# Manual for Utilizing Scripts Produced by Espaillat Disk Group Extraordinaires

October 19, 2017

**Abstract**

The Espaillat Disk Group Extraordinaires (EDGE) have put together Python based analysis tools in order to facilitate usage of the D'Alessio Iradiated Disk (`DIAD`) models. The main analysis tools are `EDGE` (EDGE.py) and `collate` (collate.py) and their objective is to standardize organization of the `DIAD` model input and output and help ease the fitting procedure.

`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, Amanda Reveles, and others. The authors are grateful for alerts to issues with the code from users like you.

For ease of reading, the manual is broken up into sections and a table of contents is located on the next page, so you can skip ahead to those relevant sections 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 the Python scripts – this can be found by examining the 'docstring' of the function in question. Note that `EDGE` uses Python 3.

# Contents

# 1 Getting started

## 1.1 First steps for new users

For first time users we recommend the following steps:
(1) send an email to cce@bu.edu to set up a Boston University (BU) Shared Computing Cluster (SCC) account
(2) follow email instructions to set up SCC account
(3) follow directions in Section 1.2 to create your working directory on the cluster
(3) follow directions in Section 1.3 to download the `EDGE` package to your local computer
(4) change your Python path as described in Section 1.4
(5) orient yourself to `EDGE` directory structure (Section 1.5)
(6) follow the demo in Section 4
(7) read the rest of the manual

## 1.2 Connecting to BU's Shared Computing Cluster

DIAD is housed on Boston University's (BU) Shared Computing Cluster (SCC). Throughout this manual it will be referred to as either "the SCC" or "the cluster."

You can log into the SCC via:

```
[UNIX] ssh -Y username@scc1.bu.edu
```

Note that in this manual commands entered into a terminal outside of python is preceded by '`[UNIX]`'.

When you log in you will be in your home directory. Here you have only 10 GB of space which will not be enough over time. Most of the your work should take place in the shared space called `/projectnb/bu-disks` which has significantly more space. Here you can create a directory where you can put your work using:

```
[UNIX] mkdir username
```

To access your backups (taken nightly up to 30 days) type

```
[UNIX] cd .snapshots
```

Note that you don't have to change permissions on files, etc. The default is that group members can see each other's files but not edit them. No one outside of the "bu-disks" group will have access.

Some useful commands:

```
[UNIX] qsub job001
```

will delete a submitted job (here called job001).

```
[UNIX] qstat -u username
```

will query the status of submitted jobs.

```
[UNIX] qdel jobid
```

will delete a submitted job (note that you have to do a qstat first to get the jobid).

```
[UNIX] scp file username@scc1.bu.edu:yourdirectory
```

will transfer files from your computer to the cluster space.

```
[UNIX] scp file username@yourcomputer.bu.edu:yourdirectory
```

will transfer files from the cluster space to your computer.

If you don't submit to the cluster via the qsub command, you can run the script locally by doing

```
[UNIX] csh job001
```

but you can only run for 15 min from the command line. This is useful for debugging and running subsets of the code. Anything longer than 15 min has to be submitted as a job to the cluster.
For more information check out:

```
http://www.bu.edu/tech/support/research/system-usage/running-jobs/
```

## 1.3 Using GitHub

First, you have to set up a GitHub account at

```
https://github.com/
```

To download `EDGE` to your local computer for the first time you have to create a local clone on your computer. Details are here:

```
https://help.github.com/articles/cloning-a-repository/
```

The repository can be found at

```
https://github.com/cespaillat/EDGE
```

When changes are announced to `EDGE` , you will want to sync with the main repository by typing the following at the command line:

```
[UNIX] git pull origin master
```

We recommend you do not place any of your own files in the `EDGE` directory so that you do not inadvertently overwrite your work when syncing with the repository.

*Note: Another useful reference for basic git commands: http://rogerdudler.github.io/git-guide/files/git_cheat_sheet.pdf*

## 1.4 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/MODULES/
```

Take care to change the directory to where ever you have placed EDGE so your python path can find the MODULES directory.

It is also recommended that you add the path to the shared EDGE directory on the SCC to your .bashrc file on the SCC. Your .bashrc file on the SCC can be found by using ssh to connect to the cluster (directions in Section 1.2). Next, type:

```
[UNIX] cd
```

which will move you to your home directory and then type:

```
[UNIX] ls -a
```

to reveal hidden files. You can then use one of the text editors available on the SCC (e.g., emacs, vi) to add the following line to the .bashrc file:

```
export PYTHONPATH="$PYTHONPATH:/project/bu-disks/shared/EDGE/MODULES"
```

You will not need to change the path here because this copy of EDGE is shared among all SCC users.

*Note: This version of EDGE should be kept up to date with the version on GitHub, but it is possible that the most recent changes may not have been pulled to the SCC.*

## 1.5 Description of EDGE package directories

Within the EDGE package, the first level contains a LICENSE file and the following five directories:

COMMON contains files that are frequently called by scripts.
DEMO has the practice demo we detail in Section 4.
MANUAL has a pdf file version of this manual. Within the latex sub-directory you will find the LaTeX file used to make this manual.
MODULES contains the main Python modules, functions, and classes (i.e., EDGE.py

and collate.py).
`SCRIPTS` has job files that call on the Python files in `MODULES`.

### 1.5.1 `EDGE` 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. All relevant functions also have a keyword that lets you supply the proper path. The paths hardcoded at the top of `EDGE` can be 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 do not need need to use them. **IMPORTANT:** If you obtain a new version of `EDGE` from GitHub, you **will** need to change the paths again.
*Note: If you are using scripts, 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.*

# 2 Intro to python

## 2.1 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. Here it is assumed that you are using ipython, which is the interactive version of python. This can be started by typing

```
[UNIX] ipython
```

which will start an `ipython` session. 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:

```
>>> edge.look(fancyStar)
```

where "`fancyStar`" is an edge observation object (which is created later in this manual). 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.

### 2.2 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)
```

### 2.3 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 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`.

## 3 Important Functions and Classes

This section will contain the important functions and classes contained within MODULES. 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.

**Important:** Older versions of the jobfile required that the jobs be collated by hand at the end of the run, but now the files are collated automatically. However, it may occasionally be necessary to re-collate jobs manually, which can be done by following the steps here.

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. *Note: This command can also be added to your `.bashrc` file on the SCC to avoid this step each time you use the SCC.*

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. If you are running `collate` on models that have already been collated, use the flag `''clob = True"` in your call to masscollate. This will cause `collate` to overwrite any fits files that have been created previously.

*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 Calvet & Gullbring (1998) 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)
```

`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 $\lambda F\lambda$ (in ergs $\mathrm{s}^{-1}\mathrm{cm}^{-2}$) 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.

### 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 $\chi^2$ 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 $\chi^2$ value as well.

### 3.12 starparam

The `starparam` module can be used to calculate various stellar parameters for a given spectral type, visual extinction, and J-band (or V-band for high mass

stars) magnitude. It calculates the stellar luminosity, mass, radius, and age of the star as well the accretion luminosity and mass accretion rate.

It also calculates Av(V-R), Av(V-I), Av(R-I), and Av(I-J), dereddened magnitudes for the input Av, and a template photosphere. This is useful if you are iteratively fitting for the extinction.

`starparam` can be run using `jobstarparam.py` which is in the `SCRIPTS` directory.

# 4 Example: 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 for 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_002.fits`').
The files within `DEMO` files are meant to be run as is with only the necessary changes to paths. As such, if you plan to modify them, we recommend that you create a new directory outside the `EDGE` directory to try this demo and not alter the files in `DEMO` so you can compare your outputs with those in `DEMO`. This can most easily be done using the `cp` command on the `DEMO` file with the `-r` flag which tells it to copy in the entire directory tree recursively.

## 4.1 Making the observation pickle

Change your paths (specpath, photpath, picklepath) in `DEMO_make_imlup.py` and do

```
[UNIX] python DEMO_make_imlup.py
```

There will be several error messages (this is normal) and a new file `imlup_obs.pkl` will be created at the location specified by picklepath.

`DEMO_make_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 de-reddened and saved to a `.pkl` file containing the de-reddened `TTS_Obs` object. This object can be looked at from 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.py`.

Change your paths (gridpath, clusterpath) in `DEMO_jobmaker.py` and do

```
[UNIX] python DEMO_jobmaker.py
```

This will create a small grid of three job files (`job001,job002,job003`) for models with different values of `alpha`, the viscosity in the disk. The job file parameters are in `DEMO_imlup_job_params.txt`.

## 4.3 Running and collating models

For the purposes of this tutorial, the jobs for this grid of models have already been uploaded and submitted to the cluster and collated for you, with the fits files you will need for analysis placed in the demo folder.

Below we outline the steps that were taken to create the fits files. You can follow these steps for practice or just read through these steps and use `.fits` files in `DEMO/model`.

In practice, the job files you just created would be moved to the `SCC` using the `UNIX` command `scp` as follows:

```
[UNIX] scp job* username@scc1.bu.edu:yourdirectory
```

For small grids, jobs can be submitted to the cluster individually with

```
[UNIX] qsub job001
[UNIX] qsub job002
[UNIX] qsub job003
```

For large grids, it is recommended that jobs are submitted using a `run_all.csh` script (also found in the `DEMO` directory). The `run_all.csh` file is created when

you run `DEMO_jobmaker.py`. You would upload `run_all.csh` to your directory and edit the path in `run_all.csh` to point to your run directory. Once the path is correct you would submit to the cluster with

```
[UNIX] qsub -t 1-3 runall.csh
```

where `1-3` in the example above should correspond to the range of job file numbers.

*Note: If you created the `run_all.csh` file with a `jobmaker` script, then it should already contain the correct path. Additionally, the `run_all.csh` script should include a commented out line containing the above command with the correct job numbers so all you need to do is copy and paste.*

You can check that your files are running by:

```
[UNIX] qstat -u username
```

It typically takes around two hours for jobs to run. Sometimes less, sometimes more. If you choose to, you can have an email sent when your files are done by changing the `runall.csh` file, I would do this in the early stages and then turn it off when you are running larger job arrays.
Once the jobs have all finished running, many files will be generated on the cluster and the results must be collated. This is now done automatically, but it may be necessary to re-collate files. This is most easily done using `masscollate`. If you have set up your `.bashrc` file correctly on the cluster (see section 1.2), your python should be able to find `collate.py`.
If collating by hand, make sure you have the latest version of collate.py and do the following:

```
[UNIX] module load anaconda3

[UNIX] python

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

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

```
>>> exit()

[UNIX] scp username@scc1.bu.edu:yourdirectory/*fits .
```

## 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 `DEMO_make_imlup.py` can be compared against the DIAD models. This will be done using the `DEMO_analysis_imlup.py` script.

Change the paths (picklepath, modelpath, figpath) in `DEMO_analysis_imlup.py` and do

```
[UNIX] ipython

[UNIX] run DEMO_analysis_imlup.py
```

This script does several things. The height of the inner wall (`altinh`) can be adjusted after models have finished running. This is useful because we can 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 $\chi^2$ value for this total emission is calculated and stored. Next the script searches for the lowest $\chi^2$ 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 $\chi^2$ is stored along with the job number and the wall height. This repeats for each job number. `.pdf` files of each job are created and the best-fit model appears in a the default python plotting GUI. Close the GUI. 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 $\chi^2$ values.

# 5 Example: using the code via the command line

## 5.1 Reading in data

The following is specifically for IRS spectra.

If you are working with a `.fits` file, do

```
>>> from astropy.io import fits
```

this imports a module that allows you to the open the fits file

```
>>> irs = fits.open('name_of_fits_file')

>>> print irs[0].header
```

this prints all of the information in the fits file
If you notice in the header there is a section that says "/DATA FLOW KEYWORDS" This tells you what columns the data you need are found in, you will typically only need the wavelength, `flux` and `flux_error`, and `nod_error` columns. You can collect the data into individual arrays like this:

`>>> irs_wl= irs[0].data[:,0] >>> irs_flux= irs[0].data[:,1] >>> irs_ferr= irs[0].data[:,3] >>> irs_noderr= irs[0].data[:,3]`

The nod and flux errors should be added in quadrature:

```
irs_err = np.sqrt(irs_ferr**2 + irs_noderr**2)
```

Next Problem: Keep in mind that the flux and flux_errors are in Jy which need to be converted into ergs/cm/s$^s$. There is a function in `util` that does that for you.

If you are working with a `.csv` file, do

```
>>> from astropy.io import ascii
```

This imports the module to read csv files. Alternatively, if the file is simple, `np.gentromtxt` will also do the job.

```
>>> data= ascii.read('filename', format='csv')
```

You can now save the data columns into an array

```
>>> wave=data['name of wavelength column in csv file']
```

Be sure to print the arrays to check that the data matches the column.

Be sure to check the units of the data. Sometimes the data will be in weird units like flux/ angstroms or some other weird thing. If your data is not in $\lambda F_\lambda$ (ergs/cm/s$^2$) then you need to convert it to get to those units.

## 5.2 Dereddening data

When de-reddening the IRS spectra, if you pay close attention to the doc strings in `EDGE` , you have to make sure that the units of your spectral data is in erg s$^-$1 cm$^-$2 cm$^-$1.

Once your data is converted you can proceed to creating the object file. The object file is where you store all of your data, spectra, photometry etc.

The process for creating one is similar in both the reddened and de-reddened cases except you need to do an extra step for the reddened case.

How to do this is outlined in Section 3 but here we outline a specific example, working with both observed and dereddened data.

For the observed data:

```
>>> PDS66obs = edge.TTS_Obs('PDS66')
```

```
>>> PDS66obs.add_photometry('Name of where photometry is from',
wavelength_array, flux_array, errors = errors_array)

>>> PDS66obs.add_spectra('Name of where spectra is from',
wavelength_array, flux_array, errors = errors_array)
```

For the dereddened data:

```
>>> PDS66red =edge.Red_Obs('PDS66')

>>> PDS66red.add_photometry('Name of where photometry is from',
wavelength_array, flux_array, errors = errors_array)

>>> PDS66red.add_spectra('Name of where spectra is from',
wavelength_array, flux_array, errors = errors_array)
```

Once you have your data stored into the object file you can save it as a pickle file! Define the path where you want the pickle file to be saved. I usually just go with the path

```
>>> picklepath = path

>>> PDS66obs.SPPickle(picklepath = path)
```

Now you can load this file back in and take a look at it with the Look function.

```
>>> PDS661 = edge.loadPickle('PDS66', path)
```

Extra steps for reddened files:

```
>>> PDS66red.SPPickle(picklepath=path)
```

Load pickle file back in to deredden

```
>>> PDS66r= edge.loadPickle('PDS66', path, red =1 )
```

Define the `edgepath` to where your `EDGE` -master folder is. For me it is like this:

```
>>> edgepath= '/Users/ugresearch/Desktop/mandar11/EDGE-master/'
```

Depending on the value of your extinction you need to choose the appropriate law. I believe there is a doc string in `EDGE` that tells you what laws correspond to what values. I know for Av $< 1$ use mathis90_rv3.1

```
>>> PDS66r.dered(Av, 'extinction_law', flux=1,picklepath=path,
lpath=edgepath)
```

Set your flux to 1 if you have it units of ergs/s/cm$^2$. If you converted it like you did the spectra leave it as 0.

This will automatically replace the red pickle file with a new object file. All you need to do now is load it back into python and view it in edge as with the

already de-reddened file.

### 5.3 Create job files and put on the SCC

Although it is strong reccomended that you use the `jobmaker.py` script for job file creation, it can be done at the command line using the `edge.job_file_create()` function in `EDGE` . Before you begin to spit out job files out after job files, make sure you are careful with the `job_sample file`. The `job_sample` file is the template from which `job_file_create` creates the job files.

For convenience you will want to change the stellar parameters in the job sample file to match that of your object so you don not have to repeat those parameters in `job_file_create`. Next, and this is the most Important, is that you must ensure that you change the labelend (around line 160 in job_sample) to that of the name of your object amd the corresponding job number you are about to make with job_file_create.

I have had many models go bad because the labelend did not match the job number DO NOT LET THIS BE YOU.

If you are using `jobmaker.py`, most of this is taken care of for you.

The commands for moving jobs to the cluster, submitting them and collating are the mostly the same as presented in section 4.3. One difference is that you will have to create your own runall.csh script.

Using the same methods of copying job files, copy the `runall.csh` script from your local computer (in `EDGE` directory) into your directory on the cluster. Check that the path in the `runall.csh` script matches the directory on the cluster where you are running the code. If it does not, you can make changes to text files on the cluster using one of the text editors there (e.g. emacs/vi). Here are the commands for vi:

```
[UNIX] vi runall.csh
```

Then make your changes.

```
[UNIX] ctrl + c

[UNIX] :wq
```

This saves the file, writes over it and exits.

After all your things are in order you can submit a job array! Retrieving files is identical to the instructions in section 4.3.

## 6 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 connorr@bu.edu. You can also raise issues about bug fixes or additional desired functionality on GitHub (https://github.com/ cespaillat/EDGE).