Trident is a Python package for creating synthetic absorption-line spectra from astrophysical hydrodynamics simulations. It utilizes the yt package to read in simulation datasets and extends it to provide realistic synthetic observations appropriate for studies of the interstellar, circumgalactic, and intergalactic media.

To avoid confusion, make sure you are viewing the correct documentation for the version of Trident you are using: stable vs. development. For more information, see Versions of Trident.
Follow these steps to successfully install Trident and its dependencies.

1.1 Versions of Trident

Previously, there were three versions of Trident: the stable version, the development version, and the demeshening version. The stable version is tried and tested, and it normally operates on a stable version of yt. The development version is actively being updated with new features, and it is also tied to the development version of yt, so occasionally unforeseen bugs can crop up as these new features are added. The demeshening version is currently in beta and active development and is used for better results on particle-based datasets.

After Trident 1.2 was released on September 19, 2019, the demeshening version was merged with the main development branch. Now, there are only the stable version and the development version.

The installation steps are slightly different between the two versions, so pay attention in the steps below. Don’t worry if you want to change later, you can always switch between the two versions easily enough by following the directions in Uninstallation or Switching Code Versions.

Note: The demeshening (development) version treats particle-based datasets more natively. The demeshening version will give faster and more accurate results with less memory overhead for particle-based datasets. For more information about the demeshening version, please see our demeshening notebook. Note, the installation instructions in the notebook should be ignored. Instead, follow the instructions for Installing the Development Version.

1.2 Step 1: Install yt

yt is a python-based software package for the analysis and visualization of a variety of different datasets, including astrophysical hydrodynamical data. yt is a dependency of Trident, so you must install it before Trident will work. There are several methods for installing yt, which are all discussed in detail in the yt installation documentation.

We find that the easiest way to install yt is with the all-in-one install script, which installs yt and its dependencies via a new conda installation:

```
$ wget https://raw.githubusercontent.com/yt-project/yt/master/doc/install_script.sh
$ ... edit the install_script.sh to mark INST_SCIPY=1, INST_ASTROPY=1,
$ ... and INST_YT_SOURCE=1
$ bash install_script.sh
$ ... update your path flag as described by the install_script.sh
```
Alternatively, if you already have conda installed, you can skip the commands above and just run the following command to get yt and its dependencies. To get the nightly build of the development version of yt, type:

```
$ conda install -c http://use.yt/with_conda/ -c conda-forge yt
```

### 1.3 Step 2: Install Trident

#### 1.3.1 Installing the Stable Release

You can install the most recent stable release of Trident using pip:

```
$ pip install trident
```

#### 1.3.2 Installing the Development Version

To get the development version, you'll pull the source code from its repository using git, which should be installed as part of your yt installation. If it isn't try: `conda install git`. After that, you'll use pip to install the source directly. The development version of Trident requires the yt-4.0 development version of yt as well. Go to your desired source code installation directory and run:

```
$ # install yt
$ git clone https://github.com/yt-project/yt --branch=yt-4.0
$ cd yt
$ pip install -e .
$ cd ..
$ # install trident
$ git clone https://github.com/trident-project/trident
$ cd trident
$ pip install -e .
```

### 1.4 Step 3: Get Ionization Table and Verify Installation

In order to calculate the ionization fractions for various ions from density, temperature, metallicity fields, you will need an ionization table datafile and a configuration file. Because this datafile can be large, it is not packaged with the main source code. The first time you try to do anything that requires it, Trident will attempt to automatically set this all up for you with a series of interactive prompts. **This step requires an internet connection the first time you run it.**

In addition, Trident provides a simple test function to verify that your install is functioning correctly. This function not only tries to set up your configuration and download your ion table file, but it will create a simple one-zone dataset, generate a ray through it, and create a spectrum from that ray. This should execute very quickly, and if it succeeds it demonstrates that your installation has been totally successful:

```
$ python
>>> import trident
>>> trident.verify()
...Series of Interactive Prompts...
```

If you cannot directly access the internet on this computer, or you lack write access to your `${HOME}` directory, or this step fails for any reason, please follow our documentation on [Manually Installing your Ionization Table](#).
1.5 Step 4: Science!

Congratulations, you’re now ready to use Trident! Please refer to the documentation for how to use it with your data or with one of our sample datasets. Please join our mailing list for announcements and when new features are added to the code.

1.6 Manually Installing your Ionization Table

If for some reason you are unable to install the config file and ionization table data automatically, you must set it up manually. When Trident runs, it looks for a configuration file called config.tri in the $HOME/.trident directory or alternatively in the current working directory (for users lacking write access to their $HOME directories). This configuration file is simple in that it tells Trident a few things about your install including the location and filename of your desired ionization table. Manually create a text file called config.tri with contents following the form:

```
[Trident]
ion_table_dir = ~/.trident
ion_table_file = hm2012_hr.h5
```

To manually obtain an ion table datafile, download and gunzip one from: http://trident-project.org/data/ion_table. While the config.tri file needs to exist in your $HOME/.trident directory or in the working directory when you import trident, the ion_table datafile can exist anywhere on the file system. Just assure that the config file points to the proper location and filename of the ion table datafile.

Now, to confirm everything is working properly, verify your installation following Step 3: Get Ionization Table and Verify Installation. If this fails or you have additional problems, please contact our mailing list.

1.7 Uninstallation or Switching Code Versions

Uninstallation of the Trident source code is easy. If you installed the stable version of the code via pip, just run:

```
$ pip uninstall trident
```

If you installed the dev version of Trident, you’ll have to delete the source as well:

```
$ pip uninstall trident
$ rm -rf </path/to/trident/repo>
```

If you want to switch between the two stable and development versions, just uninstall your version of the code as above, and then install the desired version as described in Step 2: Install Trident.

To fully remove the code from your system, remember to remove any ion table datafiles you may have downloaded in your $HOME/.trident directory, and follow the instructions for how to uninstall yt.
1.8 Updating to the Latest Version

If you want more recent features, you should periodically update your Trident codebase.

1.8.1 Updating to the Latest Stable Release

If you installed the “stable” version of the code using pip, then you can easily update your trident and yt installations:

```
$ pip install -U trident
$ yt update
```

1.8.2 Updating to the Latest Development Version

If you installed the “development” version of the code, it’s slightly more involved:

```
$ cd <path/to/trident/repo>
$ git pull origin master
$ pip install -e .
$ yt update
```

For more information on updating your yt installation, see the yt update instructions.
The best way to get a feel for what Trident can do is to go through an annotated example of its use. This section will walk you through the steps necessary to produce a synthetic spectrum based on simulation data and to view its path through the parent dataset. The following example, available in the source code itself, can be applied to datasets from any of the different simulation codes that Trident and yt support, although it may require some tweaking of parameters for optimal performance. If you want to recreate the following analysis with the exact dataset used, it can be downloaded here.

The basic process for generating a spectrum and overplotting a sightline’s trajectory through the dataset goes in three steps:

1. Generate a LightRay from the simulation data representing a sightline through the data.
2. Define the desired spectrum features and use the LightRay to create a corresponding synthetic spectrum.
3. Create a projected image and overplot the path of the LightRay.

### 2.1 Simple LightRay Generation

A LightRay is a 1D object representing the path a ray of light takes through a simulation volume on its way from some bright background object to the observer. It records all of the gas fields it intersects along the way for use in construction of a spectrum.

In order to generate a LightRay from your data, you need to first make sure that you've imported both the yt and Trident packages, and specify the filename of the dataset from which to extract the light ray:

```python
import yt
import trident
fn = 'enzo_cosmology_plus/RD0009/RD0009'
```

We need to decide the trajectory that the LightRay will take through our simulation volume. This arbitrary trajectory is specified with coordinates in code length units (e.g. \([x_{\text{start}}, y_{\text{start}}, z_{\text{start}}]\) to \([x_{\text{end}}, y_{\text{end}}, z_{\text{end}}]\)). Probably the simplest trajectory is cutting diagonally from the origin of the simulation volume to its outermost corner using the yt domain_left_edge and domain_right_edge attributes. Here we load the dataset into yt to get access to these attributes:

```python
ds = yt.load(fn)
ray_start = ds.domain_left_edge
ray_end = ds.domain_right_edge
```

Let’s define what lines or species we want to be added to our final spectrum. In this case, we want to deposit all hydrogen, carbon, nitrogen, oxygen, and magnesium lines to the resulting spectrum from the dataset:
line_list = ['H', 'C', 'N', 'O', 'Mg']

We can now generate the light ray using the `make_simple_ray` function by passing the dataset and the trajectory endpoints to it as well as telling trident to save the resulting ray dataset to an HDF5 file. We explicitly instruct trident to pull all necessary fields from the dataset in order to be able to add the lines from our `line_list`. Lastly, we set the `ftype` keyword as the field type of the fields where Trident will look to find density, temperature, and metallicity for building the required ion fields:

```python
ray = trident.make_simple_ray(ds,
                           start_position=ray_start,
                           end_position=ray_end,
                           data_filename="ray.h5",
                           lines=line_list,
                           ftype='gas')
```

**Warning:** It is imperative that you set the `ftype` keyword properly for your dataset. An `ftype` of ‘gas’ is adequate for grid-based codes, but not particle. Particle-based datasets must set `ftype` to the field type of their gas particles (e.g. ‘PartType0’) to assure that Trident builds the ion fields on the particles themselves before smoothing these fields to the grid. By not setting this correctly, you risk bad ion values by building from smoothed gas fields.

### 2.2 Overplotting a LightRay’s Trajectory on a Projection

Here we create a projection of the density field along the x axis of the dataset, and then overplot the path the `LightRay` takes through the simulation, before saving it to disk. The `annotate_ray()` operation should work for any volumetric plot, including slices, and off-axis plots:

```python
p = yt.ProjectionPlot(ds, 'x', 'density')
p.annotate_ray(ray, arrow=True)
p.save('projection.png')
```
2.3 Spectrum Generation

Now that we have our LightRay we can use it to generate a spectrum. To create a spectrum, we need to make a SpectrumGenerator object defining our desired wavelength range and bin size. You can do this by manually setting these features, or just using one of the presets for an instrument. Currently, we have three pre-defined instruments, the G130M, G160M, and G140L observing modes for the Cosmic Origins Spectrograph aboard the Hubble Space Telescope: COS-G130M, COS-G160M, and COS-G140L. Notably, instrument COS aliases to COS-G130M.

We then use this SpectrumGenerator object to make a raw spectrum according to the intersecting fields it encountered in the corresponding LightRay. We save this spectrum to disk, and plot it:

```python
sg = trident.SpectrumGenerator('COS-G130M')
sg.make_spectrum(ray, lines=line_list)
sg.save_spectrum('spec_raw.txt')
sg.plot_spectrum('spec_raw.png')
```
From here we can do some post-processing to the spectrum to include additional features that would be present in an actual observed spectrum. We add a background quasar spectrum, a Milky Way foreground, apply the COS line spread function, and add gaussian noise with SNR=30:

```python
sg.add_qso_spectrum()
sg.add_milky_way_foreground()
sg.apply_lsf()
sg.add_gaussian_noise(30)
```

Finally, we use plot and save the resulting spectrum to disk:

```python
sg.save_spectrum('spec_final.txt')
sg.plot_spectrum('spec_final.png')
```

which produces:

To create more complex or ion-specific spectra, refer to *Advanced Spectrum Generation*. 
2.4 Compound LightRays

In some cases (e.g. studying redshift evolution of the IGM), it may be desirable to create a LightRay that covers a range in redshift that is larger than the domain width of a single simulation snapshot. Rather than simply sampling the same dataset repeatedly, which is inherently unphysical since large scale structure evolves with cosmic time, Trident allows the user to create a ray that samples multiple datasets from different redshifts to produce a much longer ray that is continuous in redshift space. This is done by using the make_compound_ray function. This function is similar to the previously mentioned make_simple_ray function, but instead of accepting an individual dataset, it takes a simulation parameter file, the associated simulation type, and the desired range in redshift to be probed by the ray, while still allowing the user to specify the same sort of line list as before:

```python
fn = 'enzo_cosmology_plus/AMRCosmology.enzo'
ray = trident.make_compound_ray(fn, simulation_type='Enzo',
                                   near_redshift=0.0, far_redshift=0.1,
                                   ftype='gas',
                                   lines=line_list)
```

In this example, we’ve created a ray from an Enzo simulation (the same one used above) that goes from \( z = 0 \) to \( z = 0.1 \). This ray can now be used to generate spectra in the exact same ways as before.

Obviously, there need to be sufficient simulation outputs over the desired redshift range of the compound ray in order to have continuous sampling. To assure adequate simulation output frequency for this, one can use yt’s plan_cosmology_splice() function. See an example of its usage in the yt_astro_analysis documentation.

We encourage you to look at the detailed documentation for make_compound_ray in the API Reference section to understand how to control how the ray itself is constructed from the available data.

**Note:** The compound ray functionality has only been implemented for the Enzo and Gadget simulation codes. If you would like to help us implement this functionality for your simulation code, please contact us about this on the mailing list.
In addition to generating a basic spectrum as demonstrated in the *annotated example*, the user can also customize the generated spectrum in a variety of ways. One can choose which spectral lines to deposit or choose different settings for the characteristics of the spectrograph, and more. The following code goes through the process of setting these properties and shows what impact it has on resulting spectra.

For this demonstration, we’ll be using a light ray passing through a very dense disk of gas, taken from the initial output from the AGORA isolated box simulation using ART-II in Kim et al. (2016). If you’d like to try to reproduce the spectra included below you can get the LightRay file from the Trident sample data using the command:

```
$ wget http://trident-project.org/data/sample_data/ART-II_ray.h5
```

Now, we’ll load up the ray using yt:

```python
import yt
import trident
ray = yt.load('ART-II_ray.h5')
```

### 3.1 Setting the spectrograph

Let’s set the characteristics of the spectrograph we will use to create this spectrum. We can either choose the wavelength range and resolution and line spread function explicitly, or we can choose one of the preset instruments that come with Trident. To list the presets and their respective values, use this command:

```python
print(trident.valid_instruments)
```

Currently, we have three settings for the Cosmic Origins Spectrograph available: COS-G130M, COS-G140L, and COS-G160M, but we plan to add more instruments soon. To use one of them, we just use the name string in the SpectrumGenerator class:

```python
sg = trident.SpectrumGenerator('COS-G130M')
```

But instead, let’s just set our wavelength range manually from 1150 angstroms to 1250 angstroms with a resolution of 0.01 angstroms:

```python
sg = trident.SpectrumGenerator(lambda_min=1150, lambda_max=1250, dlambda=0.01)
```

From here, we can pass the ray to the `SpectrumGenerator` object to use in the construction of a spectrum.
3.2 Choosing what absorption features to include

There is a `LineDatabase` class that controls which spectral lines you can add to your spectrum. Trident provides you with a default `LineDatabase` with 213 spectral lines commonly used in CGM and IGM studies, but you can create your own `LineDatabase` with different lines. To see a list of all the lines included in the default line list:

```python
ldb = trident.LineDatabase('lines.txt')
print(ldb)
```

which is reading lines from the ‘lines.txt’ file present in the data directory (see where is Trident installed?) We can specify any subset of these spectral lines to use when creating the spectrum from our master line list. So if you’re interested in just looking at neutral hydrogen lines in your spectrum, you can see what lines will be included with the command:

```python
print(ldb.parse_subset('H I'))
```

As a first pass, we’ll create a spectrum that just include lines produced by hydrogen:

```python
sg.make_spectrum(ray, lines=['H'])
sg.plot_spectrum('spec_H.png')
```

The resulting spectrum contains a nice, big Lyman-alpha feature.

If, instead, we want to shows the lines that would be in our spectral range due to carbon, nitrogen, and oxygen, we can do the following:

```python
sg.make_spectrum(ray, lines=['C', 'N', 'O'])
going.plot_spectrum('spec_CNO.png')
```

And now we have:
We can see how these two spectra combined when we include all of the same lines:

```python
sg.make_spectrum(ray, lines=['H', 'C', 'N', 'O'])
sg.plot_spectrum('spec_HCNO.png')
```

which gives:

![Graph showing combined spectra](image)

We can get even more specific, by generating a spectrum that only contains lines due to a single ion species. For example, we might just want the lines from four-times-ionized nitrogen, N V:

```python
sg.make_spectrum(ray, lines=['N V'])
sg.plot_spectrum('spec_NV.png')
```

This spectrum only shows a couple of small lines on the right hand side.

![Graph showing spectrum with N V lines](image)
But if that level of specificity isn’t enough, we can request individual lines:

```python
sg.make_spectrum(ray, lines=['C I 1193', 'C I 1194'])
sg.plot_spectrum('spec_CI_1193_1194.png')
```

And we end up with:

Or we can just include all of the available lines in our `LineDatabase` with:

```python
sg.make_spectrum(ray, lines='all')
sg.plot_spectrum('spec_all.png')
```

Giving us:

To understand how to further customize your spectra, look at the documentation for the `SpectrumGenerator` and `LineDatabase` classes and other API documentation.

### 3.3 Setting Wavelength Bounds Automatically

If you are interested in creating a spectrum that contains all possible absorption features for a given set of lines, the `SpectrumGenerator` can be configured to automatically enlarge the wavelength window until all absorption is captured. This is done by setting the `lambda_min` and `lambda_max` keywords to ‘auto’ and specifying a bin size with the `dlambda` keyword:

```python
sg = trident.SpectrumGenerator(lambda_min='auto', lambda_max='auto',
                               dlambda=0.01)
```

(continues on next page)
Note, the above example is for a different ray than is used in the previous examples. The resulting spectrum will minimally contain all absorption present in the ray. This should be used with care when depositing multiple lines as this can lead to an extremely large spectrum.

### 3.4 Making Spectra in Velocity Space

Trident can be configured to create spectra in velocity space instead of wavelength space where velocity corresponds to the velocity offset from the rest wavelength of a given line. This can be done by providing the keyword `bin_space='velocity'` to the `SpectrumGenerator`:

```python
sg = trident.SpectrumGenerator(lambda_min='auto', lambda_max='auto', dlambda=1., bin_space='velocity')
sg.make_spectrum("ray.h5", lines=['H I 1216'])
sg.plot_spectrum('spec_velocity.png')
```

When working in velocity space, limits and bin sizes should be provided in km/s. If more than one transition is added to the spectrum (e.g., Ly-a and Ly-b), the zero point will correspond to the rest wavelength of the first transition added.
3.5 Making Spectra from a Subset of a Ray

The situation may arise where you want to see the spectrum that is generated by only a portion of the gas along a line of sight. For example, you may want to see the spectrum of only the cold gas. This can be done by creating a \texttt{YTCutRegion} from a loaded ray dataset:

```python
import trident
import yt

ds = yt.load('ray.h5')
all_data = ds.all_data()
cold_gas = ds.cut_region(all_data, 'obj["gas", "temperature"] < 10000')

sg = trident.SpectrumGenerator(lambda_min=1200, lambda_max=1225,
dlambd=0.01)

# spectrum of entire ray
sg.make_spectrum(all_data, lines=['H I 1216'])
all_spectrum = sg.flux_field[:]

# spectrum of cold gas
sg.make_spectrum(cold_gas, lines=['H I 1216'])
cold_spectrum = sg.flux_field[:]

trident.plot_spectrum(sg.lambda_field, [all_spectrum, cold_spectrum],
                      label=['all gas', 'cold gas'], stagger=None)
```

![Plot of spectra for all gas and cold gas](image_url)
In addition to being able to create absorption spectra, Trident can be used to postprocess datasets to add fields for ions not explicitly tracked in the simulation. These can later be analyzed using the standard yt analysis packages. This page provides some examples as to how these fields can be generated and analyzed.

### 4.1 How does it work?

When you installed Trident, you were forced to download an ion table, a data table consisting of dimensions in density, temperature, and redshift. This ion table was constructed by running many independent Cloudy instances to approximate the ionization states of all ionic species of the first 30 elements. The ionic species were calculated assuming collisional ionization equilibrium based on different density and temperature values and photoionization from a metagalactic ultraviolet background unique to each ion table. The currently preferred ion table uses the Haardt Madau 2012 model. You can change your default ionization model by changing your config file (see: *Manually Installing your Ionization Table*), or by specifying it directly in the `ionization_table` keywords of the following functions.

By following the process below, you will add different ion fields to your dataset based on the above assumptions using the dataset’s redshift, and the values of density, temperature, and metallicity found for each gas parcel in your dataset.

### 4.2 Generating species fields

As always, we first need to import yt and Trident and then we load up a dataset:

```python
import yt
import trident
fn = 'enzo_cosmology_plus/RD0009/RD0009'
ds = yt.load(fn)
```

To add ion fields we use the `add_ion_fields` function. This will add fields for whatever ions we specify in the form of:

- Ion fraction field. e.g. `Mg_p1_ion_fraction`
- Number density field. e.g. `Mg_p1_number_density`
- Density field. e.g. `Mg_p1_density`
- Mass field. e.g. `Mg_p1_mass`
Note: Trident follows yt’s naming convention for atomic, molecular, and ionic species fields. In short, the ionic prefix consists of the element and the number of times ionized it is: e.g. H I = H_p0, Mg II = Mg_p1, O VI = O_p5 (p is for plus).

Let’s add fields for O VI (five-times-ionized oxygen):

```python
trident.add_ion_fields(ds, ions=['O VI'], ftype="gas")
```

Warning: Make sure you are using the appropriate value for `ftype` in adding your ion fields to a dataset. To get best results, the ion interpolation must take place on the gas fields provided by your simulation output. For grid-based codes, these fields are typically aliased to the gas fields (e.g., (“gas”, “density”)), so using the default `ftype="gas"` is fine. But for particle-based codes, this is not usually the case, and the particle-based gas fields differ based on the code (e.g. PartType0, Gas, etc.). Inspection of your dataset may be necessary (print(ds. field_list)). Set ftype correctly to make sure ion generation takes place on the particle first, before being deposited to the grid-based fields, or you may get incorrect results.

To show how one can use this newly generated field, we’ll make a projection of the O VI number density field to show its column density map:

```python
proj = yt.ProjectionPlot(ds, "z", "O_p5_number_density")
proj.save()
```

which produces:
We can similarly create a phase plot to show where the O VI mass lives as a function of density and temperature:

```python
# we need to create a data object from the dataset to make a phase plot
ad = ds.all_data()
phase = yt.PhasePlot(ad, "density", "temperature", ["O_p5_mass"],
                      weight_field="O_p5_mass", fractional=True)
phase.save()
```

resulting in:
These internal classes and related extensions used to be part of yt but are now contained within Trident. The internal classes are documented below and can be used independently, but the primary Trident interfaces outlined in the main documentation are recommended.

### 5.1 Internal Classes

#### 5.1.1 Light Ray Generator

The *LightRay* is the one-dimensional object representing the pencil beam of light traveling from the source to the observer. Light rays can stack multiple datasets together to span a redshift interval larger than the simulation box.
A ray segment records the information of all grid cells intersected by the ray as well as the path length, $d_l$, of the ray through the cell. Column densities can be calculated by multiplying physical densities by the path length.

### Configuring the Light Ray Generator

Below follows the creation of a light ray from multiple datasets stacked together. The primary Trident interface to this is covered in *Compound LightRays*. A light ray can also be made from a single dataset. For information on this, see *Light Rays Through Single Datasets*.

The arguments required to instantiate a `LightRay` are the simulation parameter file, the simulation type, the nearest redshift, and the furthest redshift.

```python
from trident import LightRay
lr = LightRay("enzo_tiny_cosmology/32Mpc_32.enzo",
               simulation_type="Enzo",
               near_redshift=0.0, far_redshift=0.1)
```

### Making Light Ray Data

Once the LightRay object has been instantiated, the `make_light_ray()` function will trace out the rays in each dataset and collect information for all the fields requested. The output file will be an yt-loadable dataset containing all the cell field values for all the cells that were intersected by the ray. A single LightRay object can be used over and over to make multiple randomizations, simply by changing the value of the random seed with the `seed` keyword.

```python
lr.make_light_ray(seed=8675309,
                  fields=['temperature', 'density'],
                  use_peculiar_velocity=True)
```

# Optionally, we can now overplot the part of this ray that intersects
# one output from the source dataset in a ProjectionPlot

ds = yt.load('enzo_tiny_cosmology/RD0004/RD0004')
p = yt.ProjectionPlot(ds, 'z', 'density')
p.annotate_ray(lr)
p.save()

### Light Rays Through Single Datasets

LightRays can also be configured for use with single datasets. In this case, one must specify the ray’s trajectory explicitly. The main Trident interface to this functionality is covered in *Simple LightRay Generation*.

```python
from trident import LightRay
import yt

ds = yt.load("IsolatedGalaxy/galaxy0030/galaxy0030")
lr = LightRay(ds)

lr.make_light_ray(start_position=ds.domain_left_edge,
                  end_position=ds.domain_right_edge,
                  solution_filename='lightraysolution.txt',
                  data_filename='lightray.h5',
                  fields=['temperature', 'density'])

# Overplot the ray on a projection.
```
Alternately, the `trajectory` keyword can be used in place of `end_position` to specify the (r, theta, phi) direction of the ray.

**Useful Tips for Making LightRays**

Below are some tips that may come in handy for creating proper LightRays.

**How many snapshots do I need?**

The number of snapshots required to traverse some redshift interval depends on the simulation box size and cosmological parameters. Before running an expensive simulation only to find out that you don’t have enough outputs to span the redshift interval you want, have a look at the guide *Planning Simulations for LightCones* or *LightRays*. The functionality described there will allow you to calculate the precise number of snapshots and specific redshifts at which they should be written.

**My snapshots are too far apart!**

The `max_box_fraction` keyword, provided when creating the Lightray, allows the user to control how long a ray segment can be for an individual dataset. Be default, the LightRay generator will try to make segments no longer than the size of the box to avoid sampling the same structures more than once. However, this can be increased in the case that the redshift interval between datasets is longer than the box size. Increasing this value should be done with caution as longer ray segments run a greater risk of coming back to somewhere near their original position.

**What if I have a zoom-in simulation?**

A zoom-in simulation has a high resolution region embedded within a larger, low resolution volume. In this type of simulation, it is likely that you will want the ray segments to stay within the high resolution region. To do this, you must first specify the size of the high resolution region when creating the LightRay using the `max_box_fraction` keyword. This will make sure that the calculation of the spacing of the segment datasets only takes into account the high resolution region and not the full box size. If your high resolution region is not a perfect cube, specify the smallest side. Then, in the call to `make_light_ray()`, use the `left_edge` and `right_edge` keyword arguments to specify the precise location of the high resolution region.

Technically speaking, the ray segments should no longer be periodic since the high resolution region is only a sub-volume within the larger domain. To make the ray segments non-periodic, set the `periodic` keyword to False. The LightRay generator will continue to generate randomly oriented segments until it finds one that fits entirely within the high resolution region. If you have a high resolution region that can move and change shape slightly as structure forms, use the `min_level` keyword to mandate that the ray segment only pass through cells that are refined to at least some minimum level.

If the size of the high resolution region is not large enough to span the required redshift interval, the LightRay generator can be configured to treat the high resolution region as if it were periodic simply by setting the `periodic` keyword to True. This option should be used with caution as it will lead to the creation of disconnected ray segments within a single dataset.
I want a continuous trajectory over the entire ray.

Set the `minimum_coherent_box_fraction` keyword argument to a very large number, like infinity (`numpy.inf`).

### 5.1.2 AbsorptionSpectrum

For documentation on the main interface to spectrum creation in Trident, see *Spectrum Generation*.

The `AbsorptionSpectrum` is the internal class for creating absorption spectra in Trident from `LightRay` objects. The `AbsorptionSpectrum` and its workhorse method `make_spectrum()` return two arrays, one with wavelengths, the other with the normalized flux values at each of the wavelength values. It can also output a text file listing all important lines.

#### Method for Creating Absorption Spectra

Once a `LightRay` has been created traversing a dataset using the `Light Ray Generator`, a series of arrays store the various fields of the gas parcels (represented as cells) intersected along the ray. `AbsorptionSpectrum` steps through each element of the `LightRay`'s arrays and calculates the column density for desired ion by multiplying its number density with the path length through the cell. Using these column densities along with temperatures to calculate thermal broadening, voigt profiles are deposited on to a featureless background spectrum. By default, the peculiar velocity of the gas is included as a doppler redshift in addition to any cosmological redshift of the data dump itself.

#### Subgrid Deposition

For features not resolved (i.e. possessing narrower width than the spectral resolution), `AbsorptionSpectrum` performs subgrid deposition. The subgrid deposition algorithm creates a number of smaller virtual bins, by default the width of the virtual bins is 1/10th the width of the spectral feature. The Voigt profile is then deposited into these virtual bins where it is resolved, and then these virtual bins are numerically integrated back to the resolution of the original spectral bin size, yielding accurate equivalent widths values. `AbsorptionSpectrum` informs the user how many spectral features are deposited in this fashion.

#### Creating an Absorption Spectrum

#### Initialization

To instantiate an `AbsorptionSpectrum` object, the arguments required are the minimum and maximum wavelengths (assumed to be in Angstroms), and the number of wavelength bins to span this range (including the endpoints)
Adding Features to the Spectrum

Absorption lines and continuum features can then be added to the spectrum. To add a line, you must know some properties of the line: the rest wavelength, f-value, gamma value, and the atomic mass in amu of the atom. That line must be tied in some way to a field in the dataset you are loading, and this field must be added to the LightRay object when it is created. Below, we will add the H Lyman-alpha line, which is tied to the neutral hydrogen field ('H_p0_number_density').

```python
my_label = 'HI Lya'
field = 'H_p0_number_density'
wavelength = 1215.6700  # Angstroms
f_value = 4.164E-01
gamma = 6.265e+08
mass = 1.00794
sp.add_line(my_label, field, wavelength, f_value, gamma, mass, label_threshold=1.e10)
```

In the the call to `add_line()` the field argument tells the spectrum generator which field from the ray data to use to calculate the column density. The label_threshold keyword tells the spectrum generator to add all lines above a column density of $10^{10}$ cm$^{-2}$ to the text line list output at the end. If None is provided, as is the default, no lines of this type will be added to the text list.

Continuum features with optical depths that follow a power law can be added with the `add_continuum()` function. Like adding lines, you must specify details like the wavelength and the field in the dataset and LightRay that is tied to this feature. The wavelength refers to the location at which the continuum begins to be applied to the dataset, and as it moves to lower wavelength values, the optical depth value decreases according to the defined power law. The normalization value is the column density of the linked field which results in an optical depth of 1 at the defined wavelength. Below, we add the hydrogen Lyman continuum.

```python
my_label = 'HI Lya'
field = 'H_p0_number_density'
wavelength = 912.323660  # Angstroms
normalization = 1.6e17
index = 3.0
sp.add_continuum(my_label, field, wavelength, normalization, index)
```

Making the Spectrum

Once all the lines and continuums are added, the spectrum is made with the `make_spectrum()` function.

```python
wavelength, flux = sp.make_spectrum('lightray.h5',
                                   output_file='spectrum.fits',
                                   line_list_file='lines.txt')
```

A spectrum will be made using the specified ray data and the wavelength and flux arrays will also be returned. If you set the optional use_peculiar_velocity keyword to False, the lines will not incorporate doppler redshifts to shift the deposition of the line features.

Three output file formats are supported for writing out the spectrum: fits, hdf5, and ascii. The file format used is based on the extension provided in the output_file keyword: .fits for a fits file, .h5 for an hdf5 file, and anything else for an ascii file.

**Note:** To write out a fits file, you must install the astropy python library in order to access the astropy.io.fits module.
You can usually do this by simply running `pip install astropy` at the command line.

### Generating Spectra in Parallel

Spectrum generation is parallelized using a multi-level strategy where each absorption line is deposited by a different processor. If the number of available processors is greater than the number of lines, then the deposition of individual lines will be divided over multiple processors.

Absorption spectrum creation can be run in parallel simply by adding the following to the top of the script and running with `mpirun`.

```python
import yt
yt.enable_parallelism()
```

For more information on parallelism in yt, see Parallel Computation With yt.

### 5.2 Extensions

#### 5.2.1 Fitting Absorption Spectra

This tool can be used to fit absorption spectra, particularly those generated using the `AbsorptionSpectrum`. For more details on its uses and implementation please see (Egan et al. (2013)). If you find this tool useful we encourage you to cite accordingly.

#### Loading an Absorption Spectrum

To load an absorption spectrum created by `AbsorptionSpectrum`, specify the output file name. It is advisable to use either an .h5 or .fits file, rather than an ascii file to save the spectrum as rounding errors produced in saving to a ascii file will negatively impact fit quality.

```python
f = h5py.File('spectrum.h5')
wavelength = f['wavelength'][:]
flux = f['flux'][:]
f.close()
```

#### Specifying Species Properties

Before fitting a spectrum, you must specify the properties of all the species included when generating the spectrum. The physical properties needed for each species are the rest wavelength, f-value, gamma value, and atomic mass. These will be the same values as used to generate the initial absorption spectrum. These values are given in list form as some species generate multiple lines (as in the OVI doublet). The number of lines is also specified on its own.

To fine tune the fitting procedure and give results in a minimal number of optimizing steps, we specify expected maximum and minimum values for the column density, doppler parameter, and redshift. These values can be well outside the range of expected values for a typical line and are mostly to prevent the algorithm from fitting to negative values or becoming numerically unstable.

Common initial guesses for doppler parameter and column density should also be given. These values will not affect the specific values generated by the fitting algorithm, provided they are in a reasonably appropriate range (ie: within the range given by the max and min values for the parameter).
For a spectrum containing both the H Lyman-alpha line and the OVI doublet, we set up a fit as shown below.

```python
HI_parameters = {'name': 'HI',
                 'f': [.4164],
                 'Gamma': [6.265E8],
                 'wavelength': [1215.67],
                 'numLines': 1,
                 'maxN': 1E22, 'minN': 1E11,
                 'maxb': 300, 'minb': 1,
                 'maxz': 6, 'minz': 0,
                 'init_b': 30,
                 'init_N': 1E14}

OVI_parameters = {'name': 'OVI',
                  'f': [.1325, .06580],
                  'Gamma': [4.148E8, 4.076E8],
                  'wavelength': [1031.9261, 1037.6167],
                  'numLines': 2,
                  'maxN': 1E17, 'minN': 1E11,
                  'maxb': 300, 'minb': 1,
                  'maxz': 6, 'minz': 0,
                  'init_b': 20,
                  'init_N': 1E12}

speciesDicts = {'HI': HI_parameters, 'OVI': OVI_parameters}
```

**Generating Fit of Spectrum**

After loading a spectrum and specifying the properties of the species used to generate the spectrum, an appropriate fit can be generated.

```python
from trident.absorption_spectrum.absorption_spectrum_fit import generate_total_fit

orderFits = ['OVI', 'HI']

fitted_lines, fitted_flux = generate_total_fit(wavelength, flux, orderFits, speciesDicts)
```

The orderFits variable is used to determine in what order the species should be fitted. This may affect the results of the resulting fit, as lines may be fit as an incorrect species. For best results, it is recommended to fit species the generate multiple lines first, as a fit will only be accepted if all of the lines are fit appropriately using a single set of parameters. At the moment no cross correlation between lines of different species is performed.

The parameters of the lines that are needed to fit the spectrum are contained in the `fitted_lines` variable. Each species given in orderFits will be a key in the `fitted_lines` dictionary. Each entry for each species key will be another dictionary containing entries for ‘N’, ‘b’, ‘z’, and ‘group#’ which are the column density, doppler parameter, redshift, and associate line complex respectively. The `i`^th^ line of a given species is then given by the parameters `N[i]`, `b[i]`, and `z[i]` and is part of the same complex (and was fitted at the same time) as all lines with the same group number as `group#[i]`.

The `fitted_flux` is an ndarray of the same size as `flux` and `wavelength` that contains the cumulative absorption spectrum generated by the lines contained in `fitted_lines`. 

5.2. Extensions 29
Saving a Spectrum Fit

Saving the results of a fitted spectrum for further analysis is accomplished automatically using the h5 file format. A group is made for each species that is fit, and each species group has a group for the corresponding N, b, z, and group# values.

Procedure for Generating Fits

To generate a fit for a spectrum `generate_total_fit()` is called. This function controls the identification of line complexes, the fit of a series of absorption lines for each appropriate species, checks of those fits, and returns the results of the fits.

Finding Line Complexes

Line complexes are found using the `_find_complexes` function. The process by which line complexes are found involves walking through the array of flux in order from minimum to maximum wavelength, and finding series of spatially contiguous cells whose flux is less than some limit. These regions are then checked in terms of an additional flux limit and size. The bounds of all the passing regions are then listed and returned. Those bounds that cover an exceptionally large region of wavelength space will be broken up if a suitable cut point is found. This method is only appropriate for noiseless spectra.

The optional parameter `complexLim` (default = 0.999), controls the limit that triggers the identification of a spatially contiguous region of flux that could be a line complex. This number should be very close to 1 but not exactly equal. It should also be at least an order of magnitude closer to 1 than the later discussed `fitLim` parameter, because a line complex where the flux of the trough is very close to the flux of the edge can be incredibly unstable when optimizing.

The `fitLim` parameter controls what is the maximum flux that the trough of the region can have and still be considered a line complex. This effectively controls the sensitivity to very low column absorbers. Default value is `fitLim = 0.99`. If a region is identified where the flux of the trough is greater than this value, the region is simply ignored.

The `minLength` parameter controls the minimum number of array elements that an identified region must have. This value must be greater than or equal to 3 as there are a minimum of 3 free parameters that must be fit. Default is `minLength = 3`.

The `maxLength` parameter controls the maximum number of array elements that an identified region can have before it is split into separate regions. Default is `maxLength = 1000`. This should be adjusted based on the resolution of the spectrum to remain appropriate. The value correspond to a wavelength of roughly 50 angstroms.

The `splitLim` parameter controls how exceptionally large regions are split. When such a region is identified by having more array elements than `maxLength`, the point of maximum flux (or minimum absorption) in the middle two quartiles is identified. If that point has a flux greater than or equal to `splitLim`, then two separate complexes are created: one from the lower wavelength edge to the minimum absorption point and the other from the minimum absorption point to the higher wavelength edge. The default value is `splitLim = .99`, but it should not drastically affect results, so long as the value is reasonably close to 1.
Fitting a Line Complex

After a complex is identified, it is fitted by iteratively adding and optimizing a set of Voigt Profiles for a particular species until the region is considered successfully fit. The optimizing is accomplished using scipy’s least squares optimizer. This requires an initial estimate of the parameters to be fit (column density, b-value, redshift) for each line.

Each time a line is added, the guess of the parameters is based on the difference between the line complex and the fit so far. For the first line this just means the initial guess is based solely on the flux of the line complex. The column density is given by the initial column density given in the species parameters dictionary. If the line is saturated (some portion of the flux with a value less than .1) than the larger initial column density guess is chosen. If the flux is relatively high (all values >.9) than the smaller initial guess is given. These values are chosen to make optimization faster and more stable by being closer to the actual value, but the final results of fitting should not depend on them as they merely provide a starting point.

After the parameters for a line are optimized for the first time, the optimized parameters are then used for the initial guess on subsequent iterations with more lines.

The complex is considered successfully fit when the sum of the squares of the difference between the flux generated from the fit and the desired flux profile is less than $\text{errBound}$. $\text{errBound}$ is related to the optional parameter to $\text{generate_total_fit()}$ maxAvgError by the number of array elements in the region such that $\text{errBound} = \text{number of elements} \times \text{maxAvgError}$.

There are several other conditions under which the cycle of adding and optimizing lines will halt. If the error of the optimized fit from adding a line is an order of magnitude worse than the error of the fit without that line, then it is assumed that the fitting has become unstable and the latest line is removed. Lines are also prevented from being added if the total number of lines is greater than the number of elements in the flux array being fit divided by 3. This is because there must not be more free parameters in a fit than the number of points to constrain them.

Checking Fit Results

After an acceptable fit for a region is determined, there are several steps the algorithm must go through to validate the fits.

First, the parameters must be in a reasonable range. This is a check to make sure that the optimization did not become unstable and generate a fit that diverges wildly outside the region where the fit was performed. This way, even if particular complex cannot be fit, the rest of the spectrum fitting still behaves as expected. The range of acceptability for each parameter is given in the species parameter dictionary. These are merely broad limits that will prevent numerical instability rather than physical limits.

In cases where a single species generates multiple lines (as in the OVI doublet), the fits are then checked for higher wavelength lines. Originally the fits are generated only considering the lowest wavelength fit to a region. This is because we perform the fitting of complexes in order from the lowest wavelength to the highest, so any contribution to a complex being fit must come from the lower wavelength as the higher wavelength contributions would already have been subtracted out after fitting the lower wavelength.

Saturated Lyman Alpha Fitting Tools

In cases where a large or saturated line (there exists a point in the complex where the flux is less than .1) fails to be fit properly at first pass, a more robust set of fitting tools is used to try and remedy the situation. The basic approach is to simply try a much wider range of initial parameter guesses in order to find the true optimization minimum, rather than getting stuck in a local minimum. A set of hard coded initial parameter guesses for Lyman alpha lines is given by the _get_test_lines function. Also included in these parameter guesses is an an initial guess of a high column cool line overlapping a lower column warm line, indicative of a broad Lyman alpha (BLA) absorber.
We maintain a series of tests in Trident to make sure the code gives consistent results and to catch accidental breakages in our source code and dependencies. These tests are run by Travis automatically and regularly to assure consistency in functionality, but you can run them locally too (see below). The tests consist of a mix of unit tests (tests to assure Trident functions don’t actively fail) and answer tests (tests comparing newly generated results against some old established results to assure consistency).

6.1 Running the Test Suite

Running the test suite requires a version of Trident installed from source (see Installing the Development Version).

The tests are run using the pytest Python module. This can be installed with conda or pip.

```
$ conda install pytest
```

The test suite requires a number of datasets for testing functionality. Trident comes with a helper script that will download all the datasets and untar them. Before running this, make sure you have the answer_test_data_dir variable set in your config file (see Step 3: Get Ionization Table and Verify Installation). This variable should point to a directory where these datasets will be stored. The helper script is located in the tests directory of the Trident source.

```
$ cd tests
$ python download_test_data.py
```

If this is your first time running the tests, then you need to generate a “gold standard” for the answer tests. Follow Generating Gold Standard Answer Test Results for Comparison before continuing with running the tests, otherwise your answer tests will fail.

Make sure you’re on the desired version of yt and trident that you want to test and use (usually the tip of the development branch i.e., master).

```
$ export TRIDENT_GENERATE_TEST_RESULTS=0
$ cd /path/to/yt/
$ git checkout master
$ pip install -e .
$ cd /path/to/trident
$ git checkout master
$ pip install -e .
```

The test suite is run by calling py.test from within the tests directory.
If a test fails for some reason, you will be given a detailed traceback and reason for it failing. You can use this to identify what is wrong with your source or perhaps a change in the code of your dependencies. The tests should take around ten minutes to run.

### 6.2 Generating Gold Standard Answer Test Results for Comparison

In order to assure the Trident codebase gives consistent results over time, we compare the outputs of tests of new versions of Trident against an older, vetted version of the code we think gives accurate results. To create this “gold standard” result from the older version of the code, you must roll back the Trident and yt source back to the older “trusted” versions of the code. You can find the tags for the most recent trusted versions of the code by running gold_standard_versions.py and then rebuilding yt and Trident with these versions of the code. Lastly, set the TRIDENT_GENERATE_TEST_RESULTS environment variable to 1 and run the tests:

```bash
$ cd tests
$ python gold_standard_versions.py

Latest Gold Standard Commit Tags
yt = 953248239966
Trident = test-standard-v2

To update to them, `git checkout <tag>` in appropriate repository

$ cd /path/to/yt
$ git checkout 953248239966
$ pip install -e .
$ cd /path/to/trident
$ git checkout test-standard-v2
$ pip install -e .
$ export TRIDENT_GENERATE_TEST_RESULTS=1
$ cd tests
$ py.test
```

The test results should now be stored in the answer_test_data_dir that you specified in your Trident configu-
ration file. You may now run the actual tests (see *Running the Test Suite*) with your current version of yt and Trident comparing against these gold standard results.

### 6.3 The Tests Failed – What Do I Do?

If the tests have failed (either locally, or through the automatically generated test from Travis), you want to figure out what caused the breakage. It was either a change in trident or a change in one of Trident’s dependencies (e.g., yt). So first examine the output from `py.test` to see if you can deduce what went wrong.

Sometimes it isn’t obvious what caused the break, in which case you may need to use `git bisect` to track down the change, either in Trident or in yt. First, start with the tip of yt, and bisect the changes in Trident since its gold standard version (see below). If that doesn’t ID the bad changeset, then do the same with yt back to its gold standard version. Once you have identified the specific commit that caused the tests to break, you have to identify if it was a good or bad change. If the unit tests failed and some functionality no longer works, then it was a bad, and you’ll want to change the code that caused the break. On the other hand, if the answer tests changed, and they did so because of an improvement to the code, then you’ll simply want to go about *Updating the Testing Gold Standard*.

### 6.4 Updating the Testing Gold Standard

Periodically, the gold standard for our answer tests must be updated as bugs are caught or new more accurate behavior is enabled that causes the answer tests to fail. The first thing to do is to identify the most accurate version of the code (e.g., changesets for yt and trident that give the desired behavior). Tag the Trident changeset with the next gold standard iteration. You can see the current iteration by looking in the `.travis.yml` file at the `TRIDENT_GOLD` entry–increment this and tag the changeset. Update the `.travis.yml` file so that the `YT_GOLD` and `TRIDENT_GOLD` entries point to your desired changeset and tag. You have to explicitly push the new tag (here-after `test-standard-v3`) to your repository (here: `origin`). Issue a pull request.

```
$ git tag test-standard-v3 <trident-changeset>
$ ... edit .travis.yml files to update YT_GOLD=<yt changeset>
$ ... and TRIDENT_GOLD=test-standard-v3
$ git add .travis.yml
$ git commit
$ git push origin test-standard-v3
$ <MAKE PULL REQUEST>
```

Once the pull request has been accepted, someone with admin access to the main trident repository (here `upstream`) will have to push the gold standard tag.

```
$ git push upstream test-standard-v3
```

Lastly, that person will have to also clear Travis’ cache, so that it regenerates new answer test results. This can be done manually here: https://travis-ci.org/trident-project/trident/caches .
CHAPTER
SEVEN

FREQUENTLY ASKED QUESTIONS

7.1 Why don’t I have any absorption features in my spectrum?

There are many reasons you might not have any absorption features in your spectrum, but we’ll cover a few of the basic explanations here.

1. Your absorbers are in a different part of the spectrum than you’re plotting. Make sure you are plotting the wavelength range where you expect to see the absorption by taking into account the wavelength of your absorption features coupled with the redshift of your dataset: \( \lambda_{obs} = (1 + z)\lambda_{rest} \). To see the wavelength of specific ionic transitions, see the line list in: `/trident/trident/data/line_lists/lines.txt`.

2. Your sightlines do not have sufficient column densities of the desired ions to actually make an absorption feature. Look at the total column density of your desired ions in your sightline by multiplying the density times the path length and summing it all up. Here is an example for showing the total O VI column density in a ray:

```python
import trident
<generate/load your ray object>
trident.add_ion_fields(ray, ['O VI'])
ad = ray.all_data()
print((ad[('gas', 'O_p5_number_density')] * ad[('gas', 'dl')]).sum())
```

Depending on the ion, you usually need to see at least \(10^{12} cm^{-2}\) to have any appreciable absorption. Try sending a sightline through a denser region in your simulation that might have more of that ion.

7.2 What version of Trident am I running?

To learn what version of Trident you’re running, type:

```bash
$ python
>>> import trident
>>> print(trident.__version__)
```

If you have a version ending in dev, it means you’re on the development branch and you should also figure out which particular changeset you’re running. You can do this by:

```bash
$ cd <path/to/trident>
$ git log --pretty=format:'%h' -n 1
```

To figure out what version of yt you’re running, type:

```bash
$ yt version
```
If you’re writing to the mailing list with a problem, be sure to include all of the above with your bug report or question.

7.3 Where is Trident installed? Where are its data files?

One can easily identify where Trident is installed:

```
$ python
>>> import trident
>>> print(trident.path)
```

The data files are located in that path with an appended /data.

7.4 How do I join the mailing list?

You can join our mailing list for announcements, bugs reports, and changes at:

https://groups.google.com/forum/#!forum/trident-project-users

7.5 How do I learn more about the algorithms used in Trident?

We have a full description of all the methods used in Trident including citations to previous related works in our Trident method paper.

7.6 How do I cite Trident in my research?

Check out our citation page.

7.7 How do I get an invite to the Trident slack channel?

Click on this link.
8.1 Generating Rays

**make_simple_ray**
Create a yt LightRay object for a single dataset (eg CGM).

**make_compound_ray**
Create a yt LightRay object for multiple consecutive datasets (eg IGM).

**LightRay**
A 1D object representing the path of a light ray passing through a simulation.

### 8.1.1 trident.make_simple_ray

```
trident.make_simple_ray(dataset_file, start_position, end_position, lines=None, ftype='gas', fields=None, solution_filename=None, data_filename=None, trajectory=None, redshift=None, field_parameters=None, setup_function=None, load_kwargs=None, line_database=None, ionization_table=None)
```

Create a yt LightRay object for a single dataset (eg CGM). This is a wrapper function around yt’s LightRay interface to reduce some of the complexity there.

A simple ray is a straight line passing through a single dataset where each gas cell intersected by the line is sampled for the desired fields and stored. Several additional fields are created and stored including \(d_l\) to represent the path length in space for each element in the ray, \(v_{\text{los}}\) to represent the line of sight velocity along the ray, and \(\text{redshift, redshift\_dopp, and redshift\_eff}\) to represent the cosmological redshift, doppler redshift and effective redshift (combined doppler and cosmological) for each element of the ray.

A simple ray is typically specified by its start and end positions in the dataset volume. Because a simple ray only probes a single output, it lacks foreground absorbers between the observer at \(z=0\) and the redshift of the dataset that one would naturally encounter. Thus it is usually only appropriate for studying the circumgalactic medium rather than the intergalactic medium.

This function can accept a yt dataset already loaded in memory, or it can load a dataset if you pass it the dataset’s filename and optionally any load\_kwargs or setup\_function necessary to load/process it properly before generating the LightRay object.

The :lines: keyword can be set to automatically add all fields to the resulting ray necessary for later use with the SpectrumGenerator class. If the necessary fields do not exist for your line of choice, they will be added to your dataset before adding them to the ray.

**Parameters**

- **Dataset_file**  string or yt Dataset object
Either a yt dataset or the filename of a dataset on disk. If you are passing it a filename, consider usage of the `load_kwargs` and `setup_function` kwargs.

**Start_position, end_position** list of floats or YTArray object

The coordinates of the starting and ending position of the desired ray. If providing a raw list, coordinates are assumed to be in code length units, but if providing a YTArray, any units can be specified.

**Lines** list of strings, optional

List of strings that determine which fields will be added to the ray to support line deposition to an absorption line spectrum. List can include things like “C”, “O VI”, or “Mg II ####”, where #### would be the integer wavelength value of the desired line. If set to ‘all’, includes all possible ions from H to Zn. :lines: can be used in conjunction with :fields: as they will not override each other. Default: None

**Ftype** string, optional

This is now deprecated and unnecessary. Default: “gas”

**Fields** list of strings, optional

The list of which fields to store in the output LightRay. See :lines: keyword for additional functionality that will add fields necessary for creating absorption line spectra for certain line features. Default: None

**Solution_filename** string, optional

Output filename of text file containing trajectory of LightRay through the dataset. Default: None

**Data_filename** string, optional

Output filename for ray data stored as an HDF5 file. Note that at present, you must save a ray to disk in order for it to be returned by this function. If set to None, defaults to ‘ray.h5’. Default: None

**Trajectory** list of floats, optional

The (r, theta, phi) direction of the LightRay. Use either end_position or trajectory, but not both. Default: None

**Redshift** float, optional

Sets the highest cosmological redshift of the ray. By default, it will use the cosmological redshift of the dataset, if set, and if not set, it will use a redshift of 0. Default: None

**Field_parameters** optional, dict

Used to set field parameters in light rays. For example, if the ‘bulk_velocity’ field parameter is set, the relative velocities used to calculate peculiar velocity will be adjusted accordingly. Default: None

**Setup_function** function, optional

A function that will be called on the dataset as it is loaded but before the LightRay is generated. Very useful for adding derived fields and other manipulations of the dataset prior to LightRay creation. Default: None

**Load_kwargs** dict, optional

Dictionary of kwargs to be passed to the yt “load” function prior to creating the LightRay. Very useful for many frontends like Gadget, Tipsy, etc. for passing in “bounding_box”, “unit_base”, etc. Default: None
Line_database string, optional

For use with the :lines: keyword. If you want to limit the available ion fields to be added to those available in a particular subset, you can use a LineDatabase. This means when you set :lines:=‘all’, it will only use those ions present in the corresponding LineDatabase. If :LineDatabase: is set to None, and :lines:=‘all’, it will add every ion of every element up to Zinc. Default: None

Ionization_table string, optional

For use with the :lines: keyword. Path to an appropriately formatted HDF5 table that can be used to compute the ion fraction as a function of density, temperature, metallicity, and redshift. When set to None, it uses the table specified in ~/.trident/config Default: None

Example

Generate a simple ray passing from the lower left corner to the upper right corner through some Gizmo dataset:

```python
>>> import trident
>>> import yt
>>> ds = yt.load('path/to/dataset')
>>> ray = trident.make_simple_ray(ds,
... start_position=ds.domain_left_edge, end_position=ds.domain_right_edge,
... lines=['H', 'O', 'Mg II'])
```

8.1.2 trident.make_compound_ray

trident.make_compound_ray (parameter_filename, simulation_type, near_redshift, far_redshift, lines=None, ftype='gas', fields=None, solution_filename=None, data_filename=None, use_minimum_datasets=True, max_box_fraction=1.0, deltaz_min=0.0, minimum_coherent_box_fraction=0.0, find_outputs=False, seed=None, setup_function=None, load_kwargs=None, line_database=None, ionization_table=None, field_parameters=None)

Create a yt LightRay object for multiple consecutive datasets (eg IGM). This is a wrapper function around yt’s LightRay interface to reduce some of the complexity there.

Note: The compound ray functionality has only been implemented for the Enzo and Gadget/Gizmo codes. If you would like to help us implement this functionality for your simulation code, please contact us about this on the mailing list.

A compound ray is a series of straight lines passing through multiple consecutive outputs from a single cosmological simulation to approximate a continuous line of sight to high redshift.

Because a single continuous ray traversing a simulated volume can only cover a small range in redshift space (e.g. 100 Mpc only covers the redshift range from z=0 to z=0.023), the compound ray passes rays through multiple consecutive outputs from the same simulation to approximate the path of a single line of sight to high redshift. By probing all of the foreground material out to any given redshift, the compound ray is appropriate for studies of the intergalactic medium and circumgalactic medium.

By default, it selects a random starting location and trajectory in each dataset it traverses, to assure that the same cosmological structures are not being probed multiple times from the same direction. In doing this, the ray becomes discontinuous across each dataset.

The compound ray requires the parameter_filename of the simulation run. This is not the dataset filename from a single output, but the parameter file that was used to run the simulation itself. It is in this parameter file that the
output frequency, simulation volume, and cosmological parameters are described to assure full redshift coverage can be achieved for a compound ray. It also requires the simulation_type of the simulation.

Unlike the simple ray, which is specified by its start and end positions in the dataset volume, the compound ray requires the near_redshift and far_redshift to determine which datasets to use to get full coverage in redshift space as the ray propagates from near_redshift to far_redshift.

Like the simple ray produced by `make_simple_ray`, each gas cell intersected by the LightRay is sampled for the desired fields and stored. Several additional fields are created and stored including `dl` to represent the path length in space for each element in the ray, `v_los` to represent the line of sight velocity along the ray, and `redshift`, `redshift_dopp`, and `redshift_eff` to represent the cosmological redshift, doppler redshift and effective redshift (combined doppler and cosmological) for each element of the ray.

The :lines: keyword can be set to automatically add all fields to the resulting ray necessary for later use with the SpectrumGenerator class.

**Parameters**

- **Parameter_filename** string
  The simulation parameter file *not* the dataset filename

- **Simulation_type** string
  The simulation type of the parameter file. At present, this functionality only works with “Enzo” and “Gadget” yt frontends.

- **Near_redshift**, **far_redshift** floats
  The near and far redshift bounds of the LightRay through the simulation datasets.

- **Lines** list of strings, optional
  List of strings that determine which fields will be added to the ray to support line deposition to an absorption line spectrum. List can include things like “C”, “O VI”, or “Mg II ####”, where #### would be the integer wavelength value of the desired line. If set to ‘all’, includes all possible ions from H to Zn. :lines: can be used in conjunction with :fields: as they will not override each other. Default: None

- **Ftype** string, optional
  This is now deprecated and unnecessary. Default: “gas”

- **Fields** list of strings, optional
  The list of which fields to store in the output LightRay. See :lines: keyword for additional functionality that will add fields necessary for creating absorption line spectra for certain line features. Default: None

- **Solution_filename** string, optional
  Output filename of text file containing trajectory of LightRay through the dataset. Default: None

- **Data_filename** string, optional
  Output filename for ray data stored as an HDF5 file. Note that at present, you *must* save a ray to disk in order for it to be returned by this function. If set to None, defaults to ‘ray.h5’. Default: None

- **Use_minimum_datasets** bool, optional
  Use the minimum number of datasets to make the ray continuous through the supplied datasets from the near_redshift to the far_redshift. If false, the LightRay solution will contain as many datasets as possible to enable the light ray to traverse the desired redshift interval. Default: True
**Max_box_fraction** float, optional

The maximum length a light ray segment can be in order to span the redshift interval from one dataset to another in units of the domain size. Values larger than 1.0 will result in LightRays crossing the domain of a given dataset more than once, which is generally undesired. Zoom-in simulations can use a value equal to the length of the high-resolution region so as to limit ray segments to that size. If the high-resolution region is not cubical, the smallest size should be used. Default: 1.0 (the size of the box)

**Deltaz_min** float, optional

The minimum delta-redshift value between consecutive datasets used in the LightRay solution. Default: 0.0

**Minimum_coherent_box_fraction** float, optional

When use_minimum_datasets is set to False, this parameter specifies the fraction of the total box width to be traversed before rerandomizing the ray location and trajectory. Default: 0.0

**Find_outputs** optional, bool

Whether or not to search for datasets in the current directory. This is useful if the number of existing datasets is different than what would be predicted by the simulation parameter file. Default: False.

**Seed** int, optional

Sets the seed for the random number generator used to determine the location and trajectory of the LightRay as it traverses the simulation datasets. For consistent results between LightRays, use the same seed value. Default: None

**Setup_function** function, optional

A function that will be called on the dataset as it is loaded but before the LightRay is generated. Very useful for adding derived fields and other manipulations of the dataset prior to LightRay creation. Default: None

**Load_kwargs** dict, optional

Dictionary of kwargs to be passed to the yt “load” function prior to creating the LightRay. Very useful for many frontends like Gadget, Tipsy, etc. for passing in “bounding_box”, “unit_base”, etc. Default: None

**Line_database** string, optional

For use with the :lines: keyword. If you want to limit the available ion fields to be added to those available in a particular subset, you can use a LineDatabase. This means when you set :lines:=‘all’, it will only use those ions present in the corresponding LineDatabase. If :LineDatabase: is set to None, and :lines:=‘all’, it will add every ion of every element up to Zinc. Default: None

**Ionization_table** string, optional

For use with the :lines: keyword. Path to an appropriately formatted HDF5 table that can be used to compute the ion fraction as a function of density, temperature, metallicity, and redshift. When set to None, it uses the table specified in ~/.trident/config Default: None

**Field_parameters** optional, dict

Used to set field parameters in light rays. For example, if the ‘bulk_velocity’ field parameter is set, the relative velocities used to calculate peculiar velocity will be adjusted accordingly. Default: None

**Example**

Generate a compound ray passing from the redshift 0 to redshift 0.05 through a multi-output enzo simulation.
Generate a compound ray passing from the redshift 0 to redshift 0.05 through a multi-output gadget simulation.

```python
>>> import trident
>>> fn = 'path/to/simulation/parameter/file'
>>> ray = trident.make_compound_ray(fn, simulation_type='Gadget',
... near_redshift=0.0, far_redshift=0.05, lines=['H', 'O', 'Mg II'])
```

### 8.1.3 trident.LightRay

**class** `trident.LightRay`

```python
class trident.LightRay(
    parameter_filename, simulation_type=None, near_redshift=None,
    far_redshift=None, use_minimum_datasets=True, max_box_fraction=1.0,
    deltax_min=0.0, minimum_coherent_box_fraction=0.0, time_data=True,
    redshift_data=True, find_outputs=False, load_kwargs=None)
```

A 1D object representing the path of a light ray passing through a simulation. LightRays can be either simple, where they pass through a single dataset, or compound, where they pass through consecutive datasets from the same cosmological simulation. One can sample any of the fields intersected by the LightRay object as it passed through the dataset(s).

For compound rays, the LightRay stacks together multiple datasets in a time series in order to approximate a LightRay’s path through a volume and redshift interval larger than a single simulation data output. The outcome is something akin to a synthetic QSO line of sight.

Once the LightRay object is set up, use `LightRay.make_light_ray` to begin making rays. Different randomizations can be created with a single object by providing different random seeds to `make_light_ray`.

**Parameters**

- **Parameter_filename** string or `Dataset`
  For simple rays, one may pass either a loaded dataset object or the filename of a dataset. For compound rays, one must pass the filename of the simulation parameter file.

- **Simulation_type** optional, string
  This refers to the simulation frontend type. Do not use for simple rays. Default: None

- **Near_redshift** optional, float
  The near (lowest) redshift for a light ray containing multiple datasets. Do not use for simple rays. Default: None

- **Far_redshift** optional, float
  The far (highest) redshift for a light ray containing multiple datasets. Do not use for simple rays. Default: None

- **Use_minimum_datasets** optional, bool
  If True, the minimum number of datasets is used to connect the initial and final redshift. If false, the light ray solution will contain as many entries as possible within the redshift interval. Do not use for simple rays. Default: True.

- **Max_box_fraction** optional, float
  In terms of the size of the domain, the maximum length a light ray segment can be in order to span the redshift interval from one dataset to another. If using a zoom-in simulation, this
parameter can be set to the length of the high resolution region so as to limit ray segments to that size. If the high resolution region is not cubical, the smallest side should be used. Default: 1.0 (the size of the box)

**Deltaz_min** optional, float

Specifies the minimum $\Delta z$ between consecutive datasets in the returned list. Do not use for simple rays. Default: 0.0.

**Minimum_coherent_box_fraction** optional, float

Use to specify the minimum length of a ray, in terms of the size of the domain, before the trajectory is re-randomized. Set to 0 to have ray trajectory randomized for every dataset. Set to np.inf (infinity) to use a single trajectory for the entire ray. Default: 0.

**Time_data** optional, bool

Whether or not to include time outputs when gathering datasets for time series. Do not use for simple rays. Default: True.

**Redshift_data** optional, bool

Whether or not to include redshift outputs when gathering datasets for time series. Do not use for simple rays. Default: True.

**Find_outputs** optional, bool

Whether or not to search for datasets in the current directory. Do not use for simple rays. Default: False.

**Load_kwargs** optional, dict

If you are passing a filename of a dataset to LightRay rather than an already loaded dataset, then you can optionally provide this dictionary as keywords when the dataset is loaded by yt with the “load” function. Necessary for use with certain frontends. E.g. Tipsy using “bounding_box” Gadget using “unit_base”, etc. Default : None

---

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong><strong>init</strong></strong></td>
<td>Initialize self.</td>
</tr>
<tr>
<td><strong>create_cosmology_splice</strong></td>
<td>Create list of datasets capable of spanning a redshift interval.</td>
</tr>
<tr>
<td><strong>make_light_ray</strong></td>
<td>Actually generate the LightRay by traversing the desired dataset.</td>
</tr>
<tr>
<td><strong>plan_cosmology_splice</strong></td>
<td>Create imaginary list of redshift outputs to maximally span a redshift interval.</td>
</tr>
</tbody>
</table>
trident Documentation, Release 1.3.dev1

trident.LightRay.__init__

LightRay.__init__(parameter_filename, simulation_type=None, near_redshift=None, far_redshift=None, use_minimum_datasets=True, max_box_fraction=1.0, deltax_min=0.0, minimum_coherent_box_fraction=0.0, time_data=True, redshift_data=True, find_outputs=False, load_kwargs=None)

Initialize self. See help(type(self)) for accurate signature.

trident.LightRay.create_cosmology_splice

LightRay.create_cosmology_splice(near_redshift, far_redshift, minimal=True, max_box_fraction=1.0, deltax_min=0.0, time_data=True, redshift_data=True)

Create list of datasets capable of spanning a redshift interval.

For cosmological simulations, the physical width of the simulation box corresponds to some Delta z, which varies with redshift. Using this logic, one can stitch together a series of datasets to create a continuous volume or length element from one redshift to another. This method will return such a list

Parameters

- **near_redshift (float)** – The nearest (lowest) redshift in the cosmology splice list.
- **far_redshift (float)** – The furthest (highest) redshift in the cosmology splice list.
- **minimal (bool)** – If True, the minimum number of datasets is used to connect the initial and final redshift. If false, the list will contain as many entries as possible within the redshift interval. Default: True.
- **max_box_fraction (float)** – In terms of the size of the domain, the maximum length a light ray segment can be in order to span the redshift interval from one dataset to another. If using a zoom-in simulation, this parameter can be set to the length of the high resolution region so as to limit ray segments to that size. If the high resolution region is not cubical, the smallest side should be used. Default: 1.0 (the size of the box)
- **deltax_min (float)** – Specifies the minimum delta z between consecutive datasets in the returned list. Default: 0.0.
- **time_data (bool)** – Whether or not to include time outputs when gathering datasets for time series. Default: True.
- **redshift_data (bool)** – Whether or not to include redshift outputs when gathering datasets for time series. Default: True.

Examples

```python
>>> co = CosmologySplice("enzo_tiny_cosmology/32Mpc_32.enzo", "Enzo")
>>> cosmo = co.create_cosmology_splice(1.0, 0.0)
```
trident documentation, release 1.3.dev1

trident.LightRay.make_light_ray

LightRay.make_light_ray(seed=None, periodic=True, left_edge=None, right_edge=None, min_level=None, start_position=None, end_position=None, trajectory=None, fields=None, setup_function=None, solution_filename=None, data_filename=None, get_los_velocity=None, use_peculiar_velocity=True, redshift=None, field_parameters=None, njobs=-1)

Actually generate the LightRay by traversing the desired dataset.

A light ray consists of a list of field values for cells intersected by the ray and the path length of the ray through those cells. Light ray data must be written out to an hdf5 file.

Parameters

Seed optional, int

Seed for the random number generator. Default: None.

Periodic optional, bool

If True, ray trajectories will make use of periodic boundaries. If False, ray trajectories will not be periodic. Default: True.

Left_edge optional, iterable of floats or YTArray

The left corner of the region in which rays are to be generated. If None, the left edge will be that of the domain. If specified without units, it is assumed to be in code units. Default: None.

Right_edge optional, iterable of floats or YTArray

The right corner of the region in which rays are to be generated. If None, the right edge will be that of the domain. If specified without units, it is assumed to be in code units. Default: None.

Min_level optional, int

The minimum refinement level of the spatial region in which the ray passes. This can be used with zoom-in simulations where the high resolution region does not keep a constant geometry. Default: None.

Start_position optional, iterable of floats or YTArray.

Used only if creating a light ray from a single dataset. The coordinates of the starting position of the ray. If specified without units, it is assumed to be in code units. Default: None.

End_position optional, iterable of floats or YTArray.

Used only if creating a light ray from a single dataset. The coordinates of the ending position of the ray. If specified without units, it is assumed to be in code units. Default: None.

Trajectory optional, list of floats

Used only if creating a light ray from a single dataset. The (r, theta, phi) direction of the light ray. Use either end_position or trajectory, not both. Default: None.

Fields optional, list

A list of fields for which to get data. Default: None.

Setup_function optional, callable, accepts a ds

This function will be called on each dataset that is loaded to create the light ray. For example, this can be used to add new derived fields. Default: None.
Solution_filename  optional, string
Path to a text file where the trajectories of each subray is written out. Default: None.

Data_filename  optional, string
Path to output file for ray data. Default: None.

Use_peculiar_velocity  optional, bool
If True, the peculiar velocity along the ray will be sampled for calculating the effective redshift combining the cosmological redshift and the doppler redshift. Default: True.

Redshift  optional, float
Used with light rays made from single datasets to specify a starting redshift for the ray. If not used, the starting redshift will be 0 for a non-cosmological dataset and the dataset redshift for a cosmological dataset. Default: None.

Field_parameters  optional, dict
Used to set field parameters in light rays. For example, if the ‘bulk_velocity’ field parameter is set, the relative velocities used to calculate peculiar velocity will be adjusted accordingly. Default: None.

Njobs  optional, int
The number of parallel jobs over which the segments will be split. Choose -1 for one processor per segment. Default: -1.

Examples
Make a light ray from multiple datasets:

```python
>>> import yt
>>> from trident import LightRay
>>> my_ray = LightRay("enzo_tiny_cosmology/32Mpc_32.enzo", "Enzo",
... 0., 0.1, time_data=False)
...
>>> my_ray.make_light_ray(seed=12345,
... solution_filename="solution.txt",
... data_filename="my_ray.h5",
... fields=["temperature", "density"],
... use_peculiar_velocity=True)
```

Make a light ray from a single dataset:

```python
>>> import yt
>>> from trident import LightRay
>>> my_ray = LightRay("IsolatedGalaxy/galaxy0030/galaxy0030")
...
>>> my_ray.make_light_ray(start_position=[0., 0., 0.],
... end_position=[1., 1., 1.],
... solution_filename="solution.txt",
... data_filename="my_ray.h5",
... fields=["temperature", "density"],
... use_peculiar_velocity=True)
```
trident.LightRay.plan_cosmology_splice

LightRay.plan_cosmology_splice(near_redshift, far_redshift, decimals=3, filename=None, start_index=0)

Create imaginary list of redshift outputs to maximally span a redshift interval.

If you want to run a cosmological simulation that will have just enough data outputs to create a cosmology splice, this method will calculate a list of redshifts outputs that will minimally connect a redshift interval.

Parameters

- **near_redshift (float)** – The nearest (lowest) redshift in the cosmology splice list.
- **far_redshift (float)** – The furthest (highest) redshift in the cosmology splice list.
- **decimals (int)** – The decimal place to which the output redshift will be rounded. If the decimal place in question is nonzero, the redshift will be rounded up to ensure continuity of the splice. Default: 3.
- **filename (string)** – If provided, a file will be written with the redshift outputs in the form in which they should be given in the enzo dataset. Default: None.
- **start_index (int)** – The index of the first redshift output. Default: 0.

Examples

```python
>>> from yt.extensions.astro_analysis.cosmological_observation.api import CosmologySplice
>>> my_splice = CosmologySplice('enzo_tiny_cosmology/32Mpc_32.enzo', 'Enzo')
>>> my_splice.plan_cosmology_splice(0.0, 0.1, filename='redshifts.out')
```

8.2 Generating Spectra

<table>
<thead>
<tr>
<th>SpectrumGenerator</th>
<th>Preferred class for generating, storing, and plotting absorption-line spectra.</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbsorptionSpectrum</td>
<td>Base class for generating absorption spectra.</td>
</tr>
<tr>
<td>Instrument</td>
<td>An instrument class for specifying a spectrograph/telescope pair.</td>
</tr>
<tr>
<td>LSF</td>
<td>A class representing a spectrograph’s line spread function.</td>
</tr>
<tr>
<td>Line</td>
<td>A class representing an individual atomic transition.</td>
</tr>
<tr>
<td>LineDatabase</td>
<td>Class for storing and selecting collections of spectral lines.</td>
</tr>
</tbody>
</table>
8.2.1 trident.SpectrumGenerator

```python
class trident.SpectrumGenerator(instrument=None, lambda_min=None, lambda_max=None, n_lambda=None, dlambda=None, lsf_kernel=None, line_database='lines.txt', ionization_table=None, bin_space='wavelength')
```

Preferred class for generating, storing, and plotting absorption-line spectra. SpectrumGenerator is a subclass of yt’s AbsorptionSpectrum class with additional functionality like line lists, post-processing to include Milky Way foreground, quasar backgrounds, applying line-spread functions, and plotting.

User first specifies the telescope/instrument used for generating spectra (e.g. ‘COS’ for the Cosmic Origins Spectrograph aboard the Hubble Space Telescope). This can be done by naming the `Instrument`, or manually setting all of the spectral characteristics including `lambda_min`, `lambda_max`, `lsf_kernel`, and `n_lambda` or `dlambda`. If none of these arguments are set, defaults to ‘COS’ as the default instrument covering 1150-1450 Angstroms with a binsize (dlambda) of 0.1 Angstroms.

Once a `SpectrumGenerator` has been initialized, pass it `LightRay` objects using `make_spectrum` to actually generate the spectra themselves. Then one can post-process, plot, or save them using `add_milky_way_foreground`, `add_qso_spectrum`, `apply_lsf`, `save_spectrum`, and `plot_spectrum`.

**Parameters**

- **Instrument** string, optional
  
  The spectrograph to use. Currently, the only named options are different observing modes of the Cosmic Origins Spectrograph ‘COS’: ‘COS-G130M’, ‘COS-G160M’, and ‘COS-G140L’ as well as ‘COS’ which aliases to ‘COS-G130M’. These automatically set the `lambda_min`, `lambda_max`, `dlambda` and `lsf_kernel`’s appropriately. If you're going to set ``lambda_min, lambda_max, et al manually, leave this set to None. Default: None

- **Lambda_min** float, YTQuantity, or ‘auto’
  
  lower wavelength bound in angstroms or velocity bound in km/s (if bin_space set to ‘velocity’). If set to ‘auto’, the lower bound will be automatically adjusted to encompass all absorption lines. The window will not be expanded for continuum features, only absorption lines.

- **Lambda_max** float, YTQuantity, or ‘auto’
  
  upper wavelength bound in angstroms or velocity bound in km/s (if bin_space set to ‘velocity’). If set to ‘auto’, the upper bound will be automatically adjusted to encompass all absorption lines. The window will not be expanded for continuum features, only absorption lines.

- **N_lambda** int
  
  The number of bins in the spectrum (inclusive), so if extrema = 10 and 20, and dlambda (binsize) = 1, then n_lambda = 11. Default: None

- **Dlambda** float
  
  size of the wavelength bins in angstroms or velocity bins in km/s. Default: None

- **Bin_space** ‘wavelength’ or ‘velocity’
  
  Sets the dimension in which spectra are created. If set to wavelength, the resulting spectra are flux (or tau) vs. observed wavelength. If set to velocity, the spectra are flux vs. velocity offset from the rest wavelength of the absorption line. Default: wavelength

- **Lsf_kernel** string, optional
The filename for the LSF kernel. Files are found in trident.__path__/data/lsf_kernels or current working directory. Only necessary if you are applying an LSF to the spectrum in postprocessing. Default: None

**Line database** string or `LineDatabase`, optional

A text file listing the various lines to insert into the line database, or a `LineDatabase` object in memory. The line database provides a list of all possible lines that could be added to the spectrum. For a text file, it should have 4 tab-delimited columns of name (e.g. MgII), wavelength in angstroms, gamma of transition, and f-value of transition. See example datasets in trident.path/data/line_lists for examples. Default: lines.txt

**Ionization table** hdf5 file, optional

An HDF5 file used for computing the ionization fraction of the gas based on its density, temperature, metallicity, and redshift. If set to None, will use the ion table defined in your Trident config file. Default: None

**Example**

Create a one-zone ray, and generate a COS spectrum from that ray.

```python
>>> import trident
>>> ray = trident.make_onezone_ray()
>>> sg = trident.SpectrumGenerator('COS')
>>> sg.make_spectrum(ray)
>>> sg.plot_spectrum('spec_raw.png')
```

Create a one-zone ray at redshift 0.5, and generate a spectrum with 1 angstrom spectral bins from 2000-4000 angstroms, then post-process by adding a MW foreground a QSO background at z=0.5 and add a boxcar line spread function of 100 angstroms width. Plot it and save the figure to ‘spec_final.png’.

```python
>>> import trident
>>> ray = trident.make_onezone_ray(redshift=0.5)
>>> sg = trident.SpectrumGenerator(lambda_min=2000, lambda_max=4000, ... dlambda=1)
>>> sg.make_spectrum(ray)
>>> sg.add_qso_spectrum(emitting_redshift=.5)
>>> sg.add_milky_way_foreground()
>>> sg.apply_lsf(function='boxcar', width=100)
>>> sg.plot_spectrum('spec_final.png')
```

**Methods**

- `__init__`: Initialize self.
- `add_continuum`: Add a continuum feature that follows a power-law.
- `add_gaussian_noise`: Postprocess a spectrum to add gaussian random noise of a given SNR.
- `add_line`: Add an absorption line to the list of lines included in the spectrum.
- `add_line_to_database`: Adds desired line to the current `LineDatabase` object.
- `add_milky_way_foreground`: Postprocess a spectrum to add a Milky Way foreground.
- `add_noise_vector`: Add an array of noise to the spectrum.

continues on next page
**trident Documentation, Release 1.3.dev1**

Table 4 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_qso_spectrum</code></td>
<td>Postprocess a spectrum to add a QSO spectrum background.</td>
</tr>
<tr>
<td><code>apply_lsf</code></td>
<td>Postprocess a spectrum to apply a line spread function.</td>
</tr>
<tr>
<td><code>clear_spectrum</code></td>
<td>Clear the current spectrum in the SpectrumGenerator.</td>
</tr>
<tr>
<td><code>error_func</code></td>
<td>Approximate the flux error for a spectrum.</td>
</tr>
<tr>
<td><code>load_spectrum</code></td>
<td>Load data arrays into an existing spectrum object.</td>
</tr>
<tr>
<td><code>make_spectrum</code></td>
<td>Make a spectrum from ray data depositing the desired lines.</td>
</tr>
<tr>
<td><code>plot_spectrum</code></td>
<td>Plot the current spectrum and save to disk.</td>
</tr>
<tr>
<td><code>save_spectrum</code></td>
<td>Save the current spectrum data to an output file.</td>
</tr>
</tbody>
</table>

**trident.SpectrumGenerator.__init__**

SpectrumGenerator.__init__(instrument=None, lambda_min=None, lambda_max=None, n_lambda=None, dlambda=None, lsf_kernel=None, line_database='lines.txt', ionization_table=None, bin_space='wavelength')

Initialize self. See help(type(self)) for accurate signature.

**trident.SpectrumGenerator.add_continuum**

SpectrumGenerator.add_continuum(label, field_name, wavelength, normalization, index)

Add a continuum feature that follows a power-law.

**Parameters**

- **Label** string
  - label for the feature.

- **Field name** string
  - field name from ray data for column densities.

- **Wavelength** float
  - line rest wavelength in angstroms.

- **Normalization** float
  - the column density normalization.

- **Index** float
  - the power-law index for the wavelength dependence.
SpectrumGenerator.add_gaussian_noise

SpectrumGenerator.add_gaussian_noise(snr, seed=None)
Postprocess a spectrum to add gaussian random noise of a given SNR.

Parameters

Snr float
The desired signal-to-noise ratio for determining the amount of gaussian noise

Seed optional, int
Seed for the random number generator. This should be used to ensure than the same noise is added each time the spectrum is regenerated, if desired. Default: None

Example

Make a one zone ray and generate a COS spectrum for it. Add noise to the spectrum as though it were observed with a signal to noise ratio of 30.

```python
>>> import trident
>>> ray = trident.make_onezone_ray(redshift=0.5)
>>> sg = trident.SpectrumGenerator('COS')
>>> sg.make_spectrum(ray)
>>> sg.add_gaussian_noise(30)
>>> sg.plot_spectrum('spec_noise_corrected.png')
```

Plot a DLA with SNR of 10.

```python
>>> import trident
>>> ray = trident.make_onezone_ray(column_densities={'H_p0_number_density':1e21})
>>> sg = trident.SpectrumGenerator(lambda_min=1200, lambda_max=1300, dlambda=0.1)
>>> sg.make_spectrum(ray, lines=['Ly a'])
>>> sg.add_gaussian_noise(10)
>>> sg.plot_spectrum('spec_noise.png')
```

SpectrumGenerator.add_line

SpectrumGenerator.add_line(label, field_name, wavelength, f_value, gamma, atomic_mass, label_threshold=None)
Add an absorption line to the list of lines included in the spectrum.

Parameters

Label string
label for the line.

Field_name string
field name from ray data for column densities.

Wavelength float
line rest wavelength in angstroms.

F_value float
line f-value.

8.2. Generating Spectra 53
Gamma float
  line gamme value.

Atomic_mass float
  mass of atom in amu.

trident.SpectrumGenerator.add_line_to_database

SpectrumGenerator.add_line_to_database(element, ion_state, wavelength, gamma, f_value, field=None, identifier=None)
  Adds desired line to the current LineDatabase object.

Parameters
  Element string
    The element of the transition using element’s symbol on periodic table
  Ion_state string
    The roman numeral representing the ionic state of the transition
  Wavelength float
    The wavelength of the transition in angstroms
  Gamma float
    The gamma of the transition in Hertz
  F_value float
    The oscillator strength of the transition
  Field string, optional
    The default yt field name associated with the ion responsible for this line Default: None
  Identifier string, optional
    An optional identifier for the transition Default: None

trident.SpectrumGenerator.add_milky_way_foreground

SpectrumGenerator.add_milky_way_foreground(flux_field=None, filename=None)
  Postprocess a spectrum to add a Milky Way foreground. Data from Charles Danforth. Median-filter of 92 normalized COS/G130M+G160M AGN spectra spanning the wavelength range of 1130 to 1800 Angstroms in 0.07 Angstrom bin size.

Parameters
  Flux_field optional, array
    Array of flux values to which the Milky Way foreground is applied. Default: None
  Filename string
    Filename where the Milky Way foreground values used to modify the flux are stored. Default: None

Example
  Make a one zone ray and generate a COS spectrum for it. Add MW foreground to it, and save it.
trident Documentation, Release 1.3.dev1

>>> import trident
>>> ray = trident.make_onezone_ray()
>>> sg = trident.SpectrumGenerator('COS')
>>> sg.make_spectrum(ray)
>>> sg.add_milky_way_foreground()
>>> sg.plot_spectrum('spec_mw_corrected.png')

Plot a naked MW spectrum.

>>> import trident
>>> sg = trident.SpectrumGenerator('COS')
>>> sg.add_milky_way_foreground()
>>> sg.plot_spectrum('spec_mw.png')

trident.SpectrumGenerator.add_noise_vector

SpectrumGenerator.add_noise_vector(noise)

Add an array of noise to the spectrum.

Parameters

Noise array of floats

The array of noise values to be added to the spectrum. This array must be of the same size as the flux array.

Example

>>> import numpy as np
>>> import trident
>>> ray = trident.make_onezone_ray(column_densities={'H_p0_number_density':1e21})
>>> sg = trident.SpectrumGenerator(lambda_min=1200, lambda_max=1300, dlambda=0.1)
>>> sg.make_spectrum(ray, lines=['Ly a'])
>>> my_noise = np.random.normal(loc=0.0, scale=0.1, size=sg.flux_field.size)
>>> sg.add_noise_vector(my_noise)
>>> sg.plot_spectrum('spec_noise.png')

trident.SpectrumGenerator.add_qso_spectrum

SpectrumGenerator.add_qso_spectrum(flux_field=None, emitting_redshift=None, observing_redshift=None, filename=None)


Parameters

Flux_field array, optional

Array of flux values to which the quasar background is applied. Default: None

Emitting_redshift float, optional

Redshift value at which the QSO emitted its light. If specified as None, use 0. Default: None

Observing_redshift float, optional

8.2. Generating Spectra
Redshift value at which the quasar is observed. If specified as None, use the observing_redshift value specified in make_spectrum() which defaults to 0. Default: None

**Filename**  
"string"  
Filename where the Milky Way foreground values used to modify the flux are stored. Default: None

**Example**

Make a one zone ray at redshift of .5 and generate a COS spectrum for it. Add z=0.5 quasar background to it, and save it.

```python
>>> import trident
>>> ray = trident.make_onezone_ray(redshift=0.5)
>>> sg = trident.SpectrumGenerator('COS')
>>> sg.make_spectrum(ray)
>>> sg.add_qso_spectrum(emitting_redshift=0.5)
>>> sg.plot_spectrum('spec_qso_corrected.png')
```

Plot a naked QSO spectrum at z=.1

```python
>>> import trident
>>> sg = trident.SpectrumGenerator('COS')
>>> sg.add_qso_spectrum(emitting_redshift=.1)
>>> sg.plot_spectrum('spec_qso.png')
```

**trident.SpectrumGenerator.apply_lsf**

`SpectrumGenerator.apply_lsf(function=None, width=None, filename=None)`  
Postprocess a spectrum to apply a line spread function. If the SpectrumGenerator already has an LSF_kernel set, it will be used when no keywords are supplied. Otherwise, the user can specify a filename of a user-defined kernel or a function+width for a kernel. Valid functions are: “boxcar” and “gaussian”.

For more information, see `LSF` and `Instrument`.

**Parameters**

- **Function**  
  string, optional  
  Desired functional form for the applied LSF kernel. Valid options are currently “boxcar” or “gaussian” Default: None

- **Width**  
  int, optional  
  Width of the desired LSF kernel in bin elements Default: None

- **Filename**  
  string, optional  
  The filename of the user-supplied kernel for applying the LSF Default: None

**Example**

Make a one zone ray and generate a COS spectrum for it. Apply the COS line spread function to it.

```python
>>> import trident
>>> ray = trident.make_onezone_ray()
>>> sg = trident.SpectrumGenerator('COS')
>>> sg.make_spectrum(ray)
>>> sg.apply_lsf()
>>> sg.plot_spectrum('spec_lsf_corrected.png')
```
Make a one zone ray and generate a spectrum with bin width = 1 angstrom. Apply a boxcar LSF to it with width 50 angstroms.

```python
>>> import trident
>>> ray = trident.make_onezone_ray()
>>> sg = trident.SpectrumGenerator(lambda_min=1100, lambda_max=1200, dlambda=1)
>>> sg.make_spectrum(ray)
>>> sg.apply_lsf(function='boxcar', width=50)
>>> sg.plot_spectrum('spec_lsf_corrected.png')
```

**trident.SpectrumGenerator.clear_spectrum**

SpectrumGenerator.clear_spectrum()

Clear the current spectrum in the SpectrumGenerator. Clears the existing spectrum’s flux and tau fields and replaces them with ones and zeros respectively. Clear the line list kept in the AbsorptionSpectrum object as well. Also clear the line_subset stored by the LineDatabase.

**trident.SpectrumGenerator.error_func**

SpectrumGenerator.error_func(flux)

Approximate the flux error for a spectrum. Many observational analysis programs require a flux error channel in addition to a flux channel. So we create a zeroth order approximation of the flux error, simply by taking the square root of the flux. Unfortunately, with flux normalized to be < 1, this would result in errors larger than the flux values themselves, so we normalize by an arbitrary signal-to-noise ratio, which by default is set to 100. This yields a typical error for a normalized spectrum of sqrt(1.0*100)/100 = 0.1. This assures our flux errors are smaller than our fluxes for most flux reasonable flux values. Note that when a signal to noise ratio is specified for adding gaussian noise, it uses this updated value for estimating the errors. SNR is set as an attribute of AbsorptionSpectrum directly (e.g., as.snr = N).

**Parameters**

- **Flux** array of floats
  - The array of flux values

**trident.SpectrumGenerator.load_spectrum**

SpectrumGenerator.load_spectrum(lambda_field=None, tau_field=None, flux_field=None)

Load data arrays into an existing spectrum object.

**Parameters**

- **Lambda_field** array
  - The array of valid wavelengths Default: None
- **Tau_field** array
  - The array of optical depths for the corresponding wavelengths Default: None
- **Flux_field** array
  - The array of flux values for the corresponding wavelengths Default: None

**Example**

Loading a custom set of data into an existing SpectrumGenerator object:
trident Documentation, Release 1.3.dev1

```python
>>> import trident
>>> import numpy as np

>>> lambda_field = np.arange(1200,1400)
>>> flux_field = np.ones(len(lambda_field))
>>> sg = trident.SpectrumGenerator('COS')
>>> sg.load_spectrum(lambda_field=lambda_field, flux_field=flux_field)
>>> sg.plot_spectrum('temp.png')
```

trident.SpectrumGenerator.make_spectrum

```python
SpectrumGenerator.make_spectrum(ray, lines='all', output_file=None, output_absorbers_file=None, use_peculiar_velocity=True, observing_redshift=0.0, ly_continuum=True, store_observables=False, min_tau=0.001, njobs='auto')
```

Make a spectrum from ray data depositing the desired lines. Make sure to pass this function a LightRay object and potentially also a list of strings representing what lines you’d like to actually have be deposited in your final spectrum.

**Parameters**

- **Ray** string, dataset, or data container
  
  If a string, the path to the ray dataset. As a dataset, this is the ray dataset loaded by yt. As a data container, this is a data object created from a ray dataset, such as a cut region.

- **Lines** list of strings
  
  List of strings that determine which lines will be added to the spectrum. List can include things like “C”, “O VI”, or “Mg II ####”, where #### would be the integer wavelength value of the desired line. If set to ‘all’, includes all lines in LineDatabase set in SpectrumGenerator. Default: ‘all’

- **Output_file** optional, string
  
  Filename of output if you wish to save the spectrum immediately without any further processing. File formats are chosen based on the filename extension. “.h5” for HDF5, “.fits” for FITS, and everything else is ASCII. Equivalent of calling `save_spectrum`. Default: None

- **Output_absorbers_file** optional, string
  
  Option to save a text file containing all of the absorbers and corresponding wavelength and redshift information. For parallel jobs, combining the lines lists can be slow so it is recommended to set to None in such circumstances. Default: None

- **Use_peculiar_velocity** optional, bool
  
  If True, include the effects of doppler redshift of the gas in shifting lines in the final spectrum. Default: True

- **Observing_redshift** optional, float
  
  This is the value of the redshift at which the observer of this spectrum exists. In most cases, this will be a redshift of 0. Default: 0.

- **Ly_continuum** optional, boolean
  
  If any H I lines are used in the line list, this assures a Lyman continuum will be included in the spectral generation. Lyman continuum begins at final Lyman line deposited (Ly 39 = 912.32 A) not at formal Lyman Limit (911.76 A) so as to not have a gap between final Lyman lines and
continuum. Uses power law of index 3 and normalization to match opacity of final Lyman lines. Default: True

**Store_observables** optional, boolean

If set to true, observable properties for each cell in the light ray will be saved for each line in the line list. Properties include the column density, tau, thermal b, and the wavelength where tau was deposited. Best applied for a reasonable number of lines. These quantities will be saved to the SpectrumGenerator attribute: ‘line_observables_dict’. Default: False

**Min_tau** optional, float

This value determines size of the wavelength window used to deposit lines or continua. The wavelength window is expanded until the optical depth at the edge is below this value. If too high, this will result in features appearing cut off at the edges. Decreasing this will make features smoother but will also increase run time. An increase by a factor of ten will result in roughly a 2x slow down. Default: 1e-3.

**Njobs** optional, int or “auto”

The number of process groups into which the loop over absorption lines will be divided. If set to -1, each absorption line will be deposited by exactly one processor. If Njobs is set to a value less than the total number of available processors (N), then the deposition of an individual line will be parallelized over (N / Njobs) processors. If set to “auto”, it will first try to parallelize over the list of lines and only parallelize the line deposition if there are more processors than lines. This is the optimal strategy for parallelizing spectrum generation. Default: “auto”

### Example

Make a one zone ray and generate a COS spectrum for it including only Oxygen VI, Mg II, and all Carbon lines, and plot to disk.

```python
generator = trident.SpectrumGenerator('COS')
generator.make_spectrum(ray, lines=['O VI', 'Mg II', 'C'])
generator.plot_spectrum('spec.png')
```

---

**trident.SpectrumGenerator.plot_spectrum**

SpectrumGenerator.plot_spectrum(filename='spectrum.png', lambda_limits=None, flux_limits=None, step=False, title=None, label=None, figsize=None, features=None, axis_labels=None)

Plot the current spectrum and save to disk.

This is a convenience method that wraps the plot_spectrum standalone function for use with the data from the SpectrumGenerator itself.

**Parameters**

**Filename** string, optional

Output filename of the plotted spectrum. Will be a png file. Default: ‘spectrum.png’

**Lambda_limits** tuple or list of floats, optional

The minimum and maximum of the lambda range (x-axis) for the plot in angstroms. If specified as None, will use whole lambda range of spectrum. Example: (1200, 1400) for 1200-1400 Angstroms. Default: None

**Flux_limits** tuple or list of floats, optional
The minimum and maximum of the flux range (y-axis) for the plot. If specified as None, limits are automatically from [0, 1.1*max(flux)]. Example: (0, 1) for normal flux range before postprocessing. Default: None

**Step** boolean, optional

Plot the spectrum as a series of step functions. Appropriate for plotting processed and noisy data.

**Title** string, optional

Optional title for plot Default: None

**Label** string, optional

Label for spectrum to be plotted. Will automatically trigger a legend to be generated. Default: None

**Features** dict, optional

Include vertical lines with labels to represent certain spectral features. Each entry in the dictionary consists of a key string to be overplot and the value float as to where in wavelength space it will be plot as a vertical line with the corresponding label.

Example: features={'Ly a': 1216, 'Ly b': 1026}

Default: None

**Axis_labels** tuple of strings, optional

Optionally set the axis labels directly. If set to None, defaults to (‘Wavelength [\$/text{\AA}\text{]}’, ‘Relative Flux’). Default: None

**Example**

Create a one-zone ray, and generate a COS spectrum from that ray. Plot the resulting spectrum highlighting the Lyman alpha feature.

```python
>>> import trident
>>> ray = trident.make_onezone_ray()
>>> sg = trident.SpectrumGenerator('COS')
>>> sg.make_spectrum(ray)
>>> sg.plot_spectrum('spec_raw.png', features={'Ly a': 1216})
```

### trident.SpectrumGenerator.save_spectrum

**save_spectrum** (filename='spectrum.h5', format=None)

Save the current spectrum data to an output file. Unless specified, the output data format will be determined by the suffix of the filename provided (‘h5’:HDF5, “fits”:FITS, all other:ASCII).

ASCII data is stored as a tab-delimited text file.

**Parameters**

**Filename** string, optional

Output filename for storing the data. Default: ‘spectrum.h5’

**Format** string, optional

Data format of the output file. Valid examples are “HDF5”, “FITS”, and “ASCII”. If None is set, selects based on suffix of filename. Default: None
Example

Save a spectrum to disk, load it from disk, and plot it.

```python
>>> import trident
>>> ray = trident.make_onezone_ray()
>>> sg = trident.SpectrumGenerator('COS')
>>> sg.make_spectrum(ray)
>>> sg.save_spectrum('temp.h5')
>>> sg.clear_spectrum()
>>> sg.load_spectrum('temp.h5')
>>> sg.plot_spectrum('temp.png')
```

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>current_tau_field</code></td>
<td>This is the optical depth array for the current absorption line being deposited.</td>
</tr>
<tr>
<td><code>lambda_field</code></td>
<td>The lambda field.</td>
</tr>
<tr>
<td><code>tau_field</code></td>
<td>This is the total optical depth of all lines and continua.</td>
</tr>
</tbody>
</table>

**trident.SpectrumGenerator.current_tau_field**

**property** `SpectrumGenerator.current_tau_field`

This is the optical depth array for the current absorption line being deposited. We will do the deposition of lines into this array, and then add it to self.tau_field at the end.

**trident.SpectrumGenerator.lambda_field**

**property** `SpectrumGenerator.lambda_field`

The lambda field.

**trident.SpectrumGenerator.tau_field**

**property** `SpectrumGenerator.tau_field`

This is the total optical depth of all lines and continua.

8.2.2 trident.absorption_spectrum.absorption_spectrum.AbsorptionSpectrum

**class** `trident.absorption_spectrum.absorption_spectrum.AbsorptionSpectrum`(`lambda_min`, `lambda_max`, `n_lambda=None`, `dlambda=None`, `bin_space='wavelength'`)

Base class for generating absorption spectra. This code was originally based in yt and more restrictive in terms of what development was allowed, so the `SpectrumGenerator` subclass has more advanced functionality built on top of this. The base algorithm and functionality for spectral generation occurs here though.

**Note:** The preferred method for generating spectra is using `SpectrumGenerator`.

8.2. Generating Spectra
Parameters

**Lambda_min**  float, YTQuantity, or ‘auto’
lower wavelength bound in angstroms or velocity bound in km/s (if bin_space set to ‘velocity’).
If set to ‘auto’, the lower bound will be automatically adjusted to encompass all absorption lines.
The window will not be expanded for continuum features, only absorption lines.

**Lambda_max**  float, YTQuantity, or ‘auto’
upper wavelength bound in angstroms or velocity bound in km/s (if bin_space set to ‘velocity’).
If set to ‘auto’, the upper bound will be automatically adjusted to encompass all absorption lines.
The window will not be expanded for continuum features, only absorption lines.

**N_lambda** optional, int
number of bins. This cannot be set when setting either lambda_min or lambda_max to auto.

**Dlambda** optional, float or YTQuantity
size of the wavelength bins in angstroms or velocity bins in km/s.

**Bin_space** ‘wavelength’ or ‘velocity’
Sets the dimension in which spectra are created. If set to wavelength, the resulting spectra are
flux (or tau) vs. observed wavelength. If set to velocity, the spectra are flux vs. velocity offset
from the rest wavelength of the absorption line. Default: wavelength

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong><strong>init</strong></strong></td>
<td>Initialize self. See help(type(self)) for accurate signature.</td>
</tr>
<tr>
<td>add_continuum</td>
<td>Add a continuum feature that follows a power-law.</td>
</tr>
<tr>
<td>add_line</td>
<td>Add an absorption line to the list of lines included in the spectrum.</td>
</tr>
<tr>
<td>error_func</td>
<td>Approximate the flux error for a spectrum.</td>
</tr>
<tr>
<td>make_spectrum</td>
<td>Make spectrum from ray data using the line list.</td>
</tr>
</tbody>
</table>

**trident.absorption_spectrum.absorption_spectrum.AbsorptionSpectrum.__init__**

AbsorptionSpectrum.__init__(lambda_min, lambda_max, n_lambda=None, dlambda=None, bin_space='wavelength')
Initialize self. See help(type(self)) for accurate signature.

**trident.absorption_spectrum.absorption_spectrum.AbsorptionSpectrum.add_continuum**

AbsorptionSpectrum.add_continuum(label, field_name, wavelength, normalization, index)
Add a continuum feature that follows a power-law.

Parameters

**Label** string
label for the feature.

**Field_name** string
field name from ray data for column densities.
**Wavelength** float

line rest wavelength in angstroms.

**Normalization** float

the column density normalization.

**Index** float

the power-law index for the wavelength dependence.

---

**trident.absorption_spectrum.absorption_spectrum.AbsorptionSpectrum.add_line**

AbsorptionSpectrum.add_line(label, field_name, wavelength, f_value, gamma, atomic_mass, label_threshold=None)

Add an absorption line to the list of lines included in the spectrum.

**Parameters**

- **Label** string
  
  label for the line.

- **Field_name** string
  
  field name from ray data for column densities.

- **Wavelength** float
  
  line rest wavelength in angstroms.

- **F_value** float
  
  line f-value.

- **Gamma** float
  
  line gamma value.

- **Atomic_mass** float
  
  mass of atom in amu.

---

**trident.absorption_spectrum.absorption_spectrum.AbsorptionSpectrum.error_func**

AbsorptionSpectrum.error_func(flux)

Approximate the flux error for a spectrum. Many observational analysis programs require a flux error channel in addition to a flux channel. So we create a zeroth order approximation of the flux error, simply by taking the square root of the flux. Unfortunately, with flux normalized to be < 1, this would result in errors larger than the flux values themselves, so we normalize by an arbitrary signal-to-noise ratio, which by default is set to 100. This yields a typical error for a normalized spectrum of sqrt(1.0*100)/100 = 0.1. This assures our flux errors are smaller than our fluxes for most flux reasonable flux values. Note that when a signal to noise ratio is specified for adding gaussian noise, it uses this updated value for estimating the errors. SNR is set as an attribute of AbsorptionSpectrum directly (e.g., as.snr = N).

**Parameters**

- **Flux** array of floats
  
  The array of flux values
AbsorptionSpectrum.make_spectrum(input_object, output_file=None, line_list_file=None, output_absorbers_file=None, use_peculiar_velocity=True, store_observables=False, subgrid_resolution=10, observing_redshift=0.0, min_tau=0.001, njobs='auto')

Make spectrum from ray data using the line list.

Parameters

Input_object string, dataset, or data container
 If a string, the path to the ray dataset. As a dataset, this is the ray dataset loaded by yt. As a data container, this is a data object created from a ray dataset, such as a cut region.

Output_file optional, string
 Option to save a file containing the wavelength, flux, and optical depth fields. File formats are chosen based on the filename extension. .h5 for hdf5, .fits for fits, and everything else is ASCII. Default: None

Output_absorbers_file optional, string
 Option to save a text file containing all of the absorbers and corresponding wavelength and redshift information. For parallel jobs, combining the lines lists can be slow so it is recommended to set to None in such circumstances. Default: None

Use_peculiar_velocity optional, bool
 if True, include peculiar velocity for calculating doppler redshift to shift lines. Requires similar flag to be set in LightRay generation. Default: True

Store_observables optional, bool
 if True, stores observable properties of each cell along the line of sight for each line, such as tau, column density, and thermal b. These quantities will be saved to the AbsorptionSpectrum attribute: ‘line_observables_dict’. Default: False

Subgrid_resolution optional, int
 When a line is being added that is unresolved (ie its thermal width is less than the spectral bin width), the voigt profile of the line is deposited into an array of virtual wavelength bins at higher resolution. The optical depth from these virtual bins is integrated and then added to the coarser spectral wavelength bin. The subgrid_resolution value determines the ratio between the thermal width and the bin width of the virtual bins. Increasing this value yields smaller virtual bins, which increases accuracy, but is more expensive. A value of 10 yields accuracy to the 4th significant digit in tau. Default: 10

Observing_redshift optional, float
 This is the redshift at which the observer is observing the absorption spectrum. Default: 0

Min_tau optional, float
 This value determines size of the wavelength window used to deposit lines or continua. The wavelength window is expanded until the optical depth at the edge is below this value. If too high, this will result in features appearing cut off at the edges. Decreasing this will make features smoother but will also increase run time. An increase by a factor of ten will result in roughly a 2x slow down. Default: 1e-3.

Njobs optional, int or “auto”
the number of process groups into which the loop over absorption lines will be divided. If set to -1, each absorption line will be deposited by exactly one processor. If njobs is set to a value less than the total number of available processors (N), then the deposition of an individual line will be parallelized over (N / njobs) processors. If set to “auto”, it will first try to parallelize over the list of lines and only parallelize the line deposition if there are more processors than lines. This is the optimal strategy for parallelizing spectrum generation. Default: “auto”

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>current_tau_field</td>
<td>This is the optical depth array for the current absorption line being deposited.</td>
</tr>
<tr>
<td>lambda_field</td>
<td>The lambda field.</td>
</tr>
<tr>
<td>tau_field</td>
<td>This is the total optical depth of all lines and continua.</td>
</tr>
</tbody>
</table>

```
trident.absorption_spectrum.absorption_spectrum.AbsorptionSpectrum.current_tau_field
```

**property** AbsorptionSpectrum.current_tau_field
This is the optical depth array for the current absorption line being deposited. We will do the deposition of lines into this array, and then add it to self.tau_field at the end.

```
trident.absorption_spectrum.absorption_spectrum.AbsorptionSpectrum.lambda_field
```

**property** AbsorptionSpectrum.lambda_field
The lambda field.

```
trident.absorption_spectrum.absorption_spectrum.AbsorptionSpectrum.tau_field
```

**property** AbsorptionSpectrum.tau_field
This is the total optical depth of all lines and continua.

**8.2.3 trident.Instrument**

```
class trident.Instrument(lambda_min, lambda_max, n_lambda=None, dlambda=None, lsf_kernel=None, bin_space='wavelength', name=None)
```

An instrument class for specifying a spectrograph/telescope pair

**Parameters**

- **Lambda_min** float or YTQuantity
  Minimum desired wavelength for generated spectrum in angstroms

- **Lambda_max** float or YTQuantity
  Maximum desired wavelength for generated spectrum in angstroms

- **N_lambda** int
  Number of desired wavelength bins for the spectrum Setting dlambda overrides n_lambda value
  Default: None
**Dlambda** float or YTQuantity

Desired bin width for the spectrum in angstroms Setting dlambda overrides n_lambda value
Default: None

**Lsf_kernel** string

The filename for the LSF kernel Default: None

**Name** string

Name assigned to the Instrument object Default: None

**Methods**

___init___

Initialize self.

**trident.Instrument.__init__**

Instrument.__init__(lambda_min, lambda_max, n_lambda=None, dlambda=None, lsf_kernel=None, bin_space='wavelength', name=None)

Initialize self. See help(type(self)) for accurate signature.

### 8.2.4 trident.LSF

**class** trident.LSF(function=None, width=None, filename=None)

A class representing a spectrograph’s line spread function.

A line spread function can be defined either by a function and width or by a filename containing a custom kernel.

**Parameters**

**Function** string, optional

The function defining the LSF kernel. valid functions are “boxcar” or “gaussian”

**Width** int, optional

The width of the LSF kernel in bins.

**Filename** string, optional

The filename of a textfile for a user-specified kernel. Each line in the textfile contains a normalized flux value of the kernel. For examples, see contents of trident.__path__/data/lsf_kernels Trident searches for these files either in the aforementioned directory or in the execution directory.

**Examples**

Generate an LSF based on a text file:

```python
>>> LSF(filename='avg_COS.txt')
```

Generate a boxcar-based LSF:

```python
>>> LSF(function='boxcar', width=30)
```

Generate a gaussian-based LSF:
Methods

__init__

Initialize self.

trident.LSF.__init__

LSF.__init__(function=None, width=None, filename=None)

Initialize self. See help(type(self)) for accurate signature.

8.2.5 trident.Line

class trident.Line(element, ion_state, wavelength, gamma, f_value, field=None, identifier=None)

A class representing an individual atomic transition. Each Line object is uniquely identified by element, ionic state, wavelength, gamma, oscillator strength, and identifier.

Parameters

Element  string
    The element of the transition using element’s symbol on periodic table Example: ‘H’, ‘C’, ‘Mg’

Ion_state  string
    The roman numeral representing the ionic state of the transition Example: ‘I’ for neutral species, ‘II’ for singly ionized, etc.

Wavelength  float
    The wavelength of the transition in angstroms Example: 1216 for Lyman alpha

Gamma  float
    The gamma of the transition in Hertz

F_value  float
    The oscillator strength of the transition

Field  string, optional
    The default yt field name associated with the ion responsible for this line Example: ‘H_p1_number_density’ for HII

Identifier  string, optional
    An optional identifier for the transition Example: ‘Ly a’ for Lyman alpha

Example

Create a Line object for the neutral hydrogen 1215 Angstroms transition.

```python
>>> HI = Line('H', 'I', 1215.67, 469860000, 0.41641, 'Ly a')
```
trident Line.__init__

Line.__init__(element, ion_state, wavelength, gamma, f_value, field=None, identifier=None)
Initialize self. See help(type(self)) for accurate signature.

8.2.6 trident.LineDatabase

class trident.LineDatabase[input_file=None]

Class for storing and selecting collections of spectral lines. These lines will be used in the SpectrumGenerator and add_ion_fields() functionality.

Without arguments, the LineDatabase will be empty, and you must manually add individual lines to it using the add_line function. If LineDatabase is provided with an optional :input_file:, it will automatically add spectral lines for each corresponding line in the list.

Once created, you can select a subset of the total lines present in the database for further use. Use the parse_subset function to accomplish this.

Parameters

Input_file string, optional
An optional input_file can be provided to pre-store a list of Line objects. input_file should be a tab delimited text file of the format:

    element, ion_state, wavelength, gamma, f_value, (name)

H,  I, 1215.67, 4.69e8, 4.16e-1, Ly a

Example

```python
>>> # Create a LineDatabase using the lines present in lines.txt
>>> ldb = LineDatabase('lines.txt')
```

```python
>>> # Parse ldb and only select Lyman alpha, Mg II and Fe lines
>>> lines = ldb.parse_subset(lines=['H I 1216', 'Mg II', 'Fe'])
>>> print(lines)
```

Methods

__init__
Initialize self.

add_line
Manually add a line to the LineDatabase.

load_line_list_from_file
Load a line list from a file into the LineDatabase.

parse_subset
Select multiple lines based on atom, ion state, identifier, and/or wavelength.

parse_subset_to_ions
Select ions based on those needed to create specific lines.

continues on next page
select_lines

Select lines based on atom, ion state, identifier, and/or wavelength.

trident.LineDatabase.__init__

LineDatabase.__init__(input_file=None)

Initialize self. See help(type(self)) for accurate signature.

trident.LineDatabase.add_line

LineDatabase.add_line(element, ion_state, wavelength, gamma, f_value, field=None, identifier=None)

Manually add a line to the LineDatabase.

Parameters

Element string

The element of the transition using element's symbol on periodic table Example: ‘H’, ‘C’, ‘Mg’

Ion_state string

The roman numeral representing the ionic state of the transition Example: ‘I’ for neutral species, ‘II’ for singly ionized, etc.

Wavelength float

The wavelength of the transition in angstroms Example: 1216 for Lyman alpha

Gamma float

The gamma of the transition in Hertz

F_value float

The oscillator strength of the transition

Field string, optional

The default yt field name associated with the ion responsible for this line Example: ‘H_p1_number_density’ for HII

Identifier string, optional

An optional identifier for the transition Example: ‘Ly a’ for Lyman alpha

Example

```python
>>> # Create a LineDatabase using the lines present in lines.txt
>>> ldb = LineDatabase('lines.txt')

>>> # Manually add the neutral hydrogen line to ldb
>>> ldb.add_line('H', 'I', 1215.67, 469860000, 0.41641, 'Ly a')

>>> print(ldb.lines_all)
```

8.2. Generating Spectra
trident.LineDatabase.load_line_list_from_file

`trident.LineDatabase.load_line_list_from_file(filename)`

Load a line list from a file into the LineDatabase. Line list file is a tab-delimited text file in the format:

- element, ion_state, wavelength, gamma, f_value, (name)

```
H, I, 1215.67, 4.69e8, 4.16e-1, Ly a
```

**Parameters**

- `filename` : string

  The filename of the list to add. First looks in trident.__path__/data/line_lists directory, then in cwd.

trident.LineDatabase.parse_subset

`trident.LineDatabase.parse_subset(subsets='all')`

Select multiple lines based on atom, ion state, identifier, and/or wavelength. Once you’ve created a LineDatabase, you can subselect certain lines from it based on line characteristics. Preferred to use this method over `select_lines`.

Will return the unique union of all lines matching the specified subsets from the LineDatabase.

**Parameters**

- `subsets` : list of strings, optional


  If set to None, selects all lines in LineDatabase. Default: None

**Returns**

- `Line subset` : list of trident.Line objects

  A list of the Lines that were selected

**Example**

```python
>>> # Get a list of all lines of Carbon, Mg II and Lyman alpha
>>> ldb = LineDatabase('lines.txt')
>>> lines = ldb.parse_subset(['C', 'Mg II', 'H I 1216'])
>>> print(lines)
```

trident.LineDatabase.parse_subset_to_ions

`trident.LineDatabase.parse_subset_to_ions(subsets=None)`

Select ions based on those needed to create specific lines. Once you’ve created a LineDatabase, you can subselect certain ions from it based on the line characteristics of atom, ion state, identifier, and/or wavelength. Similar to `parse_subset` but outputs a list of ion tuples (e.g. (`H`, 1), (`Fe`, 2)), instead of a list of Line objects.

Will return the unique union of all ions matching the specified subsets from the LineDatabase.

**Parameters**
Subsets  list of strings, optional

List strings matching possible lines. Strings can be of the form:

If set to None, selects ions necessary to produce all lines in LineDatabase. Default: None

Returns

Ion subset  list of ion tuples

A list of the ions necessary to produce the desired lines Each ion tuple is of the form (‘H’, 1) = neutral hydrogen

Example

Get a list of all ions necessary to generate lines for Carbon, Mg II and Lyman alpha

```python
>>> ldb = LineDatabase('lines.txt')
>>> ions = ldb.parse_subset_to_ions(['C', 'Mg II', 'H I 1216'])
>>> print(ions)
```

trident.LineDatabase.select_lines

LineDatabase.select_lines(element=None, ion_state=None, wavelength=None, identifier=None, source_list=None)

Select lines based on atom, ion state, identifier, and/or wavelength. Once you’ve created a LineDatabase, you can subselect certain lines from it based on line characteristics. Recommended to use parse_subset instead which allows selecting of multiple sets of lines simultaneously.

Parameters

Element  string, optional

The element of the transition using element’s symbol on periodic table Example: ‘H’, ‘C’, ‘Mg’ Default: None

Ion_state  string, optional

The roman numeral representing the ionic state of the transition Example: ‘I’ for neutral species, ‘II’ for singly ionized, etc. Default: None

Wavelength  float, optional

The wavelength of the transition in angstroms Example: 1216 for Lyman alpha Default: None

Identifier  string, optional

An optional identifier for the transition Example: ‘Ly a’ for Lyman alpha Default: None

Source_list  list of Line objects, optional

The source list from which to select lines. If set to None, use the LineDatabase’s list ‘lines_all’. Default: None

Returns

Selected_lines  list

A list of which lines were selected.

Example
```python
>>> ldb = LineDatabase('lines.txt')
>>> selected_lines = ldb.select_lines(element='Mg', ion_state='II')
```

# 8.3 Plotting Spectra

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>load_spectrum</code></td>
<td>Load a previously saved spectrum from disk.</td>
</tr>
<tr>
<td><code>plot_spectrum</code></td>
<td>Plot a spectrum or a collection of spectra and save to disk.</td>
</tr>
</tbody>
</table>

## 8.3.1 trident.load_spectrum

`trident.load_spectrum(filename, format='auto', instrument=None, lsf_kernel=None, line_database='lines.txt', ionization_table=None)`

Load a previously saved spectrum from disk.

### Parameters

- **Filename** string
  - Filename of the saved spectrum.

- **Format** string
  - File format of the saved spectrum file. Valid values are: “auto”, “hdf5”, “fits”, and “ascii”. If you select “auto”, the code will attempt to auto-detect the file format from the extension of the data file: “.h5” or “.hdf5” -> hdf5, “.fits” or “.FITS” -> fits, all other -> ascii. Default: “auto”

- **Instrument** string, optional
  - The telescope+instrument combination to use for the loaded spectrum. Default: None

- **Lsf_kernel** string, optional
  - The filename for the LSF kernel to use for the loaded spectrum. Default: None

- **Line_database** string, optional
  - A text file listing the various lines to insert into the line database to use for the loaded spectrum. Default: None

- **Ionization_table** hdf5 file, optional
  - An HDF5 file used for computing the ionization fraction of the gas based on its density, temperature, metallicity, and redshift. Default: None

### Example

Create a simple spectrum, save it to disk, and load it back as a new SpectrumGenerator object.

```python
>>> import trident
>>> ray = trident.make_onezone_ray()
>>> sg = trident.SpectrumGenerator('COS')
>>> sg.make_spectrum(ray)
>>> sg.save_spectrum('spec.h5')
>>> sg_copy = trident.load_spectrum('spec.h5')
```
8.3.2 trident.plot_spectrum

```
trident.plot_spectrum(wavelength, flux, filename='spectrum.png', lambda_limits=None,
flux_limits=None, title=None, label=None, figsize=None, step=False,
stagger=0.2, features=None, axis_labels=None)
```

Plot a spectrum or a collection of spectra and save to disk.

This function wraps some Matplotlib plotting functionality for plotting spectra generated with the `SpectrumGenerator`. In its simplest form, it accepts a wavelength array consisting of wavelength values and a corresponding flux array consisting of relative flux values, and it plots them and saves to disk.

In addition, it can plot several spectra on the same axes simultaneously by passing a list of arrays to the `wavelength`, `flux` arguments (and optionally to the `label` and `step` keywords).

Returns the Matplotlib Figure object for further processing.

**Parameters**

- **Wavelength** array of floats or list of arrays of floats
  - Wavelength values in angstroms. Either as an array of floats in the case of plotting a single spectrum, or as a list of arrays of floats in the case of plotting several spectra on the same axes.

- **Flux** array of floats or list of arrays of floats
  - Relative flux values (from 0 to 1) corresponding to wavelength array. Either as an array of floats in the case of plotting a single spectrum, or as a list of arrays of floats in the case of plotting several spectra on the same axes.

- **Filename** string, optional
  - Output filename of the plotted spectrum. Will be a png file. Default: ‘spectrum.png’

- **Lambda_limits** tuple or list of floats, optional
  - The minimum and maximum of the wavelength range (x-axis) for the plot in angstroms. If specified as None, will use whole lambda range of spectrum. Example: (1200, 1400) for 1200-1400 Angstroms Default: None

- **Flux_limits** tuple or list of floats, optional
  - The minimum and maximum of the flux range (y-axis) for the plot. If specified as None, limits are automatically from [0, 1.1*max(flux)]. Example: (0, 1) for normal flux range before postprocessing. Default: None

- **Step** boolean or list of booleans, optional
  - Plot the spectrum as a series of step functions. Appropriate for plotting processed and noisy data. Use a list of booleans when plotting multiple spectra, where each boolean corresponds to the entry in the `wavelength` and `flux` lists.

- **Title** string, optional
  - Optional title for plot Default: None

- **Label** string or list of strings, optional
  - Label for each spectrum to be plotted. Useful if plotting multiple spectra simultaneously. Will automatically trigger a legend to be generated. Default: None

- **Stagger** float, optional
  - If plotting multiple spectra on the same axes, do we offset them in the y direction? If set to None, no. If set to a float, stagger them by the flux value specified by this parameter.
Features dict, optional

Include vertical lines with labels to represent certain spectral features. Each entry in the dictionary consists of a key string to be overplot and the value float as to where in wavelength space it will be plot as a vertical line with the corresponding label.

Example: features={'Ly a' : 1216, 'Ly b' : 1026}

Default: None

Axis_labels tuple of strings, optional

Optionally set the axis labels directly. If set to None, defaults to ('Wavelength [$\text{\AA}$]', 'Relative Flux'). Default: None

Returns

Matplotlib Figure object for further processing

Example

Plot a flat spectrum

```python
>>> import numpy as np
>>> import trident

wavelength = np.arange(1200, 1400)
flux = np.ones(len(wavelength))
trident.plot_spectrum(wavelength, flux)
```

Generate a one-zone ray, create a Lyman alpha spectrum from it, and add gaussian noise to it. Plot both the raw spectrum and the noisy spectrum on top of each other.

```python
>>> import trident

ray = trident.make_onezone_ray(column_densities={'H_p0_number_density':1e21})
sg_final = trident.SpectrumGenerator(lambda_min=1200, lambda_max=1300,
dlambda=0.5)
sg_final.make_spectrum(ray, lines=['Ly a'])
sg_final.save_spectrum('spec_raw.h5')
sg_final.add_gaussian_noise(10)
sg_raw = trident.load_spectrum('spec_raw.h5')
trident.plot_spectrum([sg_raw.lambda_field, sg_final.lambda_field],
[sg_raw.flux_field, sg_final.flux_field], stagger=0, step=[False, True],
label=['Raw', 'Noisy'], filename='raw_and_noise.png')
```

8.4 Adding Ion Fields

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_ion_fields</td>
<td>Preferred method for adding ion fields to a yt dataset.</td>
</tr>
<tr>
<td>add_ion_fraction_field</td>
<td>Add ion fraction field to a yt dataset for the desired ion.</td>
</tr>
<tr>
<td>add_ion_number_density_field</td>
<td>Add ion number density field to a yt dataset for the desired ion.</td>
</tr>
<tr>
<td>add_ion_density_field</td>
<td>Add ion mass density field to a yt dataset for the desired ion.</td>
</tr>
<tr>
<td>add_ion_mass_field</td>
<td>Add ion mass field to a yt dataset for the desired ion.</td>
</tr>
</tbody>
</table>
8.4.1 trident.add_ion_fields

trident.add_ion_fields(ds, ions, ftype='gas', ionization_table=None, field_suffix=False, line_database=None, sampling_type='local', particle_type=None)

Preferred method for adding ion fields to a yt dataset.

Select ions based on the selection indexing set up in parse_subset_to_ions function, that is, by specifying a list of strings where each string represents an ion or line. Strings are of one of three forms:

- `<element>`
- `<element> <ion state>`
- `<element> <ion state> <line_wavelength>`

If a line_database is selected, then the ions chosen will be a subset of the ions present in the equivalent LineDatabase, nominally located in trident.__path__/data/line_lists.

For each ion species selected, four fields will be added (example for Mg II):

- Ion fraction field. e.g. (“gas”, ‘Mg_p1_ion_fraction’)
- Number density field. e.g. (“gas”, ‘Mg_p1_number_density’)
- Density field. e.g. (“gas”, ‘Mg_p1_density’)
- Mass field. e.g. (“gas”, ‘Mg_p1_mass’)

This function is the preferred method for adding ion fields to one’s dataset, but for more fine-grained control, one can also employ the add_ion_fraction_field, add_ion_number_density_field, add_ion_density_field, add_ion_mass_field functions individually.

Fields are added assuming collisional ionization equilibrium and photoionization in the optically thin limit from a redshift-dependent metagalactic ionizing background using the ionization_table specified.

**Parameters**

**Ds** yt dataset object

This is the dataset to which the ion fraction field will be added.

**Ions** list of strings


If set to ‘all’, creates all ions for the first 30 elements: (ie hydrogen to zinc). If set to ‘all’ with line_database keyword set, then creates all ions associated with the lines specified in the equivalent LineDatabase.

**Ionization_table** string, optional

Path to an appropriately formatted HDF5 table that can be used to compute the ion fraction as a function of density, temperature, metallicity, and redshift. When set to None, it uses the table specified in ~/.trident/config Default: None

**Field_suffix** boolean, optional

Determines whether or not to append a suffix to the field name that indicates what ionization table was used. Useful when using generating ion_fields that already exist in a dataset.

**Line_database** string, optional

Ions are selected out of the set of ions present in the line_database constructed from the line list filename specified here. See LineDatabase for more information.
Ftype  string, optional
   This is deprecated and no longer necessary since all relevant fields are aliased to the ‘gas’ ftype.
   Default: ‘gas’

Sampling_type  string, optional
   This is deprecated and no longer necessary. Default: ‘local’

Particle_type  boolean, optional
   This is deprecated and no longer necessary. Default: ‘auto’

Example
To add ionized hydrogen, doubly-ionized Carbon, and all of the Magnesium species fields to a dataset, you
would run:

```python
>>> import yt
>>> import trident

>>> ds = yt.load('path/to/file')

>>> trident.add_ion_fields(ds, ions=['H II', 'C III', 'Mg'])
```

8.4.2 trident.add_ion_fraction_field

Add ion fraction field to a yt dataset for the desired ion.

Note: The preferred method for adding ion fields to a dataset is using `add_ion_fields`,

For example, `add_ion_fraction_field('O', 6, ds)` creates a field called `O_p5_ion_fraction` for dataset `ds`, which
represents 5-ionized oxygen (O plus 5 = O VI = ‘O’, 6).

Fields are added assuming collisional ionization equilibrium and photoionization in the optically thin limit from
a redshift-dependent metagalactic ionizing background using the `ionization_table` specified.

Parameters

Atom  string Atomic species for desired ion fraction (e.g. ‘H’, ‘C’, ‘Mg’)

Ion  integer Ion number for desired species (e.g. 1 = neutral, 2 = singly ionized, 3 = doubly ionized,
    etc.)

Ds  yt dataset object This is the dataset to which the ion fraction field will be added.

Ftype  string, optional
   This is deprecated and no longer necessary since all relevant fields are aliased to the ‘gas’ ftype.
   Default: ‘gas’

Ionization_table  string, optional Path to an appropriately formatted HDF5 table that can be used
to compute the ion fraction as a function of density, temperature, metallicity, and redshift. By
default, it uses the table specified in ~/.trident/config

Field_suffix  boolean, optional Determines whether or not to append a suffix to the field name that
indicates what ionization table was used

Sampling_type  string, optional
   This is deprecated and no longer necessary. Default: ‘local’
**Particle_type**  boolean, optional

This is deprecated and no longer necessary. Default: ‘auto’

**Example**

Add C IV (triply-ionized carbon) ion fraction field to dataset

```python
>>> import yt
>>> import trident
>>> ds = yt.load('path/to/file')
>>> trident.add_ion_fraction_field('C', 4, ds)
>>> yt.ProjectionPlot(ds, 'x', 'C_p3_ion_fraction').save()
```

### 8.4.3 trident.add_ion_number_density_field

```python
trident.add_ion_number_density_field(atom, ion, ds, ftype='gas', ionization_table=None, field_suffix=False, sampling_type='local', particle_type=None)
```

Add ion number density field to a yt dataset for the desired ion.

**Note:** The preferred method for adding ion fields to a dataset is using `add_ion_fields`.

For example, `add_ion_number_density_field('O', 6, ds)` creates a field called `O_p5_number_density` for dataset `ds`, which represents 5-ionized oxygen (O plus 5 = O VI).

Fields are added assuming collisional ionization equilibrium and photoionization in the optically thin limit from a redshift-dependent metagalactic ionizing background using the ionization_table specified.

**Parameters**

- **Atom** string
  Atomic species for desired ion fraction (e.g. ‘H’, ‘C’, ‘Mg’)

- **Ion** integer
  Ion number for desired species (e.g. 1 = neutral, 2 = singly ionized, 3 = doubly ionized, etc.)

- **Ds** yt dataset object
  This is the dataset to which the ion fraction field will be added.

- **Ftype** string, optional
  This is deprecated and no longer necessary since all relevant fields are aliased to the ‘gas’ ftype. Default: ‘gas’

- **Ionization_table** string, optional
  Path to an appropriately formatted HDF5 table that can be used to compute the ion fraction as a function of density, temperature, metallicity, and redshift. By default, it uses the table specified in ~/.trident/config

- **Field_suffix** boolean, optional
  Determines whether or not to append a suffix to the field name that indicates what ionization table was used

- **Sampling_type** string, optional
  This is deprecated and no longer necessary. Default: ‘local’

8.4. Adding Ion Fields
**trident Documentation, Release 1.3.dev1**

**Particle_type** boolean, optional

This is deprecated and no longer necessary. Default: ‘auto’

**Example**

Add C IV (triply-ionized carbon) number density field to dataset

```python
>>> import yt
>>> import trident
>>> ds = yt.load('path/to/file')
>>> trident.add_ion_number_density('C', 4, ds)
>>> yt.ProjectionPlot(ds, 'x', 'C_p3_number_density').save()
```

### 8.4.4 trident.add_ion_density_field

**trident.add_ion_density_field**

```
trident.add_ion_density_field(atom, ion, ds, ftype='gas', ionization_table=None, field_suffix=False, sampling_type='local', particle_type=None)
```

Add ion mass density field to a yt dataset for the desired ion.

**Note:** The preferred method for adding ion fields to a dataset is using `add_ion_fields`.

For example, `add_ion_density_field('O', 6, ds)` creates a field called `O_p5_density` for dataset `ds`, which represents 5-ionized oxygen (O plus 5 = O VI).

Fields are added assuming collisional ionization equilibrium and photoionization in the optically thin limit from a redshift-dependent metagalactic ionizing background using the `ionization_table` specified.

**Parameters**

- **Atom** string
  
  Atomic species for desired ion fraction (e.g. ‘H’, ‘C’, ‘Mg’)

- **Ion** integer
  
  Ion number for desired species (e.g. 1 = neutral, 2 = singly ionized, 3 = doubly ionized, etc.)

- **Ds** yt dataset object
  
  This is the dataset to which the ion fraction field will be added.

- **Ftype** string, optional
  
  This is deprecated and no longer necessary since all relevant fields are aliased to the ‘gas’ ftype. Default: ‘gas’

- **Ionization_table** string, optional
  
  Path to an appropriately formatted HDF5 table that can be used to compute the ion fraction as a function of density, temperature, metallicity, and redshift. By default, it uses the table specified in `~/.trident/config`.

- **Field_suffix** boolean, optional
  
  Determines whether or not to append a suffix to the field name that indicates what ionization table was used.

- **Sampling_type** string, optional
  
  This is deprecated and no longer necessary. Default: ‘local’
**Particle_type** boolean, optional

This is deprecated and no longer necessary. Default: ‘auto’

**Example**

Add C IV (triply-ionized carbon) mass density field to dataset

```python
>>> import yt
>>> import trident

>>> ds = yt.load('path/to/file')

>>> trident.add_ion_density_field('C', 4, ds)

>>> yt.ProjectionPlot(ds, 'x', 'C_p3_density').save()
```

### 8.4.5 trident.add_ion_mass_field

`trident.add_ion_mass_field(atom, ion, ds, ftype='gas', ionization_table=None, field_suffix=False, sampling_type='local', particle_type=None)`

Add ion mass field to a yt dataset for the desired ion.

**Note:** The preferred method for adding ion fields to a dataset is using `add_ion_fields`.

For example, `add_ion_mass_field('O', 6, ds)` creates a field called O_p5_mass for dataset ds, which represents 5-ionized oxygen (O plus 5 = O VI).

Fields are added assuming collisional ionization equilibrium and photoionization in the optically thin limit from a redshift-dependent metagalactic ionizing background using the ionization_table specified.

**Parameters**

- **Atom** string
  - Atomic species for desired ion fraction (e.g. ‘H’, ‘C’, ‘Mg’)

- **Ion** integer
  - Ion number for desired species (e.g. 1 = neutral, 2 = singly ionized, 3 = doubly ionized, etc.)

- **Ds** yt dataset object
  - This is the dataset to which the ion fraction field will be added. will be added.

- **Ftype** string, optional
  - This is deprecated and no longer necessary since all relevant fields are aliased to the ‘gas’ ftype. Default: ‘gas’

- **Ionization_table** string, optional
  - Path to an appropriately formatted HDF5 table that can be used to compute the ion fraction as a function of density, temperature, metallicity, and redshift. By default, it uses the table specified in ~/.trident/config

- **Field_suffix** boolean, optional
  - Determines whether or not to append a suffix to the field name that indicates what ionization table was used

- **Sampling_type** string, optional
  - This is deprecated and no longer necessary. Default: ‘local’
Particle_type  boolean, optional

This is deprecated and no longer necessary. Default: 'auto'

Example

Add C IV (triply-ionized carbon) mass field to dataset

```python
>>> import yt
>>> import trident
>>> ds = yt.load('path/to/file')
>>> trident.add_ion_mass_field('C', 4, ds)
>>> yt.ProjectionPlot(ds, 'x', 'C_p3_mass').save()
```

8.5 Miscellaneous Utilities

<table>
<thead>
<tr>
<th>make_onezone_dataset</th>
<th>Create a one-zone hydro dataset for use as test data.</th>
</tr>
</thead>
<tbody>
<tr>
<td>make_onezone_ray</td>
<td>Create a one-zone ray object for use as test data.</td>
</tr>
<tr>
<td>to_roman</td>
<td>Convert an integer to a Roman numeral.</td>
</tr>
<tr>
<td>from_roman</td>
<td>Convert a Roman numeral to an integer.</td>
</tr>
<tr>
<td>trident_path</td>
<td>Return the path where the trident source is installed.</td>
</tr>
<tr>
<td>trident</td>
<td>Print a Trident ASCII logo to the screen.</td>
</tr>
<tr>
<td>verify</td>
<td>Verify that the bulk of Trident’s functionality is working.</td>
</tr>
<tr>
<td>generate_total_fit</td>
<td>Fit an absorption-line spectrum into line profiles.</td>
</tr>
</tbody>
</table>

8.5.1 trident.make_onezone_dataset

trident.make_onezone_dataset (density=1e-26, temperature=1000, metallicity=0.3, domain_width=10.0)

Create a one-zone hydro dataset for use as test data. The dataset consists of a single cubicle cell of gas with hydro quantities specified in the function kwargs. It makes an excellent test dataset through which to send a sightline and test Trident’s capabilities for making absorption spectra.

Using the defaults and passing a ray through the full domain should result in a spectrum with a good number of absorption features.

Parameters

- **Density**  float, optional
  The gas density value of the dataset in g/cm**3** Default: 1e-26

- **Temperature**  float, optional
  The gas temperature value of the dataset in K Default: 10**3

- **Metallicity**  float, optional
  The gas metallicity value of the dataset in Zsun Default: 0.3

- **Domain_width**  float, optional
  The width of the dataset in kpc Default: 10.

Returns

Example
Create a simple one-zone dataset, pass a ray through it, and generate a COS spectrum for that ray.

```python
>>> import trident

>>> ds = trident.make_onezone_dataset()

>>> ray = trident.make_simple_ray(ds,
...     start_position=ds.domain_left_edge,
...     end_position=ds.domain_right_edge,
...     fields=['density', 'temperature', 'metallicity'])

>>> sg = trident.SpectrumGenerator('COS')

>>> sg.make_spectrum(ray)

>>> sg.plot_spectrum('spec_raw.png')
```

8.5.2 `trident.make_onezone_ray`

`trident.make_onezone_ray` function:

- **Density**: float, optional
  - The gas density value of the ray in g/cm**3 Default: 1e-26
- **Temperature**: float, optional
  - The gas temperature value of the ray in K Default: 10**3
- **Metallicity**: float, optional
  - The gas metallicity value of the ray in Zsun Default: 0.3
- **Length**: float, optional
  - The length of the ray in kpc Default: 10.
- **Redshift**: float, optional
  - The redshift of the ray Default: 0
- **Filename**: string, optional
  - The filename to which to save the ray to disk. Due to the mechanism for passing rays, the ray data must be saved to disk. Default: ‘ray.h5’
- **Column_densities**: dict, optional
  - The user can create a dictionary which adds more number density ion fields to the ray. Each key in the dictionary should be the desired ion field name according to the field name format: i.e. “<ELEMENT>_p<IONSTATE>_number_density” e.g. neutral hydrogen = “H_p0_number_density”. The corresponding value for each key should be the desired column density of that ion in cm**-2. See example below. Default: None

Returns

A YT LightRay object

8.5. Miscellaneous Utilities
Example

Create a one-zone ray, and generate a COS spectrum from that ray.

```python
>>> import trident
>>> ray = trident.make_onezone_ray()
>>> sg = trident.SpectrumGenerator('COS')
>>> sg.make_spectrum(ray)
>>> sg.plot_spectrum('spec_raw.png')
```

Create a one-zone ray with an HI column density of 1e21 (DLA) and generate a COS spectrum from that ray for just the Lyman alpha line.

```python
>>> import trident
>>> ds = trident.make_onezone_ray(column_densities={'H_p0_number_density': 1e21})
>>> sg = trident.SpectrumGenerator('COS')
>>> sg.make_spectrum(ray, lines=['Ly a'])
>>> sg.plot_spectrum('spec_raw.png')
```

8.5.3 trident.to_roman

`trident.to_roman(n)`

Convert an integer to a Roman numeral. Only works for integers > 0.

Parameters

- N `int`
  - Integer to convert to Roman numeral

Returns

- String representing the roman numeral equivalent of argument n.

Example

```python
>>> num = to_roman(5)
```

8.5.4 trident.from_roman

`trident.from_roman(s)`

Convert a Roman numeral to an integer.

Parameters

- S `string`

Returns

- Integer value equivalent to s Roman numeral argument.

Example

```python
>>> num = from_roman('V')
```
8.5.5 trident.trident_path

trident.trident_path()
Return the path where the trident source is installed. Useful for identifying where data files are (e.g. path/data). Note that ion table datafiles are downloaded separate and placed in another location according to the ~/.trident/config.tri file.

Example

```python
>>> print(trident_path())
```

8.5.6 trident.trident

trident.trident()
Print a Trident ASCII logo to the screen.

8.5.7 trident.verify

trident.verify(save=False)
Verify that the bulk of Trident’s functionality is working. First, it ensures that the user has a configuration file and ion table datafile, and creates/downloads these files if they do not exist. Next, it creates a single-cell grid-based dataset in memory, generates a ray by sending a sightline through that dataset, then makes a spectrum from the ray object. It saves all data to a tempdir before deleting it.

Parameters

- **Save**  boolean, optional
  By default, verify saves all of its outputs to a temporary directory and then removes it upon completion. If you would like to see the resulting data from verify(), set this to be True and it will save a light ray, and raw and processed spectra in the current working directory. Default: False

Example

Verify Trident works.

```python
>>> import trident
>>> trident.verify()
```
8.5.8 trident.absorption_spectrum.absorption_spectrum_fit.generate_total_fit

trident.absorption_spectrum.absorption_spectrum_fit.generate_total_fit(x, fluxData, orderFits, speciesDicts, minError=0.0001, complexLim=0.995, fitLim=0.97, minLength=3, maxLength=1000, splitLim=0.99, output_file=None)

Fit an absorption-line spectrum into line profiles.

Fits the spectrum into absorption complexes and iteratively adds and optimizes voigt profiles for each complex.

**Parameters**

**X**

(N) ndarray

1d array of wavelengths

**FluxData**

(N) ndarray

array of flux corresponding to the wavelengths given in x. (needs to be the same size as x)

**OrderFits** list

list of the names of the species in the order that they should be fit. Names should correspond to the names of the species given in speciesDicts. (ex: ['lya', 'OVI'])

**SpeciesDicts** dictionary

Dictionary of dictionaries (I’m addicted to dictionaries, I confess). Top level keys should be the names of all the species given in orderFits. The entries should be dictionaries containing all relevant parameters needed to create an absorption line of a given species (f, Gamma, lambda0) as well as max and min values for parameters to be fit

**ComplexLim** float, optional

Maximum flux to start the edge of an absorption complex. Different from fitLim because it decides extent of a complex rather than whether or not a complex is accepted.

**FitLim** float, optional

Maximum flux where the level of absorption will trigger identification of the region as an absorption complex. Default = .98. (ex: for a minSize=.98, a region where all the flux is between 1.0 and .99 will not be separated out to be fit as an absorbing complex, but a region that contains a point where the flux is .97 will be fit as an absorbing complex.)
**MinLength**  int, optional  
number of cells required for a complex to be included. Default is 3 cells.

**MaxLength**  int, optional  
number of cells required for a complex to be split up. Default is 1000 cells.

**SplitLim**  float, optional  
if attempting to split a region for being larger than maxlength the point of the split must have a flux greater than splitLim (ie: absorption greater than splitLim). Default= .99.

**Output_file**  string, optional  
location to save the results of the fit.

**Returns**

**AllSpeciesLines**  dictionary  
Dictionary of dictionaries representing the fit lines. Top level keys are the species given in order. Fits and the corresponding entries are dictionaries with the keys ‘N’, ‘b’, ‘z’, and ‘group#’. Each of these corresponds to a list of the parameters for every accepted fitted line. (ie: N[0], b[0], z[0] will create a line that fits some part of the absorption spectrum). ‘group#’ is a similar list but identifies which absorbing complex each line belongs to. Lines with the same group# were fit at the same time. group#’s do not correlate between species (ie: an lya line with group number 1 and an OVI line with group number 1 were not fit together and do not necessarily correspond to the same region)

**YFit**  
(N) ndarray  
array of flux corresponding to the combination of all fitted absorption profiles. Same size as x.
If you use Trident for a research application, please cite the Trident method paper in your work with the bibtex entry below:

```latex
@ARTICLE{2017ApJ...847...59H,
    author = {{Hummels}, C.-B. and {Smith}, B.-D. and {Silvia}, D.-W.},
    title = "{Trident: A Universal Tool for Generating Synthetic Absorption Spectra from Astrophysical Simulations}",
    journal = {\apj},
    archivePrefix = "arXiv",
    eprint = {1612.03935},
    primaryClass = "astro-ph.IM",
    keywords = {cosmology: theory, methods: data analysis, methods: numerical, radiative transfer },
    year = 2017,
    month = sep,
    volume = 847,
    eid = {59},
    pages = {59},
    doi = {10.3847/1538-4357/aa7e2d},
    adsurl = {http://adsabs.harvard.edu/abs/2017ApJ...847...59H},
    adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```
This document summarizes changes to the codebase with time for Trident.

10.1 Contributors

The CREDITS file has an updated list of contributors to the codebase.

10.2 Version 1.2.3 (March 18, 2020)

This is a bug fix release.

10.2.1 Enhancements

• Move testing to circleci. (PR 109)

10.2.2 Bug Fixes

• Allow access to find_outputs kwarg from make_compound_ray. (PR 126)
• Fix periodic rays. (PR 125)
• Be more careful about limiting size of line deposition array. (PR 106)

10.3 Version 1.2.2 (November 14, 2019)

This is a bug fix release.
10.3.1 Bug Fixes

• Shift wavelength of velocity center to redshift from light ray solution (PR 102)

10.4 Version 1.2.1 (October 1, 2019)

This is a bug fix release.

10.4.1 Bug Fixes

• Logging info doesn’t use correct units (PR 99)

10.5 Version 1.2 (September 19, 2019)

10.5.1 New Features

• Add support for creating spectra in velocity space (PR 90)
• Add ability to set wavelength limits to ‘auto’ (PR 87)
• Add ability to make spectrum from data container (PR 91)

10.5.2 Bug Fixes

• off by one error in subgrid index calculation (PR 85)
• fixing error in _ion_mass (PR 81)
• H_p0_number_density to default, H_number_density to alias (PR 78)
• Implements a better way of calculating the redshift on the light ray (PR 71)
• Assures that ion_fraction field reflects on-disk fields (PR 64)
• Fix atomic data for Si II 1260 according to Morton 2003 (PR 43)
• A check to avoid failure when no continuum absorbers were found in the ray (PR 39)
• Auto-addition of H_nuclei_density to a LightRay when present in the base dataset (PR 39)
• Adding max_box_fraction as a kwarg to make_compound_ray (PR 37)
• Updated trident_path() to be OS Independent (PR 36)
• simplify setting up ion fields using the “local” field type (PR 30)
• split and join filenames using os.sep instead of assuming unix (PR 29)
• updated oscillator strengths and gamma’s for Si II 1206 and Si III 1260 (PR 25)
10.5.3 Minor Enhancements

- Calculating LOS velocity with relative velocities to account for bulk motion (PR 93)
- Enabling use of `output_absorbers_file` kwarg in SpectrumGenerator (PR 58)
- Switching imports from `yt.analysis_modules` to `yt_astro_analysis` (PR 55)
- Enable passing in-memory LineDatabase to SpectrumGenerator (PR 42)
- Added equivalent width calculation to line_observables_dict (PR 40)
- Numerous documentation updates
- Updates and fixes to testing

10.6 Version 1.1 (November 18, 2017)

- Trident development has changed from mercurial to git, and the source has moved from bitbucket to github. This was done in recognition that more people interact with git/github than do with hg/bitbucket, as well as to follow our major dependency yt in making the same transition. All previous repository history (e.g., commits, versions, tags, etc.) is retained under this transition. For users operating on the development branch of Trident, you must re-install Trident in order to continue to get updates. The installation instructions were updated accordingly.

- We totally rebuilt the testing interface to Trident, which includes more coverage in unit tests and answer tests over both grid-based and particle-based datasets. We now have continuous integration through Travis that tests the code daily and with each new pull request to assure consistent code results and to minimize bugs. For more information, see Testing.

- Much of the original Trident codebase was developed in yt as the base classes AbsorptionSpectrum and LightRay. We have now stripped these classes out of yt and moved them entirely into Trident for more flexibility, stability, and autonomy moving forward. This should not affect the user as these changes were behind the scenes.

- Added store_observables keyword to make_spectrum() to store a dictionary of observable properties (e.g., tau, column density, and thermal_b) for each cell along a line of sight for use in post-processing. See source of SpectrumGenerator for more information.

- Added an approximate flux_error field to output spectra, since many observational tools require its presence. See error_func() for more details.

- Made min_tau a keyword to make_spectrum() to enable higher precision (although more time intensive) absorption line deposition.

- Added ability to specify an arbitrary noise vector with add_noise_vector().

- A bugfix was made in yt to the temperature field for Gadget-based code outputs. The internal energy field was mistakenly being read in co-moving instead of physical units, which led to gas temperatures being low by a factor of (1+z). This is now resolved in yt dev and thus we recommend Trident users use yt dev until yt 3.5 stable is released.

- Another bugfix was made in Trident dependency astropy to the convolve function, which is used in apply_lsf(). This may cause slight backwards-incompatible changes when applying line spread functions to post-process spectra.

- Replaced internal instances of particle_type with sampling_type to match similar yt conversion.
10.7 Version 1.0 (November 16, 2017)

Initial release. See our method paper for details.

- Create absorption-line spectra for any trajectory through a simulated data set mimicking both background quasar and down-the-barrel configurations.
- Reproduce the spectral characteristics of common instruments like the Cosmic Origins Spectrograph.
- Operate across the ultraviolet, optical, and infrared using customizable absorption-line lists.
- Trace simulated physical structures directly to spectral features.
- Approximate the presence of ion species absent from the simulation outputs.
- Generate column density maps for any ion.
- Provide support for all major astrophysical hydrodynamical codes.
If you run into problems with any aspect of Trident, please follow the steps below. Don’t worry, we’ll help you get it sorted out.

11.1 Update the Code

The documentation is built for the latest version of Trident. Try Updating to the Latest Version to assure your code matches what the documentation describes. Remember to update to the latest version of yt too.

11.2 Search Documentation and Mailing List Archives

Most use cases for Trident can be found in our documentation and method paper. Try searching through the documentation using the search window in the upper left part of the screen.

If that doesn’t work, try looking at specific problems we might have addressed in our Frequently Asked Questions.

Lastly, try searching the archives of our mailing list. Chances are that someone else may have encountered the problem that you have and already wrote to the list. You can search the list here.

11.3 Contact our Mailing List

Compose a message to our low-volume mailing list. Remember to include details like the operating system you’re using, the type of dataset you’re trying to reduce, the version of Trident and yt you’re using (find it out here), and of course, a description of the problem you’re having with any relevant traceback errors. Our mailing list is located here:

https://groups.google.com/forum/#!forum/trident-project-users

11.4 Join our Slack Channel

We have a slack channel for help and discussions amongst the users and developers of Trident. You can generate an invite for yourself by clicking on this link and following the instructions.
INDEX

Symbols

__init__() (trident.Instrument method), 66
__init__() (trident.LSF method), 67
__init__() (trident.LightRay method), 46
__init__() (trident.Line method), 68
__init__() (trident.LineDatabase method), 69
__init__() (trident.SpectrumGenerator method), 52
__init__() (trident.absorption_spectrum.absorption_spectrum.AbsorptionSpectrum method), 62
A
AbsorptionSpectrum (class in trident.absorption_spectrum.absorption_spectrum), 61
add_continuum() (trident.absorption_spectrum.absorption_spectrum.AbsorptionSpectrum method), 62
add_continuum() (trident.SpectrumGenerator method), 52
add_gaussian_noise() (trident.SpectrumGenerator method), 53
add_ion_density_field() (in module trident), 78
add_ion_fraction_field() (in module trident), 75
add_ion_mass_field() (in module trident), 79
add_ion_number_density_field() (in module trident), 77
add_line() (trident.absorption_spectrum.absorption_spectrum.AbsorptionSpectrum method), 63
add_line() (trident.LineDatabase method), 69
add_line() (trident.SpectrumGenerator method), 53
add_line_to_database() (trident.SpectrumGenerator method), 54
add_milky_way_foreground() (trident.SpectrumGenerator method), 54
add_noise_vector() (trident.SpectrumGenerator method), 55
add_qso_spectrum() (trident.SpectrumGenerator method), 55
apply_lsf() (trident.SpectrumGenerator method), 56
C
clear_spectrum() (trident.SpectrumGenerator method), 57
create_cosmology_splice() (trident.LightRay method), 46
current_tau_field() (trident.absorption_spectrum.absorption_spectrum.AbsorptionSpectrum method), 65
current_tau_field() (trident.SpectrumGenerator property), 61
E
eroerror_func() (trident.absorption_spectrum.absorption_spectrum.AbsorptionSpectrum method), 63
error_func() (trident.SpectrumGenerator method),
F
from_roman() (in module trident), 82
G
generate_total_fit() (in module trident.absorption_spectrum.absorption_spectrum_fit), 84
I
Instrument (class in trident), 65
lambda_field() (trident.absorption_spectrum.absorption_spectrum.AbsorptionSpectrum method), 65
lambda_field() (trident.SpectrumGenerator property), 61
LightRay (class in trident), 44
Line (class in trident), 67
LineDatabase (class in trident), 68
load_line_list_from_file() (trident.LineDatabase method), 70
load_spectrum() (in module trident), 72
load_spectrum() (trident.SpectrumGenerator method), 57
LSF (class in trident), 66

M
make_compound_ray() (in module trident), 41
make_light_ray() (trident.LightRay method), 47
make_onezone_dataset() (in module trident), 80
make_onezone_ray() (in module trident), 81
make_simple_ray() (in module trident), 39
make_spectrum() (trident.absorption_spectrum.absorption_spectrum.AbsorptionSpectrum method), 64
make_spectrum() (trident.SpectrumGenerator method), 58

P
parse_subset() (trident.LineDatabase method), 70
parse_subset_to_ions() (trident.LineDatabase method), 70
plan_cosmology_splice() (trident.LightRay method), 49
plot_spectrum() (in module trident), 73
plot_spectrum() (trident.SpectrumGenerator method), 59

S
save_spectrum() (trident.SpectrumGenerator method), 60
select_lines() (trident.LineDatabase method), 71
SpectrumGenerator (class in trident), 50

T
tau_field() (trident.absorption_spectrum.absorption_spectrum.AbsorptionSpectrum property), 65
tau_field() (trident.SpectrumGenerator property), 61
to_roman() (in module trident), 82
trident() (in module trident), 83
trident_path() (in module trident), 83

V
verify() (in module trident), 83