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

Specifics of running the code are discussed in Section 3. Other than the generated plots (and the data contained within the python class during runtime), the primary output is a file saved in the Output subdirectory named for the planet, type and date/time as `Planet_Type_YYYYMMDD_HHMM.dat`.

There are three basic output types:

1. spec - standard spectrum at one or more pointings
2. profile - a radial profile at one or more frequencies
3. image - an image at one frequency

2 Configuration File

The program reads in a configuration file from the appropriate planet directory (Jupiter or Neptune for now) called 'config.par'. It consists of keywords (tokens) followed by value(s). Characters from a # to the end of the line are ignored. Below is an example, followed by a discussion of each keyword.

```
# Format:  token  value [value...]  #comment
# Currently tokens have to match exactly (but can be upper or lower)
# Order doesn't matter but the last one read will be in effect.
# Parameters not included take the hard-coded default value
### gas data
gasfile jupiter.paulSolar 1
constituents Z T P H2 HE CH4 NH3 H2O H2S SOLN OTHER PH3 CO CO13 HCN DZ
### cloud data
cloudfile jupiter.paulclSolar 7
clouds Z T P SOLN H2O NH4SH NH3 H2S CH4 AR PH3 DZ
### other data, tweak and regrid
tweakFile JupiterTweak
regridtype 2000
pmin 0.001
pmax 50000.0
#
### gravity values
```

```
p_ref    1.0      bars
Req      71492.0  km
Rpol     66854.0  km
RJ       71492.0  km
GM       12.6686538e7
Jn       0.0 0.0 1.4697e-2 0.0 -5.84e-4 0.0 0.31e-4
omega    1.7585e-4
zonal    zonalJupiter.dat
gtype    ellipse
#
### observations
#--- vanilla
distance 5.2 AU
orientation 0.0 0.0 deg
#
### alpha
doppler 0
h2state e
water 1.0E-4
ice 1.0E-4
nh4sh 1.0E-4
nh3ice 1.0E-4
h2sice 1.0E-4
ch4  1.0E-4
```

2.1 Constituents

2.1.1 gasfile

jupiter.paulSolar 1

2.1.2 Constituents

Z T P H2 HE CH4 NH3 H2O H2S SOLN OTHER PH3 CO CO13 HCN DZ

2.2 Clouds

2.2.1 cloudfile

jupiter.paulclSolar 7

2.2.2 clouds

Z T P SOLN H2O NH4SH NH3 H2S CH4 AR PH3 DZ

2.3 Tweaking/Gridding

2.3.1 tweakFile

JupiterTweak

2.3.2 regridtype

2000

2.3.3 pmin

0.001

2.3.4 pmax

50000.0

2.4 Gravity/Shape

2.4.1 p_ref

1.0 bars

2.4.2 Req

71492.0 km

2.4.3 Rpol

66854.0 km

2.4.4 RJ

71492.0 km

2.4.5 GM

12.6686538e7

2.4.6 Jn

0.0 0.0 1.4697e-2 0.0 -5.84e-4 0.0 0.31e-4

2.4.7 omega

1.7585e-4

2.4.8 gtype

This is the type of shape to be used. Options are:

- gravity Calculates gravity using many of the terms above
- reference Not quite sure
- ellipse Uses an ellipse with Req and Rpol
- circle Assumes Rpol = Req

2.5 Observations

2.5.1 distance

5.2 AU

2.5.2 orientation

0.0 0.0 deg

2.6 Opacity Formalisms

Whether to use the opacity and which formalism to use is set by a file called `use.txt` in the appropriate sub-directory. The name of the desired Python module (without the .py) is the first line of that file. Currently this is just hand edited. To not use it at all, currently you just rename that file nouse.txt.

A script called `use.py` will show if and which formalism and will toggle a given constituent off or on. Typing `use.py` will show a list and typing `use.py --`*constituent* (where *constituent* is *e.g.* nh3, h2s, ...) will toggle it off or on.

2.6.1 doppler

0

2.6.2 zonal

zonalJupiter.dat

2.6.3 h2state

e

2.6.4 water

1.0E-4

2.6.5 ice

1.0E-4

2.6.6 nh4sh

1.0E-4

2.6.7 nh3ice

1.0E-4

2.6.8 h2sice

1.0E-4

2.6.9 ch4

1.0E-4

3 Recipes

This section provides some recipes to generate and analyze 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 (must be this for an image, see below).
- ⋈ 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
- ⋈ string - it reads in a file of frequency values (one per line)

→ `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 equatorial radius of the projected face.

- ⋈ float - this will generate a grid with the given spacing as set by the block. It sets `isImg = True` if there is only one frequency. Otherwise it is a lot of spectra and no image. It will extend ±1.5 to give some "empty space" around the planet, but this is quick, since they'll miss.
- ⋈ list of length 2 (doublet list) - this will assume one given location on the face (e.g. `[-0.1,0.1]`)
- ⋈ list of length ¿ 2 this will assume that you want a line of those values at an angle given by the first term (in degrees - 0 is equator)
- ⋈ 'disc' - disc averaged value
- ⋈ 'stamp' - postage stamp, query for values
- ⋈ otherwise it assumes a list of locations (list of doublet lists, 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`: doublet list that sets the orientation of the planet (typically to match an observation).

→ `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. (If the second number of block is < 0, then it will print out the value of block after `bRequest`. This was put in for debugging. It stays negative, so beware.)

3.1 Images

For images, b is set to a float (or the list of N^2 doublets) 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.

3.1.1 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 paired x-y lists: `imgClass.fix([[x1,x2,x3,...,xn],[y1,y2,y3,...,yn]])`. (`fixDeriv` hasn't been implemented.)

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.

3.1.2 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()`

3.2 Spectra