package

View Package is where the GUI lives. But also is where the logic of our program is. **UUTrack** was built as a graphical program for controlling cameras, but in principle many experiments don't need a GUI, a command line interface would suffice. View is the most complex part of the program, since it handles a lot of asynchronous tasks, user interactions and more. The starting point for the View is the module `cameraMain`. The module holds the main window and all the interactions between the different parts of the code.

Threading

Acquiring from cameras can be slow, for example one can set an exposure time of several seconds. It can also be very fast, acquiring an image every couple of milliseconds. The first example poses the problem of how to acquire without freezing the GUI presented to the user. The last example poses the problem of how to keep high acqusition framerates if the user will never see more than 30fps. The solution to both problems is Threading. Qt comes with a very handy threading class that is implemented in the `workThread`. The worker runs in a separate thread and therefore its execution will not block the main GUI. When there is data available, it will emit a signal. This signal will be catch in the main program by the function `getData()`. This function stores the data in a variable called *tempImage*; if the proper parameters are set, the data is accumulated in a Queue.

The refreshing of the GUI happens at a fixed framerate given by a Timer. The function responsible is `updateGUI()`. This function will display the data available in the *tempImage* variable. It is important to note that this ensures a fixed framerate to the user, regardless of the acquisition done by the camera. If the data is being acquired much faster than what the user can see, there is no point at displaying it, and if the acquisition is too slow, there is no point in freezing the interaction until it is fetched.

Threading in Qt is a very powerful tool that has to be implemented in all the GUI programs. It ensures that the main Thread is responsive, while a background thread is busy acquiring, or performing some other operation, for example downloading data from the internet. Python offers threading, but without the signalling capabilities of Qt. Since the program is built around PyQt4 there is no point in not using it.

For stopping a Thread, the best strategy is to change the status of a variable that the thread checks periodically. In the case of `workThread` is *self.keep_acquiring*. This strategy is used in `stopMovie()`. As an example on how to extend this, `specialTaskWorker` implements a tracking algorithm and emits signals accordingly. It is very basic, but it pinpoints the direction that needs to be followed.

Subpackages

UUTrack.View.Camera package

All the visualization of the camera is centralized in this package. It defines the windows and widgets and more importantly the `Worker Thread`.

Submodules

Submodules

UUTrack.config_dir module

config_dir

Just stores a variable with the name of the current directory, that is the base directory of the entire filesystem.

Config File

The config file is a Yaml file that doesn't have predefined needs. Whatever is in it is passed to the session variable of the program. New fields can be added. However the program relies on some of the attributes for proper working; for example the camera model is used to import the proper model. The exposure time is set to the camera at the beginning. The path and filename for saving are also important.:

```
%YAML 1.2
---
# Default parameters for the Tracking program
# All parameters can be changed to accommodate user needs.
# All parameters can be changed at runtime with the appropriate config window
User:
  name: Test Subject

Saving:
  auto_save: False
  directory: C:\data\Testing
```

```
  filename_video: Video # Can be the same filename for video and photo
  filename_photo: Snap

GUI:
  length_waterfall: 100 # Total length of the Waterfall (lines)
  refresh_time: 100 # Refresh rate of the GUI (in ms)

Camera:
  camera: dummyCamera # the camera to use
  model: dummyModel # This hasn't been implemented, but is useful for metadata␣
↪storing.
  exposure_time: 200 # Initial exposure time (in ms)
  binning_x: 1 # Binning
  binning_y: 1
  roi_x1: 0 # Leave at 0 for full camera
  roi_x2: 0
  roi_y1: 0
  roi_y2: 0
  background: '' # Full path to background file, or empty for none.
  background_method: [Method1, Method2]

Tracking: # Not yet implemented, will show up in the config window
  param_1: 0.
  param_2: 0
```

Setting up a Python working environment

This guide is thought for users on Windows willing to either use python 2.7 or 3+

1. Download the version of python you want from https://www.python.org/downloads/windows/ and install it
2. It may be that in Windows after the installation, python is not added to the path, don't worry things are going to be sorted out later.
3. Get pip from:

   ```
   bootstrap.pypa.io/get-pip.py
   ```

4. Run:

   ```
   path/to/python/python.exe get-pip.py
   ```

5. Go to `path/to/python/Scripts`
6. Run:

   ```
   pip.exe install virtualenv
   pip.exe install virtualenvwrapper-powershell
   ```

   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 --python=path\to\python\python.exe
   ```

The last piece is important, because it will allow you to select the exact version of python you want to run, it can be either `python2` or `python 3` and also it can be Python 64 or 32 bit. You will also create a folder called Testing, in which all the packages you are going to install are going to be kept.

8. Go to the folder `Testing\Scripts`. Try to run `activate.bat` 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. Now you are working on a safe development environment. If you just type python you will see that you are running the exact version you wanted. The same goes for packages, you can download specific versions, completely isolated from what is happening in the computer. Imagine there is more than one user and one decides to use numpy 64-bit but you need numpy 32-bit, you both can work isolated from each other. Moreover, if you run:

```
pip freeze > requirements.txt
```

You are going to generate a file (requirements.txt) with all the installed packages at that given time

10. For developing GUI's, most likely we are going to use PyQt. Since there is no official repository to install it through pip, we need to download the appropriate wheel from:

```
http://www.lfd.uci.edu/~gohlke/pythonlibs/#pyqt4
```

Afterwards, just run (replacing the last part of the command by the wheel you have just downloaded):

```
pip install PyQt44.11.4cp36cp36mwin32.whl
```

This last bit is useful for people dealing with large datasets, or people who would like to store not only data but also metadata in a future-proof binary file format.

11. For saving data, specially when dealing with big datasets, there is almost nothing better than using HDF5 (https://support.hdfgroup.org/HDF5/). For installing, follow the same procedures than with PyQt, you can find the wheel here: `http://www.lfd.uci.edu/~gohlke/pythonlibs/#h5py`

Note: *h5py* requires to have some Visual Basic distributables. Go to `http://landinghub.visualstudio.com/visual-cpp-build-tools` to download and install. HDF5 is particularly useful when the dataset is bigger than the memory available, since it writes/reads to disk but to the user everything is presented as an array. For example saving to disk is just asigning a value to a variable such as:

```
dset[:,:,i] = img
```

This line would be writing to disk the 2D array img. When reading:

```
img = dset[:,:,1]
```

Would load to memory only one 2D array. For the documentation and understanding of how HDF5 works, I highly suggest reading the website:

```
http://docs.h5py.org/en/latest/quick.html
```

Installation instructions

To install UUTrack it is important to be inside of a virtual environment. If you want to set up a working environment, I suggest you to check *Setting up a Python working environment*. From the command line you can run the following command:

```
pip install -U https://github.com/aquilesC/UUTrack/archive/master.zip
```

Remember that in this case master refers to the branch you are installing. In case you want to work with specific branches of the code, you should change it.

If you are planning to develop code (you need to change, correct a bug or whatever is present), you need to install the package in an editable way. Just run:

```
pip install -e git+git@github.com:aquilesC/UUTrack.git#egg=UUTrack
```

This will install the package inside of your virtual environment and will generate a copy of the repository in virtualenv/src/UUTrack that you can edit and push to the repository of your choice. This is very handy when you want to test new features, etc. It is also possible to work with different branches, making it very easy to keep track of the changes in the upstream code.

After you have installed the program, you can check how to *Start the program*

Start the program

To run the program you can just import the startCamera module from the root of UUTrack and trigger it with a config file:

```
from UUTrack import startCamera

ConfigDir = 'Path/to/config'
ConfigFile = 'config.yml'
startCamera.start(ConfigDir, ConfigFile)
```

Note that the splitting between the config directory and the config file is done to allow users to have different config files in the same directory, for example for different configurations of the setup. It also allows to include a pre-window to select with a GUI the desired configuration and then trigger the `startCamera.start` method.

It is important to have a Config file ready for the program to run properly. You can check the example *Config File*

Improving the code

Section author: Aquiles Carattino <aquiles@aquicarattino.com>

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. It was popularized for the creation of websites, where the user is in front of his computer, triggering actions in a remote server. Without the factor of the distance, and experiment is the same: the user triggers actions on a device from his computer. Before starting to change the code, it is important to understand the structure of **UUTrack**.

Model

Model is a subpackage of the UUTrack program. Models define the way the user will interact with the devices. For example when dealing with a camera, one of the most likely actions is to set the exposure time, trigger an acquisition and read the image. How this is achieved is dependent on every camera.

Therefore in *Model package* we will place 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 UUTrack.Model.Cameras.Hamamatsu import camera
```

With:

```
from UUTrack.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`, one has 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, 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.

View

View Package is where the GUI lives. But also is where the logic of our program is. **UUTrack** was built as a graphical program for controlling cameras, but in principle many experiments don't need a GUI, a command line interface would suffice. View is the most complex part of the program, since it handles a lot of asynchronous tasks, user interactions and more. The starting point for the View is the module `cameraMain`. The module holds the main window and all the interactions between the different parts of the code.

Threading

Acquiring from cameras can be slow, for example one can set an exposure time of several seconds. It can also be very fast, acquiring an image every couple of milliseconds. The first example poses the problem of how to acquire without freezing the GUI presented to the user. The last example poses the problem of how to keep high acqusition framerates if the user will never see more than 30fps. The solution to both problems is Threading. Qt comes with a very handy threading class that is implemented in the `workThread`. The worker runs in a separate thread and therefore its execution will not block the main GUI. When there is data available, it will emit a signal. This signal will be catch in the main program by the function `getData()`. This function stores the data in a variable called *tempImage*; if the proper parameters are set, the data is accumulated in a Queue.

The refreshing of the GUI happens at a fixed framerate given by a Timer. The function responsible is `updateGUI()`. This function will display the data available in the *tempImage* variable. It is important to note that this ensures a fixed framerate to the user, regardless of the acquisition done by the camera. If the data is being acquired much faster than what the user can see, there is no point at displaying it, and if the acquisition is too slow, there is no point in freezing the interaction until it is fetched.

Threading in Qt is a very powerful tool that has to be implemented in all the GUI programs. It ensures that the main Thread is responsive, while a background thread is busy acquiring, or performing some other operation, for example downloading data from the internet. Python offers threading, but without the signalling capabilities of Qt. Since the program is built around PyQt4 there is no point in not using it.

For stopping a Thread, the best strategy is to change the status of a variable that the thread checks periodically. In the case of `workThread` is *self.keep_acquiring*. This strategy is used in `stopMovie()`. As an example on how to extend this, `specialTaskWorker` implements a tracking algorithm and emits signals accordingly. It is very basic, but it pinpoints the direction that needs to be followed.

Controller

The last part of the program are the *controllers* for different devices. The focus of the entire UUTrack program are cameras. Controllers for cameras normally rely on library files (.dll files on Windows) that can be more or less documented. For example `hamamatsu` uses the *DCAM-API*, while `PhotonicScience` uses *scmoscam.dll*. The idea of having a Controller module separated from the Model module is the ability to copy pasting code from other sources. For example the Hamamatsu code is available on Zhuangs lab github repository, while the Photonic Science code was sent by the company itself.

Having separate modules for the controller and the model allows to share code between different setups making it more transparent for the users. For example, one may not need to set the ROI of the camera, therefore should not worry about implementing it. However learning from the *Models* of others can be extremely useful; for instance, Hamamatsu only allows to set ROI parameters that are multiple of 4. Moreover if you don't reset the ROI before changing it, the dll crashes. Photonic Science has its own share with setting the gain.

Between the controllers there is a module named `keysight` that holds the drivers for an oscilloscope and function generator. It works, but was never implemented into the main window. The idea is to use it in the `specialTaskWorker` for generating signals or acquiring fast timetraces.

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/uutrack/checkouts/latest/UUTrack/Controller/devices
of UUTrack.Controller.devices.hamamatsu.hamamatsu_camera, line 11.)

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/uutrack/checkouts/latest/UUTrack/Controller/devices
of UUTrack.Controller.devices.hamamatsu.hamamatsu_camera, line 13.)

Todo

Implement this method

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/uutrack/checkouts/latest/UUTrack/Model/Cameras/P
of UUTrack.Model.Cameras.PSI.camera.getParameters, line 7.)

Todo

`UUTrack.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/uutrack/checkouts/latest/UUTrack/Model/Cameras/P
of UUTrack.Model.Cameras.PSI.camera.initializeCamera, 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/uutrack/checkouts/latest/UUTrack/Model/Cameras/P
of UUTrack.Model.Cameras.PSI.camera.setExposure, line 3.)

Todo

not implemented

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/uutrack/checkouts/latest/UUTrack/Model/Cameras/P
of UUTrack.Model.Cameras.PSI.camera.setupCamera, line 9.)

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

u

Index

I

R

S

T

U

W