Xamidimura Observing Software Documentation

Jessica A. Evans

March 26, 2019

Contents

1 Roof & PLC Control

This section provides information about the software for interaction with PLC and roof system. There are two different systems in place for controlling the roof, which will described in more detail shortly, but first, it should be highlighted that both use the `roof_control_function.py` script. This script is responsible for all the low-level interaction with the PLC itself.

All commands and information sent to and from the PLC box are encoded in messages that start with an '@' symbol and end with '∗\r'. Each of the response codes can be split up into different sections, where each section has a specific meaning, for example an error code, or a timeout value. Full details on how to understand the response code can be found in the Superwasp Intelligent Roof Controller Technical Manual. For general use however, the user should not need to know how to decode these response codes. The `roof_control_function.py` script handles all the decoding and encoding of these response codes. It can extract the required status information and calculate new codes to send to the PLC to request a change in state. This script is very similar to how the PLC commands operated with previous roof control system, however the code has now been updated to handle the new tilt status check that are in place for the new system.

1.1 The individual PLC scripts

In the directory which stores all the software for the new telescope,

`/home/wasp/xamidimuraSoftware/`

there is the folder `plc_scripts`. This folder contains numerous executable python scripts that can be used to interact with the PLC, without worrying about understanding response codes etc. These scripts are as follows:

- **plc_open_roof.py** - Open the roof.
- **plc_close_roof.py** - Close the roof.
- **plc_stop_roof.py** - Stop roof movement.
- **plc_get_roof_status.py** - List the current status of various detectors related to the roof. This includes the open-roof detector, the close-roof detector, roof-movement, rain detected, the motor-stop, where mains power or battery power is selected, if the main door is open, if the extractor fan is on, whether the building temperature is high, if the PLC has control of the mount, if the roof control is local or remote, open and close proximity sensors, and whether the roof has been forced to close due to rain or a power failure.
- **plc_get_plc_status.py** - Check whether or not there is a response from the PLC and state what mode it is set to.
- **plc_get_rain_status.py** - Check whether the roof is set to ignore the fact that the PLC thinks it's raining.

- **plc_get_tilt_status.py** - NEW - Check how the HA axis of the telescope is tilted. Needed for making sure it is safe to close the roof.
- **plc_request_roof_control.py** - Change the roof control from local to remote, so it can be controlled from the observer machine.
- **plc_request_telescope_drive_control.py** - NEW - Change the telescope drive control to be the observer machine rather than the PLC box.
- **plc_select_main.py** - Select mains power to operate the roof control. Currently the only method of moving the roof.
- **plc_select_battery.py** - Select the battery backup. Note this is a residual command from the old roof system. The battery is not currently connected and so this will not do anything.
- **plc_set_comms_timeout.py** - Set the timeout time for communications between the PLC and the observer machine. After this amount of time without any response, the roof will be asked to close.
- **plc_set_power_timeout.py** - Set the timeout time for a power failure and the observer machine. After this amount of time without any power, the roof will be asked to close.
- **plc_reset_watchdog.py** - The watchdog is a timer that is used measure the time since the last communication. It should be reset after a command is sent to the PLC. Generally this is done automatically in the other scripts, but this reset script allow the user to reset it manually.

These scripts are all almost identical to the old PHP that were previously used to interact with the PLC and the roof, however there are now scripts to work with the PLC's new functions. NOTE there are NO tilt checks, to check the telescopes are parked and it is safe to close the roof. So it is vital that the telescope is PARKED before the roof is asked to close. There maybe a safety built in that will make the telescope to move slowly to a safe location when the roof is asked to close, but it is unclear if this is definite.

The scripts mentioned above contain very little in the way of code, they all call functions from the `PLC_interaction_functions.py` script. The close roof function will check is the roof control is set to remote, the motor-stop is not pressed, that there is no power failure with the AC motor selected or that the AC motor has tripped. When the roof is asked to open, the following are checked before opening: the roof control is set to remote, the motor-stop is not pressed, it's not raining and there is no power failure.

To run one of the scripts, navigate to the folder that contains the plc scripts and run one of the following:

```
$ python plc_get_roof_status.py
$./plc_get_roof_status.py
```

1.2 Alternative method

This second method for opening and closing the roof is a newly developed system, which may still have some issues to resolve, but for the most part works. It is easier to use than the first method, but does have a higher chance of something not working. This method still uses the `roof_control_functions.py` script for basic communication with the PLC box, and it uses some functions from `PLC_interaction_functions.py` for some of the checks. Most of the work however is carried out by the `plcd.py` and `specify_roof_state.py` scripts. The `specify_roof_state.py` script is responsible for handling any request to change the roof state, while the `plcd.py` script is responsible for checking it is safe to make the change and then carry out the request.

The `plcd.py` script should be running continuously Still needs to be set up. The script monitors a memory-mapped file for a change in the character that is stored there. The specific location is set by the `PLC_MEMORY_MAP_FILE_LOC` parameter in `settings_and_error_codes.py`. There are three character that are valid requests: 'c', indicating the roof should close; 'o', indicating the roof should open; 's', indicating that the roof movement should stop. When the `plcd` script detects that the state has changed, it will perform checks and then carry out the desired change.

When opening the roof it will check to see if the roof is already open, it will check to see if the PLC thinks it's raining, it will check to make sure the telescopes are parked (it will not move them if they are not). It will check that roof control is set to remote control and will request control if not. It will check to see if the motor-stop is pressed - it cannot do anything about this. It will check to see if there is a power failure, and finally it will check and request drive of the telescope mount.

When closing the roof it will check to see if it's already closed. It will then check how the telescope is tilted and will attempt to park them if they are not already in a parked position. The code then check the roof control is set to remote and will request control is not. It will check that motor-stop is not pressed, but it cannot do anything about this if it is. It will then try to make sure that there is suitable power for the move. Currently this is incomplete as the UPS checks are not in place - will need to change this once the UPS can successfully control the roof. The code will also request telescope drive control. No checks are performed when stopping the roof.

The `specify_roof_state` script is responsible for sending open/close/stop request to the memory-mapped file, which will be picked up by `plcd.py`. There are a couple of ways these ways the request can be made. The first method would be to navigate to the directory that contain the code `/home/wasp/xamidimuraSoftware/` and then run the executable script, for example:

```
$ ./specify_roof_state.py s
$ ./specify_roof_state.py c
$ ./specify_roof_state.py o
```

These commands will stop, close and open the roof, respectively.

The easiest way to request a change in roof status is by using the alias command that have been set up. These can be used in any directory. The commands to use are

- `open_roof`
- `close_roof`
- `stop_roof`

The `stop_roof` command also calls the `plc_stop_roof.py` script, because for some reason the stop request in `plcd` wasn't working. It has worked previously, but there was no time to investigate this issue further. It is also worth noting that because the code looks for changes in the roof state, if for some reason a close or open roof command fails, a stop roof command will need to be issued before a new open/close roof command will be responded to.

2 Observing

This sections describes all the scripts that maybe used during a typical observing session. It covers topics such as the focusers and filter wheels, flat-fields and how to start observing.

2.1 Basic exposures

One way to quickly start taking exposures of a target is to use the `expose.py` script. It is a fairly manual way to carry out the observations and assume that the telescope is already pointing in the appropriate location. Basic instructions on it's use are described in the comments at the top of the script.

To begin, an observing recipe will need to be created with a name that is appropriate, e.g. 'target.obs' if observing the object called 'target'. Please see Section 2.6 for more information how complete an observing recipe. Once complete, ensure the telescope is pointing at the appropriate target. This will need to be done with the controls on the TCS machine, either directly or through the `tcs_control.py` script.

Next, move to the directory containing the `expose.py` script and edit the three parameters named `target_name`, `target_coords` and `target_type` to appropriate value for the target being observed. `target_name` is used to identify the target and needs to match the name given to the observing recipe, and should be written in quote marks. `target_coords` is the right ascension and declination of target, written as two values in quotation marks. The hours, minutes and seconds for RA, or degrees, minutes and seconds for DEC, should be separated by spaces. An example of the format is given below:

```
['10 50 31','-32 20 43']
```

The coordinates are not used to point the telescope, they are only used in the fits header of the images to help identify the target.

Finally, there is the `target_type` parameter. Again this is only used in the fits headers to help identify the target, and should have something to identify the type of target, for example, 'EB' if the target is an eclipsing binary or 'planet' for an exoplanet target. In reality, this parameter can be any form of string, but it is worth making in it something meaningful.

Once the parameters for the target have been setup, the script can be used to take exposures. In reality, the script can be used without first setting up the target parameters, however details in the fits headers will be incorrect, and then trying to identify images for a specific target could be difficult. In order to use the script, first in a terminal window, move to the directory containing the software using

```
$ cd /home/wasp/xamidimuraSoftware/
```

Run python by typing `python`. The `expose.py` script needs to be imported before the functions inside can be used. To import the script:

```
>>> import expose
```

Before an exposure can be taken, there are a few things that need to be setup. These include starting the filter wheels and focusers, and starting to cool the cameras. To do this run the following in the python terminal:

```
>>> expose.startup()
```

The cameras will take a while to cool to their optimal temperature of -50°C, but images can still be taken in this time. The status of the images will be marked as '1', if the cameras are warmer than -20°C when an image is taken.

Once setup, exposures of the target can be taken using

```
>>> expose.expose()
```

This will call a function from the main observing script, which will handle connecting to the database for logging exposure requests, loading the observing recipe and actually taking the exposure. Note that calling this expose command will get the script to loop through the observing recipe and taking multiple images in different filters if that is how the recipe has been set up. To repeat the recipe multiple times, it will need to placed in loop by using code similar to what is shown in Listing 1.

Listing 1: Loop example to repeat the observing recipe exposure request.

```
>>> i = 0 # Initialise a count for times it will repeat
>>> while i < 100: # Change 100 to be the number of times to repeat
...     print('Loop number:', i)
...     expose.expose()
...     i+=1
```

This method for repeat exposures is not the most efficient, and is design for a quick test of an observing recipe or such. This is because the function loads the observing recipe each time and other tasks that are not require for repeat exposures. A better method for repeat exposures will be described in Section 2.5.

Finally, once the observing is complete, run

```
>>> expose.shutdown()
```

to turn off the camera cooling, shutdown the filter wheels and focusers, and disconnect from the database.

To run exposures for a different target after already completing exposures, there are two options. Either exit the python terminal and start a new one with up dated target parameter at the start of the script, or run the following after changing the target name, target coordinates and target type:

```
>>> from importlib import reload
>>> reload(expose)
```

In the first case, the startup procedure will need to be run again, however in the second case it should be possible to just run the `expose.expose()` command again.

2.2 Filter wheels

This section will describe how to control the filter wheels. It will cover the very basic operation, so how to change the filters without running the full observing script, and also describe how filter changes are handled within the main observing script. Please also refer to the manuals[1] for the filter wheels for more information on the setup and installation of the filters and filter wheels.

First, ensure that the filter wheels are switched on. There are two boxes on the table in the telescope room marked 'ifw', one for the South-telescope and one for the North-telescope. If switched on the small screen will be illuminated with the name of the current filter, e.g 'WX' or 'RX'. If the filter wheel has lost its filter names from memory the names maybe 'A1',' A2', etc. If the screen is not illuminated, the is a switch on the front of the box that will need to be switched on. Also check the cables, it's quite easy for the power cable to come loose. If communication is tried with the filter wheels switched off, then the software will just hang as it will be waiting for a response that is never going to turn up. A message about no response should be printed after some 25 seconds, but the code will still throw an error message. Once switched on, the box will try to initialise and home the filter wheel. Note that homing a filter wheel can take up to 20 seconds.

The each control box has buttons that could be used to change filter, and home the filter wheel.

[1]https://www.optecinc.com/astronomy/catalog/ifw/resources/ifw_technical_manual.pdf

2.2.1 Basic operation

This section will give instructions for the basic operation of the filter wheels without interaction with the larger `observing.py` script.

Open a terminal window and navigate to where the telescope software is stored:

```
$ cd /home/wasp/xamidimuraSoftware/
```

Run python in the terminal using

```
$ python
```

and then import the module containing all the functions for controlling the filter wheels:

```
>>> import filter_wheel_control as fwc
```

The first thing that will need to be done is run the startup procedure for each filter wheel using the following:

```
>>> ifw1_port, ifw1_dict = fwc.filter_wheel_startup('ifw1-south.cfg')
>>> ifw2_port, ifw2_dict = fwc.filter_wheel_startup('ifw2-north.cfg')
```

`ifw1_port` is what is used to pass information to the filter wheel on the South-telescope. If for some reason the port gets close, re-run the command above to open it again. `ifw1_dict` contains the information loaded in from the configuration file. More detail on the configuration file will be given in Section 2.2.3. The `ifw2_port` and `ifw2_dict` contain the same port and configuration information but for the filter wheel on the North-telescope instead. The startup procedure for the filter wheels, will open a connection to the filter wheel, run the initialisation procedure and finally home the wheel. It will need to be done separately for both wheels.

Once setup, the open port for each filter wheel can be used to issue commands. For day-to-day operation, the only function the will need to be used will be the `change_filter()` function. As the name suggests it is used to modify which filter is currently being used. Listing 2 provides examples on how to used the function to change the filter. This code will need to be run in the same python terminal as where the startup procedure was carried out.

Listing 2: Usage example to change the filters

```
# On South-telescope, hange to 'RX' filter
>>> fwc.change_filter('RX', ifw1_port, ifw1_dict)

# On North-telescope, hange to 'GX' filter
>>> fwc.change_filter('GX', ifw2_port, ifw2_dict)
```

The function will check to see if a change is actually necessary. The filter wheels use numbers to identify the slots on the wheel. Configuration information (ifw1_dict/ ifw2_dict) is past to the function to allow the conversion between the filter name and the position number. In general, it is easier to remember the name of the filter than a number associated with its position.

Once all observing is complete, the connection to the filter wheels can be closed. There are two options, one use the shut down functions or two, just immediately close the connection. To close the port immediately use

```
>>> ifw1_port.close() # for the South telescope
>>> ifw2_port.close() # for the North telescope
```

To use the shutdown function run:

```
>>> fwc.filter_wheel_shutdown(ifw1_port) # for the South telescope
>>> fwc.filter_wheel_shutdown(ifw2_port) # for the North telescope
```

The shutdown procedure will home the telescope before closing the connection. The four previous commands will need to run in a Python terminal where the filter wheel have previously been initialised.

2.2.2 Other commands for the filter wheels

Other commands that can be sent to the filter wheel will be used rarely, and will mostly be for configuration purposes. This section will briefly mention their uses. Only examples for the South-filter wheel will be given, but the same commands can be sent to the North filter wheel, by changing ifw1_port to ifw2_port and ifw1_dict to ifw2_dict.

```
>>> fwc.get_stored_filter_names(ifw1_port, formatted_dict=True)
```

Use this function to find out what names are currently stored on the filter wheel's memory. When `formatted_dict` is set to True, each of the names will be paired with the appropriate slot number for each filter. When false, the names are returned as one long string.

There are two functions `form_filter_names_string_from_config_dict()` and `pass_filter_names()` used for formatting and storing a set of names on the filter wheel's memory. In reality, it would be easier to modify to configuration file and then use the `initial_filter_wheel_setup()` to store the new names (see Section 2.2.4).

The `fwc.get_current_position(ifw1_port)` will return the slot number of the current position. The `fwc.goto_home_position(ifw1_port)` will send the filter wheel to its home position. Finally, the `fwc.goto_filter_position(1, ifw1_port)` can be used to send the filter wheel to a specific slot position, in this case slot 1, which corresponds to the home position.

2.2.3 The configuration file

Configuration files for both the filter wheels are stored in

```
/home/xamidimuraSoftware/configs
```

Listing 3 contains an example of the config files for one of the filter wheels.

Listing 3: Example configuration file for South filter wheel

```
name ifw1-south
port_name
    /dev/serial/by-id/usb-Optec__Inc._Optec_USB_Serial_Cable_OP10EELI
        -if00-port0

baud_rate 19200 #The baud_rate, data_bits,
data_bits 8   #stop bits and parity are
stop_bits 1 #preset. SHOULD NOT BE CHANGED
parity N      #possible values: N,E,O,M,S

filter_wheel_ID A
no_of_filters 9 #Total number

1 WX  # 50mm, Astronomik Deep-Sky Red
2 BX  # 50mm, Astronomik Deep-Sky Green
3 GX  # 50mm, Astronomik Deep-Sky Blue
4 RX  # 50mm, Astronomik L-3 UV-IR (white)
5 IX  # 50mm, Astronomik Proplanet 642 BP (infrared)
6 BLANK
7 BLANK
8 BLANK
9 BLANK

home_pos A
warning-low-temp -40
warning-high-temp 85
```

Not all of these parameters are currently used in the software, but most provide useful information.

Below the filter wheel name is the port name. This is where the PC looks to communicate with the filter wheel. Each filter wheel will have a different port name. For the filter wheels, each wheel's port has a unique ID that has been used over the name of individual ports (e.g. /dev/ttyUSB?) and the device can swap between named ports when the PC is reset, but the ID should remain the same.

The baud rate, data bits, stop bits and parity parameters are specific for the port communication. These values are taken from the manual for filter wheel and should not change.

Next are the filterwheel ID and number of filters in the wheel. For our telescope the filterwheel ID should remain set to 'A'. There is the option to use multiple wheels for each telescope, but for now that will not be the case here. The number of filters slot should not change either, unless the entire filter wheel instrument is changed. It is worth noting the number, as it is possible the get wheels with different numbers of slots.

The next section of the configuration file contains information about which filters are currently held in the wheel. It is currently organised as 'Slot number', 'name' and then a more detailed name in the comments following the '#' symbol. Names must not be longer than 8 characters in length and contain only the following symbols:

```
[‘0’,‘1’,‘2’,‘3’,‘4’,‘5’,‘6’,‘7’,‘8’,‘9’,‘A’,‘B’,‘C’,‘D’,
  ‘E’,‘F’,‘G’,‘H’,‘I’,‘J’,‘K’,‘L’,‘M’,‘N’,‘O’,‘P’,‘Q’,‘R’,
  ‘S’,‘T’,‘U’,‘V’,‘W’,‘X’,‘Y’,‘Z’,‘=’,‘.’,‘#’,‘/’,‘-’,‘%’, ‘ ’]
```

The final three parameters in the config file are not used.

2.2.4 Updating filter names

If the filters are updated, the new names will need to be stored in the memory on the control box. All the work required to do this has been packaged into one function to make updating the names easier.

In a terminal that is running python run the following:

```
>>> import filter_wheel_control as fwc
>>> fwc.inital_filter_wheel_setup(‘ifw1-south.cfg’)
```

This assumes that the new configuration file is called ‘ifw1-south.cfg’ and is stored in the same place. Use the `config_file_loc` parameter in the functions argument to change the path to the config file is required. The default is `configs/`. The initial setup will need to be done separately for the filter wheel on the North-telescope.

2.2.5 The filter wheel changes in `observing.py`

The request to change the filters occurs once a new exposure has been requested. The `take_exposure()` function calls the `change_filter_loop()` function, which in turn calls the functions from `filter_wheel_control.py`. The `change_filter_loop()` will handle any errors that may occur with the filter wheel, and return a status code to be used in image fit headers where appropriate.

2.3 Flat fields

The current automatic flat field script still needs some work, but it should at least allow some flats to be taken.

`autoflat.py` is the script that is responsible for taking flat field images in the evening and in the morning. Both have been tested, and work once the sky has reach the correct brightness. So far it has not been possible to confirm that the script will wait until the sky is the appropriate brightness before starting the sky flat procedure.

2.3.1 Manual start autoflat.py

The following instructions assumes none of the require setup has been carried out, and a new terminal window will be opened.

In a new terminal window, navigate to the telescope software directory using

```
$ cd /home/wasp/xamidimuraSoftware
```

and run python.

```
$ python
```

Import `observing.py`:

```
>>> import observing
```

Carry out the necessary setup to start the logging and instruments.

```
>>> focuser1_info, focuser2_info, ifw1_config, ifw1_port, \
... ifw2_config, ifw2_port = observing.connect_to_instruments()
>>>
>>> datestr, file_dir = observing.setup_file_logs_storage()
>>>
>>> observing.evening_startup() # Run camera startup
>>>
>>> time_mess, t_remain, k_time = \
... observing.getAlmanac.decide_observing_time()
```

Point the telescope at a suitable location on the sky for acquiring flats, remembering to open the roof before trying to move the telescopes. Then use one of the following to carry out either evening or morning flats as appropriate.

```
>>> observing.autoflat.do_flats_evening()
>>> observing.autoflat.do_flats_morning()
```

2.3.2 How the script works

When a request for a set of flats is made, first a sequence of filter names is loaded depending on whether evening are morning flats are requested. The filter sequences are defined at the top of the `autoflat.py` script by the `evening_filter_order` and `morning_filter_order` parameters. Currently these are set to `['BX', 'GX', 'WX', 'RX', 'IX']` for the evening and `['IX', 'RX', 'WX', 'GX', 'BX']` for the morning. Initial testing may suggest swapping 'WX' and 'RX', although any filters that do not get sufficient coverage in the evening are covered in the morning. The script will then cycle through the following instructions for each filter.

Once the filter change is complete, scratchmode is enabled. Scratchmode allows images to taken by the cameras, but the resulting images do not get saved and do not acquire an image number by the DAS machines. Test images with exposure time 0.1 sec are taken repeatedly until the estimate of a 1 second exposure (evening) or 20 sec exposure (morning) gives a reasonable count level. If it is already too dark for a 1 second exposure (evening)/ bright for 20 sec exposure (morning) it will try to estimate a better exposure value for the filter. The `lastsky` command on the TCS is used to get the sky brightness of the last images from the two telescopes. The values from the two telescopes are averaged and this average value is used for calculations by the autoflat script. Scratchmode will then be turned off, and while exposure times sit between a maximum and minimum exposure length, flat field images will be taken for that filter. Each time an image is completed, the sky brightness is checked and a estimate of a new exposure time is made.

For evening exposure, the new estimate is found by multiplying the current exposure time by a factor. The factor is found by rounding the ideal_count/current_count ratio up to nearest integer. For morning flats a slightly different method is used for the factor:

```
factor = round_up((ideal_counts/current_counts)*10) / 10
```

There is code written that will then apply an offset between each of the flat field images, but it is currently commented out as it has not been fully tested. The total number of flats for a filter are recorded, as is the number of 'good' flats. Good flats are those where the counts fall between the minimum count level and the saturation level.

All the minimum, maximum, ideal count levels and minimum, maximum exposure lengths can be set using the parameters at the top of the script.

2.3.3 Current known shortcomings

The current method of using a factor to calculate the next exposure time, is not necessary the best method if we want to get many flats for different filters. It would be better to have a function the can predict the best exposure time taking into account the falling sky brightness. It would make the process more efficient, however some calibration will be required for each of the filters.

As already mentioned, there is a chance that the code is not waiting until the sky is the right brightness before continuing. It may be related to how the sky brightness reads when the image is over exposed. This is speculation and it really needs to be checked.

2.3.4 Flat fields from `observing.py`

Starting the evening and morning flat request from `observing.py` is done via a simple function call. In `observing.py`, the `main()` function will call either `autoflat.do_flats_evening()` or `autoflat.do_flats_morning()` when required. The `main()` function will also find an appropriate blank sky field, point the telescope and open the roof if needed.

2.4 Focusers

This section details the basic operations for using the focusers. Most of the functions that can be used to interact with the focusers are in `focuser_control.py`, with the only exceptions being the functions to read the configuration files and to read response from a serial port. These particular functions are in the `common.py` script as they are shared with the filter wheels. Please also refer to the manuals for the focusers more details on what each of the commands do.

First, ensure that the focuser boxes are switched on, and all the cables are plugged in. The should be a red light on if the box is switched on. There is a switch on the side of the box to power it on. It also quite easy for the power cable to fall out of the socket on the box. If there an issue with the connection to the boxes, the software will hang and eventually raise an error as it sits and waits for a response that will not come.

2.4.1 Basic operation

Open a terminal window and navigate to the where the directory with all the software, and run Python.

```
$ cd /home/wasp/xamidimuraSoftware/
$ python
```

Then import `focuser_control.py`

```
>>> import focuser_control as fc
```

Once imported the first thing to do is to run the startup procedure for each focuser. This can be done using the commands below in the active Python terminal.

```
>>> num1, focus1 = fc.startup_focuser('focuser1-south.cfg')
>>> num2, focus2 = fc.startup_focuser('focuser2-north.cfg')
```

`focus1` and `focus2` are parameters that store the open serial port connection to the south and north focuser, respectively. The `num1` and `num2` parameters stores the focuser number used by each of the focuser control boxes. It so happens that this is one for both of the focusers, but in principle they could have been '2' as well. The `focuser1-south.cfg` and `focuser2-north.cfg` file are configuration files, which are assumed to be stored in `/home/wasp/xamidimuraSoftware/configs/`. More details on the configuration file will be given in Section 2.4.2.

The startup procedure will load the configuration file and open the communication port. Before continuing each focuser will need to be homed. The telescopes must be in a parked position before carrying out the homing or centring procedures. Large focuser movements without the telescopes being parked could damage the focuser mechanisms, small movements are fine. To home the focusers run the following in the python terminal:

```
>>> fc.home_focuser(focus1) #For south-focuser
>>> fc.home_focuser(focus2) #For north-focuser
```

The other commands to move the focusers will not work if they have not been homed.

At the end of an observing sessions, the shutdown procedure should be used to close the communication port.

```
>>> fc.shutdown_focuser(focus1)
>>> fc.shutdown_focuser(focus2)
```

It is also possible to use `focus1.close()` or `focus2.close()` to close the port without running the shutdown procedure.

There are numerous commands that can be sent to the focusers they have been homed. Below is a brief summary of these commands. Only examples for the south focuser are provided, just change `focus1` to `focus2` to change the commands to be for the north focuser.

- `fc.get_focuser_status(focus1)` - Get the current status for the focuser. Useful to see if it is still moving, or if it's homed etc.
- `fc.get_stored_focuser_config(focus1)` - Get the current configuration information for the focuser. Useful to see if the temperature compensation, or backlash compensation is enabled.
- `fc.move_to_position(<new_position>, focus1)` - Move to a position between 0 and 112000.
- `fc.move_focuser_in(focus1)` - Move the focuser in. It will keep going until it hits its limit (0), or the user asks it to stop.
- `fc.move_focuser_out(focus1)` - Move the focuser out. It will keep going until the limit (112000) is reached or the user request that it stops.
- `fc.end_relative_move(focus1)` - Stop the current movement.
- `fc.center_focuser(focus1)` - Move focuser to it's central position (56000).

If the configuration file is changed the new configuration will need to passed the the focuser control box. All the configuration functions have been grouped together into one function in `focuser_control.py`, so to update the configuration, run

```
# for south-focuser
>>> fc.focuser_initial_configuration('focuser1-south.cfg')
# OR for north-focuser
>>> fc.focuser_initial_configuration('focuser2-north.cfg')
```

2.4.2 The configuration file

Listing 4 contains an example of the config files for one of the focusers.

Listing 4: Example configuration file for South focuser

```
focuser_name focuser1-south
focuser_no 1 #Either 1 or 2
port_name
    /dev/serial/by-id/usb-Optec__Inc._Optec_USB_Serial_Cable_OP2I4TDC
        -if00-port0

baud_rate 115200 #The baud_rate, data_bits,
data_bits 8    #stop bits and parity are
stop_bits 1 #preset. SHOULD NOT BE CHANGED
parity N       #possible values: N,E,O,M,S


device_type OB #Need to check this is correct
LED_brightness 10 #Between 0-100. 0 is off

center_position 56000
min_position 0 #from manual
max_position 112000   #from manual

temp_compen False #true/false
temp_compen_mode A #A,B,C,D or E see manual
temp_compen_at_start False
temp_coeffA 86 #Just a guess, needs calibrating
temp_coeffB 86
temp_coeffC 86
temp_coeffD 86
temp_coeffE 86

backlash_compen False
backlash_steps 40
```

The first few lines of the configuration file set out the information required for the serial port connection. The max, min and centre positions are not actually used, but it is useful to have them stated. Finally there is information related to the temperature compensation and the backlash compensation. The temperature compensation is currently disabled for both focusers, as the coefficients need to be calibrated for out particular setup. Allowing the focusers to adapt based on temperature changes, should help keep the focus stable throughout the night.

2.4.3 Focusers in `observing.py.py`

Currently none of the functions for the focusers are included in `observing.py`. Currently the focuser are manually set to a value and then it kept the same throughout the night.

2.5 Observing

Most of the instructions discussed in this section have probably already been discussed, in part, in some of the other sections. However, it is worth mentioning the information again in a way that the information can be easily followed, step-by-step. This section will describe how to use the new software to get the telescope to take observations of a target. It will also discuss how the code should do this automatically when the `observing.py` script is run, although the full automated operation has not yet been achieved, most of the individual parts have been tested, and it should not require much work to get it fully automated. Currently there are certain checks that have not been automated.

2.5.1 Manual startup operations

This section describes how to manually get the telescope to observe a target, from the time of the sunsetting to being on target.

On the 'observer' machine, open a new terminal and navigate to the telescope's software directory. Run Python.

```
$ cd /home/wasp/xamidimuraSoftware
$ python
```

Import `observing.py` using

```
>>> import observing
```

Run all the setup commands as shown below. This will setup the focusers, filter wheels, cameras and database connections.

```
>>> focuser1_info, focuser2_info, ifw1_config, ifw1_port, \
... ifw2_config, ifw2_port = observing.connect_to_instruments()
>>>
>>> datestr, file_dir = observing.setup_file_logs_storage()
>>>
>>> observing.evening_startup() # Run camera startup
>>>
>>> time_mess, t_remain, k_time = \
... observing.getAlmanac.decide_observing_time()
```

There is a python script on the 'observer' machine which will look for any new fits headers that are created. When a new header is found, it will go find the matching image on the appropriate DAS machine. The image file and the fits header get combined and then the new file is stored in the location specified by the `FINAL_DATA_DIRECTORY` parameter in `settings_and_error_codes.py`. Other subdirectory are created here to help segregate the different image types, e.g. bias, flat, object. For the image to end up in the correct directory with the correct header information, the `das_fits_file_handler.py` script needs to be running. To do this, open a new terminal window and navigate to the software directory. Run the script using:

```
$ python das_fits_file_handler.py
```

This script should just run in the background throughout the observing session.

Meanwhile on the TCS computer, the telescope software needs to be running. Use the `starttel` command to start the software if it is not already running. In practice it maybe necessary to power cycle to mount at this point, especially if the PLC has previously taken control of the telescope drive. This would involve killing the xobservatory software with `killtel`, unplugging the mount power, waiting a few seconds, plugging it back in and then restarting the xobservatory software with `starttel`

On the observer machine, it is should be ready to take bias frames. In the python terminal where `observing.py` was imported use

```
>>> observing.take_bias_frames(datestr, file_dir)
```

Once complete the roof can be opened, and the telescope homed. If the `plcd.py` script is running, it should be possible to type `open_roof` in a new terminal window to get the roof to open as long as there are no issues and the motor stop button is not pressed. Alternatively, use the `plc_open_roof.py` executable found in

```
/home/wasp/xamidimuraSoftware/plc_scripts/
```

The roof can take a good few minutes to open. Logs of what the roof is doing should be located in `/home/wasp/logs/plc.log`.

The telescope homing will need to be done but sending command on the TCS, or by using the buttons on the xobservatory window. Probably easiest is by sending `zeroset ra` to home the RA axis, and then `zeroset dec` to home the DEC axis, but only once the RA axis is complete. Asking it to zeroset the DEC to soon will make it abandon it RA homing attempt. It may also be necessary to run the DEC axis zeroset twice, as it tends to timeout on the first attempt. Probably, the timeout for the DEC axis needs to be increased, or the dec axis speed should be increased. The mount should be watched during homing to make sure that it moves, and that it does not hit anything.

Once homed, it should be possible to take some flat fields, assuming the sky brightness is suitable. First, located a suitable region of the sky to take the flat fields. One of the easiest methods would be to use the blank sky field locator. On the observer machine, in the python terminal where the bias frames were taken, run

```
>>> arow = observing.find_best_blank_sky.find_best_field()
```

The table row that gets stored in the parameter arow, contains the RA and DEC of the best field. The RA and DEC will be needed to slew to the field. This can be done using:

```
>>> observing.go_to_target(arow['RA(hms)'], arow['DEC(dms)'])
```

The automated flat field script, for evening flats, can then be started using

```
>>> observing.autoflat.do_flats_evening(k_time, arow, datestr,
... filedir)
```

For morning flats the command is similar, just substitute 'evening' with 'morning', and use a new result blank sky field search. The automated script will continue taking images until the light-levels are no longer suitable.

While it is getting dark, start setting up for the target. To observe the target, you will need it's coordinates, a name, a target type (so planet, EB, etc) and an observing recipe. See Section 2.6 for details on creating an observing recipe. Assuming the observing recipe exists you can use the code similar to Listing 5 to create a `target_object` for the target. Here the details for WASP0928-37 have been used as an example.

Listing 5: Creating `target_object` for WASP0928-37

```
>>> target_name = 'WASP0928-37'
>>> target_coords = ['09 28 34.39', '-37 04 48.0']
>>> target_type = 'EB'
>>> image_type = observing.get_image_type(target_name)
>>> target_recipe = observing.get_observing_recipe(target_name)
>>> target = observing.target_obj(target_name, target_coords,
    target_type)
```

From there, to get the software to loop through the observing recipe once, use:

```
>>> observing.take_exposure(target_recipe,image_type, target,
... datestr=datestr, fits_folder = file_dir)
```

The easiest way to get the observing recipe to be repeated is the put the `take_exposure()` request in a loop. An example is given in Listing 6.

Listing 6: Loop to repeat observing recipe.

```
>>> i=0
>>> while i < 100: # change 100 to be number of times repeated
... print(i)
... observing.take_exposure(target_recipe,image_type, target,
    datestr=datestr, fits_folder = file_dir)
... i+=1
...
```

Use `ctrl-c` to exit the loop at any point. The code in Listings 5 and 6 is all that will need to be edited to change target.

2.5.2 Manual shutdown operations

At the end of the night, once all observations are complete, the appropriate shutdown procedures will need to be followed.

After stopping any imaging, click 'stop' on the TCS xobservatory software, and then 'stow'. The 'stop' button will stop the telescope tracking its target, while the 'STOW' will move the telescopes to a position where it should be save to close the roof. From testing this request, the current stow position is not far enough to the west for the roof to close. This means that when you try to close the roof, the PLC will take control of the telescope and will VERY VERY SLOWLY move the telescope to a safe position, and the roof will then close.

Once safely parked, the roof can be closed. If the plcd.py script is running, type `close_roof` in a terminal window. Otherwise, use the plc_close_roof.py executable found in `/home/wasp/xamidimuraSoftware/plc_scripts/`. The final tasks are to shutdown the instrument, cameras and disconnect from the database. This can be done by running the following in the Python terminal on the observer machine:

```
>>> observing.morning_shutdown()
>>> observing.shutdown_instruments()
>>> observing.disconnect_database
```

2.5.3 Automated observations

In principle it will be much easier to allow the telescope to go carry out the tasks set out in the last two sections automatically. Most of the code to allow this is in place, but not all of it. This section will describe how the automation currently runs, and mention some of the areas that need to be completed in order to achieve this.

To allow the telescope to run automatically, in principle just run the `observing.py` script. In a new terminal window, navigate to the telescope software directory and use

```
$ python observing.py
```

This will get it to run the `main()` function in `observing.py`. This will setup all the instruments, such as filter wheels, focusers and camera, it will connect to the database and workout the date-folder for where the new data should be saved. It will also run the almanac so it can keep track of what time of day it is. The times-of-day are described in more detail in Section 3.1.2, but this function will take different actions depending on the result of the time check. If it's `daytime`, the code will just keep running the `decide_observing_time()` function from `getAlmanac.py` every 60 seconds until the sun has set.

Once the sun has set, the message should switch to `afterSunset`. Here the telescope will take some bias frames, and then check if it should open the roof to prepare for taking flats. **Here lies one of the issues with the automated script - It should check to make sure the weather is ok before opening. The roof won't open if the RAINING indicator it set on the PLC, but it should check other things like the wind and humidity etc. Whether it has done it's checks or it not save to open, the `main()` with then keep running the `decide_observing_time()` function every 15 seconds to wait for the beginning of evening civil twilight.

The message will change to `afterCivil` once evening civil twilight has started. Again, it will check to see if the roof is open, and will attempt to open it if it is not open and it is safe to do so. If the roof is open, the code will set up to take sky flats including finding an appropriate blank-sky field and slewing to the spot. **Note at this point the telescope has not been homed or power-cycled if required. This is probably something that should be done manually, however the script does not consider this at all. The script will not attempt to start taking flats if there is less than 5 minutes until astronomical twilight, but this value can be changed in the settings script. Once it has finished with the flats, it will keep checking the `decide_observing_time()` function every 10 seconds until the start of astronomical twilight.

The message will change to `night` once evening astronomical twilight has started. Again it will check to see if the roof is open, and try to open it if it is not open and is safe to open. If the roof is open, the code will run the scheduler to find a target. Currently, this does not happen as the scheduler is not yet ready. There is just one target ID hard-coded in at this point. Once a suitable target has been found, it will look up the target ID in the `target_info` table in the database, and return information about the target's coordinates, target type, target name, etc. The target name will be used to load an observing script. The script will use the coordinates to move to the target, and then start exposure based on the loaded observing script. Currently the code will crash if there is not an observing script with a matching name. Probably better if we create a default recipe that gets loaded if no match is found. Currently the exposure request and the go-to-target request is commented out, at least until the scheduler work is complete. We also need to consider for what conditions will we stop observing the target - currently the scheduler get check each time, but we might want to consider other conditions, weather, target finished eclipsing etc.

The code will continue scheduling targets etc until the end of morning astronomical twilight. At this point the message will change to be `beforeCivil` and it will attempt to carry out the morning sky flat procedure. This is very similar to the evening automated flat procedure, but with the filters in the reverse order and with exposure getting shorter rather than longer. This will occur until the end of morning civil twilight. The message will then change to `beforeSunrise`. The code will then carry out any morning shutdown procedures. There is the potential to add in extra checks/tasks into the morning shutdown/evening startup function if it is desired. **SHOULD PROBABLY PUT A CLOSE ROOF COMMAND IN TOO**

Finally the code will shutdown the instruments and disconnect from the database, before ending the `main()` function.

2.6 Observing recipes

The observing recipes are how the telescopes know which filters to use and the exposure times to be used in each case. Listing 7 shows an example of an observing recipe for the eclipsing binary WASP 0928-37 (TYC 7166-257-1). Many of the parameters are not yet used, but have potential to be used in the future.

Listing 7: WASP0928-37.obs - An example of an observing recipe

```
TAR_NAME WASP0928-37
IMG-RA 09:28:36.63 # Nominal image center J2000 RA
IMG-DEC -36:59:32.96 # Nominal image center J2000 DEC

FILTERS BX,GX,WX # Filters to be used
FOCUS_POS 50000,50000,50000 #Focuser position for each filter
```

```
EXPTIME 30.0, 30.0 # Exposure time to match pattern, use float

N_PATT 0,0  # Refers to the index for the filter name
S_PATT 1,1  # Breaks if only one index is use e.g. 1

DO_FFIELD N #Y/N - carry out the flat fielding during photometry
    processing
COMSTARS 3 # Number of comparison stars
COM1 ??? # Not sure how these will be referenced
COM2 ??? #
COM3 ??? #
```

Each observing recipe is named by the target name followed by '.obs', so for the example above, the observing recipe would be called `WASP0928-37.obs`. The target name can be state at the top of the file, although this is not used in the main script.

The `IMG-RA` and `IMG-DEC` are not used either, but eventually could be used to specify an offset between where the target is and where the telescope should point for the optimal positioning of comparison targets etc. In the example above, the target needed to be positioned in one corner of the CCDs so it would appear on both CCDs. Once the two telescope are co-aligned, it will be if a different centre should be different from those specified by the targets coordinates.

The `FILTERS` parameter lists the filters to be used for a target. Extra filters can be listed. `FOCUS_POS` isn't currently used. The plan had been to specify the focus for each filter, but in reality the focus for each telescope is more important. It could be modified accordingly.

`N_PATT` and `S_PATT` specify the pattern of filter to be used for each telescope. `N_PATT` refers to the North-telescope, while `S_PATT` refers to the South-telescope. For example, if

```
FITLERS BX, GX
N_PATT 0, 0
S_PATT 1,1
```

the North-telescope will use only the 'BX' filter, while the South-telescope will only use the 'GX' filter. Alternativly, if you were to specify the following:

```
FITLERS BX, GX
N_PATT 0, 1
S_PATT 1,0
```

the two telescopes would alternated between the 'BX' and 'GX' filters. The North-telescope would use 'BX' then 'GX', while the South-telescope would use 'GX' followed by 'BX'.

`EXPTIME` is the exposure time in seconds for each image in the patterns specified. Currently there is no way of specifying different exposures for the two telescopes, SO THE SAME EXPOSURE HAS TO BE USED FOR BOTH TELESCOPES and probably two different filters. This needs to be considered when picking filters a target. Errors are logged if the number of exposure times does not match the filter patterns specified by `N_PATT` and `S_PATT`, or if invalid number are entered into the patterns. `N_PATT` and `S_PATT` should only contain values from and including 0, up to but not including, the number of filters in the `FILTERS` list of the observing recipe. When cycling through each exposure request, the code uses the exposure pattern for the North-telescope for both telescope.

The rest of the observing recipe is not currently implemented, but has the potential to be used during reduction. `DO_FFIELD` is a Yes/No parameter, dictating whether or not the final images should be flat-fielded. `COMSTARS` state how many comparison stars should be used during the photometry reduction, and then the next parameters `COM1`, `COM2`, etc. specify which stars should be used as the comparison stars.

2.6.1 Creating a new observing recipe

Probably the easiest way to create a new observing recipe, would be to copy/paste the contents of an existing recipe in to a new text document, and save with the target name and the extension '.obs'. Note that the target name must match what is in the target information database, with the exception of the BIAS recipe. The BIAS recipe must have BIAS at the start of the name.

Simply change the parameters so they are appropriately set for the new target, and save. When the observing recipe is loaded for target it will take this new information with it. Currently if no observing recipe is available, an error will be raised. All observing recipes should be saved in

```
/home/xamidimuraSoftware/obs_recipes/
```

2.7 Pointing offsets

NOTE

This information is based on the situation when this documentation was written and is correct as of 21/3/19

A pointing offset is the difference where the telescope believes it is pointing and it actual pointing on the sky (having carried out astrometry on an image). To help minimise variations in photometry from the differences in pixel sensitivity, we want to try to keep the target star on the same pixels on the CCD. This means monitoring for any drifts in the actual pointing of the telescope and applying offsets to correct it's pointing.

In terms of the `observing.py` script, it will check to see if a pointing offset needs to be applied when a request to the `take_exposure()` function is request. For the check to be done, using pointing offsets needs to be enabled using the `USE_PONITING_OFFSETS` parameter in `settings_and_error_codes.py`. The offsets will be check for OBJECT type images, after any filter change has been made.

In `observing.py`, if a pointing offsets are in use, the `read_offset_values()` function is called from `update_point_off.py`. As long as the values that are returned are not zero, then a request will be made to the telescope (via the TCS) to apply these offsets. Currently, it assumes the offsets are supplied in degrees. To change this, the `unit` parameter in the `change_point_offset_need()` function of `observing.py` will need to be changed. Valid options are: `deg`, `arcmin` or `arcsec` for degrees, arcminutes and arcseconds, respectively.

2.7.1 Ensuring the pointing offset is up-to-date

The pointing offsets bring together two major parts of the software for this telescope. On one side is the reduction software processing the images to actually calculate the point offset, and on the other there is the observing software interacting with the telescope and requesting exposures.

The role of the `update_point_off.py` script is to keep track of the most recent offset values, whilst ensuring the observing software has access to them whenever it requires. This is done by using a memory mapped file[2]. The two main functions in the script `read_offset_values()` and `update_offset_values()` are responsible for interacting with the memory mapped location. Unsurprisingly, the `read_offset_values()` function will retrieve the current offset values, while the `update_offset_values()` will write new values to the memory location. The location of the mapped file can be specified by the `POINT_OFF_MEM_MAP_FILE_LOC` parameter in `settings_and_error_codes.py`.

The `read_offset_values()` function will open the specified location, and read in the values. If there is an issue reading in the values, for example the mapped file does not exist, then an error is logger and 0.0 is return for both the RA and DEC. **NOTE** Currently the `update_offset_values()` function is not implemented on the data processing side, so whenever a pointing offset is requested, 0.0 is return for both RA and DEC axes. The `read_offset_values()` function only has read access the file. The `update_offset_values()` function will create a file if required, and write offset that are supplied to it, to the mapped location.

When the observing software request a pointing offset, it needs to know that the pointing offset is an up-to-date value and not from say a previous target. To ensure a value is up-to-date, before it get sent to the observing script the modification date of the mapped file is checked. If the last modification is greater that a specified time limit, then the read values are not returned, and 0.0 is return for both RA and DEC instead. The time limit is set in `settings_and_error_codes.py` using the `time_limit_since_last_pointing_update` parameter.

2.7.2 From a terminal

The pointing offset can also be updated by calling the script from a terminal window. Either `cd` to the software directory, or use the full path name. The pointing offset is specified by two floating-point numbers, the first represent the RA offset, and the second the DEC offset. The offsets should be in degrees. Below are two examples of how the it maybe used.

[2]https://en.wikipedia.org/wiki/Memory-mapped_file

```
$cd /home/wasp/xamidimuraSoftware/
$python update_point_off 0.003 -0.045
```

OR:

```
$python /home/wasp/xamidimuraSoftware/update_point_off 0.003 -0.045
```

3 Other Useful Tools & Scripts

3.1 Almanac

The almanac essentially serves two purposes: one is to calculate the time of day i.e. day-time, civil twilight, astronomical twilight, after sunset etc. and the second is to calculate the time at which things such as sunset, sunrise, moonrise will occur. The first of these purposes is used primarily within the main observing script so it can judge whether it needs to be taking flats/bias frame or not open at all). The second will be useful if an observer wishes to find out what time twilight will end etc.

All code used by the almanac is contained within the `getAlmanac.py` script.

3.1.1 Running from the terminal

The script has been made an executable, and an alias added to the `.bashrc` folder in `/home/wasp/`. This means it can be run by typing `almanac` into a terminal window. This can be done in any directory.

Listing 8 gives an example of the output when the `almanac` command is run in a terminal.

Listing 8: Output from the `almanac` terminal command

```
[wasp@swaspobs ~]$ almanac
Fetching almanac times......
~~~~~~~~~~~~~~~~~~~~~~~~~
Times are UTC
~~~~~~~~~~~~~~~~~~~~~~~~~
Next Sunset            2019-03-14 16:52:12.787
Civil twilight         2019-03-14 17:20:42.797
Nautical twilight      2019-03-14 17:49:26.742
Astronomical twilight  2019-03-14 18:18:34.437
~~~~~~~~~~~~~~~~~~~~~~~~~
Next Sunrise           2019-03-15 04:40:07.762
Civil twilight         2019-03-15 04:11:35.818
Nautical twilight      2019-03-15 03:42:50.255
Astronomical twilight  2019-03-15 03:13:41.151
~~~~~~~~~~~~~~~~~~~~~~~~~
Moonrise               2019-03-14 11:45:41.572
Moonset                2019-03-14 22:06:07.883
~~~~~~~~~~~~~~~~~~~~~~~~~
Current (UTC):         2019-03-14 11:23:35.489
```

```
Current (SAST):        2019-03-14 13:23:35.489
Current LST:           00:14:0.730
```

Note that all the sunset, twilight, etc times are in UTC, and are generated based on the current UTC time, which is also listed). The current SAST time is calculate by added 2-hours to the current UTC time. The local mean sidereal time (LST) is calculated using the longitude, latitude and altitude parameters in the `settings_and_error_codes.py` file, and functions from astropy[3]. It it uses the default model which at the time of writing was "IAU2006". The times for sunset, sunrise etc show the next time these events will occur.

3.1.2 Using the functions in the main observing script

The main function from `getAlmanac.py` that is used in the main observing script is the `decide_observing_time()` function. It's job is to determine what time of day it is, i.e. is it daytime, after sunset etc, so the observing script can decide what task it needs to doing (flats, bias, object, waiting for sunset, etc.).

The `decide_observing_time()` function itself uses two other main functions from `getAlmanac.py`, `get_timingsISO()` and `calc_time_differences()`. The first, `get_timingsISO()`, calculates the time of the next sunset, sunrise, twilight times for the site at SAAO as specified by the longitude, latitude and altitude parameters from the `settings_and_error_codes.py` file. These times then get passed to the second function that was mentioned, `calc_time_differences()`. It calculates the difference between the current time and these sunset/sunrise times (time-events). It returns a message denoting the time, the remaining of the current time of day and an the time of a time-event (the specific time-event is different for different times of day). The possible messages that can be returned are as follows:

- **daytime** - Between sunrise and sunset.
- **afterSunset** - After sunset and before evening civil twilight.
- **afterCivil** - After the start of evening civil twilight and before morning nautical twilight.
- **night** - After the start of evening astronomical twilight until the end of morning astronomical twilight.
- **beforeCivil** - After the end of morning astronomical twilight and before the start of morning civil twilight.
- **beforeSunrise** - After the starts of morning civil twilight and before sunrise.

[3]http://docs.astropy.org/en/stable/api/astropy.time.Time.html#astropy.time.Time.sidereal_time

3.1.3 General information about the script

The script uses the python package Astroplan[4] to calculate the timings, e.g sunset, sunrise. This package needs to have an up to date version of the IERS Bulletin A table for the calculations, and if the table is more than 14-day only it normally raises an `OldEarthOrientationDataWarning` warning. Currently the script tries to handle this by automatically downloading a new table if it detects that the warning was raised.

The `decide_observing_time()` function is one of the most useful functions to be used elsewhere, or separately from the main observing script. This maybe useful if you want to find out what time of day it is. To run this function, first open a terminal window and cd to the directory containing the observing software, and run python:

```
[wasp@swaspobs ~]$ cd /home/wasp/xamidimuraSoftware
[wasp@swaspobs ~]$ python
```

Then in the python terminal run the following:

```
>>> import getAlmanac
>>> getAlmanac.decide_observing_time()
```

As an example, running this will yield something similar to what is shown in Listing 9.

Listing 9: Output from the `decide_observing_time` python function

```
Night time, 9.90h until nautical twilight (morning)
('night', 0.41247805, <Time object: scale='utc', format='iso',
 value=2019-03-15 03:13:41.083>)
```

The first line states the current time of day, and then the hours until the time-of-day. It then returns the message, time remaining in days, a the time of a particular time-event. In principle, these could be assign to variables and used as appropriate.

3.2 Blank Sky Fields

The idea behind the Blank-Sky field search tool relatively simple. It will search through a list of blank-sky fields and pick the one that is closest to the zenith position at the time it was run. All the code required for this tool is contained within the `find_best_blank_sky.py` script. There are two ways in which the tool can be used, although both rely on the same functions performing the calculations.

The information about the different blank sky fields is currently stored in:

```
/home/wasp/xamidimuraSoftware/database/Blank_sky_regions.csv
```

The location and file name can be updated using the `settings_and_error_codes.py`.

[4]https://astroplan.readthedocs.io/en/latest/

3.2.1 Functions in the observing script

A key function within `find_best_blank_sky.py` is the `find_best_field()` function, as it uses most of the other functions in that script to calculate which field is closest to zenith at the time the function is called, and returns this as the 'best field'. It will not consider anything like clouds or the position of the moon.

First, the table of blank sky is loaded, converting the RA and DEC values from the typical HH:MM:SS/DD:MM:SS format to a decimal representation. The mean local sidereal time is calculated the sidereal function[5] in Astropy and the location set by the longitude, latitude and altitude in `setting_and_error_codes.py`. It then calculate the hour angle of each field for this sidereal time using the `calc_HA()` function and the equation

$$h = (ST - RA)\%24 \tag{1}$$

where h is the hour angle, ST is the mean local sidereal time and RA is the right ascension of the field in question in decimal hours.

Finally, the zenith distance is calculated. The formula is Eq. 23 from page 57 of "The CCD photometric calibration cookbook"[6], and is shown in Eq. 2.

$$\sec(z) = \frac{1}{(\sin\psi\sin\delta + \cos\psi\cos\delta\cos h)} \tag{2}$$

Here z is the zenith distance, ψ is the latitude of the telescope, δ is the declination of the field and h is the hour angle. Note all values are converted to decimal degrees and then to radians. In our case rather than calculating the inverse secant to find z, the formula is modified to what is shown in Eq. 3,

$$z = \arccos(\sin\psi\sin\delta + \cos\psi\cos\delta\cos h) \tag{3}$$

using $\sec(z) = 1/\cos(z)$. The 'best' blank sky field is taken to be to one where z is minimal.

The function returns the row from the blank sky field table that had the minimal distance, but will also record the best values in the `observingScript` log file.

3.2.2 Running from a terminal

The tool can be run from a terminal window in any directory by using the command `blank_sky`. The command has been set as alias in the `.bashrc` file in `/home/wasp`. Listing 10 is an example of the output from running the `blank_sky` command.

Listing 10: Output from the `blank_sky` command

```
Finding blank field near zenith....
```

[5]http://docs.astropy.org/en/stable/api/astropy.time.Time.html#astropy.time.Time.sidereal_time

[6]http://starlink.rl.ac.uk/star/docs/sc6.pdf

```
Current Sidereal Time: 00:37:49.198
Best blank sky field: RA = 00:42:28.8, DEC = -35:07:22.1, with limit =
    10.0
```

Running the command, execute the `main()` function in `find_best_blank_sky.py`, which along with printing information, makes a call the same `find_best_field()` as when the function get used by the observing script as described in Section 3.2.1.

3.3 Database

The database is currently stored in `/home/wasp/xamidimuraSoftware/database/` and has the name `xamidimura.db`. It currently contains three tables. The `obslog2` table contains a log of all the images that have been taken. It is designed to act as an observing log. The `target_info` contains all the targets that could be observed by the telescope. It contains details such as their coordinates, name, type, and for eclipsing binaries, details relating to the period and orbital parameters. Finally, there is the `priority_table`, which contains the priority information for each of the targets.

As way to help manage the database tables, there is the `connect_database.py`. It contains a number of functions that can be used to modify the table, or run searches on the databases.

3.4 Settings and error codes

In truth, this script is fairly self explanatory. It is where all the directory paths, timeout parameters, etc get stored. By keeping them in one place, if one of the parameters needs to be changed, it only needs to be changed once, and there will be no need to go hunting through all the different scripts to find out where that parameter appears. It also allows the parameters to be shared between all the other scripts without needing to redefine it for each new script.

3.5 TCS commands

All commands that get sent to the TCS machine, get sent using the `send_command()` in the `tcs_control.py` script. In most case the `send_command()` function is incorporated into other functions, so that any error messages are handled. All of the following commands are used to send their equivalent command to the TCS.

All the commands can be used by either importing `observing.py` into a Python terminal and putting `observing.tcs.` in front of the commands, or alternatively importing `tcs_control` into a python terminal and putting `tcs_control.` in front of the commands.

- **telshow_command()** - Returns status information from the telescope, e.g where it is currently pointing, what it's target is.
- **get_tel_target()** - Return the telescope's target RA and DEC.

- **get_tel_pointing()** - Gets where the telescope is currently pointing and whether the telescope is slewing, tracking, homing etc.
- **get_homed_status()** - Returns the homed status of HA-axis, DEC-axis. Other statuses apply to instruments that are not connected.
- **get_camera_status()** - returns if the cameras are exposing, reading or idle, and also what the cooler statuses are.
- **stop_telescope_move()** - Stop any current motion that the telescope has.
- **startTel()** - Start the telescope software on the TCS (not tested).
- **killTel()** - Kill the telescope software on the TCS (not tested).
- **slew_or_track_target(coords, track_target=True)** - Slew to a position in the sky and if `track_target = True` the telescope will track that target. Coordinate should be in the form `['12 32 13.5', '-23 21 35.1']`
- **apply_offset_to_tele(ra_alt_off, dec_az_off, units='arcsec')** - Offset the telescope by an amount equal to `ra_alt_off` and `dec_az_off`.
- **tcs_exposure_request(type, duration, number)** - Command to start an exposure. Specify 'DARK', 'BIAS', 'OBJECT', etc for the type of image.
- **camstart()** - Start up the cameras
- **waspstat()** - Get the current status of the cameras
- **stopwasp()** - Stop the cameras running.
- **scratchmode(state = 'off')** - Toggle whether scratchmode is on/off
- **lastsky()** - Get the sky count and background level of the last image taken with both cameras

4 Things that still need to be done/considered

- Calibrate the temperature coefficients for the focusers' temperature compensation option.
- Integrate the `update_offset_values` function from `update_point_off.py` into the reduction software, and test the pointing offsets actually work. Currently no offset is applied, as expected because an error is raised as there is no file to read.
- Code to handle wanting images not centred on the target coordinates. There are parameters (IMG-RA, IMG-DEC) present in the observing recipes that could be used to specify a different pointing, but will need a bit of code in the observing script to compare the values in the target information database,

and here. If they aren't the same use the pointing from the observing recipe, otherwise the target information database.

- Decide whether or not to specify the focus of a target in the observing recipe. Currently not implemented anyway, but all it would need is a call to the `move_to_position()` function in the `focuser_control.py` in the `take_exposure()` function in `observing.py`, probably around when the pointing offset is applied.
- Modifying the code get the camera to take exposures so the could take exposures of different length. This is probably far from trivial, and additional modifications would then need to be make to the observing script.
- Use the parameter in the observing recipes to control the comparison stars, do flat-fielding options? Decide how the comparison star will be referenced, if used.
- Perhaps have a default recipe that gets used, if a specific recipe is not available for a target.
- Improve the flat field exposure time estimations.
- Make sure the autoflat scripts are waiting properly when the sky is not at an appropriate brightness.
- Put weather checks in `main()` of `observing.py`.
- Decide what to do about the potential of automatically homing the telescope.
- In automated observing script, put in the code to run the scheduler to get a target ID.
- Create 'default' observing recipe that will get loaded if no recipes match a particular target name.
- Consider conditions for stopping observing a scheduled target.
- Write code to handle weather interruptions, and subsequent shutdown procedures
- What happen if something gets stuck...
- Any extra end of night/ start of night procedures
- Close roof at end of night
- Set up plcd.py as a cronjob
- Investigate the stop roof issue