pyPlanet: The Manual

1 Introduction

pyPlanet is a modular open-source planetary atmospheric radiative transfer code that is currently written for the outer planets.

- ./ root directory
  - planet.py - executive module for the full pipeline
  - atmosphere.py - module to read/compute the atmospheric structure: can be run separately
  - alpha.py - module to compute the opacity at each layer: must have an atmosphere
  - brightness.py - module to compute the radiometric brightness temperature: must have an atmosphere and opacity
- constituents

2 Recipes

This section provides some recipes to generate and analysze data. Common to all of them are setting up and loading the class. It assumes a file named `config.par` in the appropriate planet directory (e.g. Neptune). It also assumes that the use.txt files under `constituents` are properly set. See the help listings for details on the other parameters that are available.

1. Make sure the config.par file is correct
2. Start `ipython --pylab`
3. `import planet`
4. `n = planet.planet('neptune')`

Then it forks based on whether you want to make images or spectra at a set of locations. The relevant function is n.run(...). Arguments are:

- `freqs`: list of frequencies to be used. For an image this should be just one.
  - scalar - it is converted into a list of length 1
  - list of length 3 - it interprets it as start, stop, step and generates that list (so beware)
  - list of any other length - it is just that list of frequencies
- `b`: "impact parameter" on the planet face. [0,0] is center of the projected face. It is scaled such that —b—=1 corresponds to the maximum radius of the projected face.
  - float - this will generate a grid with the given spacing as set by the block. It will extend ±1.5 to give some "empty space" around the planet.
  - list of length 2 - this will assume one given location on the face (e.g. `[-0.1,0.1]`)
  - list of length other than 2 - this will assume that you want a line of those values along the equator
  - matrix of length 8 - why did I do this?
  - string (word 'disc') - disc averaged value
  - otherwise it assumes a list of locations (e.g. `[[-0.1,-0.1],[0.0,0.0],[0.1,0.1]]`)

- `freqUnit`: string specifying the unit, default to 'GHz' (and hopefully if not, it correctly propagates through...)
- `orientation`:
- `block`: Unfortunately, if more than about 6000 pixels are needed the code (as run on my computer) crashes due to memory issues. So this allows the full grid to be done in blocks e.g. `b=0.01, block = [1,15]`. Default is [1,1], so that it runs the full square grid, which generally works for b¿0.05.

2.1 Images

For images, b is set to a float and then the blocks are done if needed. If b=0.01, then 15 blocks are generally needed. Blocks 1-4 and 12-15 can be run while in the same session of ipython, however one must exit the program after those and after every single block in the middle. It will write filenames specifying which block.

When all blocks are completed, you will need to put them together and also likely fix some bad points (near the edge where some rays blow up - which should be fixable!). For this restart `ipython --pylab` and `import img`.

Type `imgClass = img.Img()` and then give it the list of files needed to make the full image (e.g. `1 2 3 4 5 6 7 8 9 10 11 12 13 14 15`). Note that `imgClass` is your variable name to use - I generally use `nc` for Neptune at C-band, etc. It will put them together and plot the image.

2.1.1 Fix

As mentioned above, there are typically bad values and `imgClass.fix` will fix them by setting them to the average of all of the surrounding points. Use `plot(imgClass.data)` to have it plot all columns and see where the jagged points are. If they are all at large values (above any realistic value) it is simple to just type `imgClass.fix(v)`, where `v` is a threshhold value greater than any realistic value but less than any bad one. Replot imgClass.data and they should be gone. If there are bad points buried in there (i.e. with values less than the realistic max image value) then you need to track them down by pixel number and fix them. Once a list of pixel values are known you pass them to fix as a paired list: `imgClass.fix([[x1,x2,x3,...,xn],[y1,y2,y3,...,yn]])`, where n≥1.

fixDeriv as well...

You should then save the reconstituted image via `imgClass.saveImage(filename='desiredFilename')`. You can then delete all of the individual block files to reduce clutter.

2.1.2 Analysis

You will then want to compare these images with observations (hopefully for which you've correctly set up for in `config.par`). For this you will probably want another program called `jove`. Move the image file you created above to the directory containing the FITS files of the observations and `jove`.

After starting `ipython --pylab`, you can `import jove`, then `import img`. Note that it must be done in this order since `jove` sets the path to find `img`. You can read in the appropriate FITS by `joveClass=jove.Img(jove.ls[num])`, where again joveClass is your name (I typically use `n` for Neptune etc) and `num` is the number in the file listing that was shown when you created `joveClass`.

You can then read in your image as before by `imgClass=img.Img()`. If you know the number of the file, you can include it in the argument, as opposed to letting it query you. Here I usually `imgClass` to be e.g. `c` for C-band etc.

You then need to convolve it to the appropriate beam. `imgClass.generateKernel(joveClass.bw[0])` `imgClass.convolve()`

2.2 Spectra