A Description of the Usage and Development of Target of Opportunity Handlers for the Huntsman Project

Bernanda Telalovic

February 24, 2017

AAO Student Fellowship 2016/17

Supervised by Dr. Lee Spitler and Dr. Anthony Horton

Special thanks to Wilfred Gee for all the coding lessons and endless patience

Abstract

Target of Opportunity (ToO) is an observational mode for the Huntsman Array triggered by external alerts. The array will receive email alerts of incoming events, which include gravity wave follow up alerts and other point source alerts such as Supernova events. The software described in this report is used to receive such alerts, handle selecting and sending the relevant targets to the Huntsman control software, POCS.

Contents

Introduction

1.1 Motivation of EM Follow-up of Gravitational Waves and Other Events of Interest

Gravitational Waves were discovered in 2016 by the LIGO and VIRGO Scientific Collaboration [1], from the merger of two black holes. With this discovery, predictions made nearly a century ago from Einstein's General Relativity were confirmed. Further study of gravitational waves if therefore crucial for further testing of the theory, perhaps leading to new physics. Electromagnetic follow-up of LIGO and VIRGO observations is key to discovering the types of cosmic events that generate clear gravitational wave signals. Mergers of binary systems containing neutron stars are particularly theorized to provide good signals in both the gravitational wave detectors and electromagnetic telescopes [2].

Other events of interest consist of alerts about point-source events which are significantly easier to locate and therefore correspondingly easier to handle in the software. Naturally, I have not done so, but for a very simple skeleton code structure of what these event handlers should look like. Examples of such events inculde Supernvae, Gamma Ray Bursts and others. For the person tasked with creating these parsers, section 5 on email parsers will be of great interest.

This report is meant to be a description of my contribution during this internship, as well as a user's guide for future developers. It is not intended as a formal report and therefore referencing will not be formalized in some cases and not all assertions will be justified.

1.2 Huntsman's Contribution

The Huntsman Array consists of optical lenses, configured for low surface brightness imaging. Component cameras can be configured to take exposure for different amounts of time, resulting in a High Dynamic Range observational mode (HDR). Sources of gravitational waves can appear at varying brightnesses and since the source is not immediately known after the trigger is given, it is practical to observe with as high a range as possible. HRD mode will therefore be used when Huntsman engages follow up on gravitational waves.

The Huntsman array has a relatively large field of view and is therefore well suited for covering a large area of sky very quickly. This trait gives it an advantage over some other telescopes observing gravitational wave alerts, as their area of coverage is much smaller. It is important that the software handing these observations gives Huntsman optimized targets that would help cover as many galaxies in a hight probability region as possible.

How We Use Galaxy Catalog and Probability Maps

The sample code provided by Singer et al [3] demonstrates how to download the fits file containing the probability map as well as a catalog from astroquery. The packages which handle gravitational wave events are loosely based on these examples, but further refined to Huntsman's case.

2.1 Galaxy Catalog

The galaxy catalog used for our purposes is the 2MASS Redshift Survey [4], covering 91% of the sky and containing good information about galaxy distances. It covers 44599 galaxies in total, with no selections made. The catalog can be downloaded using the following steps:

```
from astroquery.vizier import Vizier

catalog, = Vizier.get_catalogs('J/ApJS/199/26/table3')
```

Listing 2.1: Importing 2MASS Galaxy Catalog using astroquesry

The catalog functions as an astropy table, which means it is possible to select on properties within the array, such as distance, redshift, RA and DEC. Keep in mind that the selection statement inside the square brackets must be a logical statement. This smaller catalog preserves the structure of the mother catalog, only with some entries removed.

```
smaller_catalog = catalog[(catalog['_RAJ2000'] < 50.0) & (catalog['_DEJ2000'] >
    -45.0)]
```

Listing 2.2: Making cuts on Right-Ascension and Declination

We use selections like the one shown above in the gravity wave event handler, to create a list of galaxies in a high probability region, as determined by the probability map.

2.2 Probability Map

The probability map is available in fits form from a given url. Downloading it is quite simple, as given in [3]. The conversion between a healpix map and an array of length of the galaxy catalog must be done before selection can be made on the catalog based on the probability map:

```
from astropy.utils.data import download_file
import healpy as hp
import numpy as np

event_data = download_file('https://dcc.ligo.org/P1500071/public/10458_bayestar.
    fits.gz')

prob, distmu, distsigma, distnorm = hp.read_map(event_data, field=range(4))

npix = len(prob)
nside = hp.npix2nside(npix)
```

```
pixarea = hp.nside2pixarea(nside)

theta = 0.5*np.pi -catalog['_DEJ2000'].to('rad').value
phi = catalog['_RAJ2000'].to('rad').value
ipix = hp.ang2pix(nside, theta, phi)

# Now the ordered probability corresponds to the catalog
ordered_prob = prob[ipix]
```

Listing 2.3: Making cuts on Right-Ascension and Declination

It is better to calculate probability density, however. We need the galaxy redshift and distance to do this, however.

```
from astropy.cosmology import WMAP9 as cosmo
from astropy.table import Column
import astropy.units as u
import astropy.constants as c

redshift = (u.Quantity(catalog['cz']) / c.c).to(u.dimensionless_unscaled)

r = cosmo.luminosity_distance(redshift).to('Mpc').value

dp_dV = prob[ipix]*distnorm[ipix]*norm(distmu[ipix], distsigma[ipix]).pdf(r) /
    pixarea
```

Listing 2.4: Making cuts on Right-Ascension and Declination

We may then plot this probability map.

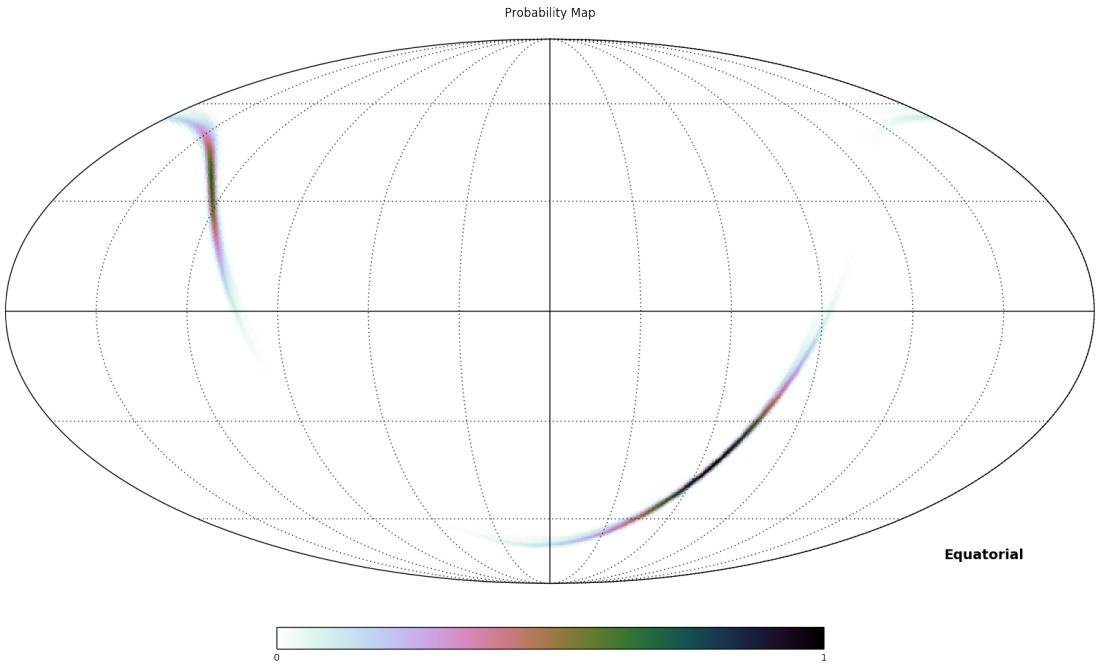

Figure 2.1: The visualisation of the highest probability density regions on the sky for this simple event.

```
# And we can cut on the catalog based on probability
high_prob_galaxies = catalog[dp_dV > np.nanpercentile(dp_dV, 95.0)]
```

Listing 2.5: Making cuts on Right-Ascension and Declination

Then we may superimpose the selected galaxies onto the probability map, showing which galaxies are submitted for selection in this particular probability distribution.

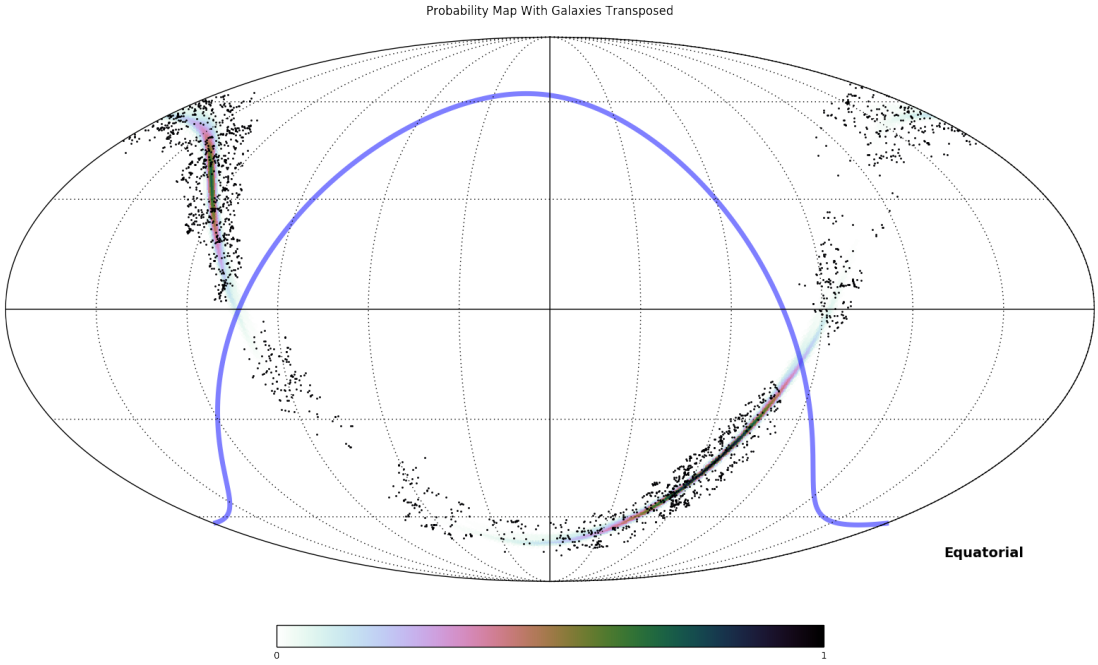

Figure 2.2: The blue line shows the galactic plane, which obscures a portion of the galaxy catalog.

Tiling Algorithm

The main challenge with so large a high-probability region is that it cannot be entirely covered with Huntsman in one night, so we must find areas the size of Huntsman's field of view that cover the most galaxies, weighted to reflect the probability for each particular galaxy in the region. Such an area will henceforth be called a *tile*.

3.1 Gravity Wave Handler

To determine the best way to select the most suitable tiles is to define what we mean by a 'good' tile. One which contains the most galaxies and the highest probability weighting would in this case fit our needs well. Therefore the algorithm implemented uses this definitions as a central criteria for finding the best tiles:

A. While some selection criteria is not met:

   1. For each galaxy in high probability region (as the one shown in Figure 2.2):

      Define a tile around it.

      Find all galaxies contained in the tile.

      Define a "score" as the sum of probabilities associated with each galaxy in the tile.

      Then:

   2. For each tile defined as above

      Find the tile with the highest score and mark it for observation. Flag all galaxies observed like this so that the next iteration doesn't take them into account.

      Return to 1.

This is the main algorithm which is implemented in the script POCS/pocs/utils/too/grav_wave/grav_wave.py, which defines the class GravityWaveEvent, the subject of the next section.

3.2 GravityWaveEvent Class

Description of given arguments

The following arguments are to be given when creating an instance of a GravityWaveEvent.

fits_file **Required**. The url of the fits file we wish to extract the probability map from. Must be an input in the form of a string. Creating an instance of a GravityWaveEvent will take care of downloading and using this file. **TODO:** make the downloader log into the website that provides real alerts. this needs to be done in the class's init method.

galaxy_catalog Default: 'J/ApJS/199/26/table3'. The Vizier address of the catalog we wish to use. Must just be the address, again, the class instance will download and interpret it.

time Default: current time. Start time of event. Must be an instance of astropy.time if given, but is usually unnecessary.

key Default: 'ra': '_RAJ2000', 'dec': '_DEJ2000'. A python dictionary which references the names of columns containing the right ascension and declination in the catalog specified. Please provide this if you change the catalog, otherwise expect errors.

frame Default: 'fk5'. The astronomical frame of reference we're working in. Inadvisable to change.

unit Default: 'deg'. The units in which the dimensions of the field of view is given.

selection_criteria Default: configurable in email_parser config file (see section 5). A python dictionary containing the criteria for when to stop the tiling algorithm. If the name is 'observable_tonight', the number of tiles does not matter, but it will tile the sky observable tonight and stop when the sum of exposure times on top of the time it started tiling is greater than the time of sun rise.

fov Default: configurable in email_parser config file (see section 5). A python dictionary containing the dimensions of the Huntsman field of view.

dist_cut Default: 50.0. The distance cut for galaxies in the catalog. Often given in the notification email for gravitational wave events. Must always be given in Mpc.

alert_pocs Default: True. Determines whether or not we alert POCS. We usually set this to False for testing purposes.

percentile Default: 95.0. The probability cut on the healpix map, determining the region where we find candidate galaxies. The lower this is set the more of the sky we cover.

altitude Default: the altitude recorded in the main config file. The altitude on the sky below which we can't observe well.

observer Default: The observation location in the main config file. The location from which we observe. Must be given as an astroplan.Observer object.

configname Default: "email_parsers" is the name of the config file we want to read the gravity wave selection criteria and field of view from. Must be located in $POCS/conf_files. Location information is read from the local_config.

The attributes of the class can be found in the class docstring.

How location plays into selection

The time taken to complete this tile selection is an important factor, since upon receiving the notice we want start the observations as soon as possible. The greatest factor which reduces the time taken to complete the selection is the number of galaxy candidates going into the loop which defines tiles. The lower this number is, the quicker the algorithm runs. Another consideration is that only a certain area of the sky is visible at the time when the notice is received. Naturally we would like to start by observing tiles in this region first. Therefore incorporating time dependency into the algorithm would allow it to make faster and smarter decisions on what we should be observing.

Time dependency is implemented by creating a variable time which is defined at the start of the loop as the maximum of the current time, the time of start of night and the time of start of event in the email notification. The Horizon class uses the methods zenith_ra_dec and horizon_range to calculate the limiting right ascension and declination given our location, start time and cut off altitude. These limits are used to make a selection on the galaxy catalog, previously selected upon by the probability map of the event. From the leftover galaxies we find the best tiles and mark them for observation. Each time a tile is marked, the time variable is incremented by the exposure time. Then when we repeat the greater loop the limiting right ascension and declination are redefined according to the new time, so that the observable sky always moves with our observations.

How to use

The code handling probability maps and tiling is complete and therefore we do not need to go very in depth into how it works, besides giving an overview of the logic and reasoning behind the algorithm, which the previous sections described. Here I will describe how to create an instance of a gravity wave event and how to extract the tiles we wish to observe.

More examples of probability maps can be found in [5]. Use only files with the extension .fits.

```
from pocs.utils.too.grav_wave.grav_wave import GravityWaveEvent
from astroplan import EarthLocation
from astropy.time import Time
import astropy.units as u

# Let's define a custom location - can be anywhere!
lat = -31.2749 * u.deg
lon = 149.0685 * u.deg
elevat = 1165 * u.m
observer = Observer(longitude=lon, latitude=lat, elevation=elevat)

# Let's define a custom time - here it's the time of writing
time = Time('2017-01-18T11:23:00', frame = 'isot', scale = 'utc')

# Defining distance cut
max_dist = 100.0 # In Mpc

# Define the fits file url
fits_file = 'https://dcc.ligo.org/P1500071/public/10458_ bayestar.fits.gz'

```

```
# Let's get the first 10 tiles visible
selection_criteria = {'name': 'first 10 tiles', 'max_tiles': 10}

# Now we create a GravityWaveEvent
grav_wave = GravityWaveEvent(fits_file, time = time, observer = observer,
    selection_criteria = selection_criteria)

# Now to get the tiles, in a python array:
10_tiles = grav_wave.tile_sky()
```

Listing 3.6: Using the GravityWaveEvent class

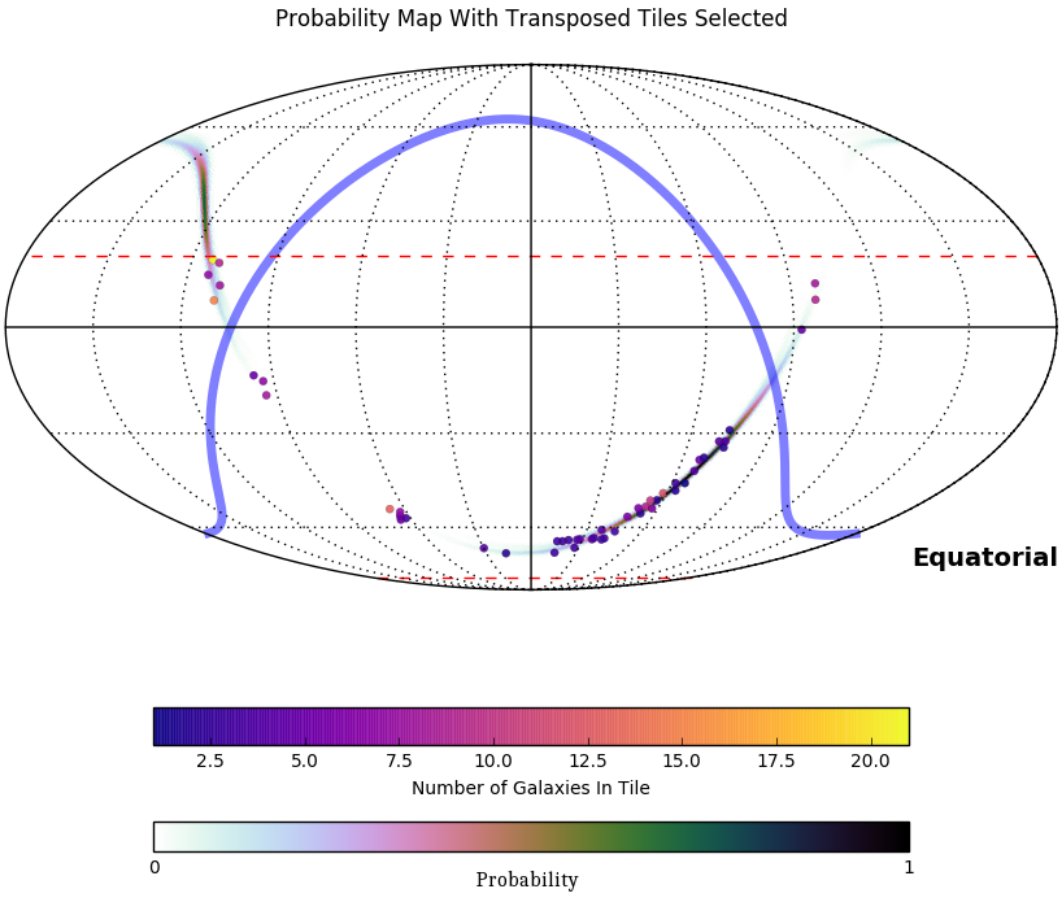

Figure 3.3: The tiles selected by the algorithm over the course of a night of observation. The dotted red lines present the limits of the observable sky below an altitude of 40.0°. The thick blue line shows the galactic plane and we see that in its vicinity the catalog is diminished so no tiles are selected there.

Figure 3.3 shows we select tiles in high probability region as well as high galaxy density regions and limits selection only to the observable patch of sky from Siding Spring Observatory.

Horizon Utility

The GravityWaveEvent class has a time dependence component used to calculate the limits of the observable sky. To that end, it implements the Horizon class, which can have other uses outside of being a utility for handling ToO events. Here I will describe the class and its methods, giving coded examples.

The working methods mainly focus on converting latitude/longitude into right ascension and declination (in FK5 format) which are then used to make selections in the tiling algorithm.

4.1 Initializing A Horizon Class

Input Arguments

observer Required. Must be an instance of astroplan.Observer.

time Default: current time. Must be passed as an astropy.Time object.

altitude Required. The minimum altitude below which we do not want to observe.

Example in code

```
from pocs.utils.too.horizon.horizon_range import Horizon
from astroplan import Observer
from astropy.time import Time
import astropy.units as u

# Let's define a custom location - can be anywhere!
lat = -31.2749 * u.deg
lon = 149.0685 * u.deg
elevat = 1165 * u.m
observer = Observer(longitude=lon, latitude=lat, elevation=elevat)

# Let's define a custom time - here it's the time of writing
tim = Time('2017-01-18T11:23:00', frame = 'isot', scale = 'utc')

# Let's define an altitude
altitude = 10.0 * u.deg

# Now we define the horizon class
horizon = Horizon(observer, altitude, time = tim)
```

Listing 4.7: Making an instance of Horizon

4.2 Methods

4.2.1 start_time

The method takes an input time and location and returns the maximum value in an array containing the current time, the time given as an argument and the time of start of night from the

given location. It is used by the GravityWaveEvent to calculate the first time instance at which we start observing.

The output is an astropy.Time object.

Input Arguments

time Default: current time

location Default: whatever location was used when Horizon was initialized. If given, must be as EarthLocation.

4.2.2 nesw_ra_dec

This method returns the right ascension and declination coordinates for North, East, South, West.

Input Arguments

time Default: current time

location Default: whatever location was used when Horizon was initialized. If given, must be as EarthLocation.

altitude Default: Whatever was given as Horizon was initialized. Must be given as a floating point number.

Example in code

```
from pocs.utils.too.horizon.horizon_range import Horizon
from astroplan import Observer
from astropy.time import Time
import astropy.units as u

# Let's define a custom location - can be anywhere!
lat = -31.2749 * u.deg
lon = 149.0685 * u.deg
elevat = 1165 * u.m
observer = Observer(longitude=lon, latitude=lat, elevation=elevat)

# Let's define an altitude
altitude = 10.0 * u.deg


horizon = Horizon(observer, altitude)

nesw_ra_dec = horizon.nesw_ra_dec()
```

Which gives the output as a python dictionary:

```
{'north:': <FK5 Coordinate (equinox=J2000.000): (ra, dec) in deg (273.72903087,
    18.71953769)>, 'west': <FK5 Coordinate (equinox=J2000.000): (ra, dec) in deg
    (219.31867938, -19.4222874)>, 'east': <FK5 Coordinate (equinox=J2000.000): (ra,
     dec) in deg (328.03253624, -19.57187037)>, 'south': <FK5 Coordinate (equinox=
    J2000.000): (ra, dec) in deg (273.11459695, -81.28347025)>}
```

Listing 4.8: Using nesw_ra_dec()

4.2.3 zenith_ra_dec

This method returns the right ascension and declination with and altitude of 90.0°, corresponding to the zenith point from the given location and time.

Input Arguments

time Default: current time

location Default: whatever location was used when Horizon was initialized. If given, must be as EarthLocation.

Example in code

```
from pocs.utils.too.horizon.horizon_range import Horizon
from astroplan import Observer
from astropy.time import Time
import astropy.units as u

# Let's define a custom location - can be anywhere!
lat = -31.2749 * u.deg
lon = 149.0685 * u.deg
elevat = 1165 * u.m
observer = Observer(longitude=lon, latitude=lat, elevation=elevat)

# Let's define an altitude
altitude = 10.0 * u.deg


horizon = Horizon(observer, altitude)
zenith = horizon.zenith_ra_dec()
```

Which gives the output as a python dictionary:

```
{'dec': <Latitude -31.28219361334 deg>, 'ra': <Longitude 273.640873886 deg>}
```

Listing 4.9: Using zenith_ra_dec()

4.2.4 horizon_range

This method returns the limiting right ascension and declination in degrees calculated with respect to the given altitude and location at the given time.

Input Arguments

zenith **Required**. Pass the zenith given by the previous method.

altitude Default: Whatever was given as Horizon was initialized. Must be given as a floating point number.

Example in code

```
from pocs.utils.too.horizon.horizon_range import Horizon
from astroplan import Observer
from astropy.time import Time
import astropy.units as u

# Let's define a custom location - can be anywhere!
lat = -31.2749 * u.deg
lon = 149.0685 * u.deg
elevat = 1165 * u.m
```

```
observer = Observer(longitude=lon, latitude=lat, elevation=elevat)

# Let's define an altitude
altitude = 10.0 * u.deg


horizon = Horizon(observer, altitude)
zenith = horizon.zenith_ra_dec()
horizon_range = horizon.horizon_range(zenith)
```

Which gives the output as a python dictionary:

```
{'min_dec': <Angle -81.282193613 deg>, 'max_ra': <Angle 323.640873886 deg>, '
    max_dec': <Angle 18.71780638665 deg>, 'min_ra': <Angle 223.6408738860 deg>}
```

Listing 4.10: Using horizon_range()

Alerter

The Alerter class contains methods which basically put the array of tiles in a format which POCS can recognise and receive as a message.

4.1 Initializing Alerter

Input Arguments

test Default: False.

port_num Default: 6500. Be very careful changing this.

4.2 Methods

4.2.1 send_alert

Sends pocs a message along the port with which the alerter was initialized.

Input Arguments

type **Required**. If the message is a retraction, please make this argument contain the full word 'retraction', either as 'retraction' or 'Retraction'.

message **Required**. Needs to be a list where each entry has a property 'coords' for coordinates, 'exp_time' for the exposure time of the observation and 'name', for obvious purposes, as well as other relevant attributes.

Example in code

```
from pocs.utils.too.alert_pocs import Alerter()

# Let's define a simple message
message = {'name': 'Some name', 'coords': '00h00m00.0s 00d00m00.0s', 'exp_time': '
    10.0'}

alerter = Alerter()

# Sending message
alerter.send_alert('add', message)
```

Listing 4.11: Using alert_pocs()

EmailParser

EmailParser is the base class for target of opportunity alerts received via email. Subclasses must be defined to handle the specific alerts. Currently the gravity wave parser is fully operational with a few minor corrections to be made and there are two skeleton subcalsses for Supernovae and Gamma Ray Bursts that need to be modified to handle the source of the alert.

5.1 Initializing EmailParser

Input Arguments

Initialized in the base class:

host **Required**. The host of the email server, given as a string. Use 'imap.gmail.com' for gmail.

address **Required**. The email address, given as a string.

password **Required**. The password for the given email, as a string.

test Default: False.

verbose Default: False. Enables print statements in both the parser and the event handler, if it exists.

Attributes specific to GravityWaveEmailParser:

selection_criteria Same as in Chapter 3.2.

observer Default: location as described in the main config file. Determines which location we observe from.

altitude Default: altitude as described in the main config. Determines the lowest altitude below which we do not observe.

fov Default: the field of view specified under grav wave in the email_parsers config.

5.2 Methods

5.2.1 get_email

This is a method of the base class and is independent of each separate parser.Gets the email with a specified subject line. This method will also mark said email as read. Returns a string containing all text and a variable that checks if the email has been read, the text of the body of the email as a single string and the a variable which tells us if the email has been marked as seen. This last returned value is important for the use by the email monitor (see 5.3) because it breaks the checking email loop if the email can't be marked as seen. If not for this check, the email monitor would keep trying to read the same email and send several instances of the same alert until manually shut down.

Input Arguments

type **Required**. The subject, given as a string.

folder Default: 'inbox'. The folder in which the email is located.

mark_as_read Default: True. If true, it will attempt to mark the email as read. For not marking the email as read, set this to False.

Example in code

```
from pocs.utils.too.email_parser.email_parser import ParseEmail

email = ParseEmail('imap.gmail.com', 'sample_email', 'sample_password')

read, text, marked_as_read = email.get_email('Some subject', folder = 'inbox,
    mark_as_read = True)
```

Listing 5.12: Using get_email()

In this case, if the email with the subject exists exists

5.2.2 read_email

This method is unique to each parser. This section describes the GravityWaveParseEmail. The method separates the text into a python dictionary. It is expecting text such as the example notifications given in [6].

Input Arguments

text **Required**. The text string from get_email.

Example in code

```
from pocs.utils.too.email_parser.email_parser import GravityWaveParseEmail

grav_email = GravityWaveParseEmail('imap.gmail.com', 'sample_email', '
    sample_password')

text, read, marked = grav_email.get_email('Some subject', folder = 'inbox',
    mark_as_read = True)

if read is True:
  message = grav_email.read_email(text)
```

Listing 5.13: Using read_email()

5.2.3 parse_event

This method is unique to each parser. This section describes the GravityWaveEmailParser. Creates a GravityWaveEvent instance using the relevant arguments from the message read by rea_email. It then calls tile_sky() which defines the tiles and alerts POCS (only if test is False). It will return the list of targets found by the event handler.

Input Arguments

text **Required**. The body of the read email. This method will internally use read_email to get the dictionary and then acquire targets.

Example in code

```
from pocs.utils.too.email_parser.email_parser import GravityWaveParseEmail

email = ParseEmail('imap.gmail.com', 'sample_email', 'sample_password')

text, read, marked = email.read_email('Some subject', folder = 'inbox')

if read is True:
  targets = grav_email.parse_event(text)
```

Listing 5.14: Using read_email()

5.3 email_monitor

The email_monitor script is not part of the EmailParser class, but rather a stand alone file which is run at the command line and has the same input arguments as the EmailParser class. If you would like a more detailed description of input arguments, simply run:

```
$ python $POCS/pocs_alerter/email_monitor.py --help
```

Listing 5.15: Using email_monitor

Which gives the output:

```
  -h, --help            show this help message and exit
  --host HOST           The email server host. Use default for gmail.
  --email EMAIL         The email address. Required.
  --password PASSWORD   The password. Required.
  --test TEST           Turns on testing.
  --rescan_interval RESCAN
                        Sets the frequency of email checks. Must be given in
                        minutes.
  --subjects SUBJECTS   The email subjects which we want to read. Must be a
                        python list containing strings which exactly match the
                        email subjects.
  --config CONFIG       The local config file containing information about the
                        Field of Vew and the selection_criteria
  --alert_pocs ALERT_POCS
                        Tells the code whether or not to alert POCS with found
                        targets
  --selection_criteria SELECTION_CRITERIA
                        The python dictionary containint our selection
                        criteria
  --verbose             Activates print statements.
```

Listing 5.16: Using email_monitor

The monitor script creates instances of all email parsers defined in the email_parsers config file and loops indefinitely (until keyboard break) to check for new emails, read them and find the targets.

5.4 email_parsers config

This config file contains information about all the external trigger parsers. As of time of writing, the gravity wave parser is complete and there are two skeleton subclasses for handling Supernovae and Gamma Ray Bursts. Knowledge of the source of these interrupts will be crucial in modifying the parsers to specifically handle these events.

An example of the email_eparsers config:

```
---
email_parsers:
-
    type: GravWaveParseEmail
    inputs:
        selection_criteria:
            name: 3_tiles
            max_tiles: 3
        fov:
            ra: 3.0
            dec: 2.0
        alert_pocs: True
    subjects:
        -
            CSV/LVC_INITIAL
-
    type: SupernovaParseEmail
    inputs:
        none: none
    subjects:
        -
            Supernova
-
    type: GRBParseEmail
    inputs:
        none: none
    subjects:
        -
            GRB
```

Listing 5.17: email_parsers.yaml example

The subjects in each of the parsers reference what subjects of emails we're looking out for. More than one can be specified.

Putting it All Together

This is the outline of how all the utilities described here function together. This chapter is essentially the lineup of how each function is called. Since it is an algorithm, t is best represented as a diagram in Figure 6.4. The methods used at each step are described within the boxes.

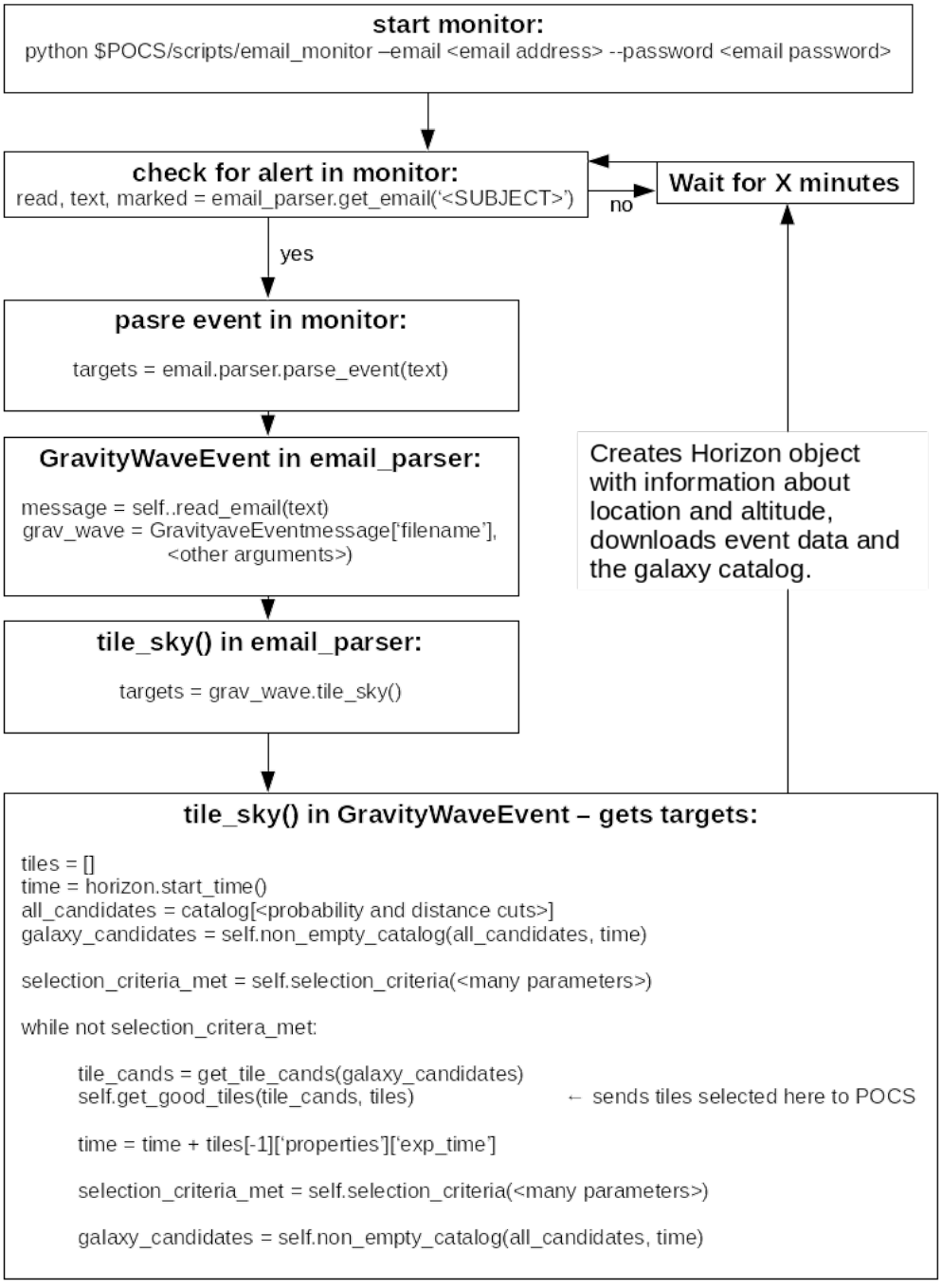

Figure 6.4: The visualisation of the full ToO event handling algorithm.

Further Work

5.1 Downloading probability maps from true Gravity Wave alerts

The emails give us links to the fitz file containing the probability map, but accessing it requires the huntsman's username and password. This will need to be handled in the gravity_wave code, and the actual account name and password should be kept in the email_parsers config file. I have not done any work on this front and I'm not sure exactly what it would entail.

5.2 Creating other ToO email parsers

The sources for the other alerts need to be determined and then the skeleton methods need to be modified to handle the incoming emails and the way the target will be sent to the control software.

5.3 Saving individual tiles as text files

Lee has expressed interest in having a list of all galaxies in a particular tile saved as a text file. This may also help with the removal of these tiles in case of a retraction. Currently there is a commented-out method in get_good_tiles which is supposed to do this, but there is an issue with python encoding strings that I haven't quite gotten around to fixing. Basically the text that needs to be printed is ready, under tile['text'], as is the title of each file, but the code doesn't actually print the tile properly.

5.4 Programming Retractions

Gravitational wave notices can also contain retractions to events. Every event has its own designation, which is carried in the name of each tile associated with the event. Therefore once the retraction has been received as an email and separated into a python dictionary, a method needs to be written which will take the keywords from the retraction message and alert POCS to remove the observations. The AlertPocs class is already equipped to handle this, but the logic or removing the tiles still needs to be written into POCS itself and into the email parser.

Suggestions

I would suggest that POCS be modified so that when it receives a remove observation message, it can remove them by name, or by substring in name. That way the email parser could just alert POCS with the name of the event, and the even can then be removed.

5.5 Modifying Targets

We will be receiving follow up notifications form other telescopes doing gravitational wave follow up. It needs to be decided whether or not we want to listen out for these and observe them. I have not done any work on this front.

5.6 Sending Notifications

As part of the agreement to subscribe to gravitational wave notices, we must send out a notice if we find anything interesting. This needs to be written and the format of the message needs to conform to the sample notices given in [6].

Bibliography

[1] B. P. Abbott et al. Physics Review Letters 116:061102 (2016)
(URL: https://physics.aps.org/featured-article-pdf/10.1103/PhysRevLett.116.061102)

[2] LIGO and VIRGO Collaboration. 'Implementation and Testing of the First Prompt Search for Gravitational Wave Transients with Electromagnetic Counterparts', Astronomy and Astrophysics manuscript (2012).
(URL: https://arxiv.org/pdf/1109.3498v2.pdf)

[3] L. P. Singer et al SUPPLEMENT: GOING THE DISTANCE: MAPPING HOST GALAXIES OF LIGO AND VIRGO SOURCES IN THREE DIMENSIONS USING LOCAL COSMOGRAPHY AND TARGETED FOLLOW-UP" (2016)
(URL: arXiv:1605.04242v3)

[4] 2MASS Galaxy Redshift Catalog
(URL: https://www.cfa.harvard.edu/ dfabricant/huchra/2mass/)

[5] Going the Distance: Mapping Host Galaxies of LIGO and Virgo Sources in Three Dimensions Using Local Cosmography and Targeted Follow-up (URL: https://dcc.ligo.org/P1500071/public)

[6] GCN/ LVC notices. (URL: https://gcn.gsfc.nasa.gov/lvc.html)