Manual for Utilizing EDGE.py and Collate.py

1 Preamble

This manual is written to introduce users to the Python code written by Dan Feldman and Connor Robinson. The object of these coding projects is to standardize and help ease the analysis and organization of the models used by the Espaillat group, as well as the DIAD researchers across the collaboration.

For ease of reading, the manual is broken up into sections, so you can skip ahead to those relevant if you need quick reference, or read through for a full overview of the code. I will not be writing up a description of each individual function or argument in the code — this can be found by examining the "docstring" of the function in question. Docstrings are bits of documentation found right after a Python function or class declaration in the code, and if you don't want to go into the code to read it, you can access the docstring directly on the command line by using the help function.

Whenever I am illustrating code that you type into the command line, I will precede the code with ">>>", however, if it's a block of code inside of a file and not at the command line, it will not have the ">>>" preceding it.

Below you'll find some Python jargon, followed by descriptions of some of the more complicated functions and classes written in the code, and then finally a step by step guide of how to use the code to load in data and a model for a T-Tauri star, and then how to plot it and calculate a reduced chi-square value.

NOTE: In most of the code that follows, unless otherwise noted, I will assume that you have imported (i.e., loaded in) the EDGE.py code into your session of Python for use. This is done by typing:

```
>>> import EDGE as edge
```

into Python. This will let you use all functions and classes in the code by typing the name of the function or class preceded by "edge", for example:

```
>>> x = edge.numCheck(2)
```

will store the output of the numCheck function into the variable x. I might therefore in future code omit the import statement from the code. Once you've imported the module, you'll never have to do it again for that session, so I will assume it's already been done.

Ok, let's get to it!

1.1 Jargon

Before we continue, there are some Python specific and non-Python specific pieces of jargon you may not know. These will be important moving forward.

object - A data structure that has associated attributes and methods. Attributes are variables associated with objects that can either describe the object or store data associated to it. Methods are built-in functions that utilize or operate on the object.

class - The numerical recipe for creating objects, along with their attributes and associated methods. So for example, if you think of classes as recipes (how to make a cake), then the cake is the object, which has a bunch of attributes (flavor, taste, cost) and perhaps a intrinsic method that can change the object (the seller changes its price).

pickle - A pickle is a serialized file containing information from a Python object or data structure. These are similar to IDL .sav files, and can be used to save information that you want to re-load later into a new Python session.

module - A Python file which contains functions and/or classes and can be imported so you can use the functions and classes contained within them. Some modules come with your Python installation (numpy, matplotlib, etc.) and some can be ones you've written yourself (EDGE). This is similar to a .pro file in IDL.

2 Paths

In the beginning of the EDGE.py file, I define a few paths (e.g., figurepath, datapath, etc.) which define where certain files exist or will exist. You do not have to define these if you don't want to — all relevant functions and classes will have a keyword that lets you supply the proper path. The paths hardcoded at the top of EDGE are very useful when you are consistently using the same directory for your data files, as you can then avoid typing in the path for each function call, but you don't need to use them.

3 Important Functions and Classes

This section will contain the important functions and classes contained within EDGE.py and collate.py that are complicated and necessary for use. This will not cover many of the smaller, independent functions utilized by the code. For information on those functions, please refer to their docstrings.

3.1 Collate

Loading in Anaconda Python on the Cluster

In order to use Python on the SCC (and therefore to use collate on the cluster), you must load in the Anaconda package. Once you have logged in to the cluster, you can load in Anaconda by typing in:

```
>>> module load anaconda
```

Once this is done, you have access to IPython, as well as all the necessary modules for Python. When you are ready to use collate, just enter IPython and follow the next section's instructions.

Collate

Collate.py is the Python version of Connor's "collate" function, which takes the output files of a given model run and stores all the information and data into one fits file for later reference. In the Python version, the collate.py file contains two functions, collate() and numCheck(). However, collate() is all you need; numCheck(), which also is written in EDGE.py, is used by the function for convenience. In order for collate to work properly, your labelend for your models (optically thin dust models or otherwise) must follow the proper naming convention, which will be outlined later. The inputs and keyword arguments can be found in the collate.py docstring. At present, you can only run collate() for one model at a time, so if you have, say 100 models to use this function on, you have to run it in a loop:

```
from collate import collate
path = 'Some path on the cluster where your model files are located'
name = 'Name of your object, specifically one used in your labelend'
for i in range(100):
   collate(path, i+1, name, path)   # This assumes your destination is same as input path
```

This should collate all of your model runs into fits files that you can later use for data analysis and plotting. You can set a destination path if you want the output in a different place. This would replace the second instance of "path" in the function call above.

3.2 Classes

TTS_Model

The TTS_Model class creates objects that contain the data from an individual model run. **Note: This is only utilized for full or transitional disk models, not for pre-transitional disk models.** In order for this class to work, all of the data and meta-data related to the model must be in a fits file created from Connor's collate.py code. TTS_Model essentially loads everything from that fits file and saves it into a Python object, as well as creating a method that can compute the total of all the model components. A list of all of the meta-data and data loaded into the module from the fits file can be found in the docstring.

TTS_Model currently has three methods. The first is the one necessary to all classes, which is the __init__ function, sometimes called the “constructor.” This creates “instances” of the class, i.e., creates individual objects. If the class is the recipe, and the object is the cake, then the __init__ function is the cook. Here it loads all of the meta-data into an object, comprised primarily of the model parameters. For example, if you want to load the third model for CVSO109 into an object, you would type:

```
>>> cvso109_3 = edge.TTS_Model(‘cvso109’, 3)
```

There are some optional keyword arguments you can supply to TTS_Model that may change what it does. For example, the dpath keyword is used to tell the class the path where the fits file is located. If you are working a lot in the same directory, you are encouraged to define the data path at the top of the code in the “datapath” variable, and then you will not have to manually supply a path to the dpath keyword. Otherwise, this is a necessity. For a list of all keywords, please see the docstring.

You may notice that I don’t explicitly call the __init__ function. That is because in Python, when you call the class, it knows to call on the __init__ function to create the object.

The second method is the dataInit method. This method loads the actual data into the object. When you call this method for TTS_Model, you need not supply any keywords. So an example call for the above object looks like this:

```
>>> cvso109_3.dataInit()
```

The third method is the calc_total method. This takes all of the components of the model and adds them together to create a “total” flux array. The method has a bunch of keywords that describe each component you may want to add to the total, and some of them are turned on by default, and others are turned off by default. The ones turned on are phot (photosphere), wall (inner wall), and disk. The one turned off is dust (optically thin dust). To turn these on and off, you just specify them when you call the method and set them to 0 or 1. For example:

```
>>> cvso109_3.calc_total(iwall=0)
```

This will keep all of the defaults except for inner wall, which I turned off.

Note: The dust keyword is an exception to this. Since there can be any number of optically thin dust files, you have to instead specify which one you want, by supplying the associated dust model number. This will also follow the convention created by collate. So if you want to utilize, for example, the fourth dust file, then you type:

```
>>> cvso109_3.calc_total(dust=4)
```

TTS_Model utilizes a nested dictionary structure to hold all the data for the model. A dictionary is a data structure in Python that hold key-value pairs. An example dictionary is as such:

```
>>> dict = {‘Key1’: 1, ‘Key2’: 2, ‘Key3’, 3}
>>> print dict[‘Key2’]
output: 2
>>> print dict.keys()
output: [‘Key1’, ‘Key2’, ‘Key3’]
```

In TTS_Model, there are two layers, where the initial set of keys are the components of the model, e.g., ‘phot’, ‘iwall’, ‘disk’, ‘total’, etc. and the second set of keys are ‘wl’ and ‘lFl’, which hold the arrays of wavelength (in microns) and λF_λ (in ergs s-1 cm-2) respectively. This nested dictionary is held in the ‘data’ attribute. So to print the flux values of the disk component, you’d type:

```
>>> print cvso109_3.data[‘disk’][‘lFl’]
```

There are a few extra keywords you can utilize in the calc_total method. Some of these include changing the altinh used for the inner wall, saving the components into a .dat file, etc. For a full list of keywords, see the docstring. *If you have scattered light component in your model, the code will always automatically include it in the total.*

PTD_Model

PTD_Model is a class almost identical to TTS_Model, except is used for pre-transitional disk models. In technical terms, PTD_Model is a class that “inherits” from the parent class of TTS_Model: this means that all of the code for PTD_Model is identical to TTS_Model except where changed in the code. The major differences are seen only in the second two methods, dataInit and calc_total.

Unlike TTS_Model, PTD_Model requires keywords when calling dataInit. The reason for this is that PTD_Model needs to match up your disk model with an inner wall model. There are two ways got do this. You can either utilize the “jobw” keyword and supply the number of the job matching the inner wall file. This is the easiest method. However, if you do not know which inner wall file is the correct one, you can supply it with keywords matching the header file with the relevant parameters matching the wall (e.g., amaxs, eps, alpha, temp, etc.). In this case, dataInit will find which model matches the wall and will then load it in. Additionally, if you have an inner wall with a different name than your disk, you can use the “altname” keyword to allow dataInit to find the correct file.

In calc_total, the procedure is the same, but there is an added keyword to turn on and off the outer wall ('owall') component of your model, as well as to change either the altinh of your inner wall and/or your outer wall ('altInner' and 'altOuter' keywords).

TTS_Obs

The TTS_Obs class creates objects that hold the observations for a given T-Tauri star. This includes all spectra and photometry. Unlike with TTS_Model, TTS_Obs's __init__ method initializes a mostly-empty object, and then requires you to utilize its methods to fill in the object with data. As such, once you have loaded in the observations to an object, you need to save it as a pickle so you can just load it in later, rather than having to build it every time.

The TTS_Obs class also utilizes a nested dictionary structure to hold the observations. Here however, there are multiple attributes which hold data, namely 'spectra' and 'photometry.' The first level of the keys holds the names of the instrument or telescope (e.g., 'DCT', 'IRS', 'PACS', etc.) and the second level of keys are 'wl,' 'lFl,' and 'err,' which holds the wavelength, λF_λ, and error arrays respectively.

When you first initialize the TTS_Obs object, you only supply it the name of your target. So if I'm working with CVSO109, I would type:

```
>>> cvso109_obs = edge.TTS_Obs('cvso109')
```

This would initialize an object with the name attribute to hold 'cvso109,' and it would have empty spectra and photometry dictionaries. It would also initialize an empty list with the attribute name 'ulim' which will potentially hold the names of data containing upper limits.

To fill in the observations, you have to make use of the add_spectra and add_photometry methods. This will take the supplied data and meta-data and fill in the nested dictionary structure for you. If you are overwriting the data, it will also ask you to make sure you wish to overwrite the data before proceeding. Later in this manual, I will show an example of how to create this type of object, so you can see later how this is done in more detail.

The last method in this class is the SPPickle function. This function will save your observations as a pickle file so you can load it back into Python later. **Note: If you reload the EDGE.py module <u>before</u> you save your object into a pickle, the SPPickle function will NOT work. So be careful of this issue when creating a new observations object.**

3.2 Functions

look

The look function is our plotting routine for TTS observations and models. The first two bits you want to provide it are observation and model objects created by the TTS_Obs and TTS_Model/PTD_Model classes (see section 2.2). The other keywords are important for various customizations, such as colors, whether or not to combine the disk and outer wall components, etc. See the docstring for full details on keywords.

loadPickle

The loadPickle function takes a pickle created by the TTS_Obs class and reloads it into your current Python session. This function is smart enough to be able to handle if you have multiple pickles for the same object, so long as you know which of them is the correct one (it will also warn you if you have multiple pickles and didn't realize it).

job_file_create

The job_file_create function will take a sample job file (to be used to run a model on the cluster) and make the desired changes to it. In the docstring you can see all of the different changes the function can handle making to the file. If you have a table with a number of changes (perhaps to make a grid), you can make a simple loop to create all of the job files for your grid. For example, lets say you want to create 3 job files for CVSO109, with the following parameters in a .txt table as shown below:

```
[in the file job_params.txt]
Alpha, Epsilon, Amax, Rdisk
0.1, 0.1, 0.25, 50
0.01, 0.1, 0.25, 100
0.01, 0.01, 0.05, 50
```

The following code can be used to load in the table, load in the EDGE function, and loop to create the desired job files. **Please note** that your job_sample file must have an amax value of 0.25 and an epsilon value of 0.0001 in order to correctly work.

```
import EDGE as edge
from astropy.io import ascii

gridpath = '/string/with/the/correct/path/goes/here/'
table = ascii.read('job_params.txt')
for i in range(3):                                    # i will loop from 0 to 2
   label = 'cvso109_' + edge.numCheck(i+1)     # makes the correct labelend value
   edge.job_file_create(i+1, gridpath, alpha=table['Alpha'][i], epsilon=table['Epsilon'][i],
                              amaxs=table['Amax'][i], rdisk=table['Rdisk'][i], labelend=label)
```

This code will create an output where the names of the job files are job001, job002, and job003. For much larger grids, you just have to change the range in the *for* loop, and if you have over 999 jobs, you need to pass the *high* keyword to the numCheck function.

If you have a really large grid you want to run, and you want to quickly create a table like the one above, you can use code like this:

```
import itertools
eps = [0.01, 0.001, 0.0001]
alpha = [0.1, 0.01, 0.001, 0.0001]
rdisk = [40, 50, 60, 70, 80, 90, 100]
amax = [0.25, 1.0, 3.0, 5.0, 100.0]
f = open('job_params.txt', 'w')
f.writelines('Job Number, Epsilon, Alpha, Rout, Amax \n')
for ind, values in enumerate(itertools.product(eps, alpha, rdisk, amax)):
    f.writelines(str(ind+1)+', '+ str(values)[1:-1]+ '\n')
f.close()
```

This will create a file called 'job_params.txt' that contains a table that can be read in similar to the above job_create code. It will contain all possible combinations of parameters used in the itertools.product loop, and the associated "number" so you can reference parameter combinations by number later.

job_optthin_create

The job_optthin_create function is similar to the job_file_create function above, except it creates job files for the optically thin dust models. The function call is identical to job_file_create, except it has different keyword arguments that you can change in the file. For a full list of these parameters, see the docstring. Again, if you wish to create a grid of models, you can modify the code in the above section to be purposed for these types of job files.

model_rchi2

The model_rchi2 function takes the observation and model objects for a given T-Tauri star and calculates the reduced chi-squared value. As an example, let's say you ran the 3 job grid in the above snippet of code and now want to calculate the chi-squared. Then you can run this code to calculate the chi-squared value and save it into an array:

```
import numpy as np
import EDGE as edge
# Going to assume the path to the model data is the same as gridpath defined above
#cvso109_obs = edge.loadPickle('cvso109', picklepath=gridpath)
chi = np.zeros(3)              # An array of zeros of length 3
for ind in range(3):
    try:
        model = edge.TTS_Model('cvso109', i+1, dpath = gridpath)
    except IOError:
        print 'Model ' + str(i+1) + ' does not exist.'
        continue
    model.calc_total(verbose=0)
    chi[i] = edge.model_rchi2('cvso109', model, gridpath)
    #edge.look(cvso109_obs, model=model, jobn=i+1, save=1)
print chi
```

This code will skip over any jobs that may have failed. So if you ran a large grid and some of the jobs are missing (because they failed), this will just have values of 0.0 for the failed jobs. If you want to create plots of the jobs as well, you can get rid of the '#' in front of the cvso109_obs line and the edge.look() line of code. If you have more than 3 jobs to loop over, you only have to adjust the input to the np.zeros function and the range of the loop.

4 Example of How to Use the Code

To get an idea of how to utilize the tools in this code to analyze data, I'm going to walk you through a full example. We will start with the assumption that you have run 1 job file for CVSO109 on the SCC cluster and would now like to 1) use collate.py to organize the output, 2) create observation and model objects for analysis of CVSO109, 3) plot the model, and 4) calculate the model's reduced chi-squared statistic.

In order for this to work, the job file should have 'cvso109_001' as the labelend parameter. If this was any other job number, it should be 'cvso109_xxx' where xxx is the 3 numbered representation of the job number (e.g., 002, 013, 134). This is crucial for using collate correctly. For other objects, just change the name before the underscore… it's only the number convention that is important.

To use collate, we get onto the cluster into a directory containing collate.py. Then, we start IPython and call the function, which takes at least 4 inputs: a path, job number, name, and destination (often the same path as the input path):

```
>>> ipython
>>> from collate import collate
>>> path = 'some path here'
>>> collate(path, '001', 'cvso109', path)
```

This should create a fits file in the path that will be called 'cvso109_001.fits'. There are some optional keywords you can use for collate — see Connor's documentation for more details.

Note: If you run a grid of over 999 runs, you will need to use 4 numbered names, i.e., name_xxxx as the labelend. When using the code in EDGE.py, you will need to use the high keyword argument to specify that the convention is 4 numbers long.

Ok, so this file now has all of the information for the model in one place. For the sake of argument, I am going to assume for this tutorial that all necessary files are all in the same path, and that I have defined this path in EDGE.py as both figurepath and datapath. As a result, I don't have to pass any arguments to certain functions' keyword 'dpath'.

What we want to do next is create the observations object using TTS_Obs. Assuming we have started Python (or IPython, which is better), and gotten into the correct directory:

```
>>> import EDGE as edge
>>> cvso109obs = edge.TTS_Obs('cvso109')
```

This created the empty object, which we will want to fill in. Let's assume we have a file that contains the dereddened IRS spectra of CVSO109, called 'CVSO109_dered_spec.txt' which has two columns, the first for wavelength (in microns) and the second for flux (in Janskys). This is the typical output of the dereddening code we have, and it will have 4 lines of comments at the top of the file. Now we want to load this into the object, but first, we will have to convert the flux to erg s-1 cm-2 units:

```
>>> import numpy as np
>>> dered_spec = np.loadtxt('CVSO109_dered_spec.txt', skiprows=4)
>>> spec_wl = dered_spec[:,0]     # All the wavelengths into a 1D array
>>> spec_flux = edge.convertJy(dered_spec[:,1], spec_wl) # Converts to flux units
>>> cvso109obs.add_spectra('IRS', spec_wl, spec_flux)
```

Now the cvso109obs object has the IRS spectra in its spectra attribute. You can double check this by typing:

```
>>> print cvso109obs.spectra['IRS']
```

If there were error values, you could store them by passing them to the 'errors' keyword of the add_spectra function.

Now let's assume we have de-reddened photometry in a file called 'CVSO109_dered_phot.txt', also with 4 lines of comments. We can read it in the same way as the spectra:

```
>>> dered_phot = np.loadtxt('CVSO109_dered_phot.txt', skiprows=4)
```

Let's also assume the first column is wavelength (in microns) and the second column is the magnitude in units of mag. For argument's sake, let's say there are 5 data points, which are DCT data (UBVRI mags). We can use the function convertMag to convert the magnitudes into fluxes. However, in its current form, it can only take one data point at a time. So we can create a new function using numpy's vectorize function:

```
>>> mag_array = dered_phot[:,1]
>>> photwl = dered_phot[:,0]
>>> newMagConvert = np.vectorize(edge.convertMag)
>>> bands = np.array([‘U’, ‘B’, ‘V’, ‘R’, ‘I’])
>>> photflux = newMagConvert(mag_array, bands)
```

Now we can add the photometry to our observations object using the add_photometry method:

```
>>> cvso109obs.add_photometry(‘DCT’, photwl, photflux)
>>> print cvso109obs.photometry          # Check it worked
```

For argument’s sake let’s say we also have an SMA upper limit to add to the object:

```
>>> smaFlux = [flux value in correct units]
>>> smaErr = [error value in correct units]
>>> smaWL = 1e3              # 1000 microns
>>> cvso109obs.add_photometry(‘SMA’, smaWL, smaFlux, errors=smaErr, ulim=1)
>>> print cvso109obs.photometry          # Should contain both entries now
>>> print cvso109obs.ulim                # Should only have SMA
```

At this point we are done with the observations, but if you had more data you could just continue the process outlined above for all data you have. See the convertMag docstring for a full list of possible magnitude conversions. If you’d like to add one (or have me add it), shoot me an email.

Since we are done, it is best to immediately save this object into a pickle so we can reload it into later Python sessions. We do this using the SPPickle method:

```
>>> picklepath = ‘path string with where you want the pickle’
>>> cvso109obs.SPPickle(picklepath)
```

Now you should have a .pkl (pickle) file in that directory. To test if it worked, you can load this pickle into a new variable using the loadPickle function:

```
>>> cvsoObs2 = edge.loadPickle(‘cvso109’, picklepath)
```

Now cvsoObs2 should be an identical object to that of cvso109obs. After we create the observations object, we should now create the model object. If you recall, this was job 1 and is located in the data path defined as ‘datapath’ at the beginning of the code. So to load it, we type:

```
>>> model109_1 = edge.TTS_Model(‘cvso109’, 1)
>>> model109_1.dataInit()
>>> model109_1.calc_total()
```

This loads in the data and meta-data as well as creating a ‘total’ component which is just the sum of all the others. Now it is ready for plotting:

```
>>> edge.look(cvso109obs, model=model109_1, jobn=1, save=1)
```

This will create a plot of your observations and model and output it into the path defined as ‘figurepath’ at the top of the code. If you wish to view and change the plot interactively, you can do so by not changing the save keyword (0 is the default). More info about the look function can be found in the docstring.

I’ve had a known issue with plotting that sometimes it comes out looking weird if you don’t first view it on your screen before saving. I’m trying to fix that, so if it occurs for you, please let me know. Hopefully I’ll have it fixed soon.

Lastly, we want to calculate the reduced chi-squared value. At this point, it’s straightforward:

```
# Assuming here that the observations are in the picklepath:
>>> chi = edge.model_rchi2(‘cvso109’, model109_1, picklepath)
>>> print chi
```

That should do it! Make sure you have run SPPickle before you calculate the reduced chi-squared, as it will try to read in the observations via the saved pickle file. Otherwise it won’t grab them.

5 Finale

This code is a living entity, and so this manual will potentially change as the code changes. If there are any questions/comments on EDGE.py or this manual, please email me at danfeldman90@gmail.com, and if there are questions/comments about collate.py, please email Connor at connorr@bu.edu. You can also raise issues about bug fixes or additional desired functionality on GitHub (https://github.com/danfeldman90/EDGE).

Dan