Draft version June 19, 2017
Preprint typeset using LaTeX style emulateapj v. 12/16/11

MANUAL FOR UTILIZING EDGE AND COLLATE

C.Robinson[1], D. Feldman
Draft version June 19, 2017

ABSTRACT

The D'Alessio Iradiated Disk (`DIAD`) models can be ungainly to work with effectively. To remedy this, we have constructed Python based analysis tools '`EDGE`' and `collate`. The objective of `EDGE` is to standardize and help ease analysis and organization of the `DIAD` models used by 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. This document does not contain a description of each individual function or argument in `EDGE` – this can be found by examining the 'docstring' of the function in question.

1. INTRODUCTION

1.1. *Basic python usage*

Whenever illustrating code that you type into the command line, the code will be preceded by '`>>>`', 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. Code entered into a terminal outside of python is preceded by '`[UNIX]`'. Most of the code that is shown here is written for use at the command line, but for reproduciability it is often better write code within scripts with `.py` file extensions. Scripts in Python can be run at the terminal using:

```
[UNIX] python name_of_script.py
```

or during a Python session with:

```
>>> run name_of_script
```

Below you will 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.

In most of the code that follows, unless otherwise noted, it is assumed that you have imported (i.e., loaded in) `EDGE` 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`. Once you have imported the module, you will not have to do it again for that session, so it will be assumed it has already been done.

1.1.1. *Adding `EDGE` to your Python path*

Note that in order for Python to find `EDGE` , the directory must be placed on your Python path. This can be done by changing your `.bashrc` (or equivalent) file generally located as a hidden file in your home directory. Typing the following command into a terminal (outside of Python),

```
[UNIX] ls -a ~/.bashrc
```

should show you this file. Adding the following line with the path to where you have saved `EDGE` to that file with the text editor of your choice will allow Python to find `EDGE` .

```
export PYTHONPATH=$PYTHONPATH:/path/to/EDGE/
```

Take care to change the directory to where ever you have placed texttt`EDGE`.

1.1.2. *Reloading a function*

If you make changes to a function and do need to reload that function, that is handled using the built-in Python function '`imp`'. To reload `EDGE`, (or any other module) do the following:

```
>>> import imp
>>> imp.reload(edge)
```

1.2. *Jargon*

Before we continue, there are some Python specific and non-Python specific pieces of jargon that are 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 infor-

connorr@bu.edu
[1] Department of Astronomy, Boston University, 725 Commonwealth Avenue, Boston, MA 02215, USA

mation 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, a few paths are defined (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. **IMPORTANT:** If you obtain a new version of `EDGE` from GitHub, you **will** need to change the paths again.

If you are using scripts (recommended), it is often better to simply include the path as a part of the call to each function, as it increases portability of the script. This also negates the need to change the paths if you re-download the code from GitHub.

3. IMPORTANT FUNCTIONS AND CLASSES

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

3.1. *Collate*

The Python module `collate` takes the output files of a given model run and stores all the information and data into one `.fits` file for later reference. 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` docstring. Note that `collate` is not included with `EDGE` and must be imported separately. Generally, `collate` is used on the **SCC**, after the models have finished running.

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:

```
[UNIX] module load anaconda3
```

Once this is done, you have access to Python, as well as all the necessary modules for Python. When you are ready to use collate, enter

```
[UNIX] python
```

into your terminal.

Models can be collated individually by entering all the required inputs (see the docstring), but the easiest way to collate models is by using `masscollate`. An example for a star named 'fancyStar' is show below:

```
>>> import collate
>>> collate.masscollate('fancyStar')
```

This should collate all of your model runs into `.fits` files that you can later use for data analysis and plotting.

For more advanced users: `collate` by default also stores information about the structure of the disk in a separate fits extension (extension 1). The data in this extension is stored as a 2D array with the labels for the columns located in the header.

`collate` is also used to gather information about the `DIAD` optically thin dust models and the (**?**) shock models. Modes for collating these types of models can be toggled using the `optthin = True` or the `shock = True` keywords respectively.

3.2. *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 `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 'fancyStar' into an object, you would type:

```
>>> fancyStar_3 = edge.TTS_Model('fancyStar',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 the the `__init__` function is not explicitly called. That is because in Python, when a class is called, 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:

```
>>> fancyStar_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. Basic usage of this function is as follows:

```
>>> fancyStar_3.calc_total
```

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:

```
>>> fancyStar_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:

```
>>> fancyStar_3.calc_total(dust=4)
```

textttTTS_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 $\lambda F\lambda$ (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.

3.3. *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`; 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.

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).

An example of this with our imaginary 'fancyStar' is shown below:

```
>>> fancyStar_PTD = edge.PTD_Model('fancyStar',1)
>>> fancyStar_PTD.dataInit(jobw = 30)
>>> fancyStar_PTD.calcTotal(altinner = 1.5)
```

In this case, we have loaded in our disk model of 'fancyStar', initialized it using the inner wall file associated with the 30^{th} model in our grid, and then calculated the total emission assuming an inner wall height of 1.5 scale heights.

3.4. *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, $\lambda F\lambda$, and error arrays respectively. When you first initialize the `TTS_Obs` object, you only supply it the name of your target. So if you are working with 'fancyStar', you would type:

```
>>> fancyStar_obs = edge.TTS_Obs('fancyStar')
```

This would initialize an object with the name attribute to hold 'fancyStar', 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.

3.5. *SPPickle*

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 module before 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. You can save pickles using the following:

```
>>> fancyStar_obs.SPPickle(clob = True)
```

The clob = True flag means that the function will 'clobber', or overwrite, any existing Pickles with the same name. This is extremely useful for scripts where you would want to create a new Pickle from scratch each time you run the script. If you are feeling more cautious, you can set the clob = False to make it so SPPickle will not overwrite the existing pickle file and instead create a new pickle with a number associated with it.

3.6. *Red_Obs*

In general, most of the actual loading data into an EDGE observation object **will not be done using** TTS_Obs, and instead will be done using Red_Obs (unless your data has already been de-reddened outside of the EDGE architecture). Red_Obs is nearly identical in function to TTS_Obs but contains an additional function that will de-redden all the data in the Red_Obs object and create a new pickle file containing a TTS_Obs object. The process for dereddening for our object 'fancyStar' is shown below.

```
>>> red = edge.Red_Obs('fancyStar')
```

Once the 'red' object is created you must then load data into it using add_photometryand add_spectra functions as before. When this is finished, the data can be de-reddened.

```
>>> Av = 0.8
>>> Av_unc = 0.2
>>> law = 'mathis90_rv3.1'
>>> picklepath = /path/to/fancyStar/pickles/'
>>> red.dered(Av, Av_unc, law, picklepath)
```

This code segment defines the extinction at Johnson V band, the uncertainity in the extinction, the destination where the newly de-reddened pickle will be stored, and finally de-reddens the data and creates the TTS_Obs pickle.

3.7. *look*

The look function is our plotting routine for TTS observations and models. Provide it with an observation object and optionally a model object created by the TTS_Obs and TTS_Model/PTD_Model classes (see section 2.2) to create plots. 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. Example code for the look function is as follows:

```
>>> edge.look(fancyStar_obs, model =
fancyStar_3)
```

3.8. *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). A pickle can be loaded with:

```
>>> fancyStar_obs = edge.loadPickle('fancyStar')
```

3.9. *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. The best way to run this command is through the jobmaker.py script, which has all the parameters in an easily editable form, and has the ability to makes large grids of models at once. More about jobmaker.py will be discussed later on in the section on scripts.

3.10. *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. A similar script to jobmaker.py has been written for these optically thin dust models as well: ojobmaker.py

3.11. *model_rchi2*

The model_rchi2 function takes the observation and model objects for a given T-Tauri star and calculates the reduced χ^2 value. This is useful as a quantitative representation of how well the model fits the data. It has the ability to weight spectra and photometry differently, and calculate a non-reduced χ^2 value as well.

4. USING THE CODE VIA SCRIPTS

The previous sections discussed some of the more important functions + modules for running an analyzing models individually. In this section, scripts that use EDGE commands in parallel with other python code are presented. Although all of what has been covered earlier can be done at the command line, scripts allow for vastly increased repeatability, transportability and easier bug solving. To get an idea of how to utilize the tools in this code to analyze data, several scripts have been included that model an object from start to finish.

Inside the DEMO folder in your EDGE distribution are the following scripts:

```
DEMO_make_imlup.py
DEMO_jobmaker.py
DEMO_analysis_imlup.py
```

along with two directories:

```
data/
models/
```

which contain (unsurprisingly) data and models. Inside the data directory is a list of photometry from Vizier in the form of a .vot file, an IRS spectra in the form of a .fits file, and a de-reddened pickle file of the observations. The model directory contains job files (e.g.,

'job005') and DIAD results in the form of collated .fits files (e.g., 'imlup_005.fits')

4.1. *Making the observation pickle*

DEMO_jobmaker_imlup.py will take in photometry and spectra from the DATA directory in order to make the Red_TTS observation object which will then be dereddened and saved to a .pkl file containing the dereddened TTS_Obs object. This object can be looked at the command line using the look function. Uncertainties associated with flux measurements can also be stored in this object/pickle.

Note: While creating this file for other objects, one must be mindful of the units that the observations are entered in. EDGE is constructed to work with units of $ergs\,s^{-1}\,cm^{-2}$ *(*$\lambda F\lambda$*). If your observations have other units, you will need to convert them. EDGE does have several useful functions for doing so, e.g., convertMag, convertJy. Units of wavelength must be in microns.*

4.2. *Making the job files*

The next step is to make the job files for DIAD. This is done using the DEMO_jobmaker_imlup.py. This file will create a small grid of models with different values of amaxs, the maximum grain size, and alpha, the viscosity in the disk. For the purposes of this tutorial, the jobs for this grid of models have already been run and collated for you, with the fits files placed in the demo folder.

4.3. *Running and collating models*

In practice, these job files would be moved to the SCC using the UNIX command scp. For large grids, it is recommended that jobs are submitted using a run_all.csh script (also found in the DEMO directory). Once the jobs have all finished running, the results must be collated. This is most easily done using masscollate. To do this, load Python on the SCC and type the following:

```
>>> import collate as c
>>> c.masscollate('fancyStar')
```

This will combine the results from each model into a single .fits file which can then transferred back to your local machine using scp again.

Since the jobs have already been run, collated and copied over, you can skip this section if you are following along with this demonstration.

4.4. *Analysis of the models*

Note: Unlike the previous two steps involving the models, it is likely that you will need to write your own analysis code depending on your needs. The steps taken to find the best fitting model for the real star IM Lup are described below, and should be taken as an example, but not necessarily as a rule.

The data that was organized and de-reddened by the DEMO_make_imlup.py can be compared against the DIAD models. This will be done using the DEMO_analysis_imlup.py script. The height of the inner wall (altinh) can be adjusted after models have finished running. This is useful because we can allows fit the height of the wall without running additional models. In this example, we will be searching for the best fitting model using the grid of models that we ran and adjusting the inner wall height. The script will loop over both job number and wall heights.

For each job number, the script loads in the model and it checks to see if the job failed using Python's built in error handling and the 'failed' tag from collate. Next it initializes the model object using the dataInit function.

We then enter the for loop over inner wall height and calculate the total emission from all of the model components using the calc_total function. The χ^2 value for this total emission is calculated and stored. Next the script searches for the lowest χ^2 value for all the different wall heights and selects the lowest value. Using this wall height, a plot is made and saved by the look function, and the value of χ^2 is stored along with the job number and the wall height. This repeats for each job number. After running the script, the best models can be found by doing the following at the command line:

```
>>> print(chi2[order])
```

where order is an array of indices found by using numpy.argsort on chi2[:,1], which contains all the χ^2 values. The best model will also be plotted as a result of running the script.

5. CONCLUSION

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 , this manual, or collate please email me at connorr@bu.edu. You can also raise issues about bug fixes or additional desired functionality on GitHub (https://github.com/ danfeldman90/EDGE).

6. ACKNOWLEDGEMENTS

EDGE and collate were primarily written and developed by D. Feldman and C. Robinson, but have since grown from contributions from many people including Catherine Espaillat, Enrique Macias, Alice Pérez, and others. The authors are grateful for alerts to issues with the code from users like **you**.