astroplan Documentation

Release 0.11.dev1+g3d966a3

Astroplan developers

Apr 05, 2024

CONTENTS

astroplan is an open source Python package to help astronomers plan observations.

The goal of astroplan is to make a flexible toolbox for observation planning and scheduling. When complete, the goal is to be easy for Python beginners and new observers to to pick up, but powerful enough for observatories preparing nightly and long-term schedules.

Features:

- Calculate rise/set/meridian transit times, alt/az positions for targets at observatories anywhere on Earth
- Built-in plotting convenience functions for standard observation planning plots (airmass, parallactic angle, sky maps).
- Determining observability of sets of targets given an arbitrary set of constraints (i.e., altitude, airmass, moon separation/illumination, etc.).
- astropy powered!
- Source code
- Docs
- Issues

License: BSD-3

Part I

Documentation

CHAPTER

ONE

INSTALLATION

1.1 Requirements

astroplan works on Linux, Mac OS X and Windows. It requires Python 3.7+ as well as numpy, astropy, pytz, and six. Additional features are available when you install Matplotlib and astroquery.

First-time Python users may want to consider an all-in-one Python installation package, such as the Anaconda Python Distribution which provides all of the above dependencies.

1.2 Installation

You can install the stable version of astroplan from PyPI with:

```
pip install astroplan
```

or from anaconda:

```
conda install -c conda-forge astroplan
```

Alternatively, you can install the latest developer version of astroplan by cloning the git repository:

```
git clone https://github.com/astropy/astroplan
```

. . . then installing the package with:

```
cd astroplan
pip install .
```

1.3 Testing

If you want to check that all the tests are running correctly with your Python configuration, run the following from the command line:

```
tox -e test
```

If there are no errors, you are good to go!

Note: If you want to run the tests that access the internet, you'll need to replace the last line above with `tox -e test -- --remote-data` and have an active connection to the internet. Also, if you want the tests that check plotting to work, you need Matplotlib and pytest-mpl.

1.4 More

astroplan follows astropy's guidelines for affiliated packages–installation and testing for the two are quite similar! Please see astropy's installation page for more information.

CHAPTER

TWO

GETTING STARTED

2.1 General Guidelines

`astroplan` is based on astropy and was built around the creation of Python objects that contain all the information needed to perform certain tasks. You, the user, will create and manipulate these objects to plan your observation. For instance, an `Target` object contains information associated with targets, such as right ascension, declination, etc.

Objects representing celestial bodies like stars (which, if we ignore proper motion, are fixed on the celestial sphere) are created (or "instantiated") via an `FixedTarget` object:

```
from astropy.coordinates import SkyCoord
from astroplan import FixedTarget

coordinates = SkyCoord('19h50m47.6s', '+08d52m12.0s', frame='icrs')
altair = FixedTarget(name='Altair', coord=coordinates)
```

Alternatively, for objects known to the CDS name resolver, you can quickly retrieve their coordinates with `from_name`:

```
altair = FixedTarget.from_name('Altair')
```

Similarly, an `Observer` object contains information about the observatory, telescope or place where you are observing, such as longitude, latitude, elevation and other optional parameters. You can initialize an `Observer` object via the `at_site` class method:

```
from astroplan import Observer
observer = Observer.at_site('subaru')
```

Or you can specify your own location parameters:

```
import astropy.units as u
from astropy.coordinates import EarthLocation
from pytz import timezone
from astroplan import Observer

longitude = '-155d28m48.900s'
latitude = '+19d49m42.600s'
elevation = 4163 * u.m
location = EarthLocation.from_geodetic(longitude, latitude, elevation)

observer = Observer(name='Subaru Telescope',
                    location=location,
                    pressure=0.615 * u.bar,
```

(continues on next page)

(continued from previous page)

```
                relative_humidity=0.11,
                temperature=0 * u.deg_C,
                timezone=timezone('US/Hawaii'),
                description="Subaru Telescope on Maunakea, Hawaii")
```

astroplan makes heavy use of certain astropy machinery, including the coordinates objects and transformations and units. Most importantly for basic use of astroplan is the representation of dates/times as Time objects (note that these are in the UTC timezone by default):

```
from astropy.time import Time
time = Time(['2015-06-16 06:00:00'])
```

2.2 Doing More

Now that you know the basics of working with astroplan, check out our *Tutorials* page for high-level examples of using astroplan, as well as the *Reference/API* section for more exhaustive documentation and lower-level usage examples.

CHAPTER

THREE

TUTORIALS

If you'd like to see how `astroplan` is used in the context of real observation planning examples, this is the page for you!

Can't find what you're looking for here? Check out our *Reference/API*.

Is there something you think we should add here? Consider posting an issue on GitHub asking for it… Or better yet, write it yourself, and become a project contributor!

We currently have the following tutorials:

3.1 Observing the Summer Triangle

Note: Your calculated rise/set and other times may differ slightly from those in this tutorial, on the order of ~1 second. This is a normal variance in precision due to several factors, including varying *IERS tables* and machine architecture.

3.1.1 Contents

3.1.2 Defining Objects

Say we want to look at the Summer Triangle (Altair, Deneb, and Vega) using the Subaru Telescope.

First, we define our `Observer` object:

```
>>> from astroplan import Observer

>>> subaru = Observer.at_site('subaru')
```

Then, we define our `FixedTarget`'s, since the Summer Triangle is fixed with respect to the celestial sphere (if we ignore the relatively small proper motion). We will use the `from_name` class method, which queries the CDS name resolver for your target's coordinates (giving you the power of SIMBAD!):

```
>>> from astropy.coordinates import SkyCoord
>>> from astroplan import FixedTarget

>>> altair = FixedTarget.from_name('Altair')
>>> vega = FixedTarget.from_name('Vega')
```

For objects that can't be resolved with `from_name`, you can enter coordinates manually:

```
>>> coordinates = SkyCoord('20h41m25.9s', '+45d16m49.3s', frame='icrs')
>>> deneb = FixedTarget(name='Deneb', coord=coordinates)
```

We also have to define a `Time` (in UTC) at which we wish to observe. Here, we pick 2AM local time, which is noon UTC during the summer:

```
>>> from astropy.time import Time

>>> time = Time('2015-06-16 12:00:00')
```

Return to Top

3.1.3 Observable?

Next, it would be handy to know if our targets are visible from Subaru at the time we settled on. In other words–are they above the horizon while the Sun is down?

```
>>> subaru.target_is_up(time, altair)
True

>>> subaru.target_is_up(time, vega)
True

>>> subaru.target_is_up(time, deneb)
True
```

…They are!

What if we weren't sure if the Sun is down at this time:

```
>>> subaru.is_night(time)
True
```

…It is!

However, we may want to find a window of time for tonight during which all three of our targets are above the horizon *and* the Sun is below the horizon (let's worry about light pollution from the Moon later).

Let's define the window of time during which all targets are above the horizon.

Note that because of the precision limitations of rise/set calculations (altitudes at these times won't equal precisely zero, but will be off by a few arc seconds), we'll manually adjust rise/set times by a few minutes.

```
>>> import numpy as np
>>> import astropy.units as u

>>> altair_rise = subaru.target_rise_time(time, altair) + 5*u.minute
```

(continues on next page)

(continued from previous page)

```
>>> altair_set = subaru.target_set_time(time, altair) - 5*u.minute

>>> vega_rise = subaru.target_rise_time(time, vega) + 5*u.minute
>>> vega_set = subaru.target_set_time(time, vega) - 5*u.minute

>>> deneb_rise = subaru.target_rise_time(time, deneb) + 5*u.minute
>>> deneb_set = subaru.target_set_time(time, deneb) - 5*u.minute

>>> all_up_start = np.max([altair_rise, vega_rise, deneb_rise])
>>> all_up_end = np.min([altair_set, vega_set, deneb_set])
```

Now, let's find sunset and sunrise for tonight (and confirm that they are indeed those for tonight):

```
>>> sunset_tonight = subaru.sun_set_time(time, which='nearest')

>>> sunset_tonight.iso
'2015-06-16 04:59:11.267'
```

This is '2015-06-15 18:59:11.267' in the Hawaii time zone (that's where Subaru is).

```
>>> sunrise_tonight = subaru.sun_rise_time(time, which='nearest')

>>> sunrise_tonight.iso
'2015-06-16 15:47:35.822'
```

This is '2015-06-16 05:47:35.822' Hawaii time.

Sunset and sunrise check out, so now we define the limits of our observation window:

```
>>> start = np.max([sunset_tonight, all_up_start])
>>> start.iso
'2015-06-16 06:28:40.126'

>>> end = np.min([sunrise_tonight, all_up_end])
>>> end.iso
'2015-06-16 15:47:35.822'
```

So, our targets will be visible (as we've defined it above) from '2015-06-15 20:28:40.126' to '2015-06-16 05:47:35.822' Hawaii time. Depending on our observation goals, this window of time may be good enough for preliminary planning, or we may want to optimize our observational conditions. If the latter is the case, go on to the Optimal Observation Time section (immediately below).

Return to Top

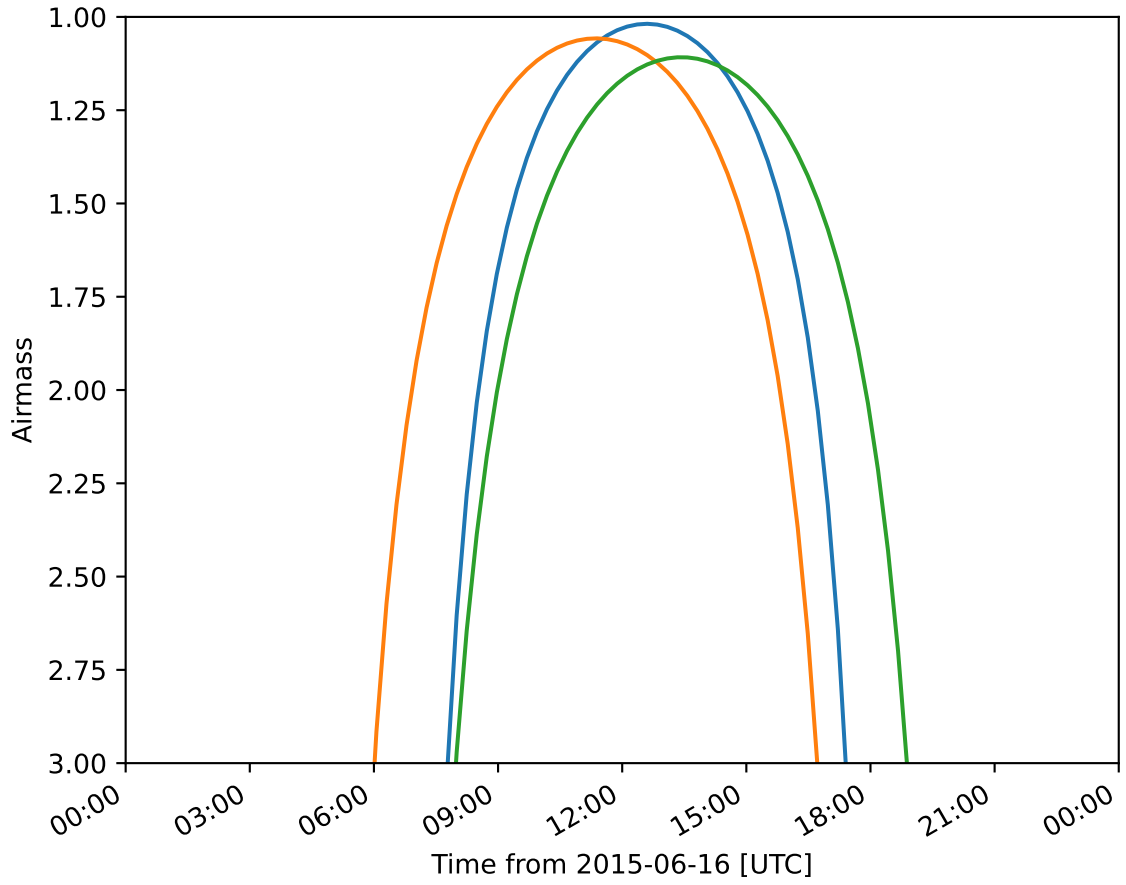
3.1.4 Optimal Observation Time

There are a few things we can look at to find the best time to observe our targets on a given night.

Airmass

To get a general idea of our targets' airmass on the night of observation, we can plot it over the course of the night (for more on plotting see *Plotting with Astroplan*):

```
>>> from astroplan.plots import plot_airmass
>>> import matplotlib.pyplot as plt

>>> plot_airmass(altair, subaru, time)
>>> plot_airmass(vega, subaru, time)
>>> plot_airmass(deneb, subaru, time)

>>> plt.legend(loc=1, bbox_to_anchor=(1, 1))
>>> plt.show()
```



We want a minimum airmass when observing, and it looks like sometime between 9:00 and 15:00 UTC (or 23:00 on the 15th to 5:00 on the 16th, US/Hawaii) would be the best time to observe all three targets.

However, if we want to define a more specific time window based on airmass, we can calculate this quantity directly. To get airmass measurements, we need to use the `AltAz` frame:

```
>>> subaru.altaz(time, altair).secz
<Quantity 1.0302347952130682>

>>> subaru.altaz(time, vega).secz
<Quantity 1.0690421636016616>

>>> subaru.altaz(time, deneb).secz
<Quantity 1.167753811648361>
```

Behind the scenes here, `subaru.altaz(time, altair)` is actually creating an `AltAz` object in the `AltAz` frame, so if you know how to work with `coordinates` objects, you can do lots more than just computing airmass.

Parallactic Angle

To get a general idea of our targets' parallactic angle on the night of observation, we can make another plot (again, see *Plotting with Astroplan* for more on customizing plots and the like):

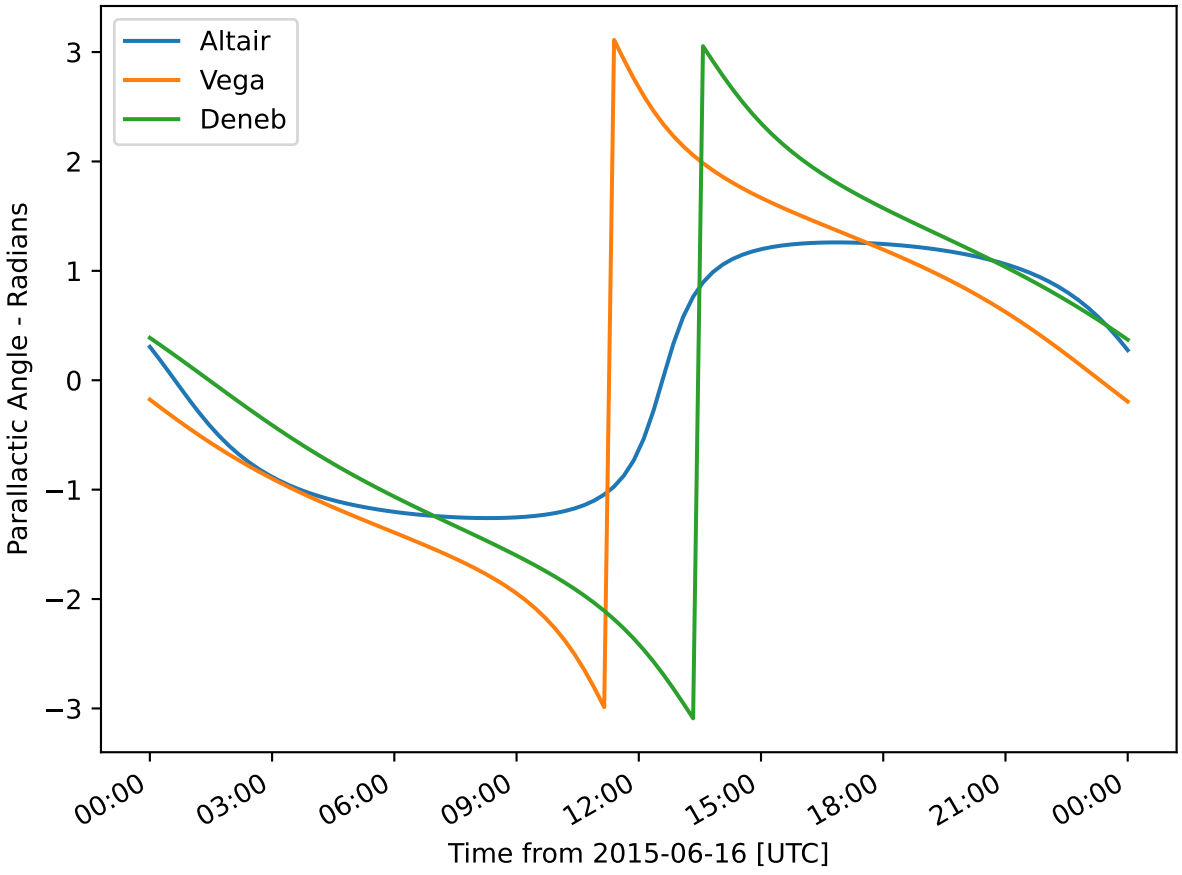
```
>>> import matplotlib.pyplot as plt
>>> from astroplan.plots import plot_parallactic

>>> plot_parallactic(altair, subaru, time)
>>> plot_parallactic(vega, subaru, time)
>>> plot_parallactic(deneb, subaru, time)

>>> plt.legend(loc=2)
>>> plt.show()
```

We can also calculate the parallactic angle directly:

```
>>> subaru.parallactic_angle(time, altair)
<Angle -0.6404957821112053 rad>

>>> subaru.parallactic_angle(time, vega)
<Angle -0.46542183982024 rad>

>>> subaru.parallactic_angle(time, deneb)
<Angle 0.7297067855978494 rad>
```

The `Angle` objects resulting from the calls to `parallactic_angle()` are subclasses of the `Quantity` class, so they can do everything a `Quantity` can do - basically they work like numbers with attached units, and keep track of units so you don't have to.

For more on the many things you can do with these, take a look at the Astropy documentation or tutorials. For now the most useful thing is to know is that `angle.degree`, `angle.hourangle`, and `angle.radian` give you back Python floats (or `numpy` arrays) for the angle in degrees, hours, or radians.


Altair
Vega
Deneb
Parallactic Angle - Radians
3
2
1
0
−1
−2
−3
00:00
03:00
06:00
09:00
12:00
15:00
18:00
21:00
00:00
Time from 2015-06-16 [UTC]

The Moon

If you need to take the Moon into account when observing, you may want to know when it rises, sets, what phase it's in, etc. Let's first find out if the Moon is out during the time we defined earlier:

```
>>> subaru.moon_rise_time(time)
<Time object: scale='utc' format='jd' value=2457190.1696768994>

>>> subaru.moon_set_time(time)
<Time object: scale='utc' format='jd' value=2457189.684134357>
```

We could also look at the Moon's alt/az coordinates:

```
>>> subaru.moon_altaz(time).alt
<Latitude -45.08860929634166 deg>

>>> subaru.moon_altaz(time).az
<Longitude 34.605498354422686 deg>
```

It looks like the Moon is well below the horizon at the time we picked before, but we should check to see if it will be out during the window of time our targets will be visible (again–as defined at the beginning of this tutorial):

```
>>> visible_time = start + (end - start)*np.linspace(0, 1, 20)

>>> subaru.moon_altaz(visible_time).alt
<Latitude [-25.21127325,-30.68088873,-35.82145644,-40.53415037,
           -44.68898859,-48.12296182,-50.64971858,-52.08946099,
           -52.31849772,-51.31548444,-49.17038499,-46.04862654,
           -42.13887599,-37.61479774,-32.61875342,-27.26048709,
           -21.62215227,-15.76463668, -9.73313141, -2.19408792] deg>
```

Looks like the Moon will be below the horizon during the entire time.

Return to Top

3.1.5 Sky Charts

Now that we've determined the best times to observe our targets on the night in question, let's take a look at the positions of our objects in the sky.

We can use `plot_sky` as a sanity check on our target's positions or even just to better visualize our observation run.

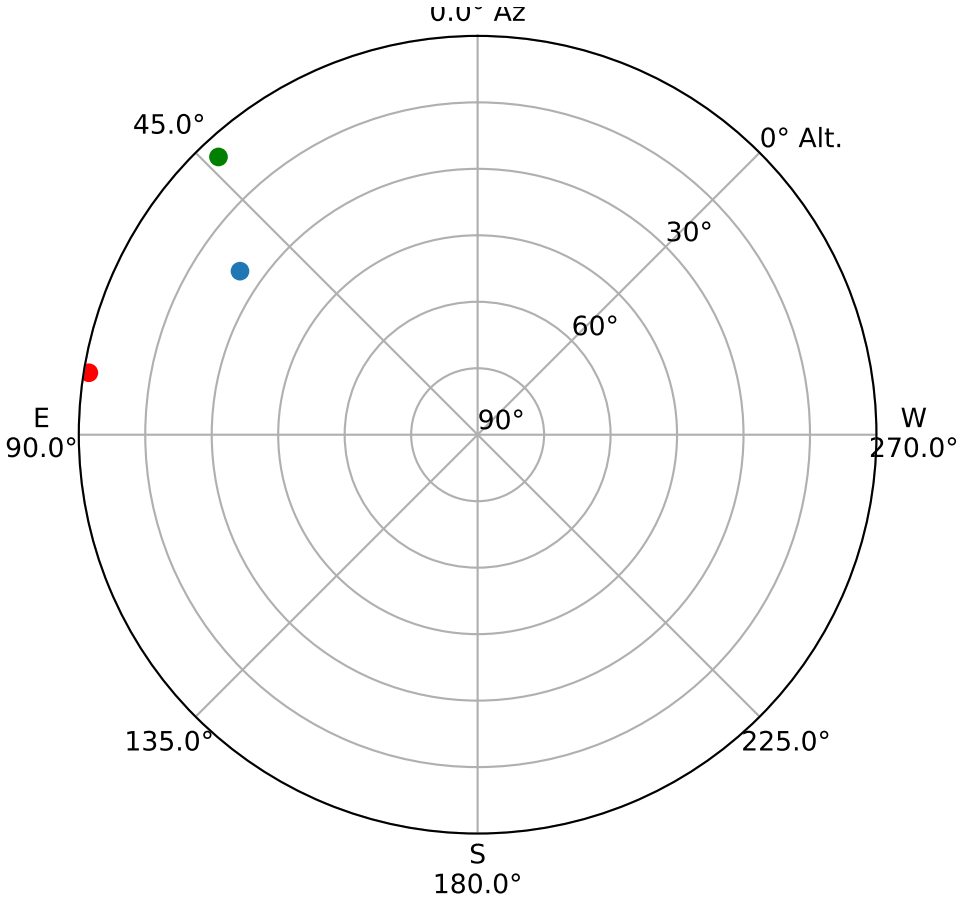
Let's take the `start` and `end` of the time window we determined *earlier* (using the most basic definition of "visible" targets, above the horizon when the sun is down), and see where our targets lay in the sky:

```
>>> from astroplan.plots import plot_sky
>>> import matplotlib.pyplot as plt

>>> altair_style = {'color': 'r'}
>>> deneb_style = {'color': 'g'}

>>> plot_sky(altair, subaru, start, style_kwargs=altair_style)
>>> plot_sky(vega, subaru, start)
>>> plot_sky(deneb, subaru, start, style_kwargs=deneb_style)
```

(continues on next page)

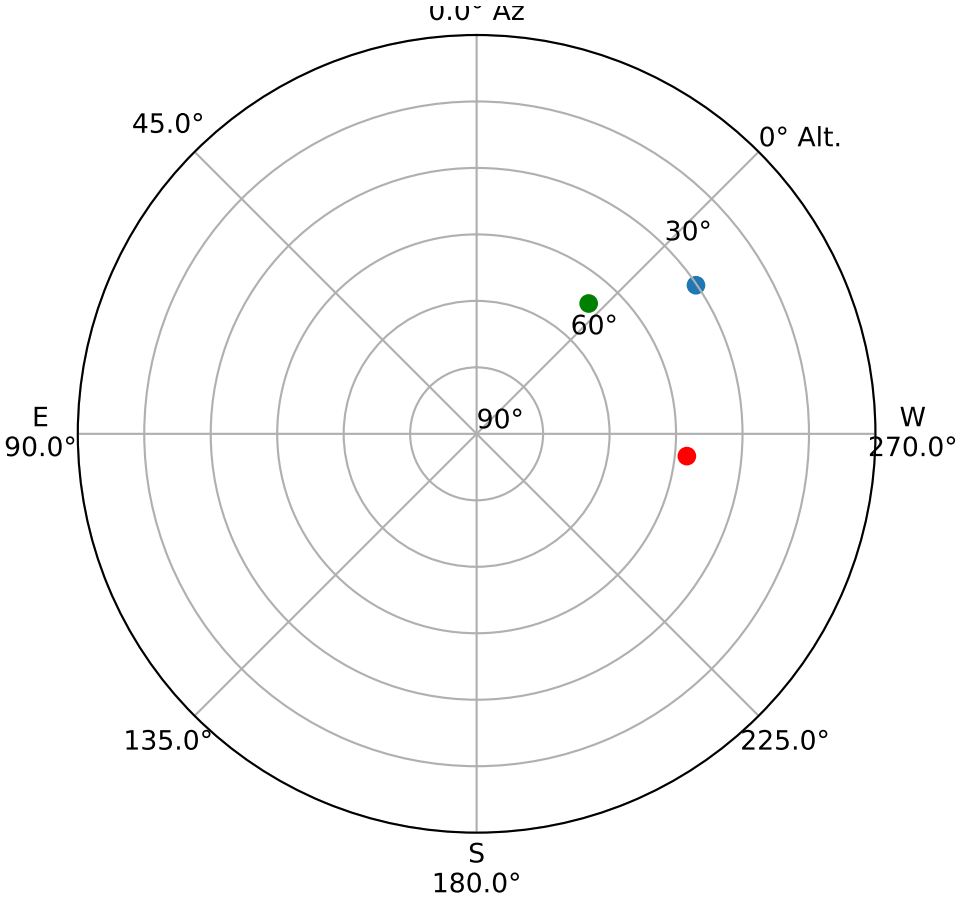
(continued from previous page)

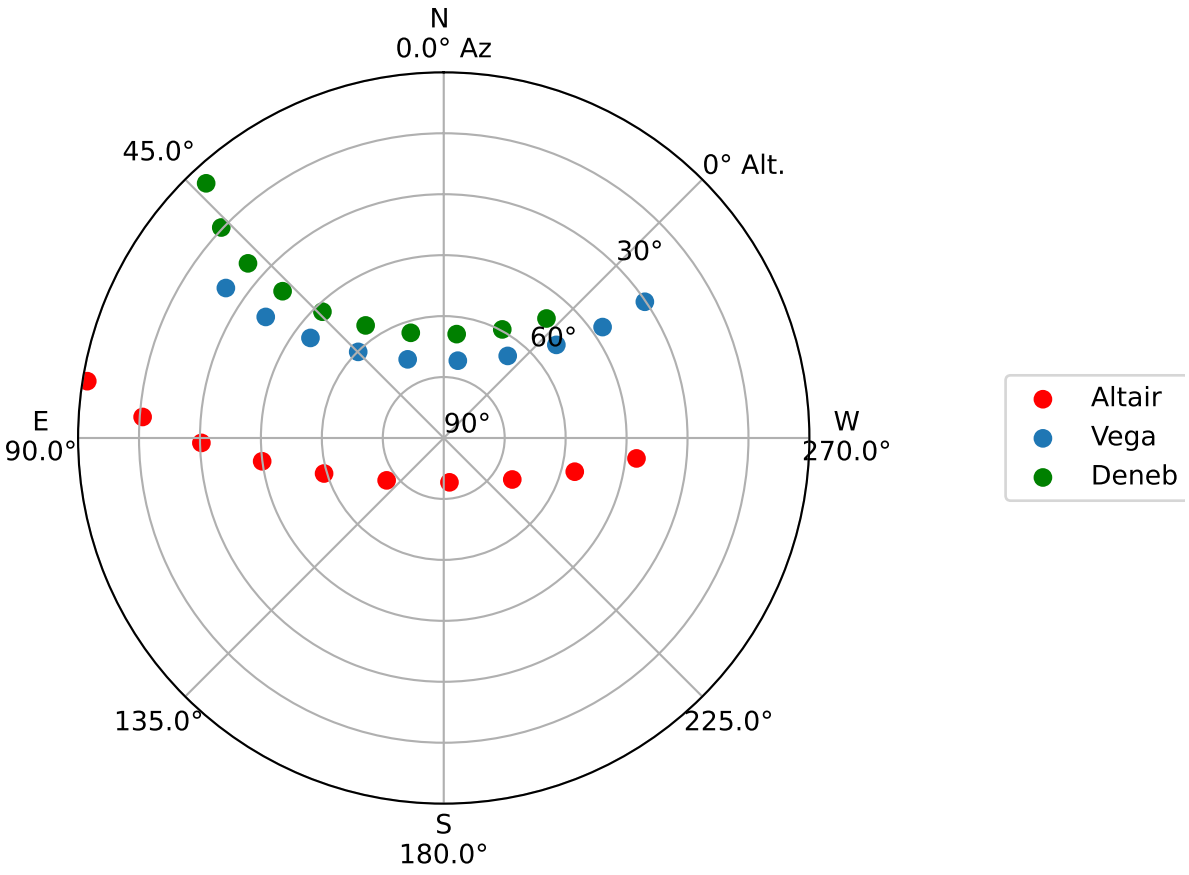
```
>>> plt.legend(loc='center left', bbox_to_anchor=(1.25, 0.5))
>>> plt.show()

>>> plot_sky(altair, subaru, end, style_kwargs=altair_style)
>>> plot_sky(vega, subaru, end)
>>> plot_sky(deneb, subaru, end, style_kwargs=deneb_style)

>>> plt.legend(loc='center left', bbox_to_anchor=(1.25, 0.5))
>>> plt.show()
```



We can also show how our targets move over time during the night in question:

```
>>> time_window = start + (end - start) * np.linspace(0, 1, 10)

>>> plot_sky(altair, subaru, time_window, style_kwargs=altair_style)
>>> plot_sky(vega, subaru, time_window)
>>> plot_sky(deneb, subaru, time_window, style_kwargs=deneb_style)

>>> plt.legend(loc='center left', bbox_to_anchor=(1.25, 0.5))
>>> plt.show()
```


0.0° Az
45.0°
0° Alt.
30°
60°
E
90.0°
90°
W
270.0°
135.0°
225.0°
S
180.0°


N
0.0° Az
45.0°
0° Alt.
30°
60°
90°
E
90.0°
W
270.0°
135.0°
225.0°
S
180.0°
Altair
Vega
Deneb

3.2 Plotting with Astroplan

astroplan currently has convenience functions for making three different types of plots: airmass vs time, parallactic angle vs time and sky charts. This plotting functionality in astroplan requires Matplotlib (although non- plotting functionality will work even without Matplotlib). The use of additional plotting packages (like Seaborn) is not explicitly prevented, but may or may not actually work.

All astroplan plots return a Axes object (which by convention is assigned to the name ax in these tutorials). You can further manipulate the returned ax object, including using it as input for more astroplan plotting functions, or you can simply display/print the plot.

3.2.1 Contents

3.2.2 Time Dependent Plots

Although all astroplan plots are time-dependent in some way, we label those that have a time-based axis as "time-dependent".

astroplan currently has a few different types of "time-dependent" plots, for example plot_airmass, plot_altitude and plot_parallactic. These take, at minimum, Observer, FixedTarget and Time objects as input.

Airmass vs time plots are made the following way:

```
>>> from astroplan.plots import plot_airmass

>>> plot_airmass(target, observer, time)
```

Parallactic angle vs time plots are made the following way:

```
>>> from astroplan.plots import plot_airmass

>>> plot_parallactic(target, observer, time)
```

Below are general guidelines for working with time-dependent plots in astroplan. Examples use airmass but apply to parallactic angle as well.

See also:

astropy.coordinates.AltAz.secz

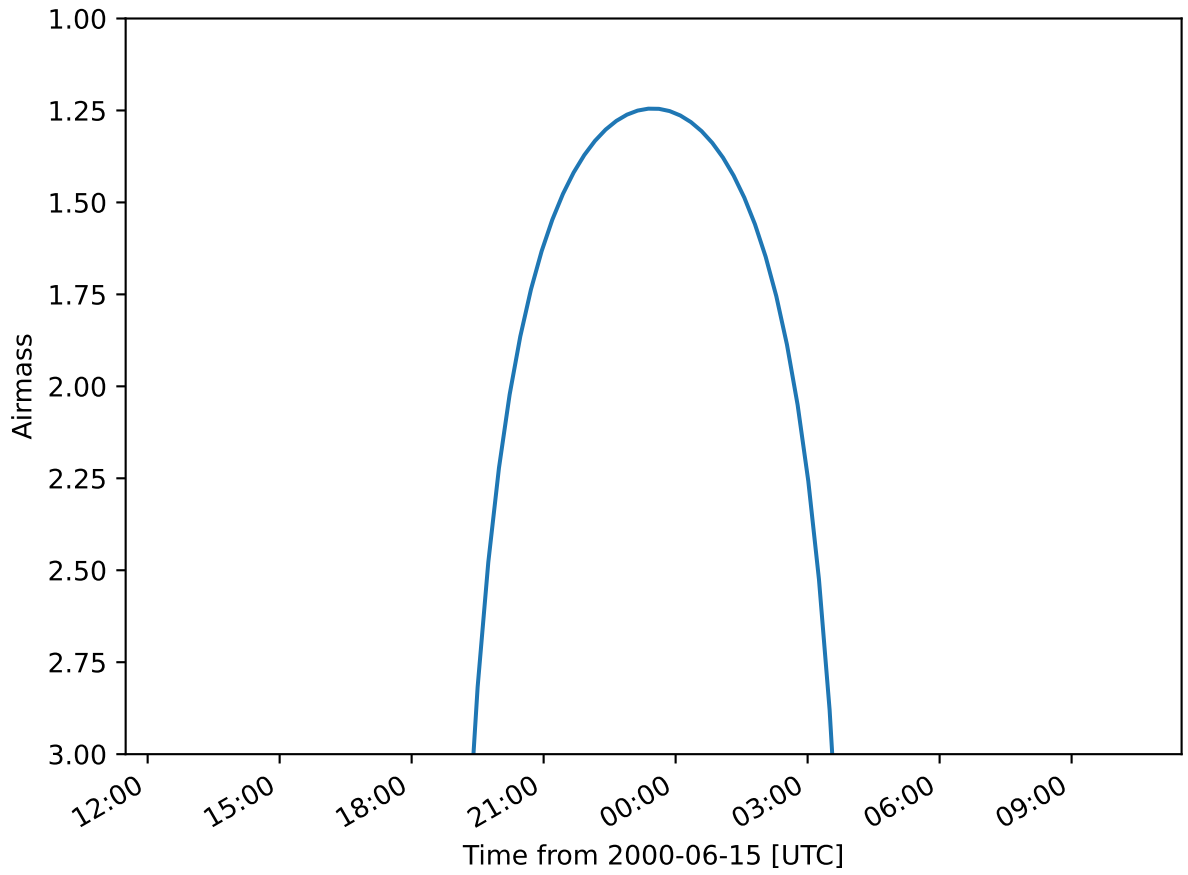
astroplan.Observer.parallactic_angle

Making a quick plot

Any plot function in `astroplan` with a time-based axis will allow you to make a quick plot over a 24-hour period.

After constructing `Observer` and `FixedTarget` objects, construct a `Time` object with a single instance in time and issue the plotting command.

```
>>> import matplotlib.pyplot as plt
>>> from astropy.time import Time
>>> from astroplan.plots import plot_airmass

>>> observe_time = Time('2000-06-15 23:30:00')

>>> plot_airmass(target, observer, observe_time)
>>> plt.show()
```

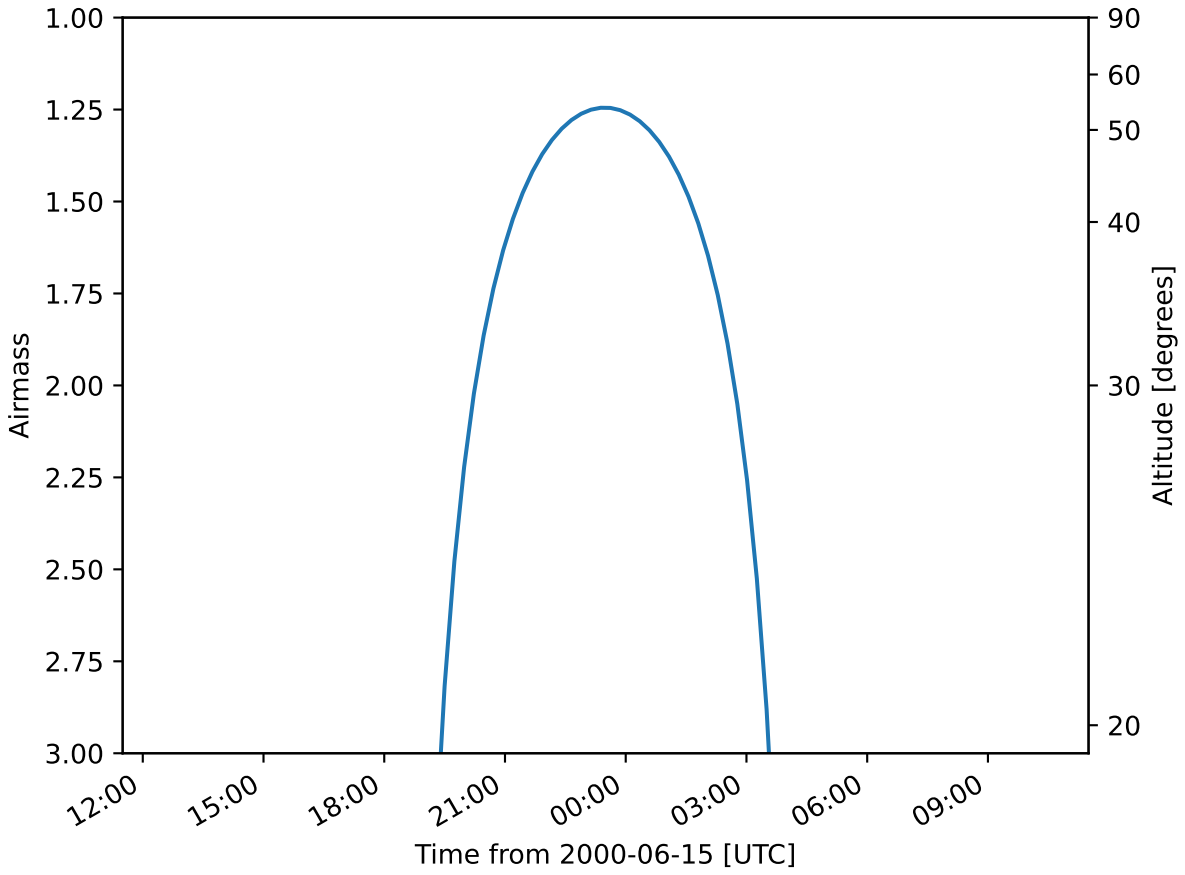


As you can see, the 24-hour plot is centered on the *time* input. You can also use array `Time` objects for these quick plots–they just can't contain more than one instance in time.

For example, these are acceptable *time* inputs:

```
Time(['2000-06-15 23:30:00'])

[Time('2000-06-15 23:30:00')]
```

You can also add a second y-axis (on the right side) which shows the corresponding altitude of the targets:



You can make altitude the primary y-axis rather than airmass by using `plot_altitude`:

Specifying a time window

If you want to see airmass plotted over a window that is not 24 hours long or you want to control the precision of the plot, you must specify every time for which you want to see an airmass plotted. Therefore, an array `Time` object is necessary.

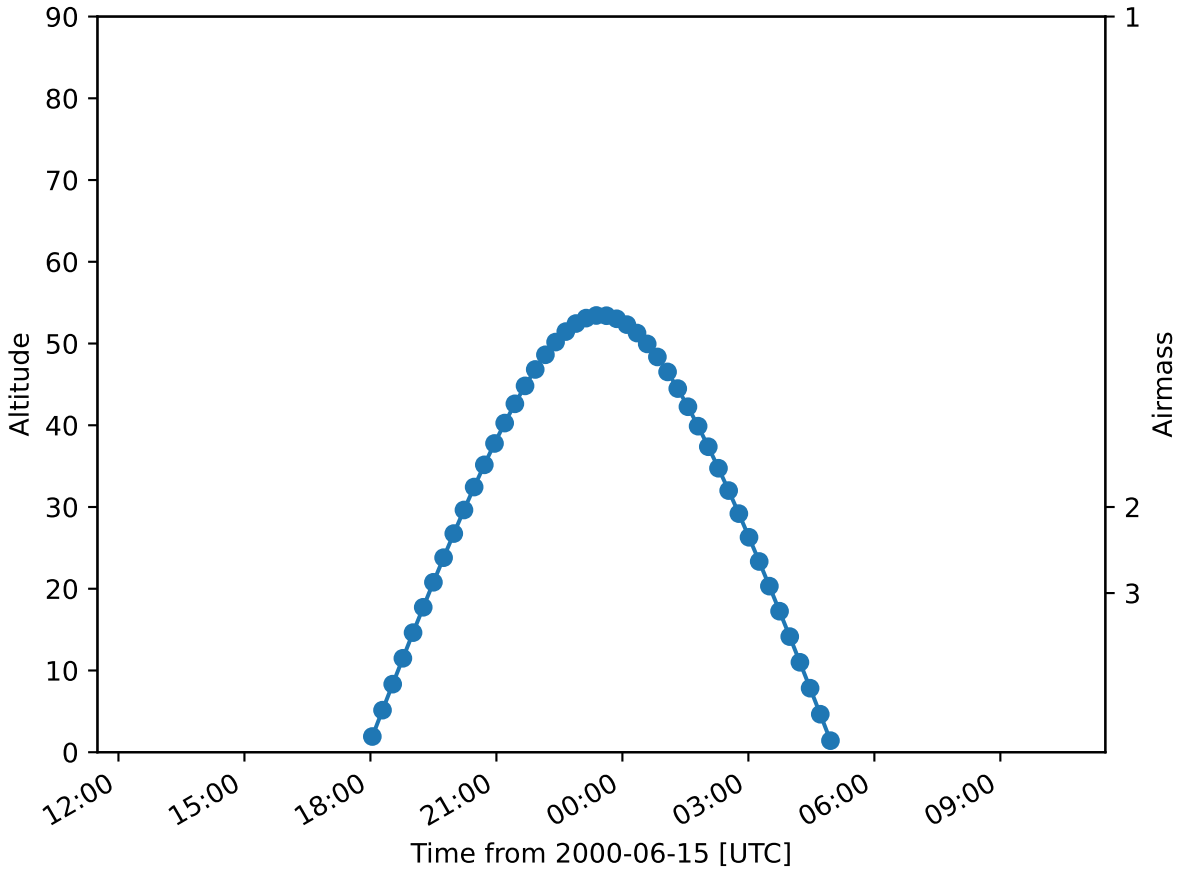
To quickly populate an `Time` object with many instances of time, use Numpy and `units`. See example below.

Centering the window at some time

To center your window at some instance in time:

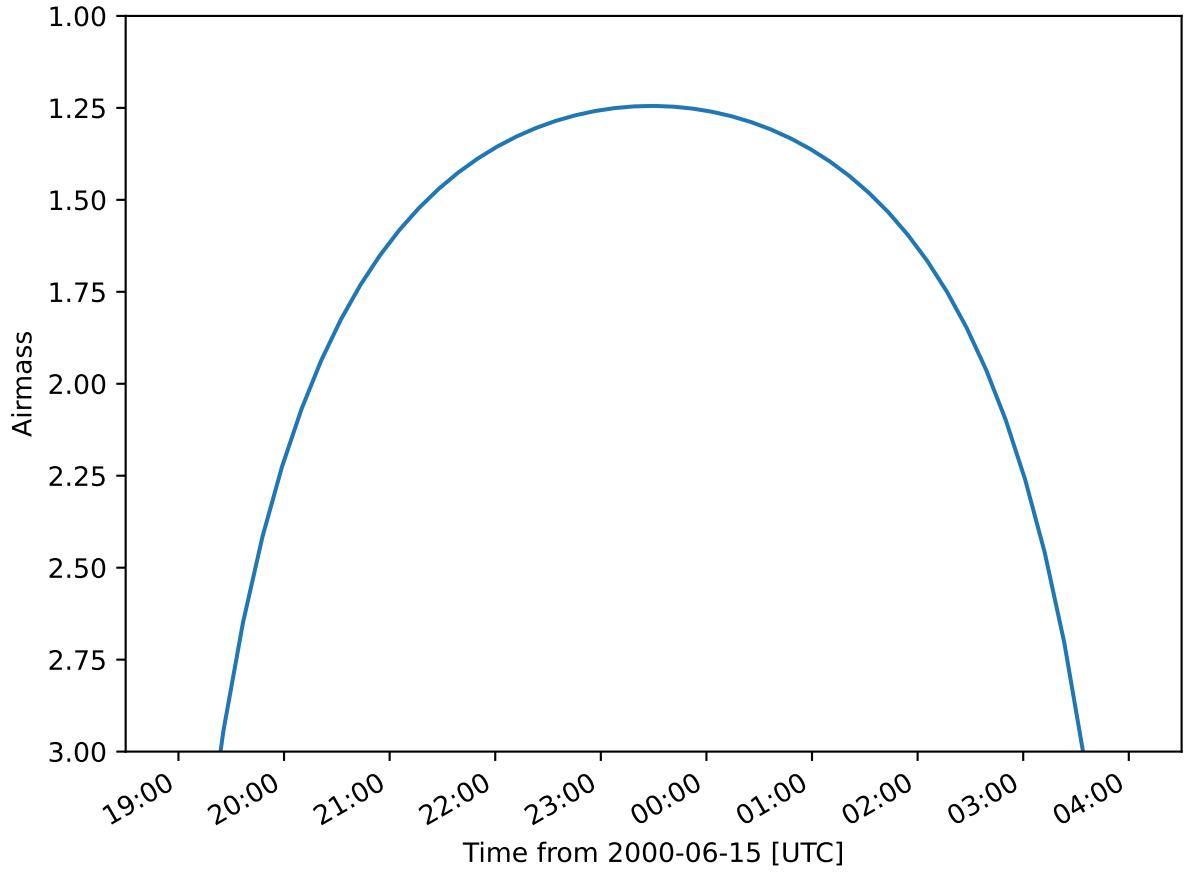
```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import astropy.units as u
>>> from astropy.time import Time
>>> from astroplan.plots import plot_airmass
```

(continues on next page)


90
80
70
60
50
40
30
20
10
0
Altitude
1
2
3
Airmass
12:00
15:00
18:00
21:00
00:00
03:00
06:00
09:00
Time from 2000-06-15 [UTC]

(continued from previous page)

```
>>> observe_time = Time('2000-06-15 23:30:00')
>>> observe_time = observe_time + np.linspace(-5, 5, 55)*u.hour

>>> plot_airmass(target, observer, observe_time)
>>> plt.show()
```



Specify start and end times

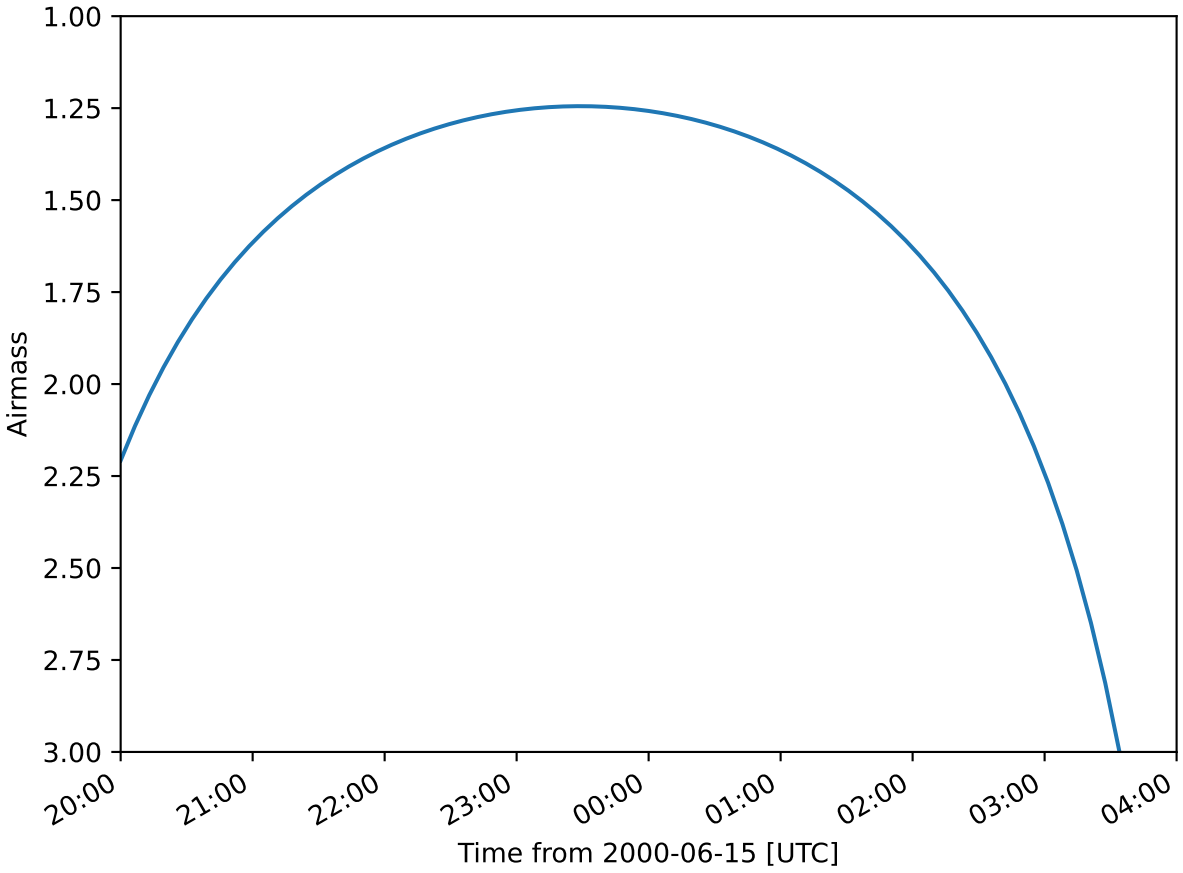
If you know the start and end times of your observation run, you can use a `TimeDelta` object to create an array for time input:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from astropy.time import Time
>>> from astroplan.plots import plot_airmass

>>> start_time = Time('2000-06-15 20:00:00')
>>> end_time = Time('2000-06-16 04:00:00')
>>> delta_t = end_time - start_time
>>> observe_time = start_time + delta_t*np.linspace(0, 1, 75)
```

(continues on next page)

(continued from previous page)

```
>>> plot_airmass(target, observer, observe_time)
>>> plt.show()
```



Plotting a quantity for multiple targets

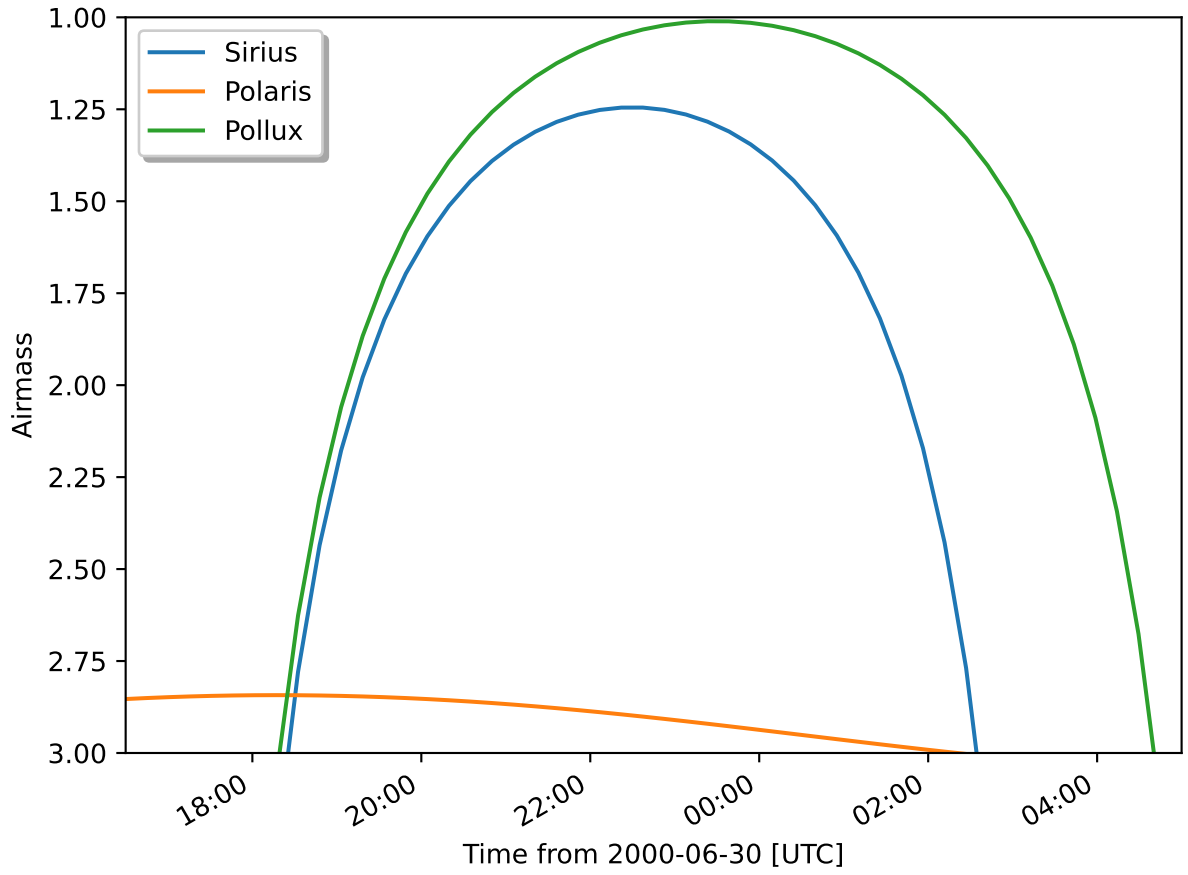
If you want to plot airmass information for multiple targets, simply reissue the `plot_airmass` command, using a different `FixedTarget` object as input this time. Repeat until you have as many targets on the plot as you wish:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from astropy.time import Time
>>> from astroplan.plots import plot_airmass

>>> observe_time = Time('2000-06-30 23:30:00') + np.linspace(-7.0, 5.5, 50)*u.hour

>>> plot_airmass(target, observer, observe_time)
>>> plot_airmass(other_target, observer, observe_time)
>>> plot_airmass(third_target, observer, observe_time)
```

(continues on next page)

(continued from previous page)

```
>>> plt.legend(shadow=True, loc=2)
>>> plt.show()
```

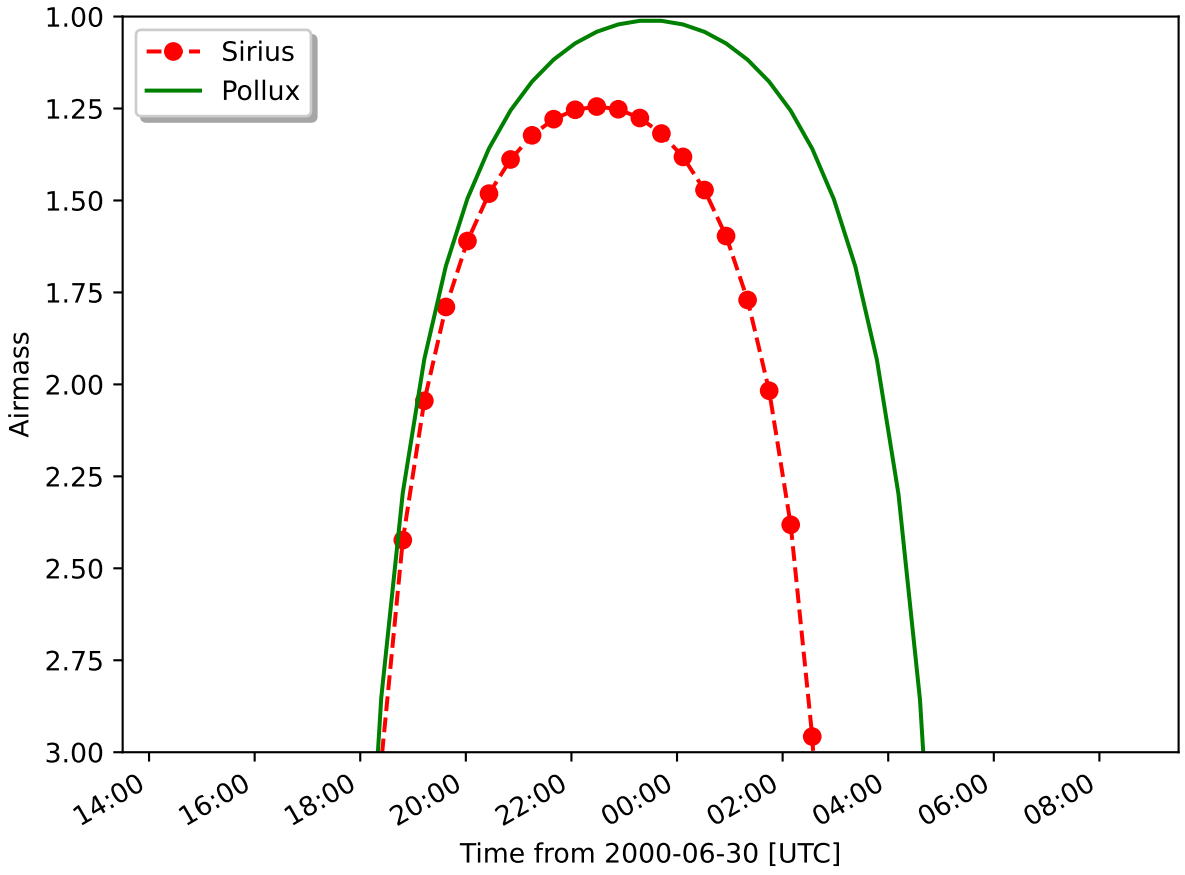


When you're ready to make a different plot, use `ax.cla()` to clear the current `Axes` object.

Changing style options

The default line for time-dependent plots is solid and the default label (should you choose to display a legend) is the name contained in the `Target` object. You can change the *linestyle*, *color*, *label* and other plotting properties by setting the *style_kwargs* option.

```
>>> import matplotlib.pyplot as plt
>>> from astroplan.plots import plot_airmass

>>> sirius_styles = {'linestyle': '--', 'color': 'r'}
>>> pollux_styles = {'color': 'g'}

>>> plot_airmass(target, observer, observe_time, style_kwargs=sirius_styles)
>>> plot_airmass(other_target, observer, observe_time, style_kwargs=pollux_styles)

>>> plt.legend(shadow=True, loc=2)
>>> plt.show()
```

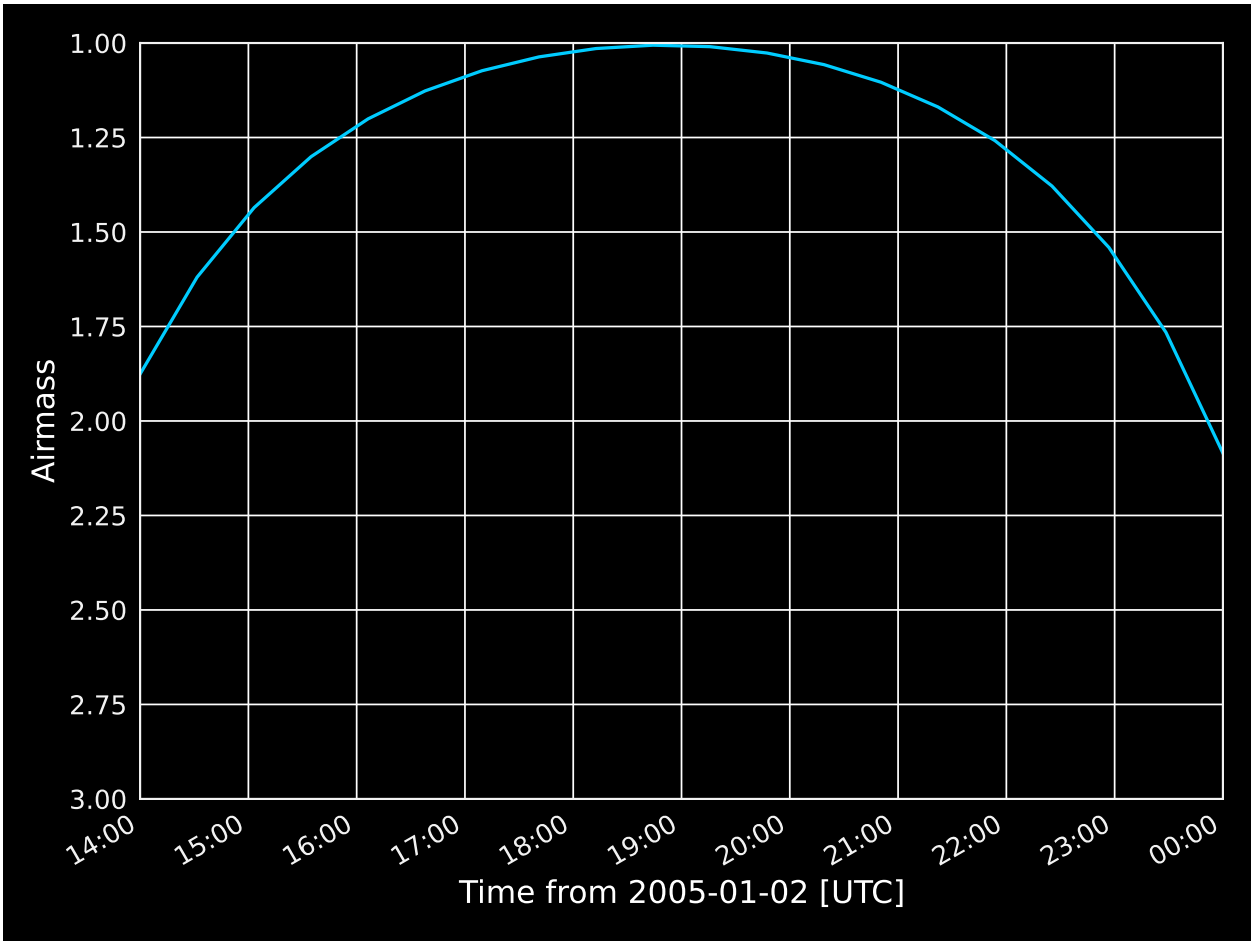

Sirius
Pollux
Airmass
1.00
1.25
1.50
1.75
2.00
2.25
2.50
2.75
3.00
14:00
16:00
18:00
20:00
22:00
00:00
02:00
04:00
06:00
08:00
Time from 2000-06-30 [UTC]

See the Matplotlib documentation for information on plotting styles in line plots.

Dark Theme Plots

By default, `astroplan` uses the Astropy style sheet for Matplotlib to generate plots with more pleasing settings than provided for by the matplotlib defaults. When using `astroplan` at night, you may prefer to make plots with dark backgrounds, rather than the default white background, to preserve your night vision. To do so, you may use the `astroplan` dark style sheet to produce dark-themed plots by using the *style_sheet* keyword argument in any plotting function:

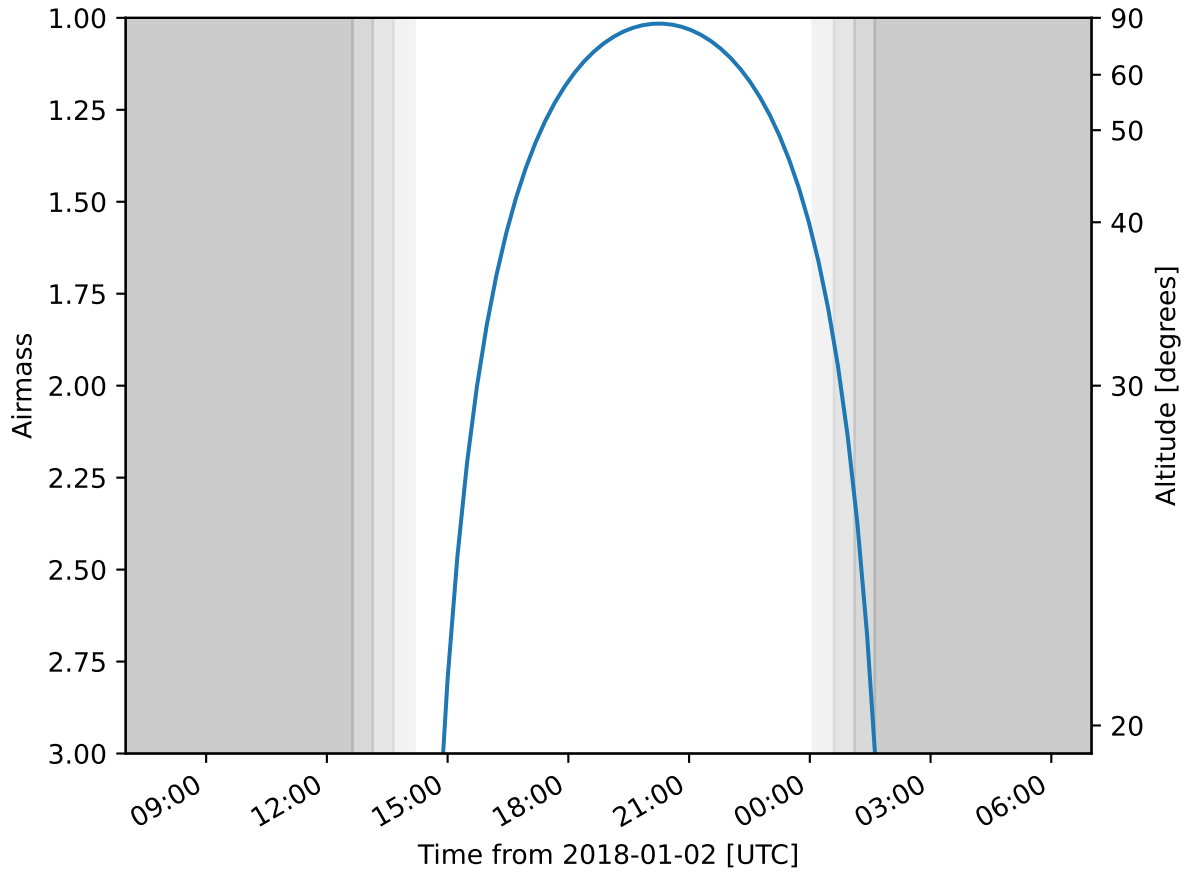
```
>>> from astroplan.plots import dark_style_sheet, plot_airmass
>>> plot_airmass(target, observatory, time, style_sheet=dark_style_sheet)
```



Additional Options

You can also shade the background according to the darkness of the sky (light shading for 0 degree twilight, darker shading for -18 degree twilight) with the `brightness_shading` keyword, and display additional y-axis ticks on the right side of the axis with the altitudes in degrees using the `altitude_yaxis` keyword:

```
>>> import matplotlib.pyplot as plt
>>> from astropy.time import Time
>>> from astroplan import FixedTarget, Observer
>>> from astroplan.plots import plot_airmass

>>> time = Time('2018-01-02 19:00')
>>> target = FixedTarget.from_name('HD 189733')
>>> apo = Observer.at_site('APO')
>>> plot_airmass(target, apo, time, brightness_shading=True, altitude_yaxis=True)
```



Return to Top

3.2.3 Sky Charts

Many users planning an observation run will want to see the positions of targets with respect to their local horizon, as well as the positions of familiar stars or other objects to act as guides.

`plot_sky` allows you to plot the positions of targets at a single instance or over some window of time. You make this plot the following way:

```
>>> from astroplan.plots import plot_sky

>>> plot_sky(target, observer, time)
```

Note: `plot_sky` currently produces polar plots in altitude/azimuth coordinates only. Plots are centered on the observer's zenith.

See also:

`astroplan.Observer.altaz`

Making a plot for one instance in time

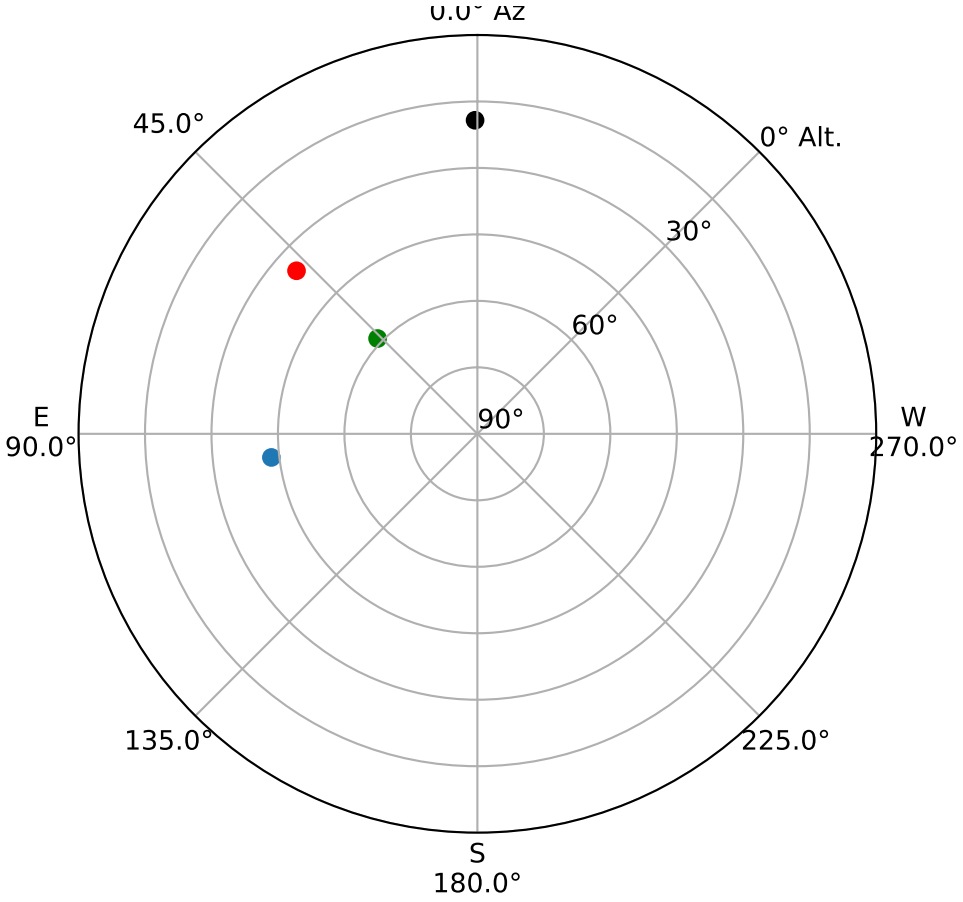
After constructing your `Observer` and `FixedTarget` objects, construct a time input using an array of length 1.

That is, either an `Time` object with an array containing one time value (e.g., `Time(['2000-1-1'])`) or an array containing one scalar `Time` object (e.g., `[Time('2000-1-1')]`).

Let's say that you created `FixedTarget` objects for Polaris, Altair, Vega and Deneb. To plot a map of the sky:

```
>>> import matplotlib.pyplot as plt
>>> from astropy.time import Time
>>> from astroplan.plots import plot_sky

>>> observe_time = Time(['2000-03-15 15:30:00'])

>>> polaris_style = {color': 'k'}
>>> vega_style = {'color': 'g'}
>>> deneb_style = {'color': 'r'}

>>> plot_sky(polaris, observer, observe_time, style_kwargs=polaris_style)
>>> plot_sky(altair, observer, observe_time)
>>> plot_sky(vega, observer, observe_time, style_kwargs=vega_style)
>>> plot_sky(deneb, observer, observe_time, style_kwargs=deneb_style)

>>> plt.legend(loc='center left', bbox_to_anchor=(1.25, 0.5))
>>> plt.show()
```

Note: Since `plot_sky` uses `scatter` (which gives the same color to different plots made on the same set of axes), you have to specify the color for each target via a style dictionary if you don't want all targets to have the same color.

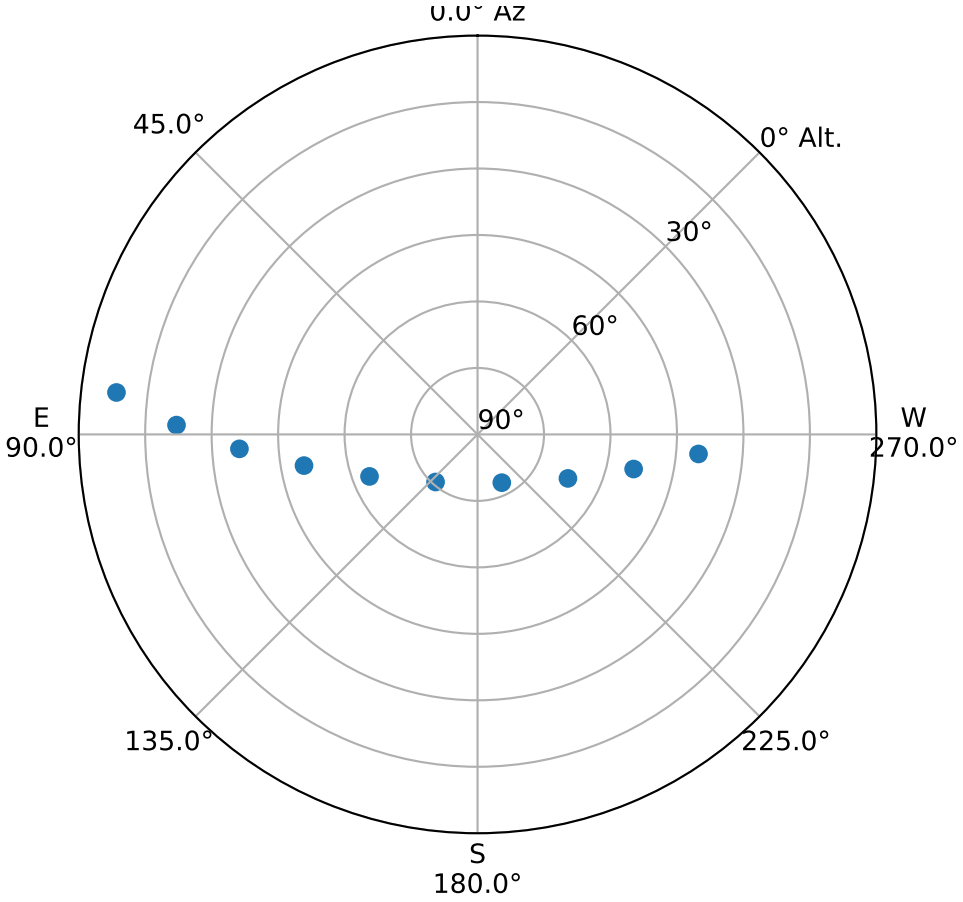

0.0° Az
45.0°
0° Alt.
30°
60°
90°
E
90.0°
W
270.0°
135.0°
225.0°
S
180.0°

Showing movement over time

If you want to see how your targets move over time, you need to explicitly specify every instance in time.

Say I want to know how Altair moves in the sky over a 9-hour period:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import astropy.units as u
>>> from astroplan.plots import plot_sky

>>> observe_time = Time('2000-03-15 17:00:00')
>>> observe_time = observe_time + np.linspace(-4, 5, 10)*u.hour

>>> plot_sky(altair, observer, observe_time)

>>> plt.legend(loc='center left', bbox_to_anchor=(1.25, 0.5))
>>> plt.show()
```



For more examples on how to populate time objects, see `Time` documentation, or *Specifying a time window*.

Note: Note that in the case of an object being under the horizon (or having negative altitude) at any of the times in your *time* input, `plot_sky` will warn you. Your object(s) will not show up on the plot for those particular times, but

any positions above the horizon will still be plotted as normal.

Customizing your sky plot

`astroplan` plots use Matplotlib defaults, so you can customize your plots in much the same way you tweak any Matplotlib plot.

Setting style options

The default marker for `plot_sky` is a circle and the default label (should you choose to display a legend) is the name contained in the `Target` object. You can change the *marker*, *color*, *label* and other plotting properties by setting the *style_kwargs* option, in the same way shown for the *time-dependent plots*.

One situation in which this is particularly useful is the plotting of guide positions, such as a few familiar stars or any body used in calibrating your telescope. You can also use this feature to set apart different types of targets (e.g., high-priority, candidates for observing run, etc.).

See the Matplotlib documentation for information on plotting styles in scatter plots.

Changing coordinate defaults

As seen in the above examples, the default position of North is at the top of the plot, and South at the bottom, with azimuth increasing counter-clockwise (CCW), putting East to the left, and West to the right.

You can't change the position of North or South (either in the actual plotting of the data, or the labels), but you can "flip" East/West by changing the direction in which azimuth increases via the *north_to_east_ccw* option:

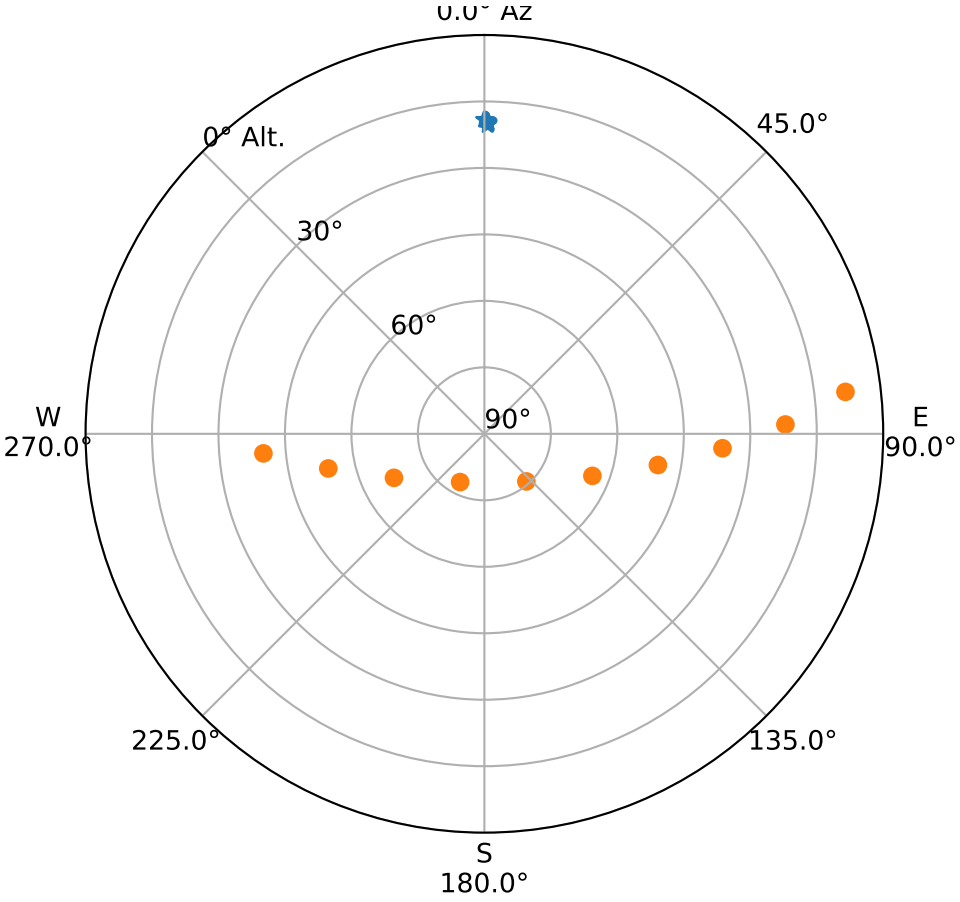
```
>>> import matplotlib.pyplot as plt
>>> from astroplan.plots import plot_sky

>>> guide_style = {'marker': '*'}

>>> plot_sky(polaris, observer, observe_time, snorth_to_east_ccw=False, style_kwargs=guide_
↪style)
>>> plot_sky(altair, observer, observe_time, north_to_east_ccw=False)

>>> plt.legend(loc='center left', bbox_to_anchor=(1.25, 0.5))
>>> plt.show()
```

Some observatories may need to offset or rotate the azimuth labels due to their particular telescope setup.

To do this, set *az_label_offset* equal to the number of degrees by which you wish to rotate the labels. By default, *az_label_offset* is set to 0 degrees. A positive offset is in the same direction as azimuth increase (see the *north_to_east_ccw* option):

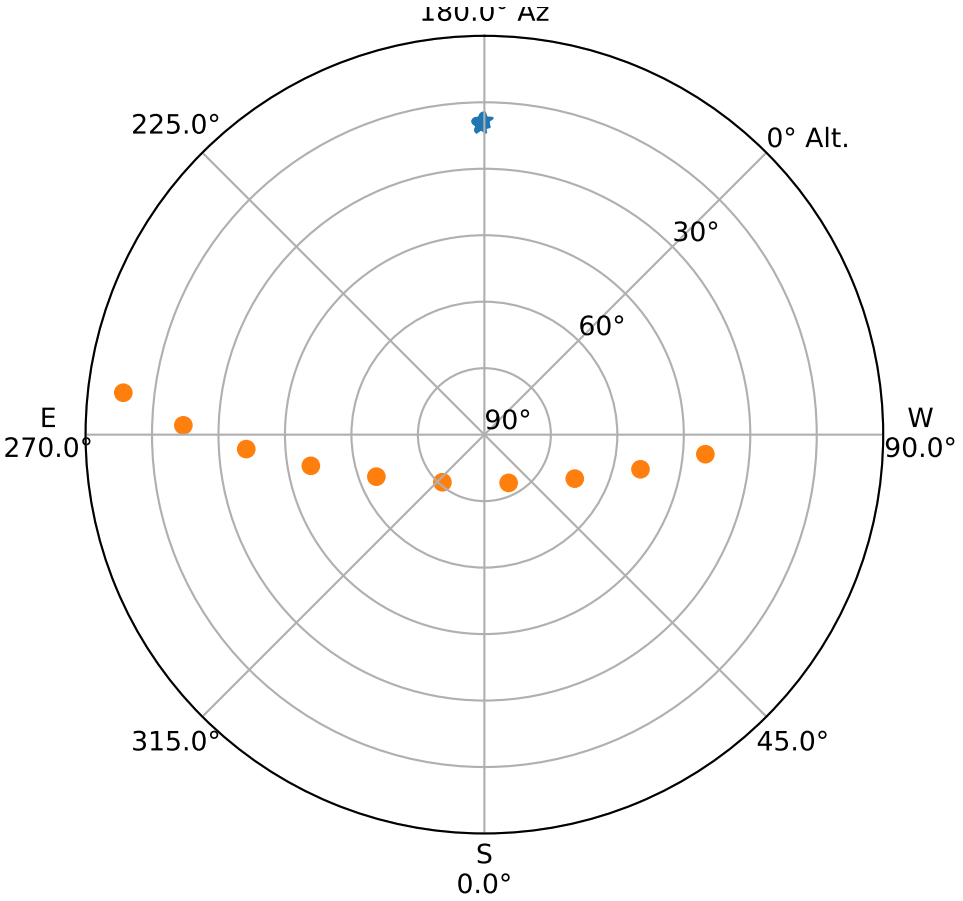
```
>>> import matplotlib.pyplot as plt
>>> from astroplan.plots import plot_sky

>>> guide_style = {'marker': '*'}

>>> plot_sky(polaris, observer, observe_time, style_kwargs=guide_style, az_label_offset=180.
↪0*u.deg)
```

(continues on next page)


0.0° Az
45.0°
0° Alt.
30°
60°
90°
W
270.0°
E
90.0°
225.0°
135.0°
S
180.0°

(continued from previous page)

```
>>> plot_sky(altair, observer, observe_time, az_label_offset=180.0*u.deg)

>>> plt.legend(loc='center left', bbox_to_anchor=(1.25, 0.5))
>>> plt.show()
```



Note: The *az_label_offset* option does not rotate the actual positions on the plot, but simply the theta grid labels (which are drawn regardless of gridline presence). Since labels are drawn with every call to `plot_sky`, we recommend you use the same *az_label_offset* argument for every target on the same plot.

It is not advised that most users change this option, as it may **appear** that your alt/az data does not coincide with the definition of altazimuth (local horizon) coordinate system.

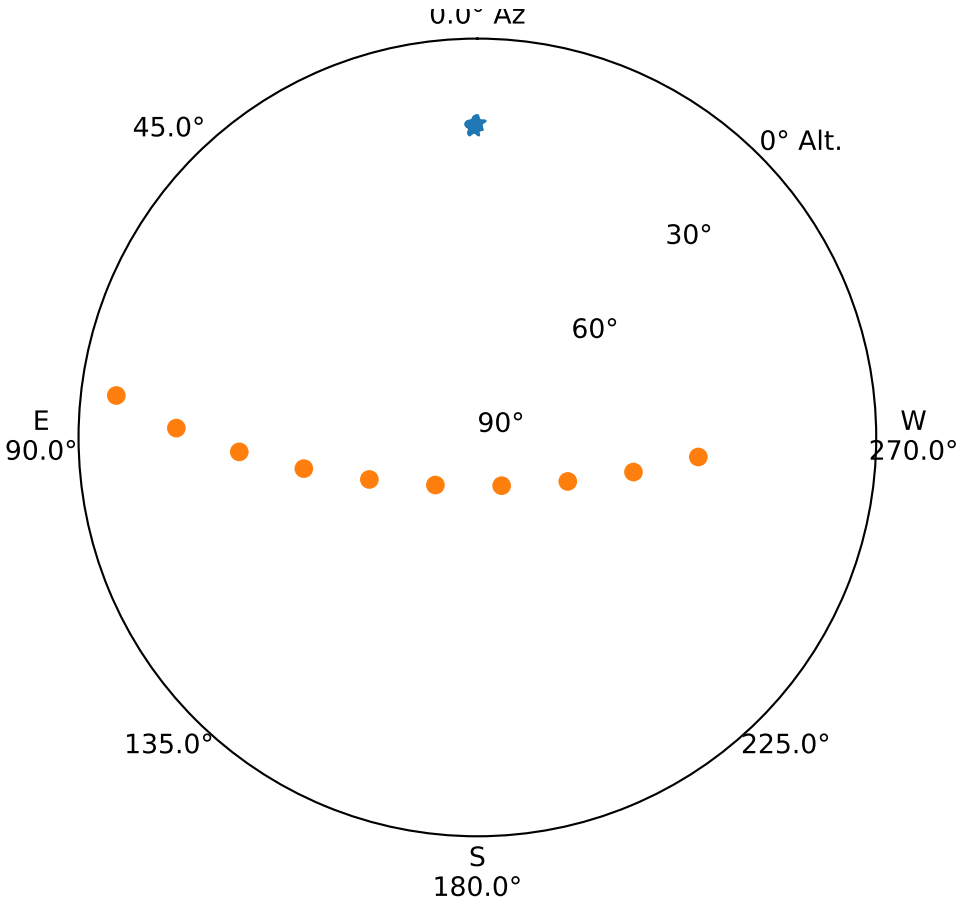
You can turn off the grid lines by setting the *grid* option to *False*:

```
>>> import matplotlib.pyplot as plt
>>> from astroplan.plots import plot_sky

>>> guide_style = {'marker': '*'}

>>> plot_sky(polaris, observer, observe_time, style_kwargs=guide_style, grid=False)
>>> plot_sky(altair, observer, observe_time, grid=False)
```

(continues on next page)

(continued from previous page)

```
>>> plt.legend(loc='center left', bbox_to_anchor=(1.25, 0.5))
>>> plt.show()
```



Note: Since grids are redrawn with every call to `plot_sky`, you must set `grid=False` for every target in the same plot.

Other tweaks

You can easily change other plot attributes by acting on the returned `matplotlib.axes.Axes` object or via `matplotlib.pyplot` calls (e.g., `plt.figure`, `plt.rc`, etc.).

For instance, you can increase the size of your plot and its font:

```
>>> # Set the figure size/font before you issue the plotting command.
>>> plt.figure(figsize=(8,6))
>>> plt.rc('font', size=14)

>>> plot_sky(polaris, observer, observe_time)
```

(continues on next page)

(continued from previous page)

```
>>> plot_sky(altair, observer, observe_time)

>>> plt.legend(loc='center left', bbox_to_anchor=(1.25, 0.5))
>>> plt.show()

>>> # Change font size back to default once done plotting.
>>> plt.rc('font', size=12)
```

Return to Top

Miscellaneous

The easiest way to reuse the `Axes` object that is the base of your plots is to just let `astroplan`'s plotting functions take care of it in the background. You do, however, have the option of explicitly passing in a named axis, assuming that you have created the appropriate type for the particular plot you want.

We can explicitly give a name to the `Axes` object returned by `plot_sky` when plotting Polaris and reuse it to plot Altair:

```
>>> my_ax = plot_sky(polaris, observer, observe_time, style_kwargs=guide_style)
>>> plot_sky(altair, observer, observe_time, my_ax)
```

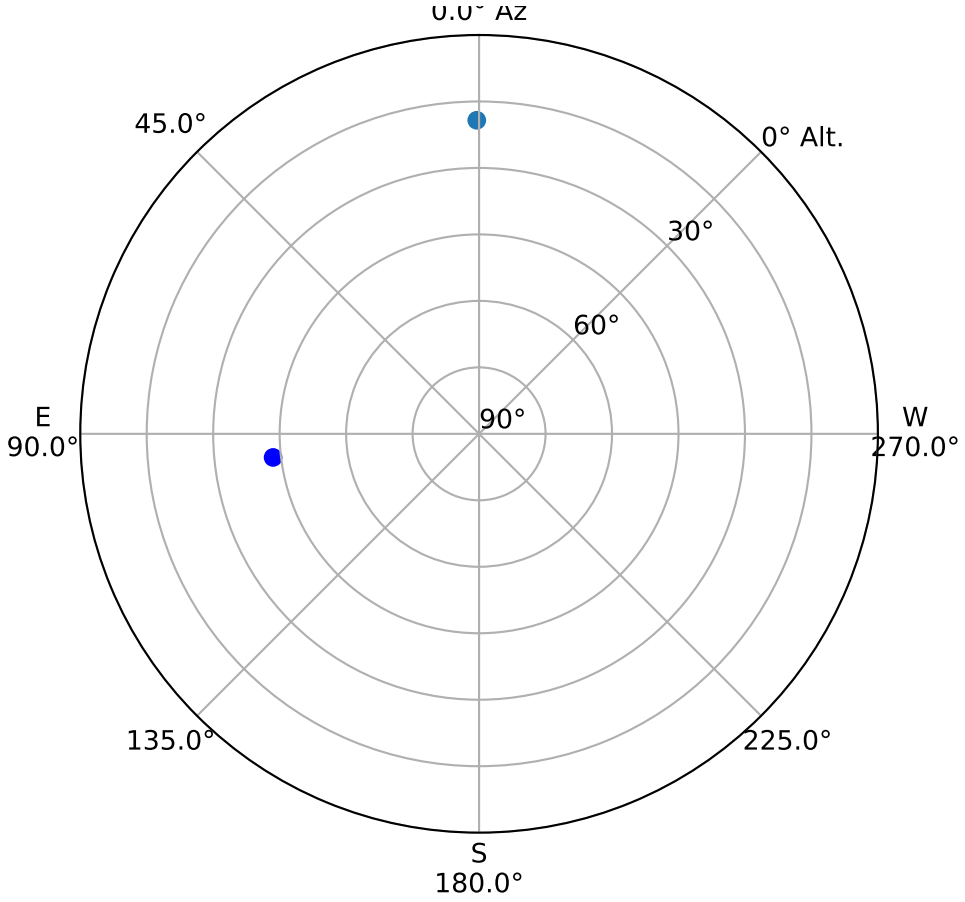
We can also create a `Axes` object entirely outside of `plot_sky`, then pass it in:

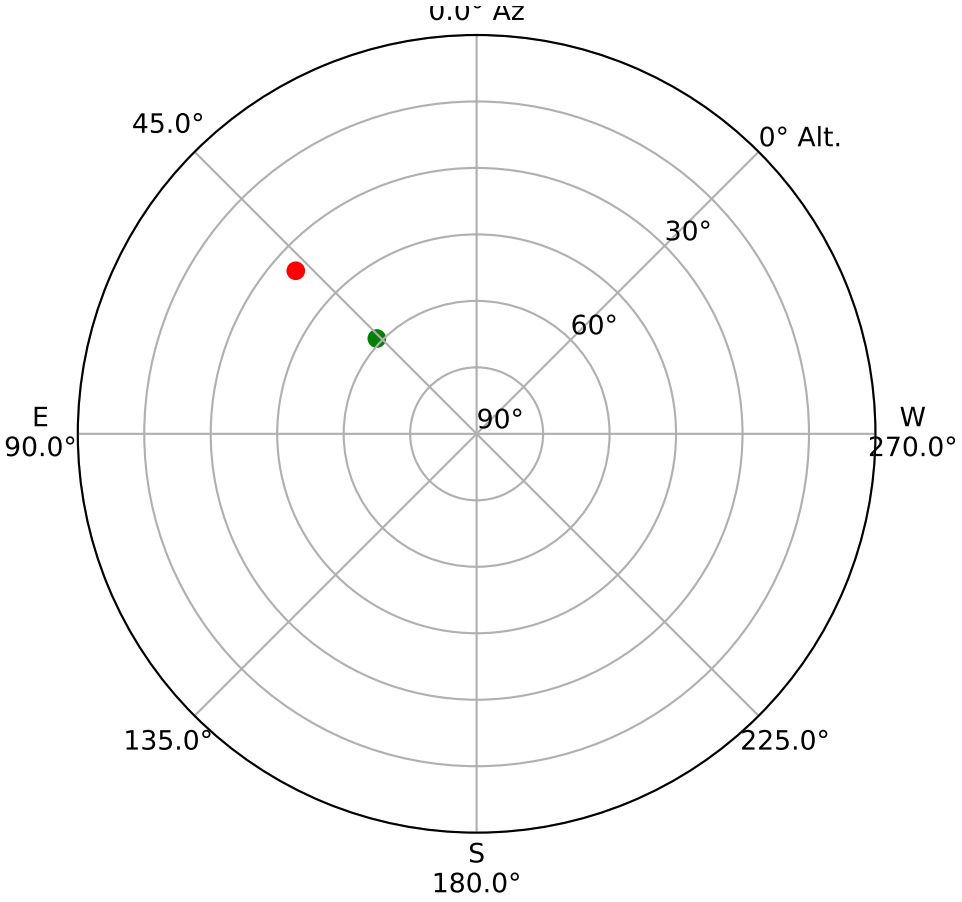
```
>>> my_ax = plt.gcf().add_subplot(projection='polar')
>>> plot_sky(polaris, observer, observe_time, my_ax)
```

Passing in named `matplotlib.axes.Axes` objects comes in handy when you want to make multiple plots:

```
>>> from astroplan.plots import plot_sky
>>> import matplotlib.pyplot as plt

>>> my_ax = plot_sky(polaris, observer, observe_time, style_kwargs=polaris_style)
>>> plot_sky(altair, observer, observe_time, my_ax, style_kwargs=altair_style)
>>> plt.legend(loc='center left', bbox_to_anchor=(1.3, 0.5))

>>> # Note that this plt.show (or another action, such as saving a figure) is critical in␣
↪maintaining two separate plots.
>>> plt.show()

>>> other_ax = plot_sky(vega, observer, observe_time, style_kwargs=vega_style)
>>> plot_sky(deneb, observer, observe_time, other_ax, style_kwargs=deneb_style)
>>> plt.legend(loc='center left', bbox_to_anchor=(1.25, 0.5))
>>> plt.show()
```

Return to Top

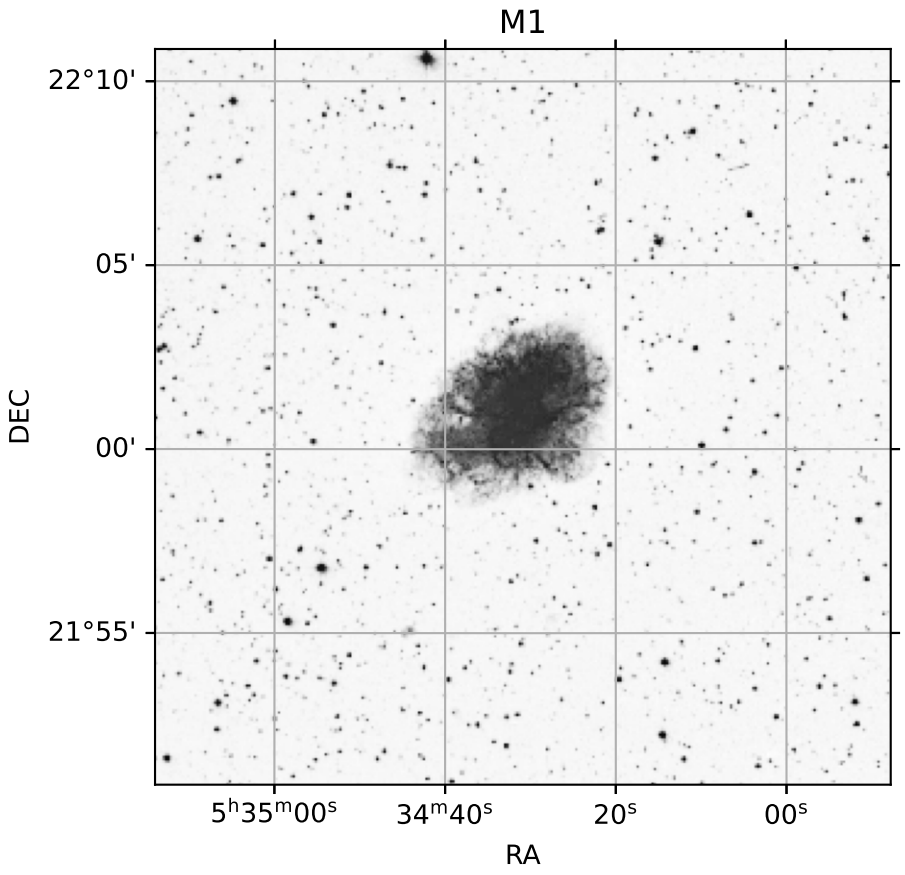

0.0° Az
45.0°
0° Alt.
30°
60°
90°
E
90.0°
W
270.0°
135.0°
225.0°
S
180.0°


0.0° Az
45.0°
0° Alt.
30°
60°
90°
E
90.0°
W
270.0°
135.0°
225.0°
S
180.0°

3.2.4 Finder Chart/Image

`astroplan` includes a function for generating quick finder images from Python, `plot_finder_image`, by querying for images from sky surveys centered on a `FixedTarget`. This function depends on astroquery (in addition to Matplotlib). In this example, we'll quickly make a finder image centered on The Crab Nebula (M1):

```
>>> from astroplan.plots import plot_finder_image
>>> from astroplan import FixedTarget
>>> import matplotlib.pyplot as plt

>>> messier1 = FixedTarget.from_name("M1")
>>> ax, hdu = plot_finder_image(messier1)
>>> plt.show()
```



Return to Top

3.3 Observing Transiting Exoplanets and Eclipsing Binaries

Note: The `periodic` module is new and under development. The API may change in upcoming versions of astroplan, and pull requests are welcome!

Warning: There are currently two major caveats in the implementation of `EclipsingSystem`. The secondary eclipse time approximation is only accurate when the orbital eccentricity is small, and the eclipse times are computed without any barycentric corrections. The current implementation should only be used for approximate mid-eclipse times for low eccentricity orbits, with event durations longer than the barycentric correction error (<=16 minutes).

3.3.1 Contents

3.3.2 Transit/Primary and secondary eclipse times

We can define the properties of an eclipsing system, such as an eclipsing binary or transiting exoplanet, using the `EclipsingSystem` object. Let's make an instance for the transiting exoplanet HD 209458 b, which has a period of 3.52474859 days, mid-transit time of JD=2452826.628514, and transit duration of 0.1277:

```
>>> from astropy.time import Time
>>> import astropy.units as u
>>> from astroplan import EclipsingSystem

>>> primary_eclipse_time = Time(2452826.628514, format='jd')
>>> orbital_period = 3.52474859 * u.day
>>> eclipse_duration = 0.1277 * u.day

>>> hd209458 = EclipsingSystem(primary_eclipse_time=primary_eclipse_time,
...                            orbital_period=orbital_period, duration=eclipse_duration,
...                            name='HD 209458 b')
```

Let's say we're observing on 2016 January 1, 00:00 UTC. We can compute the next transit and secondary eclipse using the `next_primary_eclipse_time` and `next_secondary_eclipse_time` methods, respectively:

```
>>> observing_time = Time('2016-01-01 00:00')
>>> hd209458.next_primary_eclipse_time(observing_time)
<Time object: scale='utc' format='iso' value=['2016-01-03 16:16:09.848']>

>>> hd209458.next_secondary_eclipse_time(observing_time)
<Time object: scale='utc' format='iso' value=['2016-01-01 21:58:20.708']>
```

You can compute the next ten mid-transit times with the `n_eclipses` keyword:

```
>>> hd209458.next_primary_eclipse_time(observing_time, n_eclipses=5)
<Time object: scale='utc' format='iso' value=['2016-01-03 16:16:09.848' '2016-01-07 04:51:48.
↪126'
                                              '2016-01-10 17:27:26.404' '2016-01-14 06:03:04.
↪682'
                                              '2016-01-17 18:38:42.960']>
```

It's often useful to know the ingress and egress times of the next transits when planning observations, which you can find with `next_primary_ingress_egress_time`:

```
>>> hd209458.next_primary_ingress_egress_time(observing_time, n_eclipses=3)
<Time object: scale='utc' format='jd' value=[[ 2457391.11404175  2457391.24174175]
                                             [ 2457394.63879034  2457394.76649034]
                                             [ 2457398.16353893  2457398.29123893]]>
```

And remember - in the current implementation, all eclipse times are computed without any barycentric corrections, and the secondary eclipse time approximation is only accurate when the orbital eccentricity is small.

3.3.3 Transit times via astroquery

The development version of astroquery allows users to query for properties of known exoplanets with three different services: `nasa_exoplanet_archive`, `exoplanet_orbit_database`, and `open_exoplanet_catalogue`. In the example below, we will query for the properties of the transiting exoplanet TRAPPIST-1 b with astroquery, and calculate the times of the next three transits with `EclipsingSystem`.

```
>>> # NASA Exoplanet Archive for planet properties
>>> import astropy.units as u
>>> from astropy.time import Time
>>> from astroquery.ipac.nexsci.nasa_exoplanet_archive import NasaExoplanetArchive
>>> planet_properties = NasaExoplanetArchive.query_object('TRAPPIST-1 b', select='*', table=
↪'pscomppars')


>>> # get relevant planet properties
>>> epoch = Time(planet_properties['pl_tranmid'], format='jd')
>>> period = planet_properties['pl_orbper']
>>> transit_duration = planet_properties['pl_trandur']

>>> # Create an EclipsingSystem object for HD 209458
>>> from astroplan import EclipsingSystem
>>> trappist1b = EclipsingSystem(primary_eclipse_time=epoch, orbital_period=period,
...                              duration=transit_duration)

>>> # Calculate next three mid-transit times which occur after ``obs_time``
>>> obs_time = Time('2017-01-01 12:00')
>>> trappist1b.next_primary_eclipse_time(obs_time, n_eclipses=3)
<Time object: scale='utc' format='iso' value=['2017-01-02 15:17:40.205' '2017-01-04 03:33:19.
↪443'
 '2017-01-05 15:48:58.681']>
```

3.3.4 When is the next observable transit?

Let's continue with the example from above, and now let's calculate all mid-transit times of HD 209458 b which are observable from Apache Point Observatory, when the target is above 30 degrees altitude, and in the "A" half of the night (roughly between sunset and midnight). First we need to create a `FixedTarget` object for the star, which contains the sky coordinate, and the `EclipsingSystem` object, which defines the transit time, period and duration:

```
>>> from astroplan import FixedTarget, Observer, EclipsingSystem
>>> apo = Observer.at_site('APO', timezone='US/Mountain')
>>> target = FixedTarget.from_name("HD 209458")

>>> primary_eclipse_time = Time(2452826.628514, format='jd')
>>> orbital_period = 3.52474859 * u.day
>>> eclipse_duration = 0.1277 * u.day

>>> hd209458 = EclipsingSystem(primary_eclipse_time=primary_eclipse_time,
...                            orbital_period=orbital_period, duration=eclipse_duration,
...                            name='HD 209458 b')
```

Then we compute a list of mid-transit times over the next year:

```
>>> n_transits = 100  # This is the roughly number of transits per year
>>> obs_time = Time('2017-01-01 12:00')
>>> midtransit_times = hd209458.next_primary_eclipse_time(obs_time, n_eclipses=n_transits)
```

Finally, we can check if the target is observable at each transit time, given our constraints on the altitude of the target (`AltitudeConstraint`) and the time of observations (`LocalTimeConstraint` and `AtNightConstraint`) with the function`~astroplan.is_event_observable`:

```
>>> from astroplan import (PrimaryEclipseConstraint, is_event_observable
...                        AtNightConstraint, AltitudeConstraint, LocalTimeConstraint)
>>> import datetime as dt
>>> import astropy.units as u
>>> min_local_time = dt.time(19, 0)  # 19:00 local time at APO (7pm)
>>> max_local_time = dt.time(0, 0)  # 00:00 local time at APO (midnight)
>>> constraints = [AtNightConstraint.twilight_civil(),
...                AltitudeConstraint(min=30*u.deg),
...                LocalTimeConstraint(min=min_local_time, max=max_local_time)]

>>> is_event_observable(constraints, apo, target, times=midtransit_times)
array([[ True, False,  True, ...,  True, False,  True, False]], dtype=bool)
```

In the above example, we only checked that the star is observable at the mid-transit time. If you were planning to do transit photometry of HD 209458 b, you might want to be sure that the entire transit is observable. Let's look for only completely observable transits:

```
>>> ing_egr = hd209458.next_primary_ingress_egress_time(observing_time, n_eclipses=n_
↪transits)
>>> is_event_observable(constraints, apo, target, times_ingress_egress=ing_egr)
array([[False, False, False, ...,  True, False, False, False]], dtype=bool)
```

Note that several of the transits that were observable at their mid-transit time are not observable at both the ingress and egress times, and therefore are not observable in the computation above.

3.3.5 Orbital Phase Constraint

It is often useful to plan observations as a function of orbital phase. You can calculate the orbital phase of an eclipsing or non-eclipsing system with the `PeriodicEvent` object, which you specify with an epoch and period. Let's create a `PeriodicEvent` object for an imagined binary star:

```
>>> from astroplan import PeriodicEvent
>>> import astropy.units as u
>>> from astropy.time import Time

>>> epoch = Time(2456001, format='jd')  # reference time of periodic event
>>> period = 3.25 * u.day  # period of periodic event
>>> duration = 2 * u.hour  # duration of event

>>> binary_system = PeriodicEvent(epoch=epoch, period=period)
```

Now let's determine when we can observe the binary given some observing constraints. We want to measure the binary's radial velocity at orbital phases between 0.4 and 0.6, while observing between astronomical twilights, and while the target is above 40 degrees altitude, for an observer in Greenwich, England on the night of January 1, 2017. For this task we can use the `PhaseConstraint` (learn more about the constraints module in *Defining Observing Constraints*):

```
>>> from astropy.coordinates import SkyCoord
>>> from astroplan import FixedTarget, Observer, is_observable
>>> target = FixedTarget(SkyCoord(ra=42*u.deg, dec=42*u.deg), name='Target')
>>> greenwich = Observer.at_site("Greenwich")
>>> start_time = Time('2017-01-01 01:00')
>>> end_time = Time('2017-01-01 06:00')

>>> from astroplan import PhaseConstraint, AtNightConstraint, AltitudeConstraint
>>> constraints = [PhaseConstraint(binary_system, min=0.4, max=0.6),
...                AtNightConstraint.twilight_astronomical(),
...                AltitudeConstraint(min=40 * u.deg)]
>>> is_observable(constraints, greenwich, target, time_range=[start_time, end_time])
array([ True], dtype=bool)
```

3.4 Defining Observing Constraints

3.4.1 Contents

3.4.2 Introduction to Built-In Constraints

Frequently, we have a long list of targets that we want to observe, and we need to know which ones are observable given a set of constraints imposed on our observations by a wide range of limitations. For example, your telescope may only point over a limited range of altitudes, your targets are only useful in a range of airmasses, and they must be separated from the moon by some large angle. The `constraints` module is here to help!

Say we're planning to observe from Subaru Observatory in Hawaii on August 1, 2015 from 06:00-12:00 UTC. First, let's set up an `Observer` object:

```
from astroplan import Observer, FixedTarget
from astropy.time import Time
subaru = Observer.at_site("Subaru")
time_range = Time(["2015-08-01 06:00", "2015-08-01 12:00"])
```

We're keeping a list of targets in a text file called `targets.txt`, which looks like this:

```
# name ra_degrees dec_degrees
Polaris 37.95456067 89.26410897
Vega 279.234734787 38.783688956
Albireo 292.68033548 27.959680072
Algol 47.042218553 40.955646675
Rigel 78.634467067 -8.201638365
Regulus 152.092962438 11.967208776
```

We'll read in this list of targets using `astropy.table`, and create a list of `FixedTarget` objects out of them:

```
# Read in the table of targets
from astropy.table import Table
target_table = Table.read('targets.txt', format='ascii')

# Create astroplan.FixedTarget objects for each one in the table
from astropy.coordinates import SkyCoord
import astropy.units as u
targets = [FixedTarget(coord=SkyCoord(ra=ra*u.deg, dec=dec*u.deg), name=name)
           for name, ra, dec in target_table]
```

We will build a bulleted list of our constraints first, then implement them in code below.

- Our observations with Subaru can only occur between altitudes of ~10-80 degrees, which we can define using the `AltitudeConstraint` class.
- We place an upper limit on the airmass of each target during observations using the `AirmassConstraint` class.
- Since we're optical observers, we only want to observe targets at night, so we'll also call the `AtNightConstraint` class. We're not terribly worried about sky brightness for these bright stars, so we'll define "night" times as those between civil twilights by using the class method `twilight_civil`:

```
from astroplan import (AltitudeConstraint, AirmassConstraint,
                       AtNightConstraint)
constraints = [AltitudeConstraint(10*u.deg, 80*u.deg),
               AirmassConstraint(5), AtNightConstraint.twilight_civil()]
```

This list of constraints can now be applied to our target list to determine:

- whether the targets are observable given the constraints at *any* times in the time range, using `is_observable`,

- whether the targets are observable given the constraints at *all* times in the time range, using `is_always_observable`
- during what months the targets are *ever* observable given the constraints, using `months_observable`:

```
from astroplan import is_observable, is_always_observable, months_observable
# Are targets *ever* observable in the time range?
ever_observable = is_observable(constraints, subaru, targets, time_range=time_range)

# Are targets *always* observable in the time range?
always_observable = is_always_observable(constraints, subaru, targets, time_range=time_
↪range)

# During what months are the targets ever observable?
best_months = months_observable(constraints, subaru, targets, time_range)
```

The `is_observable` and `is_always_observable` functions will return boolean arrays which tell you whether or not each target is observable given your constraints. Let's print these results in tabular form:

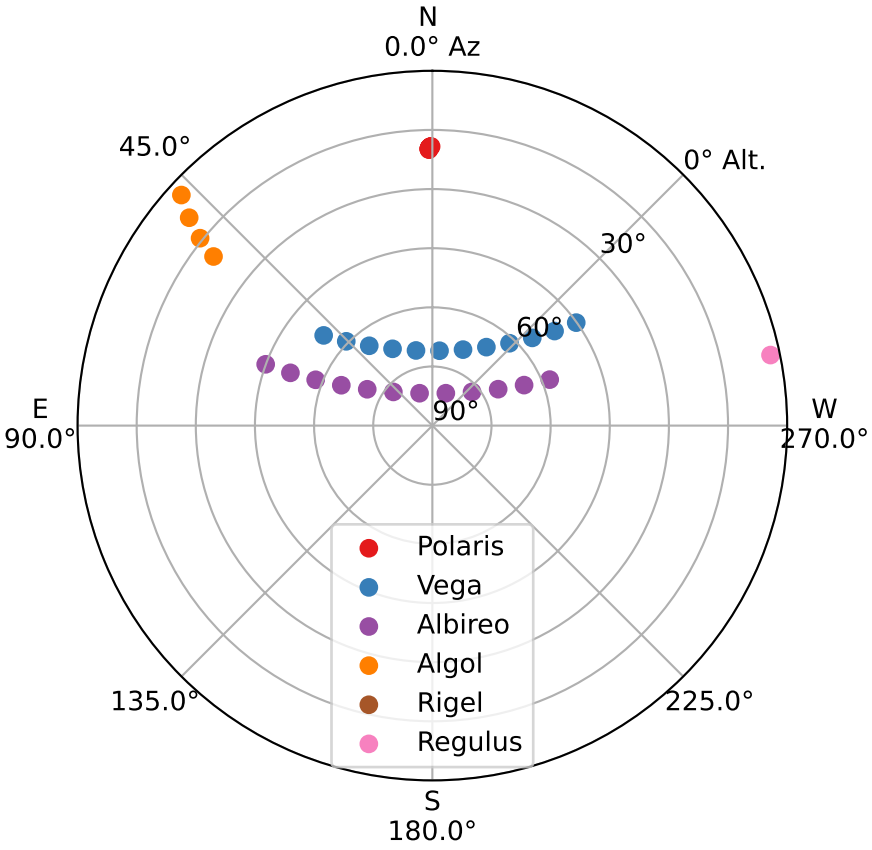
```
>>> from astropy.table import Table
>>> import numpy as np
>>> observability_table = Table()
>>> observability_table['targets'] = [target.name for target in targets]
>>> observability_table['ever_observable'] = ever_observable
>>> observability_table['always_observable'] = always_observable
>>> print(observability_table)
<Table length=6>
targets ever_observable always_observable
  str7        bool             bool
------- --------------- -----------------
Polaris            True              True
   Vega            True              True
Albireo            True             False
  Algol            True             False
  Rigel           False             False
Regulus           False             False
```

Now we can see which targets are observable! You can also use the `observability_table` method to do the same calculations and store the results in a table, all in one step:

```
>>> from astroplan import observability_table
>>> table = observability_table(constraints, subaru, targets, time_range=time_range)
>>> print(table)
target name ever observable always observable fraction of time observable
----------- --------------- ----------------- --------------------------
    Polaris            True              True                        1.0
       Vega            True              True                        1.0
    Albireo            True             False               0.833333333333
      Algol            True             False               0.166666666667
      Rigel           False             False                        0.0
    Regulus           False             False                        0.0
```

Let's sanity-check these results using `plot_sky` to plot the positions of the targets throughout the time range:

We can see that Vega is in the sweet spot in altitude and azimuth for this time range and is always observable. Albireo is not always observable given these criteria because it rises above 80 degrees altitude. Polaris hardly moves and is


N
0.0° Az
45.0°
0° Alt.
30°
60°
90°
E
90.0°
W
270.0°
135.0°
225.0°
S
180.0°
Polaris
Vega
Albireo
Algol
Rigel
Regulus

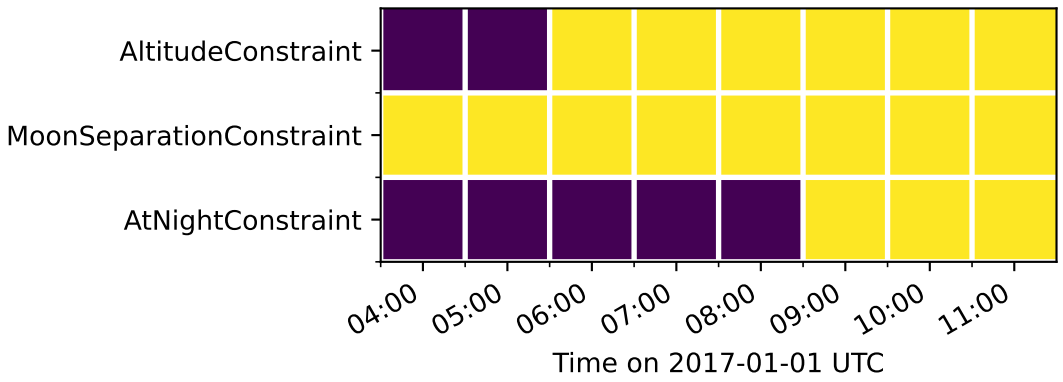
therefore always observable, and Algol starts out observable but sets below the lower altitude limit, and then the airmass limit. Rigel and Regulus never rise above those limits within the time range.

3.4.3 Visualizing Constraints

Suppose an observer is planning to observe low-mass stars in Praesepe in the optical and infrared from the W.M. Keck Observatory. The observing constraints require all observations to occur (i) between astronomical twilights; (ii) while the Moon is separated from Praesepe by at least 45 degrees; and (iii) while Praesepe is above the lower elevation limit of Keck I, about 33 degrees. These observing constraints can be specified with the `AtNightConstraint`, `MoonSeparationConstraint`, and `AltitudeConstraint` objects, like this:

We can evaluate the constraints at one hour intervals in a loop, to create an observability grid like so:

This kind of grid can be useful for visualizing what's happening under-the-hood when you use `is_observable` or `is_always_observable`. Click the link below for the source code to produce the observability grid shown below. Dark squares represent times when the observing constraint is not satisfied.



3.4.4 User-Defined Constraints

There are many possible constraints that you could find useful which have not been implemented (yet) in astroplan. This example will walk you through creating your own constraint which will be compatible with the tools in the `constraints` module.

We will begin by defining an observer at Subaru and reading the text file of stellar coordinates defined in the example above:

```
from astroplan import Observer, FixedTarget
from astropy.time import Time
subaru = Observer.at_site("Subaru")
time_range = Time(["2015-08-01 06:00", "2015-08-01 12:00"])

# Read in the table of targets
from astropy.io import ascii
target_table = ascii.read('targets.txt')

# Create astroplan.FixedTarget objects for each one in the table
from astropy.coordinates import SkyCoord
import astropy.units as u
targets = [FixedTarget(coord=SkyCoord(ra=ra*u.deg, dec=dec*u.deg), name=name)
           for name, ra, dec in target_table]
```

In the previous section, you may have noticed that constraints are assembled by making a list of calls to the initializers for classes like `AltitudeConstraint` and `AirmassConstraint`. Each of those constraint classes is subclassed from the abstract `Constraint` class, and the custom constraint that we're going to write must be as well.

In this example, let's design our constraint to ensure that all targets must be within some angular separation from Vega – we'll call it `VegaSeparationConstraint`. Two methods, `__init__` and `compute_constraint` must be written for our constraint to work:

- The `__init__` method will accept the minimum and maximum acceptable separations a target could have from Vega.
- We'll also define a method `compute_constraints` which takes three arguments: a `Time` or array of times to test, an `Observer` object, and some targets (a `SkyCoord` object representing a single target or a list of targets). `compute_constraints` will return an array of booleans that describe whether or not each target meets the constraints. The super class `Constraint` has a `__call__` method which will run your custom class's `compute_constraints` method when you check if a target is observable using `is_observable` or `is_always_observable`. This `__call__` method also checks the arguments, converting single `FixedTarget` or lists of `FixedTarget` objects into an `SkyCoord` object. The `__call__` method ensures the returned array of booleans is the correct shape, so `compute_constraints` should not normally be called directly - use the `__call__` method instead.
- We also want to provide the option of having the constraint output a non-boolean score. Where being closer to the minimum separation returns a higher score than being closer to the maximum separation.

Here's our `VegaSeparationConstraint` implementation:

```
from astroplan import Constraint, is_observable, min_best_rescale
from astropy.coordinates import Angle
import astropy.units as u

class VegaSeparationConstraint(Constraint):
    """
    Constraint the separation from Vega
```

(continues on next page)

(continued from previous page)

```
    """
    def __init__(self, min=None, max=None, boolean_constraint=True):
        """
        min : `~astropy.units.Quantity` or `None` (optional)
            Minimum acceptable separation between Vega and target.  `None`
            indicates no limit.
        max : `~astropy.units.Quantity` or `None` (optional)
            Minimum acceptable separation between Vega and target.  `None`
            indicates no limit.
        """
        self.min = min if min is not None else 0*u.deg
        self.max = max if max is not None else 180*u.deg
        self.boolean_constraint = boolean_constraint

    def compute_constraint(self, times, observer, targets):

        vega = SkyCoord(ra=279.23473479*u.deg, dec=38.78368896*u.deg)

        # Calculate separation between target and vega
        # Targets are automatically converted to SkyCoord objects
        # by __call__ before compute_constraint is called.
        vega_separation = vega.separation(targets)

        if self.boolean_constraint:
            mask = ((self.min < vega_separation) & (vega_separation < self.max))
            return mask

        # if we want to return a non-boolean score
        else:
            # rescale the vega_separation values so that they become
            # scores between zero and one
            rescale = min_best_rescale(vega_separation, self.min,
                                       self.max, less_than_min=0)
            return rescale
```

Then as in the earlier example, we can call our constraint:

```
>>> constraints = [VegaSeparationConstraint(min=5*u.deg, max=30*u.deg)]
>>> observability = is_observable(constraints, subaru, targets,
...                               time_range=time_range)
>>> print(observability)
[False False  True False False False]
```

The resulting list of booleans indicates that the only target separated by 5 and 30 degrees from Vega is Albireo. Following this pattern, you can design arbitrarily complex criteria for constraints.

By default, calling a constraint will try to broadcast the time and target arrays against each other, and raise a `ValueError` if this is not possible. To see the (target x time) array for the constraint, there is an optional `grid_times_targets` argument. Here we find the (target x time) array for the non-boolean score:

```
>>> constraint = VegaSeparationConstraint(min=5*u.deg, max=30*u.deg,
...                                       boolean_constraint=False)
>>> print(constraint(subaru, targets, time_range=time_range,
```

(continues on next page)

(continued from previous page)

```
...                 grid_times_targets=True))
[[ 0.          0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.        ]
 [ 0.57748686  0.57748686  0.57748686  0.57748686  0.57748686  0.57748686
   0.57748686  0.57748686  0.57748686  0.57748686  0.57748686  0.57748686]
 [ 0.          0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.        ]]
```

The score of .5775 for Albireo indicates that it is slightly closer to the 5 degree minimum than to the 30 degree maximum.

3.5 Scheduling an Observing Run

Note: Some terms used have been defined in *Terminology*.

3.5.1 Contents

- *Defining Targets*
- *Creating Constraints and Observing Blocks*
- *Creating a Transitioner*
- *Scheduling*
- *User-Defined Schedulers*
- *Using the Scorer*

3.5.2 Defining Targets

We want to observe Deneb and M13 in the B, V and R filters. We are scheduled for the first half-night on July 6 2016 and want to know the order we should schedule the targets in.

First we define our `Observer` object (where we are observing from):

```
>>> from astroplan import Observer

>>> apo = Observer.at_site('apo')
```

Now we want to define our list of targets (`FixedTarget` objects), any object that is in SIMBAD can be called by an identifier.

```
>>> from astroplan import FixedTarget

>>> # Initialize the targets
>>> deneb = FixedTarget.from_name('Deneb')
>>> m13 = FixedTarget.from_name('M13')

>>> deneb
<FixedTarget "deneb" at SkyCoord (ICRS): (ra, dec) in deg (310.35797975, 45.28033881)>

>>> m13
<FixedTarget "M13" at SkyCoord (ICRS): (ra, dec) in deg (250.423475, 36.4613194)>
```

We also need to define bounds within which our blocks will be scheduled using `Time` objects. Our bounds will be from the noon before our observation, to the noon after (19:00 UTC). Later we will account for only being able to use the first half of the night.

```
>>> from astropy.time import Time

>>> noon_before = Time('2016-07-06 19:00')
>>> noon_after = Time('2016-07-07 19:00')
```

Return to Top

3.5.3 Creating Constraints and Observing Blocks

An in-depth tutorial on creating and using constraints can be found in the *constraint tutorial*.

Constraints, when evaluated, take targets and times, and give scores that indicate how well the combination of target and time fulfill the constraint. We want to make sure that our targets will be high in the sky while observed and that they will be observed during the night. We don't want any object to be observed at an airmass greater than 3, but we prefer a better airmass. Usually constraints scores are boolean, but with `boolean_constraint = False` the constraint will output floats instead, indicated when it is closer to ideal.

```
>>> from astroplan.constraints import AtNightConstraint, AirmassConstraint

>>> # create the list of constraints that all targets must satisfy
>>> global_constraints = [AirmassConstraint(max = 3, boolean_constraint = False),
...                       AtNightConstraint.twilight_civil()]
```

Now that we have constraints that we will apply to every target, we need to create an `ObservingBlock` for each target+configuration combination. An observing block needs a target, a duration, and a priority; configuration information can also be given (i.e. filter, instrument, etc.). For each filter we want 16 exposures per target (100 seconds for M13 and 60 seconds for Deneb) and the instrument has a read-out time of 20 seconds. The half night goes from 7PM local time to 1AM local time, in UTC this will be from 2AM to 8AM, so we use `TimeConstraint`.

```
>>> from astroplan import ObservingBlock
>>> from astroplan.constraints import TimeConstraint
>>> from astropy import units as u

>>> # Define the read-out time, exposure duration and number of exposures
>>> read_out = 20 * u.second
>>> deneb_exp = 60*u.second
>>> m13_exp = 100*u.second
```

(continues on next page)

(continued from previous page)

```
>>> n = 16
>>> blocks = []

>>> half_night_start = Time('2016-07-07 02:00')
>>> half_night_end = Time('2016-07-07 08:00')
>>> first_half_night = TimeConstraint(half_night_start, half_night_end)
>>> # Create ObservingBlocks for each filter and target with our time
>>> # constraint, and durations determined by the exposures needed
>>> for priority, bandpass in enumerate(['B', 'G', 'R']):
...     # We want each filter to have separate priority (so that target
...     # and reference are both scheduled)
...     b = ObservingBlock.from_exposures(deneb, priority, deneb_exp, n, read_out,
...                                       configuration = {'filter': bandpass},
...                                       constraints = [first_half_night])
...     blocks.append(b)
...     b = ObservingBlock.from_exposures(m13, priority, m13_exp, n, read_out,
...                                       configuration = {'filter': bandpass},
...                                       constraints = [first_half_night])
...     blocks.append(b)
```

3.5.4 Creating a Transitioner

Now that we have observing blocks, we need to define how the telescope transitions between them. The first parameter needed is the slew_rate of the telescope (degrees/second) and the second is a dictionary that tells how long it takes to transition between two configurations. You can also give a default duration if you aren't able to give one for each pair of configurations.

```
>>> from astroplan.scheduling import Transitioner

>>> # Initialize a transitioner object with the slew rate and/or the
>>> # duration of other transitions (e.g. filter changes)
>>> slew_rate = .8*u.deg/u.second
>>> transitioner = Transitioner(slew_rate,
...                             {'filter':{('B','G'): 10*u.second,
...                                        ('G','R'): 10*u.second,
...                                        'default': 30*u.second}})
```

The transitioner now knows that it takes 10 seconds to go from 'B' to 'G', or from 'G' to 'R' but has to use the default transition time of 30 seconds for any other transition between filters. Non-transitions, like 'g' to 'g', will not take any time though.

3.5.5 Scheduling

Now all we have left is to initialize the scheduler, input our list of blocks and the schedule to put them in. There are currently two schedulers to choose from in astroplan.

The first is a sequential scheduler. It starts at the start_time and scores each block (constraints and target) at that time and then schedules it, it then moves to where the first observing block stops and repeats the scoring and scheduling on the remaining blocks.

```
>>> from astroplan.scheduling import SequentialScheduler
>>> from astroplan.scheduling import Schedule

>>> # Initialize the sequential scheduler with the constraints and transitioner
>>> seq_scheduler = SequentialScheduler(constraints = global_constraints,
...                                     observer = apo,
...                                     transitioner = transitioner)
>>> # Initialize a Schedule object, to contain the new schedule
>>> sequential_schedule = Schedule(noon_before, noon_after)

>>> # Call the schedule with the observing blocks and schedule to schedule the blocks
>>> seq_scheduler(blocks, sequential_schedule)
```

The second is a priority scheduler. It sorts the blocks by their priority (multiple blocks with the same priority will stay in the order they were in), then schedules them one-by-one at the best time for that block (highest score).

```
>>> from astroplan.scheduling import PriorityScheduler

>>> # Initialize the priority scheduler with the constraints and transitioner
>>> prior_scheduler = PriorityScheduler(constraints = global_constraints,
...                                     observer = apo,
...                                     transitioner = transitioner)
>>> # Initialize a Schedule object, to contain the new schedule
>>> priority_schedule = Schedule(noon_before, noon_after)

>>> # Call the schedule with the observing blocks and schedule to schedule the blocks
>>> prior_scheduler(blocks, priority_schedule)
```

Now that you have a schedule there are a few ways of viewing it. One way is to have it print a table where you can show, or hide, unused time and transitions with `show_transitions` and `show_unused` (default is showing transitions and not unused).

```
>>> priority_schedule.to_table()
     target         start time (UTC)         end time (UTC)     duration (minutes)        ra␣
↪           dec         configuration
     str15               str23                   str23               float64           ␣
↪str32          str32            object
--------------- ---------------------- ---------------------- ------------------ ----------
↪----- -------------- -----------------
            M13 2016-07-07 03:49:20.019 2016-07-07 04:21:20.019               32.0  ␣
↪250d25m24.51s 36d27m40.7498s   {'filter': 'R'}
TransitionBlock 2016-07-07 04:21:20.019 2016-07-07 04:22:00.019     0.666666666667          ␣
↪                     ['filter:R to B']
            M13 2016-07-07 04:25:20.021 2016-07-07 04:57:20.021               32.0  ␣
↪250d25m24.51s 36d27m40.7498s   {'filter': 'B'}
TransitionBlock 2016-07-07 04:57:20.021 2016-07-07 04:57:40.021     0.333333333333          ␣
↪                     ['filter:B to G']
            M13 2016-07-07 04:57:40.021 2016-07-07 05:29:40.021               32.0  ␣
↪250d25m24.51s 36d27m40.7498s   {'filter': 'G'}
TransitionBlock 2016-07-07 05:29:40.021 2016-07-07 05:31:00.021      1.33333333333          ␣
↪                     ['filter:G to R']
          Deneb 2016-07-07 06:44:00.026 2016-07-07 07:05:20.026      21.3333333333 310d21m28.
↪7271s 45d16m49.2197s   {'filter': 'R'}
```

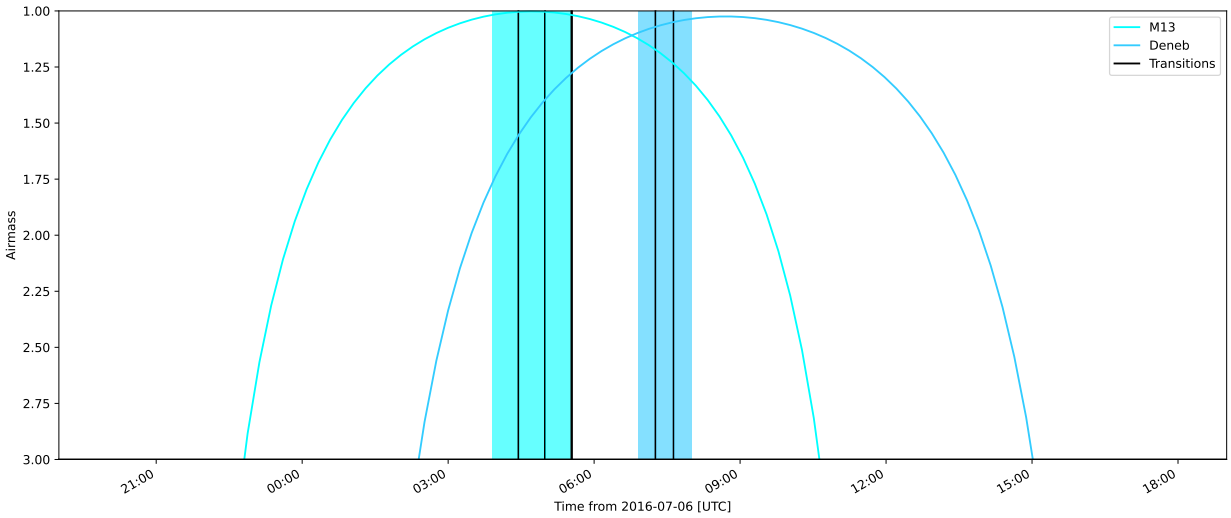
(continues on next page)

(continued from previous page)

```
TransitionBlock 2016-07-07 07:05:20.026 2016-07-07 07:06:00.026     0.666666666667          ␣
↪                    ['filter:R to G']
         Deneb 2016-07-07 07:09:20.027 2016-07-07 07:30:40.027      21.3333333333 310d21m28.
↪7271s 45d16m49.2197s   {'filter': 'G'}
TransitionBlock 2016-07-07 07:30:40.027 2016-07-07 07:31:20.027     0.666666666667          ␣
↪                    ['filter:G to B']
         Deneb 2016-07-07 07:34:40.028 2016-07-07 07:56:00.028      21.3333333333 310d21m28.
↪7271s 45d16m49.2197s   {'filter': 'B'}
```

The other way is to plot the schedule against the airmass of the targets.

```
>>> from astroplan.plots import plot_schedule_airmass
>>> import matplotlib.pyplot as plt

>>> # plot the schedule with the airmass of the targets
>>> plt.figure(figsize = (14,6))
>>> plot_schedule_airmass(priority_schedule)
>>> plt.legend(loc = "upper right")
>>> plt.show()
```



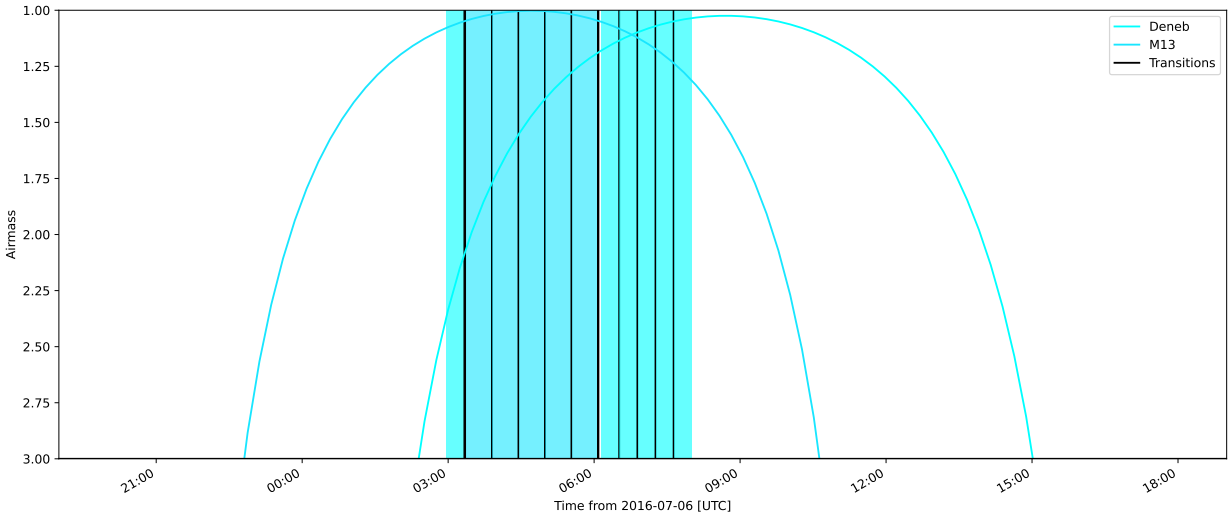
There is a lot of unfilled space in our schedule currently. We can fill that time with more observations of our targets by calling our scheduler using the same blocks and the schedule we already added to.

```
>>> prior_schedule(blocks, priority_schedule)
>>> plt.figure(figsize = (14,6))
>>> plot_schedule_airmass(priority_schedule)
>>> plt.legend(loc = "upper right")
>>> plt.show()
```

We want to check if there is any way that we could observe Alpha Centauri A as well during our time slot. So we create a new block for it with priority over the others, add it to our list of blocks and run the priority scheduler again.

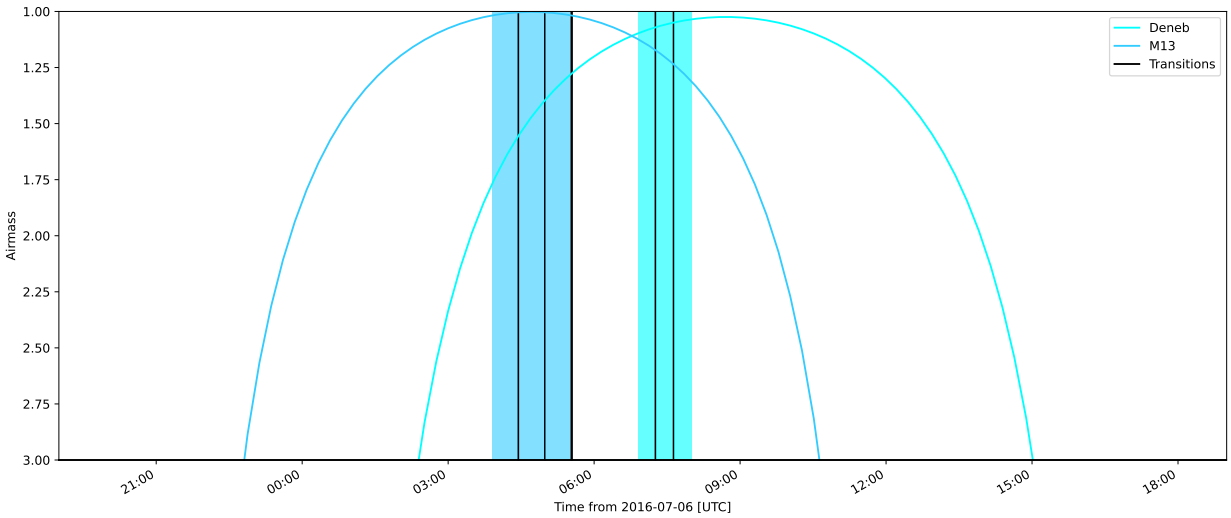
```
>>> alpha_cen = FixedTarget.from_name('Alpha Centauri A')
>>> # ObservingBlocks can also be called with arguments: target, duration, priority
>>> blocks.append(ObservingBlock(alpha_cen, 20*u.minute, -1))
```

(continues on next page)



(continued from previous page)

```
>>> # Initialize a new schedule for this test
>>> schedule = Schedule(start_time, end_time)
>>> prior_scheduler(blocks, schedule)

>>> plt.figure(figsize = (14,6))
>>> plot_schedule_airmass(priority_schedule)
>>> plt.legend(loc = "upper right")
>>> plt.show()
```



Nothing new shows up because Alpha Centauri isn't visible from APO.

3.5.6 User-Defined Schedulers

There are many ways that targets can be scheduled, only two of which are currently implemented. This example will walk through the steps for creating your own scheduler that will be compatible with the tools of the `scheduling` module.

As you may have noticed above, the schedulers are assembled by making a call to the initializer of the class (e.g. `PriorityScheduler`). Each of the schedulers is subclassed from the abstract `astroplan.scheduling.Scheduler` class, and our custom scheduler needs to be as well.

A scheduler needs to be able to schedule observing blocks where they have a non-zero score (i.e. they satisfy all of their constraints). For our scheduler, we will make one that schedules `ObservingBlocks` at the first unoccupied place they have a score greater than zero: a `SimpleScheduler`. We need to include two methods, `__init__` and `_make_schedule` for it to work:

- The `__init__` is already defined by the super class, and accepts global constraints, the `Observer`, the `Transitioner`, a `gap_time`, and a `time_resolution` for spacing during the creation of the schedule.
- It also needs a `_make_schedule` to do the heavy lifting. This takes a list of `ObservingBlock` objects and a `Schedule` object to input them into. This method needs to be able to check whether a block can be scheduled in a given spot, and be able to insert it into the schedule once a suitable spot has been found. For score evaluation we will use the built-in `Scorer`.

Here's the `SimpleScheduler` implementation:

```
from astroplan.scheduling import Scheduler, Scorer
from astroplan.utils import time_grid_from_range
from astroplan.constraints import AltitudeConstraint
from astropy import units as u

import numpy as np

class SimpleScheduler(Scheduler):
    """
    schedule blocks randomly
    """
    def __init__(self, *args, **kwargs):
        super(SimpleScheduler, self).__init__(*args, **kwargs)

    def _make_schedule(self, blocks):
        # gather all the constraints on each block into a single attribute
        for b in blocks:
            if b.constraints is None:
                b._all_constraints = self.constraints
            else:
                b._all_constraints = self.constraints + b.constraints

            # to make sure the Scorer has some constraint to work off of
            # and to prevent scheduling of targets below the horizon
            if b._all_constraints is None:
                b._all_constraints = [AltitudeConstraint(min=0*u.deg)]
                b.constraints = [AltitudeConstraint(min=0*u.deg)]
            elif not any(isinstance(c, AltitudeConstraint) for c in b._all_constraints):
                b._all_constraints.append(AltitudeConstraint(min=0*u.deg))
                if b.constraints is None:
                    b.constraints = [AltitudeConstraint(min=0*u.deg)]
                else:
```

(continues on next page)

(continued from previous page)

```
                b.constraints.append(AltitudeConstraint(min=0*u.deg))
            b.observer = self.observer

        # before we can schedule, we need to know where blocks meet the constraints
        scorer = Scorer(blocks, self.observer, self.schedule,
                        global_constraints=self.constraints)
        score_array = scorer.create_score_array(self.time_resolution)
        # now we have an array of the scores for the blocks at intervals of
        # ``time_resolution``. The scores range from zero to one, some blocks may have
        # higher scores than others, but we only care if they are greater than zero

        # we want to start from the beginning and start scheduling
        start_time = self.schedule.start_time
        current_time = start_time
        while current_time < self.schedule.end_time:
            scheduled = False
            i=0
            while i < len(blocks) and scheduled is False:
                block = blocks[i]
                # the schedule starts with only 1 slot
                if len(self.schedule.slots) == 1:
                    test_time = current_time
                # when a block is inserted, the number of slots increases
                else:
                    # a test transition between the last scheduled block and this one
                    transition = self.transitioner(schedule.observing_blocks[-1],
                                                   block, current_time, self.observer)
                    test_time = current_time + transition.duration
                # how many time intervals are we from the start
                start_idx = int((test_time - start_time)/self.time_resolution)
                duration_idx = int(block.duration/self.time_resolution)
                # if any score during the block's duration would be 0, reject it
                if any(score_array[i][start_idx:start_idx+duration_idx] == 0):
                    i +=1
                # if all of the scores are >0, accept and schedule it
                else:
                    if len(self.schedule.slots) >1:
                        self.schedule.insert_slot(current_time, transition)
                    self.schedule.insert_slot(test_time, block)
                    # advance the time and remove the block from the list
                    current_time = test_time + block.duration
                    scheduled = True
                    blocks.remove(block)
            # if every block failed, progress the time
            if i == len(blocks):
                current_time += self.gap_time
        return schedule
```

Then to use our new scheduler, we just need to call it how we did up above:

```
>>> from astroplan.constraints import AtNightConstraint
>>> from astroplan.scheduling import Schedule, ObservingBlock
```

(continues on next page)

(continued from previous page)

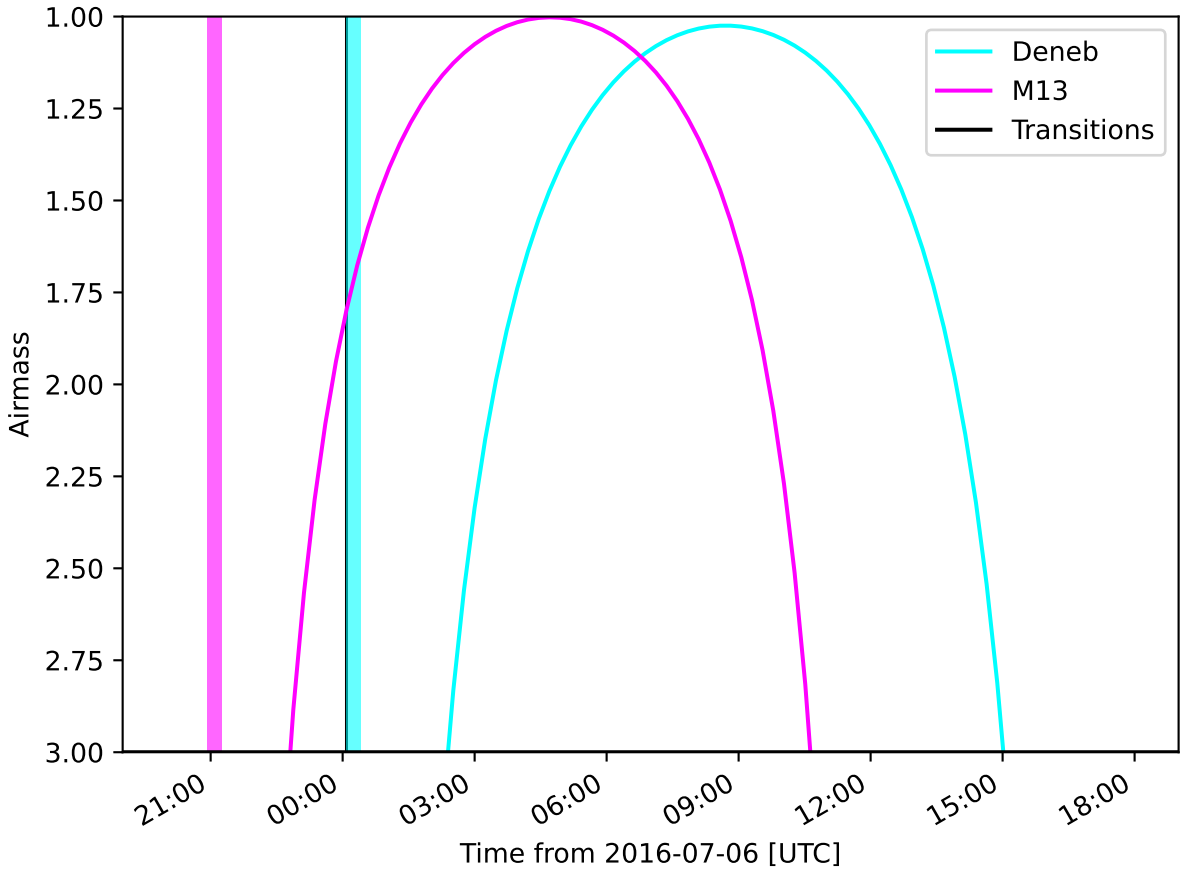
```
>>> from astroplan import FixedTarget, Observer, Transitioner
>>> from astropy.time import Time

>>> # Initialize the observer and targets, and create observing blocks
>>> apo = Observer.at_site('apo')
>>> deneb = FixedTarget.from_name('Deneb')
>>> m13 = FixedTarget.from_name('M13')
>>> blocks = [ObservingBlock(deneb, 20*u.minute, 0)]
>>> blocks.append(ObservingBlock(m13, 20*u.minute, 0))

>>> # For a telescope that can slew at a rate of 2 degrees/second
>>> transitioner = Transitioner(slew_rate=2*u.deg/u.second)

>>> # Schedule the observing blocks using the simple scheduler
>>> schedule = Schedule(Time('2016-07-06 19:00'), Time('2016-07-07 19:00'))
>>> scheduler = SimpleScheduler(observer = apo, transitioner = transitioner,
...                             constraints = [])
>>> scheduler(blocks, schedule)

>>> # Plot the created schedule
>>> import matplotlib.pyplot as plt
>>> from astroplan.plots import plot_schedule_airmass
>>> plot_schedule_airmass(schedule)
>>> plt.legend()
>>> plt.show()
```

We gave the scheduler no constraints, global or local, so it added the default `AltitudeConstraint` which is only satisfied when the targets are above the horizon. Therefore the ObservingBlocks are scheduled at the first available time after the target rises, which occurs at much higher airmass than the plot shows.

3.5.7 Using the Scorer

The Scheduler defined above uses `create_score_array`, which creates an array with dimensions (# of blocks, schedule duration/`time_resolution`). The Score of any element (block, time) in that array is made by multiplying the scores returned by all of the constraints for that target and time.

If you wish to use a different method of score evaluation, you can add a new method to the `Scorer`. The general framework of the `create_score_array` method will ensure evaluation of all of the constraints, but change how it combines the scores from the separate constraints (e.g. add the reciprocals of the scores together and then use smaller values as better). If you create a method that might be generically useful to other users, consider submitting it so that others will be able to use it as well.


1.00
1.25
1.50
1.75
2.00
2.25
2.50
2.75
3.00
Airmass
21:00
00:00
03:00
06:00
09:00
12:00
15:00
18:00
Time from 2016-07-06 [UTC]
Deneb
M13
Transitions

3.6 Speeding up astroplan

Some users of astroplan may find it useful to trade-off a bit of precision in the rise/set/transit times of targets in exchange for computational efficiency. In this short tutorial, we show you how to speed up astroplan in exchange for a bit of time precision, which is especially useful when planning many observations over a long period of time.

3.6.1 Rise/set/transit times

The rise, set, and transit time methods on the `Observer` object take an optional keyword argument called `n_grid_points` as of astroplan version 0.6 (in earlier versions of astroplan, `n_grid_points` is fixed to 150). To understand `n_grid_points` you first need to know how target rise/set/transit times are computed in astroplan.

Astroplan computes rise/set times relative to a given reference time by computing the altitude of the target on a grid which spans a period of 24 hours before/after the reference time. The grid is then searched for horizon-crossings, and astroplan interpolates between the two nearest-to-zero altitudes to approximate the target rise/set times.

The `n_grid_points` keyword argument dictates the number of grid points on which to compute the target altitude. The larger the `n_grid_points`, the more precise the rise/set/transit time will be, but the operation also becomes more computationally expensive. As a general rule of thumb, if you choose `n_grid_points=150` your rise/set time precisions will be precise to better than one minute; this is the default if you don't specify `n_grid_points`. If you choose `n_grid_points=10` you'll get significantly faster rise/set time computations, but your precision degrades to better than five minutes.

3.6.2 Examples

Let's see some simple examples. We can compute a very accurate rise time for Sirius over Apache Point Observatory, by specifying `n_grid_points=1000`:

```
>>> from astroplan import Observer, FixedTarget
>>> from astropy.time import Time

>>> time = Time('2019-01-01 00:00')
>>> sirius = FixedTarget.from_name('Sirius')
>>> apo = Observer.at_site('APO')

>>> rise_time_accurate = apo.target_rise_time(time, sirius, n_grid_points=1000)
>>> rise_time_accurate.iso
'2019-01-01 01:52:13.393'
```

That's the rise time computed on a grid of 1000 altitudes in a 24 hour period, so it should be very accurate, but we can run the `timeit` function on the above code snippet to see how slow this is:

```
290 ms ± 4.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Now let's compute a lower precision, but much faster rise time, using `N=10` this time:

```
>>> rise_time_fast = apo.target_rise_time(time, sirius, n_grid_points=10)
>>> rise_time_fast.iso
'2019-01-01 01:54:09.946'
```

And timing the above snippet, we find:

```
27.3 ms ± 709 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

You can see that the rise time returned by `target_rise_time` with `n_grid_points=10` is only two minutes different from the prediction with `n_grid_points=1000`, so it looks like we haven't lost much precision despite the drastically different number of grid points and an order-of-magnitude speedup.

CHAPTER

FOUR

FAQ

We've assembled some caveats and gotchas in this Frequently Asked Questions section. If your question isn't addressed here or on the astroplan Issues page, post a new issue.

4.1 What are the IERS tables and how do I update them?

4.1.1 Contents

4.1.2 Background

Keeping accurate, consistent time systems is rough. We intuitively want time to be described by a continuous cyclic clock like the clock on your wall or the calendar, but the Earth's rate of rotation varies measurably on human timescales. As a result, the time measured by an atomic clock and the time that you would infer from measuring how long it has been since the last solar noon are constantly becoming offset from one another with stochastic changes in Earth's moment of inertia and slowly acting tidal forces.

For this reason, there is a time system that keeps track of seconds like an atomic clock, TAI, a more commonly used one that works like an atomic clock with a varying offset of some integer number of seconds, UTC, and one that matches up with the Earth's rotation, UT1. The difference between UT1 and UTC is constantly changing as the Earth's rotation changes and as leap seconds get added to UTC to compensate.

In order to accurately predict the apparent position of a celestial object from the Earth, for example in altitude and azimuth coordinates, one needs to know the orientation of the Earth at any point in time, and since the Earth's rotation does not change in a predictable way, we must rely on observations of the Earth's orientation as a function of time to accurately calculate UT1-UTC.

4.1.3 What are IERS Bulletins A and B?

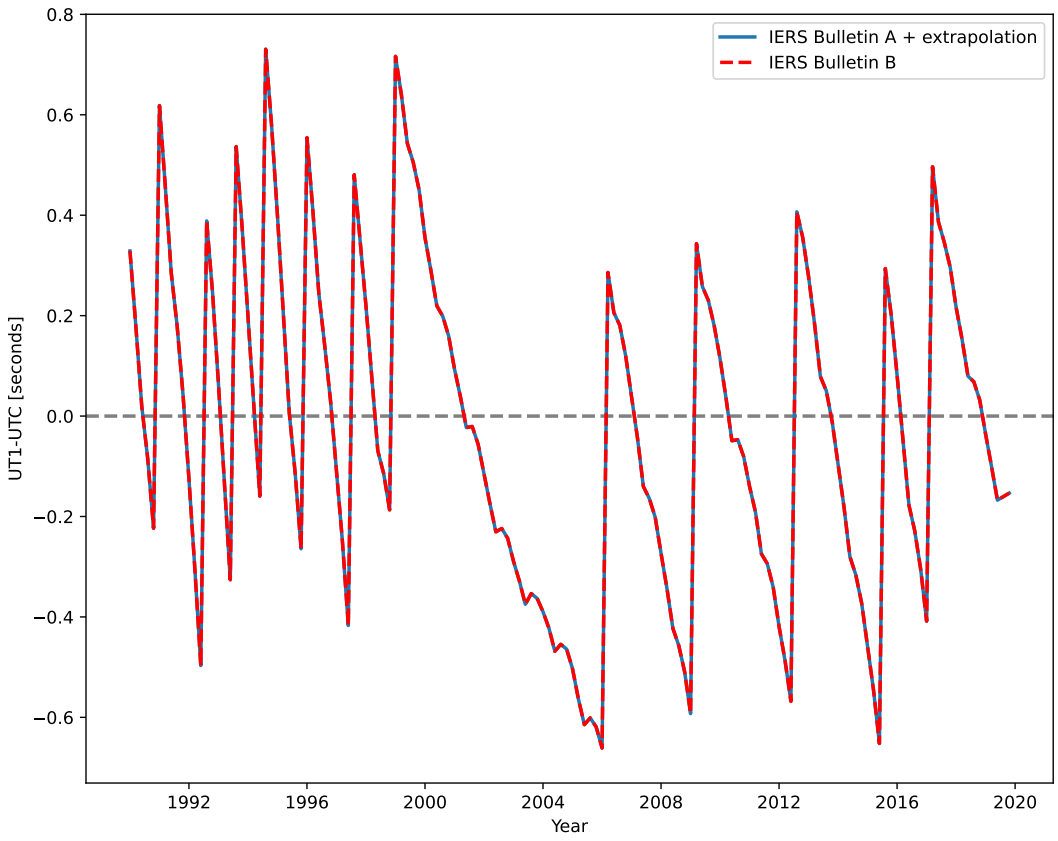
The International Earth Rotation and Reference Systems Service (IERS) is responsible for measuring the Earth's orientation as a function of time, and making predictions for the Earth's orientation in the near future (accounting for scheduled leap seconds). The data products released by the IERS used by astroplan are the IERS Bulletins A and B.

- IERS Bulletin B is a table with Earth orientation observations from the last few decades up through nearly the present time. Astropy relies on IERS B to compute UT1-UTC, and by default will raise an error if you try to compute UT1-UTC for a time that is outside the bounds of the IERS Bulletin B table (see the Astropy docs on the UT1/UTC transformation offsets for more details), like this:

```
>>> from astropy.time import Time
>>> Time('2040-01-01', scale='utc').ut1    # Convert from UTC to UT1
IndexError: (some) times are outside of range covered by IERS table.
```

- IERS Bulletin A encompasses the observations of recent Earth orientation contained in Bulletin B, while also making extrapolations into the past, before the Bulletin B tables begin, and into the future, after the Bulletin B tables end. The future predictions include leap second additions scheduled to be added.

Let's plot the UT1-UTC from IERS Bulletins A and B to show the difference using astropy's IERS machinery:



4.1.4 How do I get IERS Bulletin A for astroplan?

Without downloading IERS Bulletin A, astroplan simply approximates UT1-UTC=0 always. This will lead to lower precision position and time calculations on the order of arcseconds or seconds, and allow you to handle times in the far future and distant past.

To download the IERS Bulletin A table for the first time, or to refresh the cached version that you already have, simply run:

```
from astroplan import download_IERS_A
download_IERS_A()
```

4.2 Why is my target above/below the horizon at the rise/set time?

Rise/set/meridian transit calculations in `astroplan` are designed to be fast while achieving a precision comparable to what can be predicted given the affects of the changing atmosphere. As a result, there may be some counter-intuitive behavior in `astroplan` methods like `astroplan.Observer.target_rise_time`, `astroplan.Observer.target_set_time` and `astroplan.Observer.target_meridian_transit_time`, that can lead to small changes in the numerical values of these computed timed (of order seconds).

For example, to calculate the rise time of Sirius, you might write:

```
from astroplan import Observer, FixedTarget
from astropy.time import Time

# Set up observer, target, and time
keck = Observer.at_site("Keck")
sirius = FixedTarget.from_name("Sirius")
time = Time('2010-05-11 06:00:00')

# Find rise time of Sirius at Keck nearest to `time`
rise_time = keck.target_rise_time(time, sirius)
```

You might expect the altitude of Sirius to be zero degrees at `rise_time`, i.e. Sirius will be on the horizon, but this is not the case:

```
>>> altitude_at_rise = keck.altaz(rise_time, sirius).alt
>>> print(altitude_at_rise.to('arcsec'))
2.70185arcsec
```

The altitude that you compute on your machine may be different from the number above by a small amount – for a detailed explanation on where the difference arises from, see *What are the IERS tables and how do I update them?*. The rise and set time methods use the following approximation:

- A time series of altitudes for the target is computed at times near `time`
- The two times when the target is nearest to the horizon are identified, and a linear interpolation is done between those times to find the horizon-crossing

This method has a precision of a few arcseconds, so your targets may be slightly above or below the horizon at their rise or set times.

4.3 How are sunrise and sunset defined?

By default, `sun_set_time` will compute an approximation to the moment when the center of the solar disk is on the horizon. This differs slightly from the conventional definition used by USNO, for example, which returns the time when the solar disk center is at -0.8333 degrees altitude to account for the solar radius and atmospheric refraction. If you want to accurately reproduce the sun rise and set times computed by USNO, you can call `sun_set_time` with the keyword argument `horizon=-0.8333*u.deg`, like so:

```
>>> from astroplan import Observer
>>> from astropy.time import Time
>>> import astropy.units as u

>>> mmt = Observer.at_site('mmt', pressure=0*u.bar)

>>> # USNO time from the MMTO Almanac:
>>> # http://www.mmto.org/sites/default/files/almanac_2019.pdf
>>> usno_sunset = Time('2019-01-01 00:31')

>>> # Compute equivalent time with astroplan
>>> astroplan_sunset = mmt.sun_set_time(usno_sunset - 10*u.min,
...                                     horizon=-0.8333*u.deg, which='next')

>>> abs(usno_sunset - astroplan_sunset) < 1 * u.min
True
```

4.4 Terminology

4.4.1 Scheduling

Scheduling observations is a common task in astronomy, but there is a lack of formal terminology. Here we define the terminology as it will be used in astroplan for scheduling tasks. This is for consistency within the code and the documentation.

The terms used are as follows:

- **observing block** (**OB**): an observation of a target for an amount of time and a particular instrument configuration.
- **priority**: number assigned to the **observing block** by the user that defines its precedence within the set of blocks to be scheduled. Should probably also be on a 0->1 (0=no good, 1=good) scale, or rescaled within the scheduler
- **rank**: like a **priority**, but defined for a specific set of **OB**s, often set by an agent other than the observer. (e.g. a proposal is **ranked** by a time allocation committee (TAC))
- **constraint**: sets limits and can yield **scores** based on inputs. Currently only boolean scores are implemented in the code.
- **score**: the value returned by evaluating a **constraint** or **constraints**. Can be boolean or floats between 0 and 1 inclusive (0=no good, 1=good), with a flag in the `Constraint` call that selects which is used. Can be combined by a **scorekeeper** into a cumulative score for an **observing block** given a start time.
- **scorekeeper**: assigns a cumulative score to the **OB** based on some function that would be applied to all the individual **scores** (result should be in (0, 1))
- **scheduler**: the entity that consumes the results of the **scorekeeper** and the **observing blocks** and produces a schedule

- **Weights** ("user defined"?): preferences for which constraint's **scores** matter most (e.g. I care more about getting dark time than getting a low airmass). **weights** can be floats between 0 and 1 inclusive (0=not important, 1 = important)

This is a list of possible schedulers, none are implemented yet. Once implemented they will have different methods for creating their schedule. Below is a list of ideas for the scheduler descriptors and the related scheduler would work. Each scheduler will be able to be called with `DescriptorScheduler(args)` using the descriptor defined below.

- **Sequential**: starts from the beginning of the time range and schedules the **OB** with the best **score**, that hasn't already been scheduled, for that time.
- **Priority**: starts from the highest **priority OB** and schedules it at the time where it has its highest **score**. Then schedules the next-highest priority without overlapping blocks

4.5 Getting Help & Contributing

4.5.1 Getting Help

If your questions are not addressed in the docs, you can reach out to us for help at the astropy-dev mailing list.

4.5.2 Contributing to `astroplan`

Contributions to improve, expand and fix `astroplan` are welcome! As part of the Astropy ecosystem, we recommend the following resources for getting started on your first contributions to astroplan:

- How to make a code contribution
- Comprehensive Coding Guidelines
- Astropy Developer Documentation

Note: `astroplan` is a very young project and we welcome new features. If you are interested in contributing, contact us via GitHub or the astropy-dev mailing list to check what's already in progress and what still needs to be started.

CHAPTER

FIVE

REFERENCE/API

5.1 astroplan Package

astroplan is an open source (BSD licensed) observation planning package for astronomers that can help you plan for everything but the clouds.

It is an in-development Astropy affiliated package that seeks to make your life as an observational astronomer a little less infuriating.

- Code: https://github.com/astropy/astroplan
- Docs: https://astroplan.readthedocs.io/

5.1.1 Functions

download_IERS_A([show_progress])	Download and cache the IERS Bulletin A table.
is_always_observable(constraints, observer, ...)	A function to determine whether targets are always observable throughout time_range given constraints in the constraints_list for a particular observer.
is_event_observable(constraints, observer, ...)	Determines if the target is observable at each time in times, given constraints in constraints for a particular observer.
is_observable(constraints, observer, targets)	Determines if the targets are observable during time_range given constraints in constraints_list for a particular observer.
max_best_rescale(vals, min_val, max_val[, ...])	rescales an input array vals to be a score (between zero and one), where the max_val goes to one, and the min_val goes to zero.
min_best_rescale(vals, min_val, max_val[, ...])	rescales an input array vals to be a score (between zero and one), where the min_val goes to one, and the max_val goes to zero.
months_observable(constraints, observer, targets)	Determines which month the specified targets are observable for a specific observer, given the supplied constraints.
moon_illumination(time[, ephemeris])	Calculate fraction of the moon illuminated.
moon_phase_angle(time[, ephemeris])	Calculate lunar orbital phase in radians.
observability_table(constraints, observer, ...)	Creates a table with information about observability for all the targets over the requested time_range, given the constraints in constraints_list for observer.
stride_array(arr, window_width)	Computes all possible sequential subarrays of arr with length = window_width
time_grid_from_range(time_range[, ...])	Get linearly-spaced sequence of times.

download_IERS_A

astroplan.**download_IERS_A**(*show_progress=True*)

Download and cache the IERS Bulletin A table.

If one is already cached, download a new one and overwrite the old. Store table in the astropy cache, and undo the monkey patching caused by earlier failure (if applicable).

If one does not exist, monkey patch `_get_delta_ut1_utc` so that `Time` objects don't raise errors by computing UT1-UTC off the end of the IERS table.

Parameters

show_progress

[bool] `True` shows a progress bar during the download.

is_always_observable

astroplan.**is_always_observable**(*constraints*, *observer*, *targets*, *times=None*, *time_range=None*, *time_grid_resolution=<Quantity 0.5 h>*)

A function to determine whether `targets` are always observable throughout `time_range` given constraints in the `constraints_list` for a particular `observer`.

Parameters

constraints

[list or `Constraint`] Observational constraint(s)

observer

[`Observer`] The observer who has constraints `constraints`

targets

[{list, `SkyCoord`, `FixedTarget`}] Target or list of targets

times

[`Time` (optional)] Array of times on which to test the constraint

time_range

[`Time` (optional)] Lower and upper bounds on time sequence, with spacing `time_resolution`. This will be passed as the first argument into `time_grid_from_range`.

time_grid_resolution

[`Quantity` (optional)] If `time_range` is specified, determine whether constraints are met between test times in `time_range` by checking constraint at linearly-spaced times separated by `time_resolution`. Default is 0.5 hours.

Returns

ever_observable

[list] List of booleans of same length as `targets` for whether or not each target is observable in the time range given the constraints.

is_event_observable

astroplan.**is_event_observable**(*constraints*, *observer*, *target*, *times=None*, *times_ingress_egress=None*)

Determines if the `target` is observable at each time in `times`, given constraints in `constraints` for a particular `observer`.

Parameters

constraints

[list or `Constraint`] Observational constraint(s)

observer

[`Observer`] The observer who has constraints `constraints`

target

[{list, `SkyCoord`, `FixedTarget`}] Target

times

[`Time` (optional)] Array of mid-event times on which to test the constraints

times_ingress_egress

[`Time` (optional)] Array of ingress and egress times for `N` events, with shape (`N`, 2).

Returns

event_observable

[`ndarray`] Array of booleans of same length as `times` for whether or not the target is ever observable at each time, given the constraints.

is_observable

astroplan.**is_observable**(*constraints*, *observer*, *targets*, *times=None*, *time_range=None*, *time_grid_resolution=<Quantity 0.5 h>*)

Determines if the `targets` are observable during `time_range` given constraints in `constraints_list` for a particular `observer`.

Parameters

constraints

[list or `Constraint`] Observational constraint(s)

observer

[`Observer`] The observer who has constraints `constraints`

targets

[{list, `SkyCoord`, `FixedTarget`}] Target or list of targets

times

[`Time` (optional)] Array of times on which to test the constraint

time_range

[Time (optional)] Lower and upper bounds on time sequence, with spacing `time_resolution`. This will be passed as the first argument into `time_grid_from_range`.

time_grid_resolution

[Quantity (optional)] If `time_range` is specified, determine whether constraints are met between test times in `time_range` by checking constraint at linearly-spaced times separated by `time_resolution`. Default is 0.5 hours.

Returns

ever_observable

[list] List of booleans of same length as `targets` for whether or not each target is ever observable in the time range given the constraints.

max_best_rescale

`astroplan.`**`max_best_rescale`**(*vals*, *min_val*, *max_val*, *greater_than_max=1*)

rescales an input array `vals` to be a score (between zero and one), where the `max_val` goes to one, and the `min_val` goes to zero.

Parameters

vals

[array-like] the values that need to be rescaled to be between 0 and 1

min_val

[float] worst acceptable value (rescales to 0)

max_val

[float] best value cared about (rescales to 1)

greater_than_max

[0 or 1] what is returned for `vals` above `max_val`. (in some cases anything higher than `max_val` should also return one, in some cases it should return zero)

Returns

array of floats between 0 and 1 inclusive rescaled so that `vals` equal to `min_val` equal 0 and those equal to `max_val` equal 1

Examples

rescale an array of altitudes to be between 0 and 1, with the best (60) going to 1 and worst (35) going to 0. For values outside the range, the rescale should return 0 below 35 and 1 above 60. >>> from astroplan.constraints import max_best_rescale >>> import numpy as np >>> altitudes = np.array([20, 30, 40, 45, 55, 70]) >>> max_best_rescale(altitudes, 35, 60) # doctest: +FLOAT_CMP array([0. , 0. , 0.2, 0.4, 0.8, 1.])

min_best_rescale

astroplan.**min_best_rescale**(*vals*, *min_val*, *max_val*, *less_than_min=1*)

rescales an input array `vals` to be a score (between zero and one), where the `min_val` goes to one, and the `max_val` goes to zero.

Parameters

vals

[array-like] the values that need to be rescaled to be between 0 and 1

min_val

[float] worst acceptable value (rescales to 0)

max_val

[float] best value cared about (rescales to 1)

less_than_min

[0 or 1] what is returned for `vals` below `min_val`. (in some cases anything less than `min_val` should also return one, in some cases it should return zero)

Returns

array of floats between 0 and 1 inclusive rescaled so that `vals` equal to `max_val` equal 0 and those equal to `min_val` equal 1

Examples

rescale airmasses to between 0 and 1, with the best (1) and worst (2.25). All values outside the range should return 0. >>> from astroplan.constraints import min_best_rescale >>> import numpy as np >>> airmasses = np.array([1, 1.5, 2, 3, 0]) >>> min_best_rescale(airmasses, 1, 2.25, less_than_min = 0) # doctest: +FLOAT_CMP array([1. , 0.6, 0.2, 0. , 0.])

months_observable

astroplan.**months_observable**(*constraints*, *observer*, *targets*, *time_range=<Time object: scale='utc' format='iso' value=['2024-01-01 00:00:00.000' '2024-12-31 00:00:00.000']>*, *time_grid_resolution=<Quantity 0.5 h>*)

Determines which month the specified `targets` are observable for a specific `observer`, given the supplied `constraints`.

Parameters

constraints

[list or `Constraint`] Observational constraint(s)

observer

[`Observer`] The observer who has constraints `constraints`

targets

[{list, `SkyCoord`, `FixedTarget`}] Target or list of targets

time_range

[`Time` (optional)] Lower and upper bounds on time sequence If `time_range` is not specified, defaults to current year (localtime)

time_grid_resolution

[`Quantity` (optional)] If `time_range` is specified, determine whether constraints are met between test times in `time_range` by checking constraint at linearly-spaced times separated by `time_resolution`. Default is 0.5 hours.

Returns

observable_months

[list] List of sets of unique integers representing each month that a target is observable, one set per target. These integers are 1-based so that January maps to 1, February maps to 2, etc.

moon_illumination

astroplan.**moon_illumination**(*time*, *ephemeris=None*)

Calculate fraction of the moon illuminated.

Parameters

time

[`Time`] Time of observation

ephemeris

[str, optional] Ephemeris to use. If not given, use the one set with `solar_system_ephemeris` (which is set to 'builtin' by default).

Returns

k

[float] Fraction of moon illuminated

moon_phase_angle

`astroplan.`**`moon_phase_angle`**(*time*, *ephemeris=None*)

Calculate lunar orbital phase in radians.

Parameters

time

[`Time`] Time of observation

ephemeris

[str, optional] Ephemeris to use. If not given, use the one set with `solar_system_ephemeris` (which is set to 'builtin' by default).

Returns

i

[`Quantity`] Phase angle of the moon [radians]

observability_table

`astroplan.`**`observability_table`**(*constraints*, *observer*, *targets*, *times=None*, *time_range=None*, *time_grid_resolution=<Quantity 0.5 h>*)

Creates a table with information about observability for all the `targets` over the requested `time_range`, given the constraints in `constraints_list` for `observer`.

Parameters

constraints

[list or `Constraint`] Observational constraint(s)

observer

[Observer] The observer who has constraints `constraints`

targets

[{list, SkyCoord, FixedTarget}] Target or list of targets

times

[Time (optional)] Array of times on which to test the constraint

time_range

[Time (optional)] Lower and upper bounds on time sequence, with spacing `time_resolution`. This will be passed as the first argument into `time_grid_from_range`. If a single (scalar) time, the table will be for a 24 hour period centered on that time.

time_grid_resolution

[Quantity (optional)] If `time_range` is specified, determine whether constraints are met between test times in `time_range` by checking constraint at linearly-spaced times separated by `time_resolution`. Default is 0.5 hours.

Returns

observability_table

[Table] A Table containing the observability information for each of the `targets`. The table contains four columns with information about the target and it's observability: `'target name'`, `'ever observable'`, `'always observable'`, and `'fraction of time observable'`. The column `'time observable'` will also be present if the `time_range` is given as a scalar. It also contains metadata entries `'times'` (with an array of all the times), `'observer'` (the `Observer` object), and `'constraints'` (containing the supplied `constraints`).

stride_array

astroplan.**stride_array**(*arr*, *window_width*)

Computes all possible sequential subarrays of arr with length = window_width

Parameters

arr

[array-like (length = n)] Linearly-spaced sequence

window_width

[int] Number of elements in each new sub-array

Returns

strided_arr

[array (shape = (n-window_width, window_width))] Linearly-spaced sequence of times

time_grid_from_range

astroplan.**time_grid_from_range**(*time_range*, *time_resolution*=<*Quantity 0.5 h*>)

Get linearly-spaced sequence of times.

Parameters

time_range

[Time (length = 2)] Lower and upper bounds on time sequence.

time_resolution

[quantity (optional)] Time-grid spacing

Returns

times

[Time] Linearly-spaced sequence of times

5.1.2 Classes

AirmassConstraint([max, min, boolean_constraint])	Constrain the airmass of a target.
AltitudeConstraint([min, max, ...])	Constrain the altitude of the target.
AstroplanWarning	Superclass for warnings used by astroplan
AtNightConstraint([max_solar_altitude, ...])	Constrain the Sun to be below horizon.
Constraint()	Abstract class for objects defining observational constraints.
EclipsingSystem(primary_eclipse_time, ...[, ...])	Define parameters for an eclipsing system; useful for an eclipsing binary or transiting exoplanet.
FixedTarget(coord[, name])	Coordinates and metadata for an object that is "fixed" with respect to the celestial sphere.
GalacticLatitudeConstraint([min, max])	Constrain the distance between the Galactic plane and some targets.
LocalTimeConstraint([min, max])	Constrain the observable hours.
MissingConstraintWarning	Triggered when a constraint is expected but not supplied
MoonIlluminationConstraint([min, max, ephemeris])	Constrain the fractional illumination of the Earth's moon.

continues on next page

Table 1 – continued from previous page

MoonSeparationConstraint([min, max, ephemeris])	Constrain the distance between the Earth's moon and some targets.
NonFixedTarget([name, ra, dec, marker])	Placeholder for future function.
Observer([location, timezone, name, ...])	A container class for information about an observer's location and environment.
ObservingBlock(target, duration, priority[, ...])	An observation to be scheduled, consisting of a target and associated constraints on observations.
OldEarthOrientationDataWarning	Using old Earth rotation data from IERS
PeriodicEvent(epoch, period[, duration, name])	A periodic event defined by an epoch and period.
PhaseConstraint(periodic_event[, min, max])	Constrain observations to times in some range of phases for a periodic event (e.g.~transiting exoplanets, eclipsing binaries).
PlotBelowHorizonWarning	Warning for when something is hidden on a plot because it's below the horizon
PlotWarning	Warnings dealing with the plotting aspects of astroplan
PrimaryEclipseConstraint(eclipsing_system)	Constrain observations to times during primary eclipse.
PriorityScheduler(*args, **kwargs)	A scheduler that optimizes a prioritized list.
Schedule(start_time, end_time[, constraints])	An object that represents a schedule, consisting of a list of Slot objects.
Scheduler(constraints, observer[, ...])	Schedule a set of ObservingBlock objects
Scorer(blocks, observer, schedule[, ...])	Returns scores and score arrays from the evaluation of constraints on observing blocks
SecondaryEclipseConstraint(eclipsing_system)	Constrain observations to times during secondary eclipse.
SequentialScheduler(*args, **kwargs)	A scheduler that does "stupid simple sequential scheduling".
Slot(start_time, end_time)	A time slot consisting of a start and end time
SunSeparationConstraint([min, max])	Constrain the distance between the Sun and some targets.
Target([name, ra, dec, marker])	Abstract base class for target objects.
TargetAlwaysUpWarning	Target is circumpolar
TargetNeverUpWarning	Target never rises above horizon
TimeConstraint([min, max])	Constrain the observing time to be within certain time limits.
TransitionBlock(components[, start_time])	Parameterizes the "dead time", e.g. between observations, while the telescope is slewing, instrument is reconfiguring, etc.
Transitioner([slew_rate, ...])	A class that defines how to compute transition times from one block to another.

AirmassConstraint

class astroplan.**AirmassConstraint**(*max=None*, *min=1*, *boolean_constraint=True*)

Bases: AltitudeConstraint

Constrain the airmass of a target.

In the current implementation the airmass is approximated by the secant of the zenith angle.

Note: The max and min arguments appear in the order (max, min) in this initializer to support the common case for users who care about the upper limit on the airmass (max) and not the lower limit.

Parameters

max

[float or `None`] Maximum airmass of the target. `None` indicates no limit.

min

[float or `None`] Minimum airmass of the target. `None` indicates no limit.

boolean_contstraint

[bool]

Examples

To create a constraint that requires the airmass be "better than 2", i.e. at a higher altitude than airmass=2:

```
AirmassConstraint(2)
```

Methods Summary

`compute_constraint`(times, observer, targets)	Actually do the real work of computing the constraint.

Methods Documentation

compute_constraint(*times*, *observer*, *targets*)

Actually do the real work of computing the constraint. Subclasses override this.

Parameters

times

[`Time`] The times to compute the constraint

observer

[`Observer`] the observaton location from which to apply the constraints

targets

[sequence of `Target`] The targets on which to apply the constraints.

Returns

constraint_result

[2D array of float or bool] The constraints, with targets along the first index and times along the second.

AltitudeConstraint

class astroplan.**AltitudeConstraint**(*min=None*, *max=None*, *boolean_constraint=True*)

Bases: Constraint

Constrain the altitude of the target.

Note: This can misbehave if you try to constrain negative altitudes, as the AltAz frame tends to mishandle negative

---

Parameters

min

[Quantity or None] Minimum altitude of the target (inclusive). None indicates no limit.

max

[Quantity or None] Maximum altitude of the target (inclusive). None indicates no limit.

boolean_constraint

[bool] If True, the constraint is treated as a boolean (True for within the limits and False for outside). If False, the constraint returns a float on [0, 1], where 0 is the min altitude and 1 is the max.

Methods Summary

compute_constraint(times, observer, targets)	Actually do the real work of computing the constraint.

Methods Documentation

compute_constraint(*times*, *observer*, *targets*)

Actually do the real work of computing the constraint. Subclasses override this.

Parameters

times

[Time] The times to compute the constraint

observer

[Observer] the observaton location from which to apply the constraints

targets

[sequence of Target] The targets on which to apply the constraints.

Returns

constraint_result

[2D array of float or bool] The constraints, with targets along the first index and times along the second.

AstroplanWarning

exception astroplan.**AstroplanWarning**

Superclass for warnings used by astroplan

AtNightConstraint

class astroplan.**AtNightConstraint**(*max_solar_altitude=<Quantity 0. deg>*, *force_pressure_zero=True*)

Bases: Constraint

Constrain the Sun to be below horizon.

Parameters

max_solar_altitude

[Quantity] The altitude of the sun below which it is considered to be "night" (inclusive).

force_pressure_zero

[bool (optional)] Force the pressure to zero for solar altitude calculations. This avoids errors in the altitude of the Sun that can occur when the Sun is below the horizon and the corrections for atmospheric refraction return nonsense values.

Methods Summary

compute_constraint(times, observer, targets)	Actually do the real work of computing the constraint.
twilight_astronomical(**kwargs)	Consider nighttime as time between astronomical twilights (-18 degrees).
twilight_civil(**kwargs)	Consider nighttime as time between civil twilights (-6 degrees).
twilight_nautical(**kwargs)	Consider nighttime as time between nautical twilights (-12 degrees).

Methods Documentation

compute_constraint(*times*, *observer*, *targets*)

Actually do the real work of computing the constraint. Subclasses override this.

Parameters

times

[Time] The times to compute the constraint

observer

[Observer] the observaton location from which to apply the constraints

targets

[sequence of Target] The targets on which to apply the constraints.

Returns

constraint_result

[2D array of float or bool] The constraints, with targets along the first index and times along the second.

classmethod twilight_astronomical(***kwargs*)

Consider nighttime as time between astronomical twilights (-18 degrees).

classmethod twilight_civil(***kwargs*)

Consider nighttime as time between civil twilights (-6 degrees).

classmethod twilight_nautical(***kwargs*)

Consider nighttime as time between nautical twilights (-12 degrees).

Constraint

class astroplan.**Constraint**

Bases: object

Abstract class for objects defining observational constraints.

Methods Summary

`__call__`(observer, targets[, times, ...])	Compute the constraint for this class
`compute_constraint`(times, observer, targets)	Actually do the real work of computing the constraint.

Methods Documentation

__call__(*observer*, *targets*, *times=None*, *time_range=None*, *time_grid_resolution=<Quantity 0.5 h>*, *grid_times_targets=False*)

Compute the constraint for this class

Parameters

observer

[`Observer`] the observation location from which to apply the constraints

targets

[sequence of `Target`] The targets on which to apply the constraints.

times

[`Time`] The times to compute the constraint. WHAT HAPPENS WHEN BOTH TIMES AND TIME_RANGE ARE SET?

time_range

[`Time` (length = 2)] Lower and upper bounds on time sequence.

time_grid_resolution

[`quantity`] Time-grid spacing

grid_times_targets

[bool] if True, grids the constraint result with targets along the first index and times along the second. Otherwise, we rely on broadcasting the shapes together using standard numpy rules.

Returns

———-

constraint_result

[1D or 2D array of float or bool] The constraints. If 2D with targets along the first index and times along the second.

abstract compute_constraint(*times*, *observer*, *targets*)

Actually do the real work of computing the constraint. Subclasses override this.

Parameters

times

[Time] The times to compute the constraint

observer

[Observer] the observaton location from which to apply the constraints

targets

[sequence of Target] The targets on which to apply the constraints.

Returns

constraint_result

[2D array of float or bool] The constraints, with targets along the first index and times along the second.

EclipsingSystem

class astroplan.**EclipsingSystem**(*primary_eclipse_time*, *orbital_period*, *duration=None*, *name=None*, *eccentricity=None*, *argument_of_periapsis=None*)

Bases: PeriodicEvent

Define parameters for an eclipsing system; useful for an eclipsing binary or transiting exoplanet.

Warning: There are currently two major caveats in the implementation of EclipsingSystem. The secondary eclipse time approximation is only accurate when the orbital eccentricity is small, and the eclipse times are computed without any barycentric corrections. The current implementation should only be used forapproximate mid-eclipse times for low eccentricity orbits, with event durations longer than the barycentric correction error (<=16 minutes).

Parameters

primary_eclipse_time

[Time] Time of primary eclipse

orbital_period

[Quantity] Orbital period of eclipsing system

duration

[Quantity (optional)] Duration of eclipse

name

[str (optional)] Name of target/event

eccentricity

[float (optional)] Orbital eccentricity. Default is `None`, which assumes circular orbit (e=0).

argument_of_periapsis

[float (optional)] Argument of periapsis for the eclipsing system, in radians. Default is `None`, which assumes pi/2.

Methods Summary

`in_primary_eclipse`(time)	Returns `True` when `time` is during a primary eclipse.
`in_secondary_eclipse`(time)	Returns `True` when `time` is during a secondary eclipse
`next_primary_eclipse_time`(time[, n_eclipses])	Time of the next primary eclipse after `time`.
`next_primary_ingress_egress_time`(time[, ...])	Calculate the times of ingress and egress for the next `n_eclipses` primary eclipses after `time`
`next_secondary_eclipse_time`(time[, n_eclipses])	Time of the next secondary eclipse after `time`.
`next_secondary_ingress_egress_time`(time[, ...])	Calculate the times of ingress and egress for the next `n_eclipses` secondary eclipses after `time`
`out_of_eclipse`(time)	Returns `True` when `time` is not during primary or secondary eclipse.

Methods Documentation

`in_primary_eclipse`(*time*)

Returns `True` when `time` is during a primary eclipse.

> **Warning:** Barycentric offsets are ignored in the current implementation.

Parameters

time

[`Time`] Time to evaluate

Returns

in_eclipse

[`ndarray` or bool] `True` if `time` is during primary eclipse

`in_secondary_eclipse`(*time*)

Returns `True` when `time` is during a secondary eclipse

If the eccentricity of the eclipsing system is non-zero, then we compute the secondary eclipse time approximated to first order in eccentricity, as described in Winn (2010) Equation 33 [1]:

The time between the primary eclipse and secondary eclipse δt_c is given by $\delta t_c \approx 0.5 \left(\frac{4}{\pi} e \cos \omega\right)$, where e is the orbital eccentricity and ω is the angle of periapsis.

Warning: This approximation for the secondary eclipse time is only accurate when the orbital eccentricity is small; and barycentric offsets are ignored in the current implementation.

Parameters

time

[Time] Time to evaluate

Returns

in_eclipse

[ndarray or bool] True if time is during secondary eclipse

References

[1]

next_primary_eclipse_time(*time*, *n_eclipses=1*)

Time of the next primary eclipse after time.

Warning: Barycentric offsets are ignored in the current implementation.

Parameters

time

[Time] Find the next primary eclipse after time

n_eclipses

[int (optional)] Return the times of eclipse for the next n_eclipses after time. Default is 1.

Returns

primary_eclipses

[Time] Times of the next n_eclipses primary eclipses after time

next_primary_ingress_egress_time(*time*, *n_eclipses=1*)

Calculate the times of ingress and egress for the next `n_eclipses` primary eclipses after `time`

> **Warning:** Barycentric offsets are ignored in the current implementation.

Parameters

time

[`Time`] Find the next primary ingress and egress after `time`

n_eclipses

[int (optional)] Return the times of eclipse for the next `n_eclipses` after `time`. Default is 1.

Returns

primary_eclipses

[`Time` of shape (`n_eclipses`, 2)] Times of ingress and egress for the next `n_eclipses` primary eclipses after `time`

next_secondary_eclipse_time(*time*, *n_eclipses=1*)

Time of the next secondary eclipse after `time`.

> **Warning:** Barycentric offsets are ignored in the current implementation.

Parameters

time

[`Time`] Find the next secondary eclipse after `time`

n_eclipses

[int (optional)] Return the times of eclipse for the next `n_eclipses` after `time`. Default is 1.

Returns

secondary_eclipses

[`Time`] Times of the next `n_eclipses` secondary eclipses after `time`

next_secondary_ingress_egress_time(*time*, *n_eclipses=1*)

Calculate the times of ingress and egress for the next `n_eclipses` secondary eclipses after `time`

Warning: Barycentric offsets are ignored in the current implementation.

Parameters

time

[`Time`] Find the next secondary ingress and egress after `time`

n_eclipses

[int (optional)] Return the times of eclipse for the next `n_eclipses` after `time`. Default is 1.

Returns

secondary_eclipses

[`Time` of shape (`n_eclipses`, 2)] Times of ingress and egress for the next `n_eclipses` secondary eclipses after `time`.

out_of_eclipse(*time*)

Returns `True` when `time` is not during primary or secondary eclipse.

Warning: Barycentric offsets are ignored in the current implementation.

Parameters

time

[`Time`] Time to evaluate

Returns

in_eclipse

[`ndarray` or bool] `True` if `time` is not during primary or secondary eclipse

FixedTarget

class astroplan.**FixedTarget**(*coord*, *name=None*, ***kwargs*)

Bases: Target

Coordinates and metadata for an object that is "fixed" with respect to the celestial sphere.

Examples

Create a FixedTarget object for Sirius:

```
>>> from astroplan import FixedTarget
>>> from astropy.coordinates import SkyCoord
>>> import astropy.units as u
>>> sirius_coord = SkyCoord(ra=101.28715533*u.deg, dec=16.71611586*u.deg)
>>> sirius = FixedTarget(coord=sirius_coord, name="Sirius")
```

Create an equivalent FixedTarget object for Sirius by querying for the coordinates of Sirius by name:

```
>>> from astroplan import FixedTarget
>>> sirius = FixedTarget.from_name("Sirius")
```

Parameters

coord

[SkyCoord] Coordinate of the target

name

[str (optional)] Name of the target, used for plotting and representing the target as a string

Methods Summary

from_name(query_name[, name])	Initialize a FixedTarget by querying for a name from the CDS name resolver, using the machinery in from_name.

Methods Documentation

classmethod from_name(*query_name*, *name=None*, ***kwargs*)

Initialize a FixedTarget by querying for a name from the CDS name resolver, using the machinery in from_name.

This

Parameters

query_name

[str] Name of the target used to query for coordinates.

name

[string or `None`] Name of the target to use within astroplan. If `None`, query_name is used as `name`.

Examples

```
>>> from astroplan import FixedTarget
>>> sirius = FixedTarget.from_name("Sirius")
>>> sirius.coord
<SkyCoord (ICRS): (ra, dec) in deg
    ( 101.28715533, -16.71611586)>
```

GalacticLatitudeConstraint

class astroplan.**GalacticLatitudeConstraint**(*min=None*, *max=None*)

Bases: `Constraint`

Constrain the distance between the Galactic plane and some targets.

Parameters

min

[`Quantity` or `None` (optional)] Minimum acceptable Galactic latitude of target (inclusive). `None` indicates no limit.

max

[`Quantity` or `None` (optional)] Minimum acceptable Galactic latitude of target (inclusive). `None` indicates no limit.

Methods Summary

`compute_constraint`(times, observer, targets)	Actually do the real work of computing the constraint.

Methods Documentation

compute_constraint(*times*, *observer*, *targets*)

Actually do the real work of computing the constraint. Subclasses override this.

Parameters

times

[Time] The times to compute the constraint

observer

[Observer] the observaton location from which to apply the constraints

targets

[sequence of Target] The targets on which to apply the constraints.

Returns

constraint_result

[2D array of float or bool] The constraints, with targets along the first index and times along the second.

LocalTimeConstraint

class astroplan.**LocalTimeConstraint**(*min=None*, *max=None*)

Bases: Constraint

Constrain the observable hours.

Parameters

min

[time] Earliest local time (inclusive). None indicates no limit.

max

[time] Latest local time (inclusive). None indicates no limit.

Examples

Constrain the observations to targets that are observable between 23:50 and 04:08 local time:

```
>>> from astroplan import Observer
>>> from astroplan.constraints import LocalTimeConstraint
>>> import datetime as dt
>>> subaru = Observer.at_site("Subaru", timezone="US/Hawaii")
>>> # bound times between 23:50 and 04:08 local Hawaiian time
>>> constraint = LocalTimeConstraint(min=dt.time(23,50), max=dt.time(4,8))
```

Methods Summary

compute_constraint(times, observer, targets)	Actually do the real work of computing the constraint.

Methods Documentation

compute_constraint(*times*, *observer*, *targets*)

Actually do the real work of computing the constraint. Subclasses override this.

Parameters

times

[Time] The times to compute the constraint

observer

[Observer] the observaton location from which to apply the constraints

targets

[sequence of Target] The targets on which to apply the constraints.

Returns

constraint_result

[2D array of float or bool] The constraints, with targets along the first index and times along the second.

MissingConstraintWarning

exception astroplan.**MissingConstraintWarning**

Triggered when a constraint is expected but not supplied

MoonIlluminationConstraint

class astroplan.**MoonIlluminationConstraint**(*min=None*, *max=None*, *ephemeris=None*)

Bases: Constraint

Constrain the fractional illumination of the Earth's moon.

Constraint is also satisfied if the Moon has set.

Parameters

min

[float or None (optional)] Minimum acceptable fractional illumination (inclusive). None indicates no limit.

max

[float or None (optional)] Maximum acceptable fractional illumination (inclusive). None indicates no limit.

ephemeris

[str, optional] Ephemeris to use. If not given, use the one set with solar_system_ephemeris (which is set to 'builtin' by default).

Methods Summary

bright([min, max])	initialize a MoonIlluminationConstraint with defaults of a minimum of 0.65 and no maximum
compute_constraint(times, observer, targets)	Actually do the real work of computing the constraint.
dark([min, max])	initialize a MoonIlluminationConstraint with defaults of no minimum and a maximum of 0.25
grey([min, max])	initialize a MoonIlluminationConstraint with defaults of a minimum of 0.25 and a maximum of 0.65

Methods Documentation

classmethod bright(*min=0.65*, *max=None*, ***kwargs*)

initialize a MoonIlluminationConstraint with defaults of a minimum of 0.65 and no maximum

Parameters

min

[float or None (optional)] Minimum acceptable fractional illumination (inclusive). None indicates no limit.

max

[float or None (optional)] Maximum acceptable fractional illumination (inclusive). None indicates no limit.

compute_constraint(*times*, *observer*, *targets*)

Actually do the real work of computing the constraint. Subclasses override this.

Parameters

times

[Time] The times to compute the constraint

observer

[Observer] the observaton location from which to apply the constraints

targets

[sequence of Target] The targets on which to apply the constraints.

Returns

constraint_result

[2D array of float or bool] The constraints, with targets along the first index and times along the second.

classmethod dark(*min=None*, *max=0.25*, ***kwargs*)

initialize a MoonIlluminationConstraint with defaults of no minimum and a maximum of 0.25

Parameters

min

[float or None (optional)] Minimum acceptable fractional illumination (inclusive). None indicates no limit.

max

[float or None (optional)] Maximum acceptable fractional illumination (inclusive). None indicates no limit.

classmethod grey(*min=0.25*, *max=0.65*, ***kwargs*)

initialize a MoonIlluminationConstraint with defaults of a minimum of 0.25 and a maximum of 0.65

Parameters

min

[float or None (optional)] Minimum acceptable fractional illumination (inclusive). None indicates no limit.

max

[float or None (optional)] Maximum acceptable fractional illumination (inclusive). None indicates no limit.

MoonSeparationConstraint

class astroplan.**MoonSeparationConstraint**(*min=None*, *max=None*, *ephemeris=None*)

Bases: Constraint

Constrain the distance between the Earth's moon and some targets.

Parameters

min

[Quantity or None (optional)] Minimum acceptable separation between moon and target (inclusive). None indicates no limit.

max

[Quantity or None (optional)] Maximum acceptable separation between moon and target (inclusive). None indicates no limit.

ephemeris

[str, optional] Ephemeris to use. If not given, use the one set with `astropy.coordinates.solar_system_ephemeris.set` (which is set to 'builtin' by default).

Methods Summary

compute_constraint(times, observer, targets)	Actually do the real work of computing the constraint.

Methods Documentation

compute_constraint(*times*, *observer*, *targets*)

Actually do the real work of computing the constraint. Subclasses override this.

Parameters

times

[Time] The times to compute the constraint

observer

[Observer] the observaton location from which to apply the constraints

targets

[sequence of Target] The targets on which to apply the constraints.

Returns

constraint_result

[2D array of float or bool] The constraints, with targets along the first index and times along the second.

NonFixedTarget

class astroplan.**NonFixedTarget**(*name=None*, *ra=None*, *dec=None*, *marker=None*)

Bases: Target

Placeholder for future function.

Defines a single observation target.

Parameters

name

[str, optional]

ra

[WHAT TYPE IS ra ?]

dec

[WHAT TYPE IS dec ?]

marker

[str, optional] User-defined markers to differentiate between different types of targets (e.g., guides, high-priority, etc.).

Observer

class astroplan.**Observer**(*location=None*, *timezone='UTC'*, *name=None*, *latitude=None*, *longitude=None*, *elevation=<Quantity 0. m>*, *pressure=None*, *relative_humidity=None*, *temperature=None*, *description=None*)

Bases: `object`

A container class for information about an observer's location and environment.

Examples

We can create an observer at Subaru Observatory in Hawaii two ways. First, locations for some observatories are stored in astroplan, and these can be accessed by name, like so:

```
>>> from astroplan import Observer
>>> subaru = Observer.at_site("Subaru", timezone="US/Hawaii")
```

To find out which observatories can be accessed by name, check out `get_site_names`.

Next, you can initialize an observer by specifying the location with `EarthLocation`:

```
>>> from astropy.coordinates import EarthLocation
>>> import astropy.units as u
>>> location = EarthLocation.from_geodetic(-155.4761*u.deg, 19.825*u.deg,
...                                        4139*u.m)
>>> subaru = Observer(location=location, name="Subaru", timezone="US/Hawaii")
```

You can also create an observer without an `EarthLocation`:

```
>>> from astroplan import Observer
>>> import astropy.units as u
>>> subaru = Observer(longitude=-155.4761*u.deg, latitude=19.825*u.deg,
...                   elevation=0*u.m, name="Subaru", timezone="US/Hawaii")
```

Parameters

location

[`EarthLocation`] The location (latitude, longitude, elevation) of the observatory.

timezone

[str or `datetime.tzinfo` (optional)] The local timezone to assume. If a string, it will be passed through `pytz.timezone()` to produce the timezone object.

name

[str] A short name for the telescope, observatory or location.

latitude

[float, str, `Quantity` (optional)] The latitude of the observing location. Should be valid input for initializing a `Latitude` object.

longitude

[float, str, `Quantity` (optional)] The longitude of the observing location. Should be valid input for initializing a `Longitude` object.

elevation

[`Quantity` (optional), default = 0 meters] The elevation of the observing location, with respect to sea level. Defaults to sea level.

pressure

[`Quantity` (optional)] The ambient pressure. Defaults to zero (i.e. no atmosphere).

relative_humidity

[float (optional)] The ambient relative humidity.

temperature

[`Quantity` (optional)] The ambient temperature.

description

[str (optional)] A short description of the telescope, observatory or observing location.

Attributes Summary

`elevation`	The elevation of the observing location with respect to sea level.
`latitude`	The latitude of the observing location, derived from the location.
`longitude`	The longitude of the observing location, derived from the location.

Methods Summary

altaz(time[, target, obswl, grid_times_targets])	Get an AltAz frame or coordinate.
astropy_time_to_datetime(astropy_time)	Convert the Time object astropy_time to a localized datetime object.
at_site(site_name, **kwargs)	Initialize an Observer object with a site name.
datetime_to_astropy_time(date_time)	Convert the datetime object date_time to a Time object.
is_night(time[, horizon, obswl])	Is the Sun below horizon at time?
local_sidereal_time(time[, kind, model])	Convert time to local sidereal time for observer.
midnight(time[, which, n_grid_points])	Time at solar midnight.
moon_altaz(time[, ephemeris])	Returns the position of the moon in alt/az.
moon_illumination(time)	Calculate the illuminated fraction of the moon.
moon_phase([time])	Calculate lunar orbital phase.
moon_rise_time(time[, which, horizon, ...])	Returns the local moon rise time.
moon_set_time(time[, which, horizon, ...])	Returns the local moon set time.
noon(time[, which, n_grid_points])	Time at solar noon.
parallactic_angle(time, target[, ...])	Calculate the parallactic angle.
sun_altaz(time)	Returns the position of the Sun in alt/az.
sun_rise_time(time[, which, horizon, ...])	Time of sunrise.
sun_set_time(time[, which, horizon, ...])	Time of sunset.
target_hour_angle(time, target[, ...])	Calculate the local hour angle of target at time.
target_is_up(time, target[, horizon, ...])	Is target above horizon at this time?
target_meridian_antitransit_time(time, target)	Calculate time at the antitransit of the meridian.
target_meridian_transit_time(time, target[, ...])	Calculate time at the transit of the meridian.
target_rise_time(time, target[, which, ...])	Calculate rise time.
target_set_time(time, target[, which, ...])	Calculate set time.
tonight([time, horizon, obswl])	Return a time range corresponding to the nearest night
twilight_evening_astronomical(time[, which, ...])	Time at evening astronomical (-18 degree) twilight.
twilight_evening_civil(time[, which, ...])	Time at evening civil (-6 degree) twilight.
twilight_evening_nautical(time[, which, ...])	Time at evening nautical (-12 degree) twilight.
twilight_morning_astronomical(time[, which, ...])	Time at morning astronomical (-18 degree) twilight.
twilight_morning_civil(time[, which, ...])	Time at morning civil (-6 degree) twilight.
twilight_morning_nautical(time[, which, ...])	Time at morning nautical (-12 degree) twilight.

Attributes Documentation

elevation

The elevation of the observing location with respect to sea level.

latitude

The latitude of the observing location, derived from the location.

longitude

The longitude of the observing location, derived from the location.

Methods Documentation

altaz(*time*, *target=None*, *obswl=None*, *grid_times_targets=False*)

Get an `AltAz` frame or coordinate.

If `target` is None, generates an altitude/azimuth frame. Otherwise, calculates the transformation to that frame for the requested `target`.

Parameters

time

[`Time` or other (see below)] The time at which the observation is taking place. Will be used as the `obstime` attribute in the resulting frame or coordinate. This will be passed in as the first argument to the `Time` initializer, so it can be anything that `Time` will accept (including a `Time` object)

target

[`FixedTarget`, `SkyCoord`, or list (optional)] Celestial object(s) of interest. If `target` is `None`, returns the `AltAz` frame without coordinates.

obswl

[`Quantity` (optional)] Wavelength of the observation used in the calculation.

grid_times_targets: bool (optional)

If True, the target object will have extra dimensions packed onto the end, so that calculations with M targets and N times will return an (M, N) shaped result. Otherwise, we rely on broadcasting the shapes together using standard numpy rules. Useful for grid searches for rise/set times etc.

Returns

`AltAz`

If `target` is `None`, returns `AltAz` frame. If `target` is not `None`, returns the `target` transformed to the `AltAz` frame.

Examples

Create an instance of the `AltAz` frame for an observer at Apache Point Observatory at a particular time:

```
>>> from astroplan import Observer
>>> from astropy.time import Time
>>> from astropy.coordinates import SkyCoord
>>> apo = Observer.at_site("APO")
>>> time = Time('2001-02-03 04:05:06')
>>> target = SkyCoord(0*u.deg, 0*u.deg)
>>> altaz_frame = apo.altaz(time)
```

Now transform the target's coordinates to the alt/az frame:

```
>>> target_altaz = target.transform_to(altaz_frame)
```

Alternatively, construct an alt/az frame and transform the target to that frame all in one step:

```
>>> target_altaz = apo.altaz(time, target)
```

astropy_time_to_datetime(*astropy_time*)

Convert the `Time` object `astropy_time` to a localized `datetime` object.

Timezones localized with pytz.

Parameters

astropy_time

[`Time`] Scalar or list-like.

Returns

datetime

Localized datetime, where the timezone of the datetime is set by the `timezone` keyword argument of the `Observer` constructor.

Examples

Convert an astropy time to a localized `datetime`:

```
>>> from astroplan import Observer
>>> from astropy.time import Time
>>> subaru = Observer.at_site("Subaru", timezone="US/Hawaii")
>>> astropy_time = Time('1999-12-31 06:00:00')
>>> print(subaru.astropy_time_to_datetime(astropy_time))
1999-12-30 20:00:00-10:00
```

classmethod at_site(*site_name*, ***kwargs*)

Initialize an `Observer` object with a site name.

Extra keyword arguments are passed to the `Observer` constructor (see `Observer` for available keyword arguments).

Parameters

site_name

[str] Observatory name, must be resolvable with `get_site_names`.

Returns

`Observer`

Observer object.

Examples

Initialize an observer at Kitt Peak National Observatory:

```
>>> from astroplan import Observer
>>> import astropy.units as u
>>> kpno_generic = Observer.at_site('kpno')
>>> kpno_today = Observer.at_site('kpno', pressure=1*u.bar, temperature=0*u.deg_C)
```

datetime_to_astropy_time(*date_time*)

Convert the `datetime` object `date_time` to a `Time` object.

Timezones localized with pytz. If the `date_time` is naive, the implied timezone is the `timezone` structure of `self`.

Parameters

date_time

[`datetime` or list-like]

Returns

`Time`

Astropy time object (no timezone information preserved).

Examples

Convert a localized `datetime` to a `Time` object. Non-localized datetimes are assumed to be UTC. <Time object: scale='utc' format='datetime' value=1999-12-31 06:00:00>

```
>>> from astroplan import Observer
>>> import datetime
>>> import pytz
>>> subaru = Observer.at_site("Subaru", timezone="US/Hawaii")
>>> hi_date_time = datetime.datetime(2005, 6, 21, 20, 0, 0, 0)
>>> subaru.datetime_to_astropy_time(hi_date_time)
<Time object: scale='utc' format='datetime' value=2005-06-22 06:00:00>
>>> utc_date_time = datetime.datetime(2005, 6, 22, 6, 0, 0, 0,
...                                   tzinfo=pytz.timezone("UTC"))
```

(continues on next page)

(continued from previous page)

```
>>> subaru.datetime_to_astropy_time(utc_date_time)
<Time object: scale='utc' format='datetime' value=2005-06-22 06:00:00>
```

is_night(*time*, *horizon=<Quantity 0. deg>*, *obswl=None*)

Is the Sun below `horizon` at `time`?

Parameters

time

[`Time` or other (see below)] Time of observation. This will be passed in as the first argument to the `Time` initializer, so it can be anything that `Time` will accept (including a `Time` object)

horizon

[`Quantity` (optional), default = zero degrees] Degrees above/below actual horizon to use for calculating day/night (i.e., -6 deg horizon = civil twilight, etc.)

obswl

[`Quantity` (optional)] Wavelength of the observation used in the calculation

Returns

sun_below_horizon

[bool or np.ndarray(bool)] `True` if sun is below `horizon` at `time`, else `False`.

Examples

Is it "nighttime" (i.e. is the Sun below `horizon`) at Apache Point Observatory at 2015-08-29 18:35 UTC?

```
>>> from astroplan import Observer
>>> from astropy.time import Time
>>> apo = Observer.at_site("APO")
>>> time = Time("2015-08-29 18:35")
>>> apo.is_night(time)
False
```

local_sidereal_time(*time*, *kind='apparent'*, *model=None*)

Convert `time` to local sidereal time for observer.

This is a thin wrapper around the `sidereal_time` method.

Parameters

time

[Time or other (see below)] Time of observation. This will be passed in as the first argument to the Time initializer, so it can be anything that Time will accept (including a Time object)

kind

[{'mean', 'apparent'} (optional)] Passed to the kind argument of sidereal_time

model

[str or None; optional] The precession/nutation model to assume - see sidereal_time for more details.

Returns

Longitude

Local sidereal time.

midnight(*time*, *which='nearest'*, *n_grid_points=150*)

Time at solar midnight.

Parameters

time

[Time or other (see below)] Time of observation. This will be passed in as the first argument to the Time initializer, so it can be anything that Time will accept (including a Time object)

which

[{'next', 'previous', 'nearest'}] Choose which noon relative to the present time would you like to calculate

n_grid_points

[int (optional)] The number of grid points on which to search for the horizon crossings of the target over a 24 hour period, default is 150 which yields midnight time precisions better than one minute.

Returns

Time

Time at solar midnight

moon_altaz(*time*, *ephemeris=None*)

Returns the position of the moon in alt/az.

Parameters

time

[Time or other (see below)] This will be passed in as the first argument to the Time initializer, so it can be anything that Time will accept (including a Time object).

ephemeris

[str, optional] Ephemeris to use. If not given, use the one set with `astropy.coordinates.solar_system_ephemeris.set` (which is set to 'builtin' by default).

Returns

altaz

[SkyCoord] Position of the moon transformed to altitude and azimuth

Examples

Calculate the altitude and azimuth of the moon at Apache Point Observatory:

```
>>> from astroplan import Observer
>>> from astropy.time import Time
>>> apo = Observer.at_site("APO")
>>> time = Time("2015-08-29 18:35")
>>> altaz_moon = apo.moon_altaz(time)
>>> print("alt: {0.alt}, az: {0.az}".format(altaz_moon))
alt: -63.72706397691421 deg, az: 345.3640380598265 deg
```

moon_illumination(*time*)

Calculate the illuminated fraction of the moon.

Parameters

time

[Time or other (see below)] This will be passed in as the first argument to the Time initializer, so it can be anything that Time will accept (including a Time object).

Returns

float

Fraction of lunar surface illuminated

Examples

How much of the lunar surface is illuminated at 2015-08-29 18:35 UTC, which we happen to know is the time of a full moon?

```
>>> from astroplan import Observer
>>> from astropy.time import Time
>>> apo = Observer.at_site("APO")
>>> time = Time("2015-08-29 18:35")
>>> apo.moon_illumination(time)
array([ 0.99972487])
```

moon_phase(*time=None*)

Calculate lunar orbital phase.

For example, phase=pi is "new", phase=0 is "full".

Parameters

time

[`Time` or other (see below)] This will be passed in as the first argument to the `Time` initializer, so it can be anything that `Time` will accept (including a `Time` object).

Returns

moon_phase_angle

[float] Orbital phase angle of the moon where pi corresponds to new moon, zero corresponds to full moon.

Examples

Calculate the phase of the moon at 2015-08-29 18:35 UTC. Near zero radians corresponds to a nearly full moon.

```
>>> from astroplan import Observer
>>> from astropy.time import Time
>>> apo = Observer.at_site('APO')
>>> time = Time('2015-08-29 18:35')
>>> apo.moon_phase(time)
<Quantity [ 0.03317537] rad>
```

moon_rise_time(*time*, *which='nearest'*, *horizon=<Quantity 0. deg>*, *n_grid_points=150*)

Returns the local moon rise time.

Compute time of the next/previous/nearest moon rise, where moon rise is defined as the time when the moon transitions from altitudes below `horizon` to above `horizon`.

Parameters

time

[Time or other (see below)] Time of observation. This will be passed in as the first argument to the Time initializer, so it can be anything that Time will accept (including a Time object)

which

[{'next', 'previous', 'nearest'}] Choose which moon rise relative to the present time would you like to calculate.

horizon

[Quantity (optional), default = zero degrees] Degrees above/below actual horizon to use for calculating rise/set times (i.e., -6 deg horizon = civil twilight, etc.)

n_grid_points

[int (optional)] The number of grid points on which to search for the horizon crossings of the target over a 24 hour period, default is 150 which yields rise time precisions better than one minute.

moon_set_time(*time*, *which='nearest'*, *horizon=<Quantity 0. deg>*, *n_grid_points=150*)

Returns the local moon set time.

Compute time of the next/previous/nearest moon set, where moon set is defined as the time when the moon transitions from altitudes below horizon to above horizon.

Parameters

time

[Time or other (see below)] Time of observation. This will be passed in as the first argument to the Time initializer, so it can be anything that Time will accept (including a Time object)

which

[{'next', 'previous', 'nearest'}] Choose which moon set relative to the present time would you like to calculate.

horizon

[Quantity (optional), default = zero degrees] Degrees above/below actual horizon to use for calculating set/set times (i.e., -6 deg horizon = civil twilight, etc.)

n_grid_points

[int (optional)] The number of grid points on which to search for the horizon crossings of the target over a 24 hour period, default is 150 which yields set time precisions better than one minute.

noon(*time*, *which='nearest'*, *n_grid_points=150*)

Time at solar noon.

Parameters

time

[Time or other (see below)] Time of observation. This will be passed in as the first argument to the Time initializer, so it can be anything that Time will accept (including a Time object)

which

[{'next', 'previous', 'nearest'}] Choose which noon relative to the present time would you like to calculate

n_grid_points

[int (optional)] The number of grid points on which to search for the horizon crossings of the target over a 24 hour period, default is 150 which yields noon time precisions better than one minute.

Returns

Time

Time at solar noon

parallactic_angle(*time*, *target*, *grid_times_targets=False*, *kind='mean'*, *model=None*)

Calculate the parallactic angle.

Parameters

time

[Time] Observation time.

target

[FixedTarget or SkyCoord or list] Target celestial object(s).

grid_times_targets: bool

If True, the target object will have extra dimensions packed onto the end, so that calculations with M targets and N times will return an (M, N) shaped result. Otherwise, we rely on broadcasting the shapes together using standard numpy rules.

kind

[str] Argument to the sidereal_time function, which may be either 'mean' or 'apparent', i.e., accounting for precession only, or also for nutation.

model

[str or None; optional] Precession (and nutation) model to use in the call to `sidereal_time`, see docs for that method for details.

Returns

`Angle`

Parallactic angle.

Notes

The parallactic angle is the angle between the great circle that intersects a celestial object and the zenith, and the object's hour circle [1].

sun_altaz(*time*)

Returns the position of the Sun in alt/az.

Parameters

time

[`Time` or other (see below)] This will be passed in as the first argument to the `Time` initializer, so it can be anything that `Time` will accept (including a `Time` object).

ephemeris

[str, optional] Ephemeris to use. If not given, use the one set with `astropy.coordinates.solar_system_ephemeris.set` (which is set to 'builtin' by default).

Returns

altaz

[`SkyCoord`] Position of the sun transformed to altitude and azimuth

sun_rise_time(*time*, *which='nearest'*, *horizon=<Quantity 0. deg>*, *n_grid_points=150*)

Time of sunrise.

Compute time of the next/previous/nearest sunrise, where sunrise is defined as when the Sun transitions from altitudes below `horizon` to above `horizon`.

Parameters

time

[`Time` or other (see below)] Time of observation. This will be passed in as the first

argument to the `Time` initializer, so it can be anything that `Time` will accept (including a `Time` object)

which

[{'next', 'previous', 'nearest'}] Choose which sunrise relative to the present `time` would you like to calculate.

horizon

[`Quantity` (optional), default = zero degrees] Degrees above/below actual horizon to use for calculating rise/set times (i.e., -6 deg horizon = civil twilight, etc.)

n_grid_points

[int (optional)] The number of grid points on which to search for the horizon crossings of the target over a 24 hour period, default is 150 which yields rise time precisions better than one minute.

Returns

`Time`

Time of sunrise

Examples

Calculate the time of the previous sunrise at Apache Point Observatory:

```
>>> from astroplan import Observer
>>> from astropy.time import Time
>>> apo = Observer.at_site("APO")
>>> time = Time('2001-02-03 04:05:06')
>>> sun_rise = apo.sun_rise_time(time, which="previous")
>>> print("ISO: {0.iso}, JD: {0.jd}".format(sun_rise))
ISO: 2001-02-02 14:02:50.554, JD: 2451943.08531
```

sun_set_time(*time*, *which='nearest'*, *horizon=<Quantity 0. deg>*, *n_grid_points=150*)

Time of sunset.

Compute time of the next/previous/nearest sunset, where sunset is defined as when the Sun transitions from altitudes below `horizon` to above `horizon`.

Parameters

time

[`Time` or other (see below)] Time of observation. This will be passed in as the first argument to the `Time` initializer, so it can be anything that `Time` will accept (including a `Time` object)

which

[{'next', 'previous', 'nearest'}] Choose which sunset relative to the present `time` would you like to calculate

horizon

[`Quantity` (optional), default = zero degrees] Degrees above/below actual horizon to use for calculating rise/set times (i.e., -6 deg horizon = civil twilight, etc.)

n_grid_points

[int (optional)] The number of grid points on which to search for the horizon crossings of the target over a 24 hour period, default is 150 which yields set time precisions better than one minute.

Returns

`Time`

Time of sunset

Examples

Calculate the time of the next sunset at Apache Point Observatory:

```
>>> from astroplan import Observer
>>> from astropy.time import Time
>>> apo = Observer.at_site("APO")
>>> time = Time('2001-02-03 04:05:06')
>>> sun_set = apo.sun_set_time(time, which="next")
>>> print("ISO: {0.iso}, JD: {0.jd}".format(sun_set))
ISO: 2001-02-04 00:35:42.102, JD: 2451944.52479
```

target_hour_angle(*time*, *target*, *grid_times_targets=False*)

Calculate the local hour angle of `target` at `time`.

Parameters

time

[`Time` or other (see below)] Time of observation. This will be passed in as the first argument to the `Time` initializer, so it can be anything that `Time` will accept (including a `Time` object)

target

[`SkyCoord`, `FixedTarget`, or list] Target celestial object(s)

grid_times_targets: bool

If True, the target object will have extra dimensions packed onto the end, so that calculations with M targets and N times will return an (M, N) shaped result. Otherwise, we rely on broadcasting the shapes together using standard numpy rules.

Returns

hour_angle

[Angle] The hour angle(s) of the target(s) at time

target_is_up(*time*, *target*, *horizon=<Quantity 0. deg>*, *return_altaz=False*, *grid_times_targets=False*)

Is target above horizon at this time?

Parameters

time

[Time or other (see below)] Time of observation. This will be passed in as the first argument to the Time initializer, so it can be anything that Time will accept (including a Time object)

target

[SkyCoord, FixedTarget, or list] Target celestial object(s)

horizon

[Quantity (optional), default = zero degrees] Degrees above/below actual horizon to use for calculating rise/set times (i.e., -6 deg horizon = civil twilight, etc.)

return_altaz

[bool (optional)] Also return the '~astropy.coordinates.AltAz' coordinate.

grid_times_targets: bool

If True, the target object will have extra dimensions packed onto the end, so that calculations with M targets and N times will return an (M, N) shaped result. Otherwise, we rely on broadcasting the shapes together using standard numpy rules.

Returns

observable

[boolean or np.ndarray(bool)] True if target is above horizon at time, else False.

Examples

Are Aldebaran and Vega above the horizon at Apache Point Observatory at 2015-08-29 18:35 UTC?

```
>>> from astroplan import Observer, FixedTarget
>>> from astropy.time import Time
>>> apo = Observer.at_site("APO")
>>> time = Time("2015-08-29 18:35")
>>> aldebaran = FixedTarget.from_name("Aldebaran")
>>> vega = FixedTarget.from_name("Vega")
>>> apo.target_is_up(time, aldebaran)
True
>>> apo.target_is_up(time, [aldebaran, vega])
array([ True, False], dtype=bool)
```

target_meridian_antitransit_time(*time*, *target*, *which='nearest'*, *grid_times_targets=False*, *n_grid_points=150*)

Calculate time at the antitransit of the meridian.

Compute time of the next/previous/nearest antitransit of the `target` object.

Parameters

time

[`Time` or other (see below)] Time of observation. This will be passed in as the first argument to the `Time` initializer, so it can be anything that `Time` will accept (including a `Time` object)

target

[`SkyCoord`, `FixedTarget`, or list] Target celestial object(s)

which

[{'next', 'previous', 'nearest'}] Choose which sunrise relative to the present `time` would you like to calculate

grid_times_targets

[bool] If True, the target object will have extra dimensions packed onto the end, so that calculations with M targets and N times will return an (M, N) shaped result. Otherwise, we rely on broadcasting the shapes together using standard numpy rules.

n_grid_points

[int (optional)] The number of grid points on which to search for the horizon crossings of the target over a 24 hour period, default is 150 which yields rise time precisions better than one minute.

Returns

`Time`

Antitransit time of target

Examples

Calculate the meridian anti-transit time of Rigel at Keck Observatory:

```
>>> from astroplan import Observer, FixedTarget
>>> from astropy.time import Time
>>> time = Time("2001-02-03 04:05:06")
>>> target = FixedTarget.from_name("Rigel")
>>> keck = Observer.at_site("Keck")
>>> rigel_antitransit_time = keck.target_meridian_antitransit_time(
...     time, target, which="next")
>>> print("ISO: {0.iso}, JD: {0.jd}".format(rigel_antitransit_time))
ISO: 2001-02-03 18:40:29.761, JD: 2451944.27812
```

target_meridian_transit_time(*time*, *target*, *which='nearest'*, *grid_times_targets=False*, *n_grid_points=150*)

Calculate time at the transit of the meridian.

Compute time of the next/previous/nearest transit of the `target` object.

Parameters

time

[`Time` or other (see below)] Time of observation. This will be passed in as the first argument to the `Time` initializer, so it can be anything that `Time` will accept (including a `Time` object)

target

[`SkyCoord`, `FixedTarget`, or list] Target celestial object(s)

which

[{'next', 'previous', 'nearest'}] Choose which sunrise relative to the present `time` would you like to calculate

grid_times_targets: bool

If True, the target object will have extra dimensions packed onto the end, so that calculations with M targets and N times will return an (M, N) shaped result. Otherwise, we rely on broadcasting the shapes together using standard numpy rules.

n_grid_points

[int (optional)] The number of grid points on which to search for the horizon crossings of the target over a 24 hour period, default is 150 which yields rise time precisions better than one minute.

Returns

`Time`

Transit time of target

Examples

Calculate the meridian transit time of Rigel at Keck Observatory:

```
>>> from astroplan import Observer, FixedTarget
>>> from astropy.time import Time
>>> time = Time("2001-02-03 04:05:06")
>>> target = FixedTarget.from_name("Rigel")
>>> keck = Observer.at_site("Keck")
>>> rigel_transit_time = keck.target_meridian_transit_time(time, target,
...                                                        which="next")
>>> print("ISO: {0.iso}, JD: {0.jd}".format(rigel_transit_time))
ISO: 2001-02-03 06:42:26.863, JD: 2451943.77948
```

target_rise_time(*time*, *target*, *which='nearest'*, *horizon=<Quantity 0. deg>*, *grid_times_targets=False*, *n_grid_points=150*)

Calculate rise time.

Compute time of the next/previous/nearest rise of the `target` object, where "rise" is defined as the time when the `target` transitions from altitudes below the `horizon` to above the `horizon`.

Parameters

time

[`Time` or other (see below)] Time of observation. This will be passed in as the first argument to the `Time` initializer, so it can be anything that `Time` will accept (including a `Time` object)

target

[`SkyCoord`, `FixedTarget`, or list] Target celestial object(s)

which

[{'next', 'previous', 'nearest'}] Choose which sunrise relative to the present `time` would you like to calculate

horizon

[`Quantity` (optional), default = zero degrees] Degrees above/below actual horizon to use for calculating rise/set times (i.e., -6 deg horizon = civil twilight, etc.)

grid_times_targets: bool

If True, the target object will have extra dimensions packed onto the end, so that calculations with M targets and N times will return an (M, N) shaped result. Otherwise, we rely on broadcasting the shapes together using standard numpy rules.

n_grid_points

[int (optional)] The number of grid points on which to search for the horizon crossings of the target over a 24 hour period, default is 150 which yields rise time precisions better than one minute.

Returns

`Time`

Rise time of target

Examples

Calculate the rise time of Rigel at Keck Observatory:

```
>>> from astroplan import Observer, FixedTarget
>>> from astropy.time import Time
>>> time = Time("2001-02-03 04:05:06")
>>> target = FixedTarget.from_name("Rigel")
>>> keck = Observer.at_site("Keck")
>>> rigel_rise_time = keck.target_rise_time(time, target, which="next")
>>> print("ISO: {0.iso}, JD: {0.jd}".format(rigel_rise_time))
ISO: 2001-02-04 00:51:23.330, JD: 2451944.53569
```

target_set_time(*time*, *target*, *which='nearest'*, *horizon=<Quantity 0. deg>*, *grid_times_targets=False*, *n_grid_points=150*)

Calculate set time.

Compute time of the next/previous/nearest set of `target`, where "set" is defined as when the `target` transitions from altitudes above `horizon` to below `horizon`.

Parameters

time

[`Time` or other (see below)] Time of observation. This will be passed in as the first argument to the `Time` initializer, so it can be anything that `Time` will accept (including a `Time` object)

target

[`SkyCoord`, `FixedTarget`, or list] Target celestial object(s)

which

[{'next', 'previous', 'nearest'}] Choose which sunset relative to the present `time` would you like to calculate

horizon

[`Quantity` (optional), default = zero degrees] Degrees above/below actual horizon to use for calculating rise/set times (i.e., -6 deg horizon = civil twilight, etc.)

grid_times_targets: bool

If True, the target object will have extra dimensions packed onto the end, so that calculations with M targets and N times will return an (M, N) shaped result. Otherwise, we rely on broadcasting the shapes together using standard numpy rules.

n_grid_points

[int (optional)] The number of grid points on which to search for the horizon crossings of the target over a 24 hour period, default is 150 which yields set time precisions better than one minute.

Returns

`Time`

Set time of target.

Examples

Calculate the set time of Rigel at Keck Observatory:

```
>>> from astroplan import Observer, FixedTarget
>>> from astropy.time import Time
>>> time = Time("2001-02-03 04:05:06")
>>> target = FixedTarget.from_name("Rigel")
>>> keck = Observer.at_site("Keck")
>>> rigel_set_time = keck.target_set_time(time, target, which="next")
>>> print("ISO: {0.iso}, JD: {0.jd}".format(rigel_set_time))
ISO: 2001-02-03 12:29:34.768, JD: 2451944.02054
```

tonight(*time=None*, *horizon=<Quantity 0. deg>*, *obswl=None*)

Return a time range corresponding to the nearest night

This will return a range of `Time` corresponding to the beginning and ending of the night. If in the middle of a given night, return times from `now` until the nearest `sun_rise_time`

Parameters

time

[`Time` (optional), default = `now`] The start time for tonight, which is allowed to be arbitrary. See description above for behavior

horizon

[`Quantity` (optional), default = zero degrees] Degrees above/below actual horizon to use for calculating rise/set times (e.g., -6 deg horizon = civil twilight, etc.)

obswl

[`Quantity` (optional)] Wavelength of the observation used in the calculation

Returns

times

[Time] A tuple of times corresponding to the start and end of current night

twilight_evening_astronomical(*time*, *which='nearest'*, *n_grid_points=150*)

Time at evening astronomical (-18 degree) twilight.

Parameters

time

[Time or other (see below)] Time of observation. This will be passed in as the first argument to the Time initializer, so it can be anything that Time will accept (including a Time object)

which

[{'next', 'previous', 'nearest'}] Choose which twilight relative to the present time would you like to calculate. Default is nearest.

n_grid_points

[int (optional)] The number of grid points on which to search for the horizon crossings of the target over a 24 hour period, default is 150 which yields twilight time precisions better than one minute.

Returns

Time

Time of twilight

twilight_evening_civil(*time*, *which='nearest'*, *n_grid_points=150*)

Time at evening civil (-6 degree) twilight.

Parameters

time

[Time or other (see below)] Time of observation. This will be passed in as the first argument to the Time initializer, so it can be anything that Time will accept (including a Time object)

which

[{'next', 'previous', 'nearest'}] Choose which twilight relative to the present time would you like to calculate. Default is nearest.

n_grid_points

[int (optional)] The number of grid points on which to search for the horizon crossings of the target over a 24 hour period, default is 150 which yields twilight time precisions better than one minute.

Returns

`Time`

Time of twilight

twilight_evening_nautical(*time*, *which='nearest'*, *n_grid_points=150*)

Time at evening nautical (-12 degree) twilight.

Parameters

time

[`Time` or other (see below)] Time of observation. This will be passed in as the first argument to the `Time` initializer, so it can be anything that `Time` will accept (including a `Time` object)

which

[{'next', 'previous', 'nearest'}] Choose which twilight relative to the present `time` would you like to calculate. Default is nearest.

n_grid_points

[int (optional)] The number of grid points on which to search for the horizon crossings of the target over a 24 hour period, default is 150 which yields twilight time precisions better than one minute.

Returns

`Time`

Time of twilight

twilight_morning_astronomical(*time*, *which='nearest'*, *n_grid_points=150*)

Time at morning astronomical (-18 degree) twilight.

Parameters

time

[`Time` or other (see below)] Time of observation. This will be passed in as the first argument to the `Time` initializer, so it can be anything that `Time` will accept (including a `Time` object)

which

[{'next', 'previous', 'nearest'}] Choose which twilight relative to the present `time` would you like to calculate

n_grid_points

[int (optional)] The number of grid points on which to search for the horizon crossings of the target over a 24 hour period, default is 150 which yields twilight time precisions better than one minute.

Returns

`Time`

Time of twilight

twilight_morning_civil(*time*, *which='nearest'*, *n_grid_points=150*)

Time at morning civil (-6 degree) twilight.

Parameters

time

[`Time` or other (see below)] Time of observation. This will be passed in as the first argument to the `Time` initializer, so it can be anything that `Time` will accept (including a `Time` object)

which

[{'next', 'previous', 'nearest'}] Choose which twilight relative to the present `time` would you like to calculate. Default is nearest.

n_grid_points

[int (optional)] The number of grid points on which to search for the horizon crossings of the target over a 24 hour period, default is 150 which yields twilight time precisions better than one minute.

Returns

`Time`

Time of sunset

twilight_morning_nautical(*time*, *which='nearest'*, *n_grid_points=150*)

Time at morning nautical (-12 degree) twilight.

Parameters

time

[Time or other (see below)] Time of observation. This will be passed in as the first argument to the Time initializer, so it can be anything that Time will accept (including a Time object)

which

[{'next', 'previous', 'nearest'}] Choose which twilight relative to the present time would you like to calculate. Default is nearest.

n_grid_points

[int (optional)] The number of grid points on which to search for the horizon crossings of the target over a 24 hour period, default is 150 which yields twilight time precisions better than one minute.

Returns

Time

Time of twilight

ObservingBlock

class astroplan.**ObservingBlock**(*target*, *duration*, *priority*, *configuration={}*, *constraints=None*, *name=None*)

Bases: object

An observation to be scheduled, consisting of a target and associated constraints on observations.

Parameters

target

[FixedTarget] Target to observe

duration

[Quantity] exposure time

priority

[integer or float] priority of this object in the target list. 1 is highest priority, no maximum

configuration

[dict] Configuration metadata

constraints

[list of Constraint objects] The constraints to apply to this particular observing block. Note that constraints applicable to the entire list should go into the scheduler.

name

[integer or str] User-defined name or ID.

Attributes Summary

constraints_scores

Methods Summary

from_exposures(target, priority, ...[, ...])

Attributes Documentation

constraints_scores

Methods Documentation

classmethod from_exposures(*target*, *priority*, *time_per_exposure*, *number_exposures*, *readout_time=<Quantity 0. s>*, *configuration={}*, *constraints=None*)

OldEarthOrientationDataWarning

exception astroplan.**OldEarthOrientationDataWarning**

Using old Earth rotation data from IERS

PeriodicEvent

class astroplan.**PeriodicEvent**(*epoch*, *period*, *duration=None*, *name=None*)

Bases: object

A periodic event defined by an epoch and period.

Parameters

epoch

[Time] Time of event

period

[Quantity] Period of event

duration

[Quantity (optional)] Duration of event

name

[str (optional)] Name of target/event

Methods Summary

phase(time)	Phase of periodic event, on interval [0, 1).

Methods Documentation

phase(*time*)

Phase of periodic event, on interval [0, 1). For example, the phase could be an orbital phase for an eclipsing binary system.

Parameters

time

[Time] Evaluate the phase at this time or times

Returns

phase_array

[ndarray] Phase at each `time`, on range [0, 1)

PhaseConstraint

class astroplan.**PhaseConstraint**(*periodic_event*, *min=None*, *max=None*)

Bases: Constraint

Constrain observations to times in some range of phases for a periodic event (e.g.~transiting exoplanets, eclipsing binaries).

Parameters

periodic_event

[PeriodicEvent or subclass] System on which to compute the phase. For example, the system could be an eclipsing or non-eclipsing binary, or exoplanet system.

min

[float (optional)] Minimum phase (inclusive) on interval [0, 1). Default is zero.

max

[float (optional)] Maximum phase (inclusive) on interval [0, 1). Default is one.

Examples

To constrain observations on orbital phases between 0.4 and 0.6, >>> from astroplan import PeriodicEvent >>> from astropy.time import Time >>> import astropy.units as u >>> binary = PeriodicEvent(epoch=Time('2017-01-01 02:00'), period=1*u.day) >>> constraint = PhaseConstraint(binary, min=0.4, max=0.6)

The minimum and maximum phase must be described on the interval [0, 1). To constrain observations on orbital phases between 0.6 and 1.2, for example, you should subtract one from the second number: >>> constraint = PhaseConstraint(binary, min=0.6, max=0.2)

Methods Summary

compute_constraint(times[, observer, targets])	Actually do the real work of computing the constraint.

Methods Documentation

compute_constraint(*times*, *observer=None*, *targets=None*)

Actually do the real work of computing the constraint. Subclasses override this.

Parameters

times

[Time] The times to compute the constraint

observer

[Observer] the observaton location from which to apply the constraints

targets

[sequence of Target] The targets on which to apply the constraints.

Returns

constraint_result

[2D array of float or bool] The constraints, with targets along the first index and times along the second.

PlotBelowHorizonWarning

exception astroplan.**PlotBelowHorizonWarning**

Warning for when something is hidden on a plot because it's below the horizon

PlotWarning

exception astroplan.**PlotWarning**

Warnings dealing with the plotting aspects of astroplan

PrimaryEclipseConstraint

class astroplan.**PrimaryEclipseConstraint**(*eclipsing_system*)

Bases: Constraint

Constrain observations to times during primary eclipse.

Parameters

eclipsing_system

[EclipsingSystem] System which must be in primary eclipse.

Methods Summary

compute_constraint(times[, observer, targets])	Actually do the real work of computing the constraint.

Methods Documentation

compute_constraint(*times*, *observer=None*, *targets=None*)

Actually do the real work of computing the constraint. Subclasses override this.

Parameters

times

[Time] The times to compute the constraint

observer

[Observer] the observaton location from which to apply the constraints

targets

[sequence of Target] The targets on which to apply the constraints.

Returns

constraint_result

[2D array of float or bool] The constraints, with targets along the first index and times along the second.

PriorityScheduler

class astroplan.**PriorityScheduler**(**args*, ***kwargs*)

Bases: Scheduler

A scheduler that optimizes a prioritized list. That is, it finds the best time for each ObservingBlock, in order of priority.

Methods Summary

attempt_insert_block(b, new_start_time, ...)

Methods Documentation

attempt_insert_block(*b*, *new_start_time*, *start_time_idx*)

Schedule

class astroplan.**Schedule**(*start_time*, *end_time*, *constraints=None*)

Bases: object

An object that represents a schedule, consisting of a list of Slot objects.

Parameters

start_time

[Time] The starting time of the schedule; the start of your observing window.

end_time

[Time] The ending time of the schedule; the end of your observing window

constraints

[sequence of `Constraint` s] these are constraints that apply to the entire schedule

Attributes Summary

`observing_blocks`	
`open_slots`	
`scheduled_blocks`	

Methods Summary

`change_slot_block`(slot_index[, new_block])	Change the block associated with a slot.
`insert_slot`(start_time, block)	Insert a slot into schedule and associate a block to the new slot.
`new_slots`(slot_index, start_time, end_time)	Create new slots by splitting a current slot.
`to_table`([show_transitions, show_unused])	

Attributes Documentation

observing_blocks

open_slots

scheduled_blocks

Methods Documentation

change_slot_block(*slot_index*, *new_block=None*)

Change the block associated with a slot.

This is currently designed to work for TransitionBlocks in PriorityScheduler The assumption is that the slot afterwards is open and that the start time will remain the same.

If the block is changed to None, the slot is merged with the slot afterwards to make a longer slot.

Parameters

slot_index

[int] The slot to edit

new_block

[TransitionBlock, default None] The new transition block to insert in this slot

insert_slot(*start_time*, *block*)

Insert a slot into schedule and associate a block to the new slot.

Parameters

start_time

[Time] The start time for the new slot.

block

[ObservingBlock] The observing block to insert into new slot.

Returns

slots

[list of Slot objects] The new slots in the schedule.

new_slots(*slot_index*, *start_time*, *end_time*)

Create new slots by splitting a current slot.

Parameters

slot_index

[int] The index of the slot to split

start_time

[Time] The start time for the slot to create

end_time

[Time] The end time for the slot to create

Returns

new_slots

[list of Slot s] The new slots created

to_table(*show_transitions=True*, *show_unused=False*)

Scheduler

class astroplan.**Scheduler**(*constraints*, *observer*, *transitioner=None*, *gap_time=<Quantity 5. min>*, *time_resolution=<Quantity 20. s>*)

Bases: `object`

Schedule a set of `ObservingBlock` objects

Parameters

constraints

[sequence of `Constraint`] The constraints to apply to *every* observing block. Note that constraints for specific blocks can go on each block individually.

observer

[`Observer`] The observer/site to do the scheduling for.

transitioner

[`Transitioner` (required)] The object to use for computing transition times between blocks. Leaving it as `None` will cause an error.

gap_time

[`Quantity` with time units] The maximum length of time a transition between Observing-Blocks could take.

time_resolution

[`Quantity` with time units] The smallest factor of time used in scheduling, all Blocks scheduled will have a duration that is a multiple of it.

Methods Summary

`__call__`(blocks, schedule)	Schedule a set of `ObservingBlock` objects.
`from_timespan`(center_time, duration, **kwargs)	Create a new instance of this class given a center time and duration.

Methods Documentation

__call__(*blocks*, *schedule*)

Schedule a set of ObservingBlock objects.

Parameters

blocks

[list of ObservingBlock objects] The observing blocks to schedule. Note that the input ObservingBlock objects will *not* be modified - new ones will be created and returned.

schedule

[Schedule object] A schedule that the blocks will be scheduled in. At this time the schedule must be empty, only defined by a start and end time.

Returns

schedule

[Schedule] A schedule objects which consists of Slot objects with and without populated block objects containing either TransitionBlock or ObservingBlock objects with populated start_time and end_time or duration attributes

classmethod from_timespan(*center_time*, *duration*, ***kwargs*)

Create a new instance of this class given a center time and duration.

Parameters

center_time

[Time] Mid-point of time-span to schedule.

duration

[Quantity or TimeDelta] Duration of time-span to schedule

Scorer

class astroplan.**Scorer**(*blocks*, *observer*, *schedule*, *global_constraints=[]*)

Bases: object

Returns scores and score arrays from the evaluation of constraints on observing blocks

Parameters

blocks

[list of `ObservingBlock` objects] list of blocks that need to be scored

observer

[`Observer`] the observer

schedule

[`Schedule`] The schedule inside which the blocks should fit

global_constraints

[list of `Constraint` objects] any `Constraint` that applies to all the blocks

Methods Summary

`create_score_array`([time_resolution])	this makes a score array over the entire schedule for all of the blocks and each `Constraint` in the .constraints of each block and in self.global_constraints.
`from_start_end`(blocks, observer, start_time, ...)	for if you don't have a schedule/ aren't inside a scheduler

Methods Documentation

create_score_array(*time_resolution=<Quantity 1. min>*)

this makes a score array over the entire schedule for all of the blocks and each `Constraint` in the .constraints of each block and in self.global_constraints.

Parameters

time_resolution

[`Quantity`] the time between each scored time

Returns

score_array

[`ndarray`] array with dimensions (# of blocks, schedule length/ `time_resolution`

classmethod from_start_end(*blocks*, *observer*, *start_time*, *end_time*, *global_constraints=[]*)

for if you don't have a schedule/ aren't inside a scheduler

SecondaryEclipseConstraint

class astroplan.**SecondaryEclipseConstraint**(*eclipsing_system*)

Bases: Constraint

Constrain observations to times during secondary eclipse.

Parameters

eclipsing_system

[EclipsingSystem] System which must be in secondary eclipse.

Methods Summary

compute_constraint(times[, observer, targets])	Actually do the real work of computing the constraint.

Methods Documentation

compute_constraint(*times*, *observer=None*, *targets=None*)

Actually do the real work of computing the constraint. Subclasses override this.

Parameters

times

[Time] The times to compute the constraint

observer

[Observer] the observaton location from which to apply the constraints

targets

[sequence of Target] The targets on which to apply the constraints.

Returns

constraint_result

[2D array of float or bool] The constraints, with targets along the first index and times along the second.

SequentialScheduler

class astroplan.**SequentialScheduler**(**args*, ***kwargs*)

Bases: Scheduler

A scheduler that does "stupid simple sequential scheduling". That is, it simply looks at all the blocks, picks the best one, schedules it, and then moves on.

Parameters

constraints

[sequence of Constraint] The constraints to apply to *every* observing block. Note that constraints for specific blocks can go on each block individually.

observer

[Observer] The observer/site to do the scheduling for.

transitioner

[Transitioner (required)] The object to use for computing transition times between blocks. Leaving it as None will cause an error.

gap_time

[Quantity with time units] The maximum length of time a transition between Observing-Blocks could take.

time_resolution

[Quantity with time units] The smallest factor of time used in scheduling, all Blocks scheduled will have a duration that is a multiple of it.

Slot

class astroplan.**Slot**(*start_time*, *end_time*)

Bases: object

A time slot consisting of a start and end time

Parameters

start_time

[Time] The starting time of the slot

end_time

[Time] The ending time of the slot

Attributes Summary

duration	

Methods Summary

split_slot(early_time, later_time)	Split this slot and insert a new one.

Attributes Documentation

duration

Methods Documentation

split_slot(*early_time*, *later_time*)

Split this slot and insert a new one.

Will return the new slots created, which can either be one, two or three slots depending on if there is space remaining before or after the inserted slot.

Parameters

early_time

[Time] The start time of the new slot to insert.

later_time

[Time] The end time of the new slot to insert.

SunSeparationConstraint

class astroplan.**SunSeparationConstraint**(*min=None*, *max=None*)

Bases: Constraint

Constrain the distance between the Sun and some targets.

Parameters

min

[Quantity or None (optional)] Minimum acceptable separation between Sun and target (inclusive). None indicates no limit.

max

[Quantity or None (optional)] Maximum acceptable separation between Sun and target (inclusive). None indicates no limit.

Methods Summary

compute_constraint(times, observer, targets)	Actually do the real work of computing the constraint.

Methods Documentation

compute_constraint(*times*, *observer*, *targets*)

Actually do the real work of computing the constraint. Subclasses override this.

Parameters

times

[Time] The times to compute the constraint

observer

[Observer] the observaton location from which to apply the constraints

targets

[sequence of Target] The targets on which to apply the constraints.

Returns

constraint_result

[2D array of float or bool] The constraints, with targets along the first index and times along the second.

Target

class astroplan.**Target**(*name=None*, *ra=None*, *dec=None*, *marker=None*)

Bases: object

Abstract base class for target objects.

This is an abstract base class – you can't instantiate examples of this class, but must work with one of its subclasses such as FixedTarget or NonFixedTarget.

Defines a single observation target.

Parameters

name

[str, optional]

ra

[WHAT TYPE IS ra ?]

dec

[WHAT TYPE IS dec ?]

marker

[str, optional] User-defined markers to differentiate between different types of targets (e.g., guides, high-priority, etc.).

Attributes Summary

dec	Declination.
ra	Right ascension.

Attributes Documentation

dec

Declination.

ra

Right ascension.

TargetAlwaysUpWarning

exception astroplan.**TargetAlwaysUpWarning**

Target is circumpolar

TargetNeverUpWarning

exception astroplan.**TargetNeverUpWarning**

Target never rises above horizon

TimeConstraint

class astroplan.**TimeConstraint**(*min=None*, *max=None*)

Bases: Constraint

Constrain the observing time to be within certain time limits.

An example use case for this class would be to associate an acceptable time range with a specific observing block. This can be useful if not all observing blocks are valid over the time limits used in calls to is_observable or is_always_observable.

Parameters

min

[Time] Earliest time (inclusive). None indicates no limit.

max

[Time] Latest time (inclusive). None indicates no limit.

Examples

Constrain the observations to targets that are observable between 2016-03-28 and 2016-03-30:

```
>>> from astroplan import Observer
>>> from astropy.time import Time
>>> subaru = Observer.at_site("Subaru")
>>> t1 = Time("2016-03-28T12:00:00")
>>> t2 = Time("2016-03-30T12:00:00")
>>> constraint = TimeConstraint(t1,t2)
```

Methods Summary

compute_constraint(times, observer, targets)	Actually do the real work of computing the constraint.

Methods Documentation

compute_constraint(*times*, *observer*, *targets*)

Actually do the real work of computing the constraint. Subclasses override this.

Parameters

times

[Time] The times to compute the constraint

observer

[Observer] the observaton location from which to apply the constraints

targets

[sequence of Target] The targets on which to apply the constraints.

Returns

constraint_result

[2D array of float or bool] The constraints, with targets along the first index and times along the second.

TransitionBlock

class astroplan.**TransitionBlock**(*components*, *start_time=None*)

Bases: object

Parameterizes the "dead time", e.g. between observations, while the telescope is slewing, instrument is reconfiguring, etc.

Parameters

components

[dict] A dictionary mapping the reason for an observation's dead time to Quantity objects with time units

start_time

[Quantity] Start time of observation

Attributes Summary

components
end_time

Methods Summary

from_duration(duration)

Attributes Documentation

components

end_time

Methods Documentation

classmethod from_duration(*duration*)

Transitioner

class astroplan.**Transitioner**(*slew_rate=None*, *instrument_reconfig_times=None*)

Bases: object

A class that defines how to compute transition times from one block to another.

Parameters

slew_rate

[Quantity with angle/time units] The slew rate of the telescope

instrument_reconfig_times

[dict of dicts or None] If not None, gives a mapping from property names to another dictionary. The second dictionary maps 2-tuples of states to the time it takes to transition between those states (as an Quantity), can also take a 'default' key mapped to a default transition time.

Methods Summary

`__call__`(oldblock, newblock, start_time, ...)	Determines the amount of time needed to transition from one observing block to another.
`compute_instrument_transitions`(oldblock, ...)	

Methods Documentation

__call__(*oldblock*, *newblock*, *start_time*, *observer*)

Determines the amount of time needed to transition from one observing block to another. This uses the parameters defined in `self.instrument_reconfig_times`.

Parameters

oldblock

[`ObservingBlock` or None] The initial configuration/target

newblock

[`ObservingBlock` or None] The new configuration/target to transition to

start_time

[`Time`] The time the transition should start

observer

[`astroplan.Observer`] The observer at the time

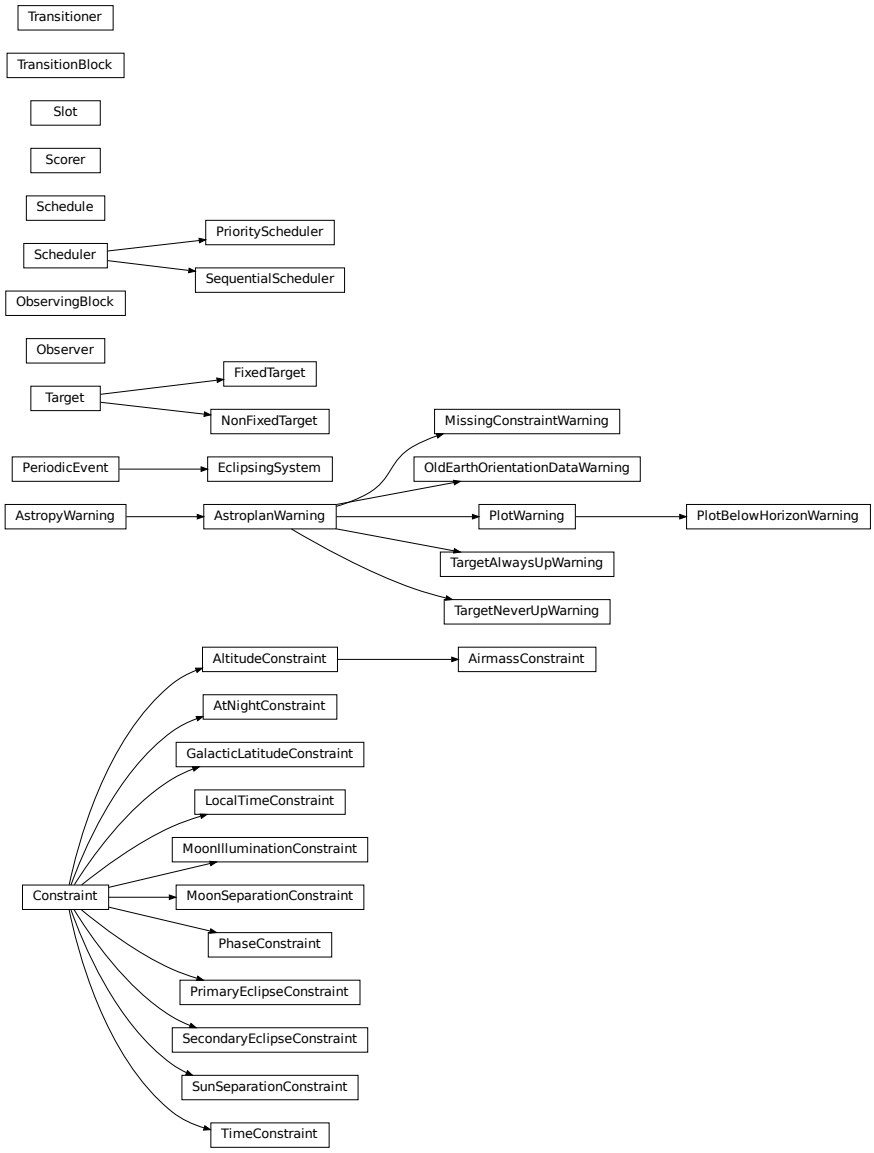
Returns

transition

[`TransitionBlock` or None] A transition to get from `oldblock` to `newblock` or `None` if no transition is necessary

compute_instrument_transitions(*oldblock*, *newblock*)

5.1.3 Class Inheritance Diagram



5.2 astroplan.plots Package

`astroplan.plots` contains functions for making plots of commonly-used quantities in observation planning (e.g., airmass vs. time), using `astroplan` and Matplotlib.

5.2.1 Functions

`plot_airmass`(targets, observer, time[, ax, ...])	Plots airmass as a function of time for a given target.
`plot_altitude`(targets, observer, time[, ax, ...])	Plots altitude as a function of time for a given target.
`plot_finder_image`(target[, survey, ...])	Plot survey image centered on `target`.
`plot_parallactic`(target, observer, time[, ...])	Plots parallactic angle as a function of time for a given target.
`plot_schedule_airmass`(schedule[, show_night])	Plots when observations of targets are scheduled to occur superimposed upon plots of the airmasses of the targets.
`plot_sky`(target, observer, time[, ax, ...])	Plots target positions in the sky with respect to the observer's location.
`plot_sky_24hr`(target, observer, time[, ...])	Plots target positions in the sky with respect to the observer's location over a twenty-four hour period centered on `time`.

plot_airmass

astroplan.plots.**plot_airmass**(*targets*, *observer*, *time*, *ax=None*, *style_kwargs=None*, *style_sheet=None*, *brightness_shading=False*, *altitude_yaxis=False*, *min_airmass=1.0*, *min_region=None*, *max_airmass=3.0*, *max_region=None*, *use_local_tz=False*)

Plots airmass as a function of time for a given target.

If a `Axes` object already exists, an additional airmass plot will be "stacked" on it. Otherwise, creates a new `Axes` object and plots airmass on top of that.

When a scalar `Time` object is passed in (e.g., `Time('2000-1-1')`), the resulting plot will use a 24-hour window centered on the time indicated, with airmass sampled at regular intervals throughout. However, the user can control the exact number and frequency of airmass calculations used by passing in a non-scalar `Time` object. For instance, `Time(['2000-1-1 23:00:00', '2000-1-1 23:30:00'])` will result in a plot with only two airmass measurements.

For examples with plots, visit the documentation of *Time Dependent Plots*.

Parameters

targets

[list of `FixedTarget` objects] The celestial bodies of interest. If a single object is passed it will be converted to a list.

observer

[`Observer`] The person, telescope, observatory, etc. doing the observing.

time

[Time] If scalar (e.g., `Time('2000-1-1')`), will result in plotting target airmasses once an hour over a 24-hour window. If non-scalar (e.g., `Time(['2000-1-1'])`, `[Time('2000-1-1')]`, `Time(['2000-1-1', '2000-1-2'])`), will result in plotting data at the exact times specified.

ax

[Axes or None, optional.] The `Axes` object to be drawn on. If None, uses the current `Axes`.

style_kwargs

[dict or None, optional.] A dictionary of keywords passed into `plot_date` to set plotting styles.

style_sheet

[dict or `None` (optional)] matplotlib style sheet to use. To see available style sheets in astroplan, print *astroplan.plots.available_style_sheets*. Defaults to the light theme.

brightness_shading

[bool] Shade background of plot to scale roughly with sky brightness. Dark shading signifies times when the sun is below the horizon. Default is `False`.

altitude_yaxis

[bool] Add alternative y-axis on the right side of the figure with target altitude. Default is `False`.

min_airmass

[float] Lower limit of y-axis airmass range in the plot. Default is `1.0`.

max_airmass

[float] Upper limit of y-axis airmass range in the plot. Default is `3.0`.

min_region

[float] If set, defines an interval between `min_airmass` and `min_region` that will be shaded. Default is `None`.

max_region

[float] If set, defines an interval between `max_airmass` and `max_region` that will be shaded. Default is `None`.

use_local_tz

[bool] If the time is specified in a local timezone, the time will be plotted in that timezone.

Returns

ax

[Axes] An `Axes` object with added airmass vs. time plot.

Notes

y-axis is inverted and shows airmasses between 1.0 and 3.0 by default. If user wishes to change these, use `ax.<set attribute>` before drawing or saving plot:

plot_altitude

astroplan.plots.**plot_altitude**(*targets*, *observer*, *time*, *ax=None*, *style_kwargs=None*, *style_sheet=None*, *brightness_shading=False*, *airmass_yaxis=False*, *min_altitude=0*, *min_region=None*, *max_altitude=90*, *max_region=None*)

Plots altitude as a function of time for a given target.

If a `Axes` object already exists, an additional altitude plot will be "stacked" on it. Otherwise, creates a new `Axes` object and plots altitude on top of that.

When a scalar `Time` object is passed in (e.g., `Time('2000-1-1')`), the resulting plot will use a 24-hour window centered on the time indicated, with altitude sampled at regular intervals throughout. However, the user can control the exact number and frequency of altitude calculations used by passing in a non-scalar `Time` object. For instance, `Time(['2000-1-1 23:00:00', '2000-1-1 23:30:00'])` will result in a plot with only two altitude measurements.

For examples with plots, visit the documentation of *Time Dependent Plots*.

Parameters

targets

[list of `FixedTarget` objects] The celestial bodies of interest. If a single object is passed it will be converted to a list.

observer

[`Observer`] The person, telescope, observatory, etc. doing the observing.

time

[`Time`] If scalar (e.g., `Time('2000-1-1')`), will result in plotting target altitudes once an hour over a 24-hour window. If non-scalar (e.g., `Time(['2000-1-1'])`, `[Time('2000-1-1')]`, `Time(['2000-1-1', '2000-1-2'])`), will result in plotting data at the exact times specified.

ax

[`Axes` or None, optional.] The `Axes` object to be drawn on. If None, uses the current `Axes`.

style_kwargs

[dict or None, optional.] A dictionary of keywords passed into `plot_date` to set plotting styles.

style_sheet

[dict or `None` (optional)] matplotlib style sheet to use. To see available style sheets in astroplan, print *astroplan.plots.available_style_sheets*. Defaults to the light theme.

brightness_shading

[bool] Shade background of plot to scale roughly with sky brightness. Dark shading signifies times when the sun is below the horizon. Default is `False`.

altitude_yaxis

[bool] Add alternative y-axis on the right side of the figure with target altitude. Default is `False`.

min_altitude

[float] Lower limit of y-axis altitude range in the plot. Default is `1.0`.

max_altitude

[float] Upper limit of y-axis altitude range in the plot. Default is `3.0`.

min_region

[float] If set, defines an interval between `min_altitude` and `min_region` that will be shaded. Default is `None`.

max_region

[float] If set, defines an interval between `max_altitude` and `max_region` that will be shaded. Default is `None`.

Returns

ax

[`Axes`] An `Axes` object with added altitude vs. time plot.

plot_finder_image

`astroplan.plots.`**`plot_finder_image`**(*target*, *survey='DSS'*, *fov_radius=<Quantity 10. arcmin>*, *log=False*, *ax=None*, *grid=False*, *reticle=False*, *style_kwargs=None*, *reticle_style_kwargs=None*)

Plot survey image centered on `target`.

Survey images are retrieved from NASA Goddard's SkyView service via `astroquery.skyview.SkyView`.

If a `Axes` object already exists, plots the finder image on top. Otherwise, creates a new `Axes` object with the finder image.

Parameters

target

[`FixedTarget`, `SkyCoord`] Coordinates of celestial object

survey

[string] Name of survey to retrieve image from. For dictionary of available surveys, use

`from astroquery.skyview import SkyView; SkyView.list_surveys()`. Defaults to `'DSS'`, the Digital Sky Survey.

fov_radius

[`Quantity`] Radius of field of view of retrieved image. Defaults to 10 arcmin.

log

[bool, optional] Take the natural logarithm of the FITS image if `True`. False by default.

ax

[`Axes` or None, optional.] The `Axes` object to be drawn on. If None, uses the current `Axes`.

grid

[bool, optional.] Grid is drawn if `True`. `False` by default.

reticle

[bool, optional] Draw reticle on the center of the FOV if `True`. Default is `False`.

style_kwargs

[dict or `None`, optional.] A dictionary of keywords passed into `imshow` to set plotting styles.

reticle_style_kwargs

[dict or `None`, optional] A dictionary of keywords passed into `axvline` and `axhline` to set reticle style.

Returns

ax

[`Axes`] Matplotlib axes with survey image centered on `target`

hdu

[`PrimaryHDU`] FITS HDU of the retrieved image

Notes

Dependencies:

In addition to Matplotlib, this function makes use of astroquery.

plot_parallactic

astroplan.plots.**plot_parallactic**(*target*, *observer*, *time*, *ax=None*, *style_kwargs=None*, *style_sheet=None*)

Plots parallactic angle as a function of time for a given target.

If a `Axes` object already exists, an additional parallactic angle plot will be "stacked" on it. Otherwise, creates a new `Axes` object and plots on top of that.

When a scalar `Time` object is passed in (e.g., `Time('2000-1-1')`), the resulting plot will use a 24-hour window centered on the time indicated, with parallactic angle sampled at regular intervals throughout. However, the user can control the exact number and frequency of parallactic angle calculations used by passing in a non-scalar `Time` object. For instance, `Time(['2000-1-1 23:00:00', '2000-1-1 23:30:00'])` will result in a plot with only two parallactic angle measurements.

For examples with plots, visit the documentation of *Time Dependent Plots*.

Parameters

target

[`FixedTarget`] The celestial body of interest.

observer

[`Observer`] The person, telescope, observatory, etc. doing the observing.

time

[`Time`] If scalar (e.g., `Time('2000-1-1')`), will result in plotting target parallactic angle once an hour over a 24-hour window. If non-scalar (e.g., `Time(['2000-1-1'])`, `[Time('2000-1-1')]`, `Time(['2000-1-1', '2000-1-2'])`), will result in plotting data at the exact times specified.

ax

[`Axes` or None, optional.] The `Axes` object to be drawn on. If None, uses the current `Axes`.

style_kwargs

[dict or None, optional.] A dictionary of keywords passed into `plot_date` to set plotting styles.

style_sheet

[dict or `None` (optional)] matplotlib style sheet to use. To see available style sheets in astroplan, print *astroplan.plots.available_style_sheets*. Defaults to the light theme.

Returns

ax

[`Axes`] An `Axes` object with added parallactic angle vs. time plot.

plot_schedule_airmass

astroplan.plots.**plot_schedule_airmass**(*schedule*, *show_night=False*)

Plots when observations of targets are scheduled to occur superimposed upon plots of the airmasses of the targets.

Parameters

schedule

[Schedule] a schedule object output by a scheduler

show_night

[bool] Shades the night-time on the plot

Returns

ax

[Axes] An Axes object with added airmass and schedule vs. time plot.

plot_sky

astroplan.plots.**plot_sky**(*target*, *observer*, *time*, *ax=None*, *style_kwargs=None*, *north_to_east_ccw=True*, *grid=True*, *az_label_offset=<Quantity 0. deg>*, *warn_below_horizon=False*, *style_sheet=None*)

Plots target positions in the sky with respect to the observer's location.

If a Axes object already exists, plots an additional target position on top. Otherwise, creates a new Axes object with a sky plot.

Can pass in a scalar Time object (e.g. `Time('2000-1-1')`) or an array of length one (e.g. `Time(['2000-1-1'])`) to get plot at one instance in time. If pass in an Time object with multiple instances of time (e.g. `Time(['2000-1-1 20:00:00', '2000-1-1 20:30:00'])`), target's position will be shown at each of these times.

For examples with plots, visit the documentation of *Sky Charts*.

Parameters

target

[FixedTarget] The celestial body of interest.

observer

[Observer] The person, telescope, observatory, etc. doing the observing.

time

[Time] If pass in an Time object with just one instance in time, whether it be a scalar or an array (e.g. Time('2000-1-1'), Time(['2000-1-1']), [Time('2000-1-1')]), plot_sky will return plot at one instance in time. If pass in an Time object with multiple instances in time (e.g. Time(['2000-1-1', '2000-1-2'])) will show positions plotted at the exact times specified.

ax

[Axes or None, optional.] The Axes object to be drawn on. If None, uses the current Axes.

style_kwargs

[dict or None, optional.] A dictionary of keywords passed into scatter to set plotting styles.

north_to_east_ccw

[bool, optional.] True by default, meaning that azimuth is shown increasing counter-clockwise (CCW), or with North at top, East at left, etc. To show azimuth increasing clockwise (CW), set to False.

grid

[bool, optional.] True by default, meaning that grid is drawn.

az_label_offset

[~astropy.units.degree, optional.] DANGER: It is not recommended that you change the default behavior, as to do so makes it seem as if N/E/S/W are being decoupled from the definition of azimuth (North from az = 0 deg., East from az = 90 deg., etc.). An offset for azimuth labels from the North label. A positive offset will increase in the same direction as azimuth (see north_to_east_ccw option).

warn_below_horizon

[bool, optional] If False, don't show warnings when attempting to plot targets below the horzion.

style_sheet

[dict or None (optional)] matplotlib style sheet to use. To see available style sheets in astroplan, print *astroplan.plots.available_style_sheets*. Defaults to the light theme.

Returns

An Axes object (ax) with a map of the sky.

Notes

Using `Time` objects:

See Astropy documentation for more details.

Coordinate defaults:

Altazimuth (local horizon) coordinate system. North is always at top of plot, South is always at the bottom, E/W can be right or left depending on the `north_to_east_cw` option.

Altitude: 90 degrees (zenith) is at plot origin (center) and 0 degrees (horizon) is at plot edge. This cannot be changed by user.

Azimuth: 0 degrees is at North (top of plot), 90 degrees at East, etc. DANGER: Azimuth labels can be changed by user via the `az_label_offset` option, but it is not recommended, as to do so makes it seem as if N/E/S/W are being decoupled from the definition of azimuth (North from az = 0 deg., East from az = 90 deg., etc.).

plot_sky_24hr

astroplan.plots.**plot_sky_24hr**(*target*, *observer*, *time*, *delta=<Quantity 1. h>*, *ax=None*, *style_kwargs=None*, *north_to_east_ccw=True*, *grid=True*, *az_label_offset=<Quantity 0. deg>*, *center_time_style_kwargs=None*)

Plots target positions in the sky with respect to the observer's location over a twenty-four hour period centered on `time`.

Parameters

target

[`FixedTarget`] The celestial body of interest.

observer: `~astroplan.Observer`

The person, telescope, observatory, etc. doing the observing.

time

[`Time`] If pass in an `Time` object with just one instance in time, whether it be a scalar or an array (e.g. `Time('2000-1-1')`, `Time(['2000-1-1'])`, `[Time('2000-1-1')]`), `plot_sky` will return plot at one instance in time. If pass in an `Time` object with multiple instances in time (e.g. `Time(['2000-1-1', '2000-1-2'])`) will show positions plotted at the exact times specified.

delta

[`Quantity` (optional)] Interval between times plotted.

ax

[`Axes` or None, optional.] The `Axes` object to be drawn on. If None, uses the current `Axes`.

style_kwargs

[dict or None, optional.] A dictionary of keywords passed into `scatter` to set plotting styles.

north_to_east_ccw

[bool, optional.] True by default, meaning that azimuth is shown increasing counter-clockwise (CCW), or with North at top, East at left, etc. To show azimuth increasing clockwise (CW), set to False.

grid

[bool, optional.] True by default, meaning that grid is drawn.

az_label_offset

[`~astropy.units.degree`, optional.] DANGER: It is not recommended that you change the default behavior, as to do so makes it seem as if N/E/S/W are being decoupled from the definition of azimuth (North from az = 0 deg., East from az = 90 deg., etc.). An offset for azimuth labels from the North label. A positive offset will increase in the same direction as azimuth (see `north_to_east_ccw` option).

center_time_style_kwargs

[dict or `None` (optional)] Dictionary of style keyword arguments to pass to `scatter` to set plotting style of the point at time `time`.

Returns

An `Axes` object (ax) with a map of the sky.

CHAPTER

SIX

CHANGELOG

6.1 0.11 (unreleased)

6.2 0.10 (2024-04-04)

- Fix compatibility with astropy v6.0.

6.3 0.9.1 (2023-09-20)

- Fix bug when `FixedTarget` objects are passed to methods that calculate lunar coordinates. [#568]

6.4 0.9 (2023-07-27)

- Fix time range in `months_observable` to not be only in 2014. Function now accepts argument `time_range` and defaults to the current year. [#458]
- Fix `Observer` not having longtitude, latitude, and elevation parameters as class attributes. They are now properties calculated from the `location`.
- Documentation revisions and theme update [#563]

6.5 0.8 (2021-01-26)

- Fix Read The Docs compatibility [#497]
- Move to APE 17 infrastructure, change to github actions [#493]
- Update conda channel in favor of conda-forge [#491]
- Fix for astropy cache compatibility [#481]

6.6 0.7 (2020-10-27)

- Fix compatibility with Astropy 4.X

6.7 0.6 (2019-10-08)

- Added documentation for reproducing MMTO sun rise/set times [#434]
- Deprecation of `MAGIC_TIME` variable, which used to be returned for targets that don't rise or set [#435]
- Replace deprecated astroquery service [#431]
- Fix for the broken IERS patch [#418, #425]
- Add `GalacticLatitudeConstraint` to constrain the galactic latitudes of targets. This can be useful for planning surveys for which crowding due to Galactic point sources is an issue. [#413]
- Add `n_grid_points` keyword argument to rise/set/transit functions which allows usersto trade off precision for speed. [#424]

6.8 0.5 (2019-07-08)

- `observability_table` now accepts scalars as `time_range` arguments, and gives `'time observable'` in this case in the resulting table. [#350]
- Bug fixes [#414, #412, #407, #401]

6.9 0.4 (2017-10-23)

- Added new `eclipsing` module for eclipsing binaries and transiting exoplanets [#315]
- Fixes for compatibility with astropy Quantity object updates [#336]
- Better PEP8 compatibility [#335]
- Using travis build stages [#330]

6.10 0.3 (2017-09-02)

- `Observer.altaz` and `Constraint.__call__` no longer returns an (MxN) grid of results when called with M `target``s and N ``times`. Instead, we attempt to broadcast the time and target shapes, and an error is raised if this is not possible. This change breaks backwards compatibility but an optional argument `grid_times_targets` has been added to these methods. If set to True, the old behaviour is recovered. All `Observer` methods for which it is relevant have this optional argument.
- Updates for compatibility with astropy v2.0 coordinates implementation [#311], updates to astropy-helpers [#309], fix pytest version [#312]

6.11 0.2.1 (2016-04-27)

- Internal changes to the way calculations are done means that astropy>=1.3 is required [#285]
- Fixed bug when scheduling block list is empty [#298]
- Fixed bug in Transitioner object when no transition needed [#295]
- Update to astropy-helpers 1.3.1 [#294] and compatibility fixes for astropy 1.3 [#283]

6.12 0.2 (2016-09-20)

- Fixed bug arising from changes to distutils.ConfigParser [#177, #187, #191]
- Removed the sites module from astroplan, since it was ported to astropy [#168]
- Removed dependence on PyEphem, now using jplephem for the solar system ephemeris [#167]
- New API for scheduling observations (still in development)
- New `plot_finder_image` function makes quick finder charts [#115]
- Updates to astropy helpers and the package template [#177, #180]

CHAPTER

SEVEN

MAINTAINERS

- Brett Morris, including contributions from Google Summer of Code 2015

CHAPTER

EIGHT

ATTRIBUTION

If you use astroplan in your work, please cite Morris et al. 2018:

```
@ARTICLE{2018AJ....155..128M,
       author = {{Morris}, Brett M. and {Tollerud}, Erik and {Sip{\H{o}}cz}, Brigitta and
↪{Deil}, Christoph and {Douglas}, Stephanie T. and {Berlanga Medina}, Jazmin and
↪{Vyhmeister}, Karl and {Smith}, Toby R. and {Littlefair}, Stuart and {Price-Whelan},␣
↪Adrian M. and {Gee}, Wilfred T. and {Jeschke}, Eric},
        title = "{astroplan: An Open Source Observation Planning Package in Python}",
      journal = {\aj},
     keywords = {methods: numerical, methods: observational, Astrophysics - Instrumentation␣
↪and Methods for Astrophysics},
         year = 2018,
        month = mar,
       volume = {155},
       number = {3},
          eid = {128},
        pages = {128},
          doi = {10.3847/1538-3881/aaa47e},
archivePrefix = {arXiv},
       eprint = {1712.09631},
 primaryClass = {astro-ph.IM},
       adsurl = {https://ui.adsabs.harvard.edu/abs/2018AJ....155..128M},
      adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

BIBLIOGRAPHY

[1] Winn (2010) https://arxiv.org/abs/1001.2010

[1] https://en.wikipedia.org/wiki/Parallactic_angle

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

T