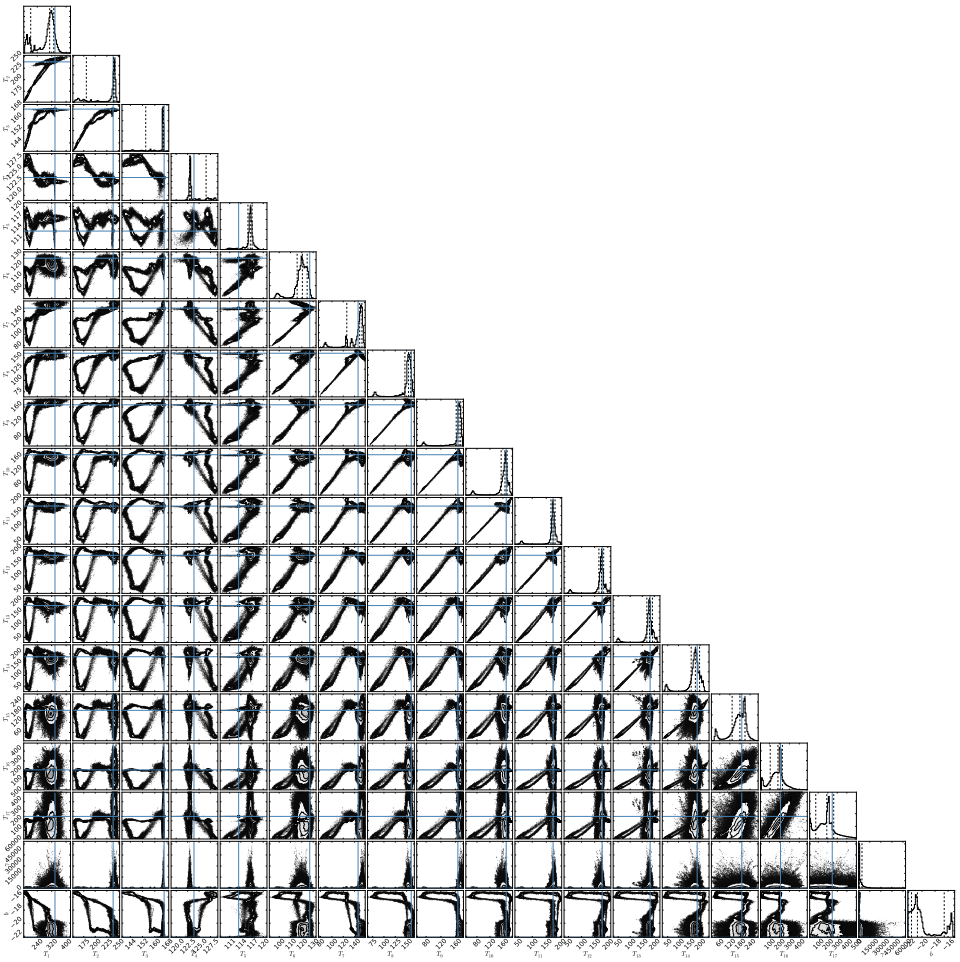
NEMCEE
The NEMESIS-EMCEE Interface

A no-nonsense guide and idiot-proof manual



Ryan Garland
ryan.garland@physics.ox.ac.uk

Contents

Chapter 1

An Intro to EMCEE and MCMC

This is taken from my first-year report and is only meant as a guide. If you truly want to get to grips with MCMC I'd recommend extensive googling and the Bayesian Data Analysis book by Gelman. For EMCEE, try Dan's website which contains a lot of info and links to the appropriate papers: http://dan.iel.fm/emcee/current/

1.1 EMCEE

1.1.1 Markov Chain Monte Carlo

Bayesian inference is about the quantification and propagation of uncertainty, defined via a probability, in light of observations of the system. They arrive at a posterior probability distribution from an *a priori* state vector. Markov Chain Monte Carlo (MCMC) techniques are methods for sampling from posterior probability distributions using Markov chains.

The main reason why we are considering MCMC over optimal estimation is because of its ability to quantify all aspects of uncertainty via probability and propagate these errors robustly in the posterior probability distribution function. This is especially desirable for low resolution or wavelength-limited spectra, which is usually the case for Y dwarfs.

Consider the sequence of random $\{x_0, x_1, x_2, ...\}$ variables, sampled from the probability distribution $p(x_{t+1}|x_t)$, then each next sample x_{t+1} depends only on the current state x_t and does not depend on the previous history $\{x_0, x_1, ..., x_{t-1}\}$. Such a sequence is called a Markov chain. MCMC techniques aim to construct cleverly sampled chains which (after a burn-in period) draw samples which are progressively more likely realisations of the distribution of interest - the posterior probability distribution function (PDF).

MCMC is a perfect match for planetary science, where the models are often expensive to calculate (because efficient sampling methods mean a lower amount of runs to realise the PDF), contain many free parameters, and have low signal-to-noise observations. MCMC has already been used to great effect by Line et al. (2015) to calculate the probability distribution functions of many relevant parameters to brown dwarf atmospheres, concluding that only low-resolution spectra are required to unambigously detect ammonia in T dwarfs. We note here that, in comparison to optimal estimation, MCMC can be orders of magnitude more computationally expensive. We are likely to therefore use both of these techniques alongside one another to determine atmospheric parameters.

One of the key advantages of Bayesian data analysis is that it is possible to marginalise over nuisance parameters (e.g., spectral noise, instrumental errors). A nuisance parameter is one that is required in order to model the process that generates the data, but is otherwise of little interest. Marginalisation is the process of integrating over all possible values of the parameter and hence propagating the effects of uncertainty about its value into the final result.

Marginalisation is represented by the marginalised probability function $p(\Theta|D)$ of the set of model parameters Θ given the set of observations D:

$$p(\Theta|D) = \int p(\Theta, \alpha|D) \mathrm{d}\alpha \tag{1.1}$$

where α is the set of nuisance parameters. Usually, the set α can be extremely large and therefore expensive to integrate. However an MCMC-generated sampling of values (Θ_t, α_t) (where t is the timestep/iteration of the chain) of the model and nuisance parameters from the joint distribution $p(\Theta, \alpha|D)$ automatically provides a sampling of values Θ_t from the marginalised PDF $p(\Theta|D)$.

Another advantage of MCMC comes from the interest in the likelihood (or prior) function of the parameters, which may come from expensive calculations. Here, efficient MCMC sampling, where few function evaluations are required to create a statistically independent sample from the posterior PDF, is very valuable. The methods presented here (Section 1.1.2) are designed for efficiency.

1.1.2 EMCEE

We use the Markov Chain Monte Carlo (MCMC) approach implemented with affine-invariant ensemble sampler EMCEE (Foreman-Mackey et al., 2013), following Line et al. (2015). Affine-invariant transformations preserve collinearity, so they transform parallel lines into parallel lines and preserve ratios of distances along parallel lines. Affine transformations allow the conversion of highly anisotropic probability distribution functions into isotropic distribution functions, which greatly reduces the difficulty of sampling the PDF. An algorithm that is affine invariant performs equally well under all linear transformations; it will therefore be insensitive to covariances among parameters. This is a significant advancement over the optimal estimation used previously in NEMESIS, as we are now able to make fewer *a priori* assumptions, and the shape of the parameter uncertainties no longer need to be Gaussian.

Algorithm

The EMCEE algorithm involves simultaneously evolving an ensemble of K *walkers* $S = \{X_k\}$ with positions X where the proposal distribution for one walker k is based on the current positions of the $K-1$ walkers in the *complementary ensemble* $S_{[k]} = \{X_j, \forall j \neq k\}$. Here, position refers to a vector in the N-dimensional, real-valued parameter space.

To update the position of a walker at position X_k, a walker X_j is drawn randomly from the remaining walkers $S_{[k]}$ and a new position Y is proposed:

$$X_{k(t)} \rightarrow Y = X_j + Z[X_{k(t)} - X_j] \tag{1.2}$$

where Z is a random variable drawn from a distribution $g(Z = z)$:

$$g(z^{-1}) \propto \begin{cases} \frac{1}{\sqrt{z}} & \text{if} z \in \left[\frac{1}{a}, a\right] \\ 0 & \text{otherwise} \end{cases}$$

and a is an adjustable scale parameter, normally set to 2.
If g satisfies

$$g(z^{-1}) = zg(z) \tag{1.3}$$

the proposal of Equation 1.2 is symmetric in the sense that

$$p(X_k(t) \to Y) = p(Y \to X_k(t)). \tag{1.4}$$

In this case, the chain will satisfy detailed balance (around any closed cycle of states, there is no net flow of probability, i.e. Equation 1.4) if the proposal is accepted with probability

$$q = \min\left(1, Z^{N-1}\frac{p(Y)}{p(X_k(t))}\right) \tag{1.5}$$

where $p(Y)$ is the probability of the proposed position Y, calculated from the likelihood function and the prior information described in subsubsection 1.1.2, and N is the dimension of the parameter space. This procedure is then repeated in series for each walker.

A huge benefit of using EMCEE, is that it may perform these iterations (with some modification to the algorithm) in parallel, and therefore the sampling may occur on multiple cores simultaneously, drastically reducing the amount of (real, not computational) time taken for each analysis. It is also configured for multicore-multinodal calculations using an MPI (message passing interface) setup, so that the calculations can be further spread between many computers and many cores at once.

NEMESIS Integration

EMCEE requires a functional form for the log of the posterior probability to perform the sampling. The posterior probability is a combination of the likelihood and the prior described as follows. Starting from Bayes' theorem,

$$p(\boldsymbol{x}|\boldsymbol{y}) = \frac{\mathcal{L}(\boldsymbol{y}|\boldsymbol{x})p(\boldsymbol{x})}{E} \tag{1.6}$$

where $\boldsymbol{x}$ is the parameter vector, $\boldsymbol{y}$ is the data vector (i.e. the spectrum), $p(\boldsymbol{x}|\boldsymbol{y})$ is the posterior probability distribution, $\mathcal{L}(\boldsymbol{y}|\boldsymbol{x})$ is the likelihood distribution which penalises poor/improbable fits to the data, $p(\boldsymbol{x})$ is the prior information which can restrict the parameter space, and E is the evidence, or marginal likelihood, which is a normalisation factor required for Bayesian model comparison, but not for parameter estimation, and is a complicated factor which requires sophisticated Monte Carlo methods to solve. We follow a similar method to Line et al. (2015), describing the log-likelihood distribution with the function:

Table 1.1.1: Summary of Priors for Each of the Parameters

Parameter	Prior
log VMR	Uniform, log VMR $\geq$ -12, $\sum$ VMR ≤ 1
Mass, M	Uniform, $1M_J \leq M \leq 80M_J$
Radius, R	Uniform, $0.7R_J \leq R \leq 1.5R_J$
γ	Inverse Gamma($\tilde{\Gamma}(\gamma;\alpha,\beta)), \alpha = 1, \beta = 5 \times 10^{-5}$)
b	$0.01 \min(\sigma_i^2) \leq 10^b \leq 100 \max(\sigma_i^2)$
T_i	See Equation (1.10)

$$\ln\mathcal{L}(\boldsymbol{y}|\boldsymbol{x}) = -\frac{1}{2}\sum_{i=1}^{n}\frac{(y_i - F_i(\boldsymbol{x}))^2}{{s_i}^2} - \frac{1}{2}\ln(2\pi {s_i}^2) \tag{1.7}$$

where the index i denotes the ith data point, y is the measured flux, $F(x)$ is the modeled flux that comes from NEMESIS's forward model. s is the data error given by:

$$s_i^2 = \sigma_i^2 + 10^b \tag{1.8}$$

where σ is the measurement error, and b is a free parameter which accounts for missing forward model physics and underestimated uncertainties.

Equation 1.7 is formed with two main terms. The first term inside the summation penalises large residuals between the model and data (i.e. chi-square). The second term is the Gaussian normalization factor, which is normally excluded as the data errors are unchanging, but because of the free parameter b the normalisation can vary and therefore must be taken into account. This normalisation ensures that the error bar inflation doesn't approach infinity (which would of course fit *any* spectra) or its maximum value set by the prior.

The prior, $p(\boldsymbol{x})$, can be broken up into several pieces as

$$p(\boldsymbol{x}) = p(\boldsymbol{T})p(\boldsymbol{x'})p(\gamma) \tag{1.9}$$

where $p(\boldsymbol{x'})$ is the prior on the log of the volume mixing ratios, mass, radius and error inflation (b), while $p(\boldsymbol{T})$ is the prior on the temperature profile and $p(\gamma)$ is the prior on the temperature profile smoothing parameter. The smoothing parameter γ (technically a hyperparameter - a parameter of a prior distribution; the term is used to distinguish them from the physical parameters of the model for the underlying system under analysis such as temperature) acts to penalise unphysical fluctuations in the temperature profile by examining the second derivative of the temperature. This smoothing is implemented as:

$$\ln p(\boldsymbol{T}) = -\frac{1}{2\gamma}\sum_{i=1}^{n}(T_{i+1} - 2T_i + T_{i-1})^2 - \frac{1}{2}\ln(2\pi\gamma). \tag{1.10}$$

The hyperprior (a probability distribution on the hyperparameter itself - or a hyperhyperparameter) on γ takes the form of an inverse gamma distribution with the properties shown in Table 1.1.1, based on experimentation (Lang & Brezger, 2004; Jullion & Lambert, 2007).

Chapter 2

NEMCEE

2.1 Installation guide

2.1.1 The Easy Way

The Virtual Environment

In order to run all of the appropriate python libraries one must use a virtual environment because our central (default) python distribution is linked to a bunch of other nodes on the network, and obviously changing those files would affect everyone (you couldn't change them if you tried anyway). So, to enter the virtual environment enter this in your terminal:

```
cd /network/aopp/oxpln97/plan/garland/codes/py27ve
source bin/activate.csh
```

The terminal should now be labelled as [py27ve] node(user), indicating that you are in the virtual environment py27ve. If you're using bash, remove the .csh from the source call.

The Shared Object NEMESIS Files

In the '/network/aopp/oxpln97/plan/garland/codes/' directory, there lie three *.so (shared object) files - one for each type of NEMESIS configured for NEMCEE. We have nemesisEMCEE.so, nemesisdiscEMCEE.so, and nemesisPTEMCEE.so, all corresponding to the (hopefully self-explanatory) different versions of NEMESIS. Essentially these 'shared object' files are dynamic libraries which contain native machine code (i.e. FORTRAN) but whose functions have the interfaces of a Python module (which lets us call it from Python). They're a black box of NEMESIS.

Also in this folder are a set of base/example codes from which to run NEMCEE. I'll pick these apart in section 2.3

2.1.2 The Hard Way

The Virtual Environment

Personally I would not recommend installing your own virtual environment, as there should be no reason you would need to alter the python library source codes. But if you really must, here's how you do it:

```
cd <desired directory, probably $HOME>
virtualenv --system-site-packages <ENVNAME>
source <ENVNAME>/bin/activate.csh
pip install --upgrade pip setuptools
pip install --upgrade emcee
pip install triangle
```

Occasionally you might find that a version of one of your libraries is not up-to-date. To update type 'pip install –upgrade LIBRARY'.

These things can be a bit fiddly - a common error is that a link goes awry and python looks in /usr/local for the appropriate files. This will cause it to fail as emcee and triangle do not exist there. The reason for this will be quite particular, so I can't guide you to fix this without seeing the error. What you can do is create a new environment and reinstall things, or go back to the 'easy way' option. Or pull your hair out for a week looking at related articles on stackoverflow.

The Shared Object NEMESIS Files

Now included in the SVN are *.comp files. These are essentially make files that compile NEMESIS using F2Py, created the dynamic libraries which are then imported to the NEMCEE codes. To compile, type, e.g.:

```
source nemesisdiscEMCEE.comp
```

This will then create the appropriate .so file, which you must then copy into the working directory of your NEMCEE run. I'll talk about this more in section '2.2.

Inside the .comp files we have the command:

```
f2py -c –m nemesisEMCEE nemesisEMCEE.f ... files called in nemesisEMCEE and subroutines.f
```

f2py is the Fortran-to-python wrapper program. The flag -c is for building an extension module, and -m is for the module name. We call only the necessary files found in nemesisEMCEE.f and its subroutines.

2.2 How to setup NEMESIS Flags and Environment before Calculation

First and foremost, setup NEMESIS like you would for any other run, and set the number of iterations to -1. Now copy from 'codes' multdirec.csh. This is a script file that will copy all of the files in the current directory into new subdirectories called '0,1,2....N'. You must edit the script to change the '0..1' to '0..N' where N must be larger than your number of parameters * nwalkers. Typically we use 10 walkers per parameter, and the number of parameters is usually ¡30 or so you should normally create 300 subdirectories. To run the script, type:

```
./multidirec.csh
```

Every one of these subdirectories MUST be identical, otherwise the EMCEE will produce some funny results. Therefore to change one file, you must change it in each subdirectory.

Now that everything is setup, copy the relevant base code (*.py) and dynamic library (*.so) files from 'codes' to the top-level (level above the 300 subdirectories). Now we have to edit the python code to make it do what we want! In section 2.3, we'll describe how the code works and how to edit it accordingly.

2.3 A Step-by-Step of the Code

2.3.1 Imports

```
import numpy as np
import emcee
import matplotlib
matplotlib.use('PDF')
import matplotlib.pyplot as plt
import pickle
import triangle
import math
import time
import scipy as sp
import nemesisdiscEMCEE
from scipy.stats import invgamma
from scipy.interpolate import interp1d
from math import log,log10
from scipy.stats.kde import gaussian_kde
#                           END OF IMPORTS
################################################################################
```

Here are the various libraries to be imported for use in the code. Important things to note here are 'matplotlib.use('PDF')' which must be imported to produce plots on nodes that we don't have access to the display, and the packaged nemesis file we are choosing to use, e.g. 'import nemesisdiscEMCEE'.

2.3.2 Definitions

```
#                       BEGINNING OF DEFINITIONS
################################################################################

font = {'family' : 'normal',
        'weight' : 'normal',
        'size'   : 20}

matplotlib.rc('font', **font)
```

Here you can alter font settings for the plots globally, although I would recommend setting them individually on each plot in the definitions to come.

```
# Define the probability function as likelihood * prior
twopi = 2*math.pi
```

```
def lnpriormass(x):

    mass = x[ntemp+nvmr+ngamma]

    if 1 < mass/mjup < 80:
        return 0    #log(1) probabilities are equal
    else:
        return -np.inf  # log(0)

def lnpriorrad(x):

    rad = x[ntemp+nvmr+ngamma+nmass]

    if 0.6 < rad/rjup < 1.4:
        return 0
    else:
        return -np.inf  # log(0)
```

Here we define the upper and lower limits of the mass and radius priors. We keep them as flat probabilities, i.e. no mass or radius is more likely than another (unless it's beyond the physical limits). As these are defined as log probabilities, if the probability is 1, then a 0 (i.e. $\ln 1$) is returned. Likewise, a probability of 0 returns $-\inf$ (i.e. $\ln 0$).

```
def lnpriorvmr(x):
# vmr = log(vmr)
    vmr = x[xrange(ntemp, ntemp+nvmr)]
    vmrsum = np.sum(10**vmr)
    h2 = h2p*(1-vmrsum)
    he = hep*(1-vmrsum)
    if all(-12<=i<=-1 for i in vmr) and vmrsum <= 0.1:
        return 0    #log(1) probabilities of vmrs are equal (flat?)
    else:
        return -np.inf  # log(0)
```

The VMR prior is also flat, and is constructed in a way such that we never need to calculate the H_2 and He VMRs directly, saving on calculation time by reducing the parameter-space. This is done by assigning standard values for H_2 and He via 'h2p' and 'hep' from which they will not stray wildly. The standard is 0.85 and 0.15 respectively.

By ensuring that the non-H2 and -He VMRs sum to 0.1, these standard values can only be altered by $\pm$ 10% (H_2 ranges from 0.765 to 0.935, He ranges from 0.135 to 0.165). Of course all of these values can be altered

by the user, and for non-Jovian-like atmospheres one can alter the code to remove this feature in this prior and some of the plotting definitions discussed later.

```
def lnpriorgamma(x):
    gamma=x[ntemp+nvmr]
    invg=invgamma.pdf(gamma,1,scale=5e-5)
    invgprob=invg
    if invgprob > 0:
        return log(invgprob)
    else:
        return -np.inf
```

This 'weakly informative' prior is used to determine the 'kinkiness' of the temperature profile (i.e. the second derivative of the temperature w.r.t. pressure). It weakly influences the prior to prefer smoother profiles via an inverse-gamma PDF. One can change the shape and size parameters of the distribution, but ultimately it shouldn't make a huge difference to the retrieval.

```
def lnpriorb(x):

    b = x[ntemp+nvmr+ngamma+nmass+nrad]
    if 0.01*(min(yerr))**2 <= 10**(b) <= 100*(max(yerr))**2:
        return 0
    else:
        return -np.inf
```

This is the flat error inflation prior. Initially the run will almost always try to maximise this value as it is the quickest route to fitting a crappy fit. Over time it will settle down to lower values as the fit becomes better.

```
def lnpriortemp(x):
    temp=x[xrange(ntemp)]
    gamma=x[ntemp+nvmr]
    dt = 0
    for i in range(len(temp)-1):
        if i > 0:
         dt += (temp[i+1]-2*temp[i] + temp[i-1])**2


    lnprobt= -0.5*( dt/gamma + np.log(twopi*gamma) )

    tcubic = cubicinterp(pres0,temp,nempres)
```

```
    if np.isfinite(lnprobt) and all(10 <= i <= 7000 for i in tcubic):
        return lnprobt
    else:
        return -np.inf
```

Here we calculate the likelihood of the temperature prior using the equation:

$$\ln p(\boldsymbol{T}) = -\frac{1}{2\gamma}\sum_{i=1}^{n}(T_{i+1} - 2T_i + T_{i-1})^2 - \frac{1}{2}\ln(2\pi\gamma). \tag{2.1}$$

We also interpolate all of the temperature points at this stage to ensure that the temperatures are physical (10 - 7000K).

```
def lnprior(x):

    lp=lnpriorvmr(x)+lnpriorgamma(x)+lnpriortemp(x)+lnpriorb(x)+lnpriorra
    if not np.isfinite(lp):
        return -np.inf
    else:
        return lp
```

Here we define all of the relevant priors in the calculation. For example, if we only wanted VMRs, mass and radius, we'd have:

```
lp = lnpriorvmr(x)+lnpriorrad(x)+lnpriormass(x)
```

If any of these parameters are defined as having zero probability, then EMCEE will automatically not calculate the spectra for that sample.

```
def cubicinterp(pres,temp,presnew):

# need to have a monotonically increasing x (pres) for function
# so we reverse order of arrays
    pres = pres[::-1]
    temp = temp[::-1]
    f = interp1d(pres, temp, kind='cubic')
    tcubic = f(presnew)
# reverse again to have initial order
    return tcubic
```

This is the cubic interpolation routine for converting the retrieved temperatures points and mapping it to the TP-profile in the NEMESIS files.

```
def lnlike(x):

    b = x[ntemp+nvmr+ngamma+nmass+nrad]
    p=np.loadtxt('position.txt')
```

```
    ith = -1
    for j in range(len(p)):
     if all(p[j] == x):
      ith = j
      break
    sigma2 = yerr**2 + 10**b
# note we return a single zero
# array when flags prevent VMR etc
# because Fortran will not accept
# a 0th-size array

    if(vflag==0):
     vmr=np.zeros(1)
    else:
     vmr = x[xrange(ntemp, ntemp+nvmr)]
     vmrsum = np.sum(10**vmr)
     h2 = h2p*(1-vmrsum)
     he = hep*(1-vmrsum)
     vmr = np.append(vmr,[np.log10(h2),np.log10(he)])
    if(gflag==0):
     mass=np.zeros(1)
     rad=np.zeros(1)
    else:
     rad = x[ntemp+nvmr+ngamma+nmass]
     mass = x[ntemp+nvmr+ngamma]
    if(tflag==0):
     tcubic=nemtemp
    else:
     temp=x[xrange(ntemp)]
     tcubic = cubicinterp(pres0,temp,nempres)

    model = nemesisdiscEMCEE.nemesisdiscemcee(runname, len(y), len(tcubic
#    model = 1
    return -0.5*( np.sum( (y-model)**2/sigma2 + np.log(twopi*sigma2) ) )
```

In order to find the directory in which to calculate the spectrum (NEMESIS can't operate multiple calculations in one directory as it needs to update the same files), the first part of the likelihood function needs to find the number of the supplied walker in the chain. It uses this number to decide which directory to calculate in when it is passed to NEMESIS.

The next section is decidated to ensuring the calculation will still operate even if not all the parameters are to be retrieved via flags.

Then all of the parameters are find into the calculation, and the likelihood is returned.

Notice also there is a commented out 'model = 1' line. A good way to debug the code is to set model = 1 and run normally. This means that, while the retrieved values are meaningless, if it completes one iteration fully and prints out all of the required information correctly, you can be sure you're not going to get an error four hours later.

```
def lnprob(x):

    lp = lnprior(x)
    if not np.isfinite(lp):
        return -np.inf
    return lp + lnlike(x)
```

The final log-probability used in the statistics, the combination of the priors and the likelihood function.

```
def save_object(file, object):
    with open(file, 'wb') as f:
        pickle.dump(object, f)

def load_object(file):
    return pickle.load(open(file, "rb"))
```

Routines to later save and load the relevant statistical information.

```
def plot_pdf_kde(ndim, names, samples):
 fig, ax = plt.subplots(figsize=(50, 10))
 rows = 2
 for j in range(ndim):
    plt.subplot(rows,int(math.ceil(float(ndim)/rows)),j)
    pmed = np.percentile(samples[:,j],50)
    pp = np.percentile(samples[:,j],84.14)
    pm = np.percentile(samples[:,j],15.87)
#     plt.hist(sampler.flatchain[:,j], 1000, color="b", histtype="step")
    dist_space = np.linspace( min(samples[:,j]), max(samples[:,j]), 10000
    kde = gaussian_kde( samples[:,j] )
    plt.plot(dist_space, kde(dist_space)/max(kde(dist_space)))
    print(pmed, pp, pm)
#     plt.fill_betweenx(kde(dist_space), pm, pp)
#     plt.annotate('%d' % pm,   xy=(pm,0), xycoords='data', fontsize='xx-la
    plt.axvline(x=pm, linestyle='dashed', color='r', label=r'$%2.2f^{+%2.
    plt.axvline(x=pp, linestyle='dashed', color='r')
    plt.axvline(x=pmed, linestyle='dashed', color='r')
    plt.title( '%s' % names[j])
```

```
#     plt.axis('off')
      plt.legend(loc='best',frameon=False, handlelength=0,numpoints=1, font
      plt.set_xlabel=('Normalised PDF')
      plt.set_ylabel=(r'$\log(VMR)$')

  plt.tight_layout()
  plt.savefig('new.pdf',transparent=True)
  plt.close()
```

This is a plotting routine to plot only the PDFs of each parameter. It uses a kernal density estimator to fit a histogram of results. One can alter the figure size and number of rows at the top of the function.

```
def plot_walkers_iter(ndim, names, chain):

  fig, axis = plt.subplots(figsize=(50, 10))
  rows = 2
  for j in range(ndim):
      plt.subplot(rows,int(math.ceil(float(ndim)/rows)),j)
      plt.plot(chain[:,:,j].T, color="k",alpha=0.8)
      plt.set_xlabel=('n')
      plt.set_ylabel=(names[j])
  plt.tight_layout()
  plt.savefig('lasttime.pdf',transparent=True)
  plt.close()
```

Another plotting routine, for plotting the position of the walkers at any given iteration. Can also adjust plots at the top.

```
def plot_spectra(percent, samples, runname, y, yerr, vflag, gflag, ntemp,

  x = map(lambda v: [v[1], v[2]-v[1], v[1]-v[0]],
                        zip(*np.percentile(samples, [15.87, 50., 84.14], axi
  pmparams = np.asarray(x).flatten()

  # median, +1 sigma (high), -1 sigma (low)
  med = pmparams[0::3]
  high = pmparams[1::3]
  low =  pmparams[2::3]

  if (tflag==1):
   tempmed = cubicinterp(pres0,med[xrange(ntemp)],nempres)
   temphigh = tempmed + cubicinterp(pres0,high[xrange(ntemp)],nempres)
   templow =  tempmed - cubicinterp(pres0,low[xrange(ntemp)],nempres)
   aptemp = cubicinterp(pres0,tempp0,nempres)
# maximum likelihood params
```

```
 mltemp = cubicinterp(pres0,par_ML[xrange(ntemp)],nempres)
 fig, ax = plt.subplots(figsize=(20, 20))
 plt.semilogy(aptemp, 10**(nempres), label='A Priori', linewidth=4, colo
 plt.semilogy(tempmed, 10**(nempres), label='Median', linewidth=4, color
 plt.semilogy(nemtemp, 10**(nempres), label='OE', linewidth=4, color='r'
 plt.semilogy(mltemp, 10**(nempres), label='Max Like', linewidth=4, colo
 plt.fill_betweenx(10**(nempres), temphigh, templow, facecolor='b', alph
 plt.gca().invert_yaxis()
 plt.legend()
 plt.savefig(str(percent)+'%'+"tempprof.pdf")
 plt.close()

else:
 tempmed, temphigh, templow = nemtemp, nemtemp, nemtemp

if (vflag==1):
 vmrmed = med[xrange(ntemp, ntemp+nvmr)]
 vmrhigh = vmrmed + high[xrange(ntemp, ntemp+nvmr)]
 vmrlow = vmrmed - low[xrange(ntemp, ntemp+nvmr)]
 vmrsummed, vmrsumhigh, vmrsumlow = np.sum(10**vmrmed), np.sum(10**vmrhi
 medh2, highh2, lowh2 = h2p*(1-vmrsummed), h2p*(1-vmrsumhigh), h2p*(1-vm
 medhe, highhe, lowhe = hep*(1-vmrsummed), hep*(1-vmrsumhigh), hep*(1-vm

 vmrmed = np.append(vmrmed,[np.log10(medh2),np.log10(medhe)])
 vmrhigh = np.append(vmrhigh,[np.log10(highh2),np.log10(highhe)])
 vmrlow = np.append(vmrlow,[np.log10(lowh2),np.log10(lowhe)])

else:
 vmrmed, vmrhigh, vmrlow = np.zeros(1), np.zeros(1), np.zeros(1)

if (gflag==1):
 massmed = med[ntemp+nvmr+ngamma]
 masshigh = massmed + high[ntemp+nvmr+ngamma]
 masslow = massmed - low[ntemp+nvmr+ngamma]
 radmed = med[ntemp+nvmr+ngamma+nmass]
 radhigh = radmed + high[ntemp+nvmr+ngamma+nmass]
 radlow = radmed - low[ntemp+nvmr+ngamma+nmass]
else:
  massmed, masshigh, masslow  = 0,0,0
  radmed, radhigh, radlow = 0,0,0

sigmamed = np.sqrt(yerr**2 + 10**med[-1])
sigmahigh = np.sqrt(yerr**2 + 10**high[-1])
sigmalow = np.sqrt(yerr**2 + 10**low[-1])
```

```
#     model = nemesisEMCEE.nemesisemcee(runname, len(y), len(tcubic), len
print(runname)
medmodel = nemesisdiscEMCEE.nemesisdiscemcee(runname, len(y), len(tempme
highmodel = nemesisdiscEMCEE.nemesisdiscemcee(runname, len(y), len(temph
lowmodel = nemesisdiscEMCEE.nemesisdiscemcee(runname, len(y), len(templo

fig, ax = plt.subplots(figsize=(20, 10))

plt.plot(wv, y, label='Data', color='g')
plt.fill_between(wv, (y-sigmamed), (y+sigmamed), facecolor='g', alpha=0.

plt.plot(wv, medmodel, label='Med', color='b')
plt.fill_between(wv, lowmodel, highmodel, facecolor='b', alpha=0.7)
# plt.fill_between(wv, medmodel-medsigma, medmodel+medsigma, facecolor='b
# plt.plot(wv, mlmodel, label='Max Like')
plt.legend()
plt.tight_layout()
# plt.ylim(0.05*min(y-yerr), 2*max(y+yerr))
plt.set_xlabel=(r'$Wavelength (\mu m)$')
plt.set_ylabel=('Luminosity (W)')
plt.savefig(str(percent)+'%'+"spectra.pdf",transparent=True)
plt.close()

#                         END OF DEFINITIONS
##############################################################################
```

Plotting function for plotting spectra. This works by finding the median and $\pm 1\sigma$ values for each parameter. Then the data is plotted with the inflated errors, along with the median spectra, with shading included for $\pm 1\sigma$. I've done by best to make this friendly to flags but not tested it extensively.

2.3.3 Inputs

```
#                         BEGINNING OF INPUTS
##############################################################################


# enter runname
runname = 'jupiter_hot'

# read spectrum and errors
input = np.loadtxt("./0/" + runname + ".spx", skiprows=4).T
```

```
wv = input[0]
y = input[1]
yerr = input[2]
# read in OE result
nemres = np.loadtxt("./0/" + runname + ".ref", skiprows=13).T
nempres = np.log10(nemres[1])
nemtemp = nemres[2]
# VMRs except H2 He
nemvmr = np.loadtxt("./0/" + runname + ".ref", skiprows=13)[0][3:10]
```

Here we enter the basename of the NEMESIS files, and read in the a priori TP-profile and VMRs, and the spectra. The number of 'skiprows' in the .spx and .ref files will change from time to time, so make sure you're capturing all of the relevant information by changing this.

Note that for the VMRs, the H_2 and He VMRs must be put to the last two columns and not recorded in the a priori.

```
# a priori / starting values

# take every 5th element of a priori from .ref file as
# starting values
#pres0=nempres[::6]
#tempp0=nemtemp[::6]
pres0 = np.append(nempres[::6], nempres[-1])
tempp0 = np.append(nemtemp[::6], nemtemp[-1])
vmrp0=np.log10(nemvmr)
```

Here we chop up the TP-profile into a smaller number of points. This takes some adjustment, depending on the initial number of layers, as the new numbers MUST begin and end at the first and last values of the initial TP profile, otherwise interpolation will fail. If there isn't a nice integer number that does this automatically, you can append the last point manually (as shown above).

```
gamma0 = 1e5

bp0 = log10(0.01*(min(yerr))**2)
# mp0 is * 1e-24 kg. 1898e24 is mass of jup
mjup = 1898.0
mp0 = 30.0*mjup
# rp0 is in km, rjup
rjup = 69911.0
rp0 = 1.0*rjup

# define h2 and he standard VMRs
h2p = 0.85
```

```
hep = 0.15
```

The starting values for the other parameter inputs go here. Hopefully these are self-explanatory.

```
# define number of params
# no need to change nmass, nrad etc as this will happen at
# flag check stage
ntemp, nvmr, ngamma, nmass, nrad, nb  = len(tempp0), len(vmrp0), 1, 1, 1,

# flags for using the VMR and grav(mass+radius).
# if = 0, then the values put in for
# VMR and grav are ignored, so that only
# temp profile can be retrieved.
# tflag = 0 uses the ref temperature
tflag = 1
vflag = 1
gflag = 1
bflag = 1

if (gflag==0):
 nmass = 0
 nrad = 0

if (tflag==0):
 ntemp = 0

if(vflag==0):
 nvmr = 0

if(bflag==0):
 nb = 0

ndim = ntemp+nvmr+ngamma+nmass+nrad+nb
```

Here we define the number of parameters included in the model. This should be done via the relevant flags, and not directly.

```
# number of walkers
nwalkers = ndim*10
# numer of iterations
niter = 10000
#number of burn-in steps
nburn = 0
```

Definitions of the number of walkers. I've been told from people in-the-know that 10 walkers per parameter is a decent number. For reference, Mike Line uses 8 per parameter.

The number of iterations should be at least 10000, if not 20000.

The number of burn steps is often arbitrary and depends totally on the system, data and initial positions. Sometimes it's been 300, sometimes 3000 iterations before the sampling reaches an equilibrium position. There are some arguments for never throwing away samples, and that burn-in isn't necessary, so I stick with 0 as default. Later you can throw away the samples manually if you want perfect gaussian PDFs without the trouble of guessing the burn-in period.

```
#initial positions

p0 = [np.zeros((ndim)) for i in xrange(nwalkers)]

for i in xrange(nwalkers):
 t0 = np.random.random(ntemp)+tempp0
 v0 = np.random.random(nvmr)*(-0.1)+vmrp0
 g0 = np.random.random(1)+gamma0
 m0 = np.random.random(1)+mp0
 r0 = np.random.random(1)+rp0
 b0 = -0.01*bp0*np.random.random(1) + bp0
 p0[i] = np.concatenate((t0,v0,g0,m0,r0,b0))
# p0[i] = v0
```

Here we setup the initial positions of the walkers - a tight N-dim ball of paramater space near what we hope is the most likely answer. Another method would be to have the parameters in each walker as vastly different from one another, although this method would theoretically take longer than the first method (presuming we are nearby the most-likely answer).

```
continuerun = False
# do not change start
start = 0

if(continuerun == True):
 pos = load_object('pos.bin')
 state = load_object('state.bin')
 prob = load_object('prob.bin')
 start = load_object('start.bin')
 intchain = load_object('chain.bin')
 intflatchain = load_object('flatchain.bin')
 intlikeflat = load_object('likeflat.bin')
```

```
#                                END OF INPUTS
#############################################################################
```

Here we can decide to continue a previous run or not. If you do wish to, the relevant parameters will be loaded in.

```
#                              BEGINNING OF MCMC
#############################################################################

# decide number of cores to use here
sampler = emcee.EnsembleSampler(nwalkers, ndim, lnprob, threads=1)
print("Running MCMC...")
```

This is where we instantiate the sampler with X number of cores (threads). Threads=10 means that the MCMC will use 10 cores for the calulcation. Note here that the maximum is going to 64 threads for 64 cores. It is possible to create a message-passing-interface in general for EMCEE, so that multiple nodes may be used (i.e. 2x64 core computers = 128 cores = 128 threads), but we have not implemented it here.

```
# the MCMC will incrementally save probabilities etc
# and make some pretty graphs every 'savenum'
# iterations
savenum = 100
interval = niter / savenum
if(continuerun == False):
 f = open('maximumprob.dat', 'w')
```

Savenum represents the number of iterations after which results are saved / plotted. 100 iterations usually takes 3-6 hours depending on the MCMC. This can be altered as desired, but a default of 100 seems reasonable.

```
# do not change sumtime
sumtime = 0

for i in xrange(interval):
# if we are continuing a run, then start will be non-zero
# and it will pick up where it left off (hopefully)
 i = i + start
 print('i, start:')
 print(i, start)

 if (i > interval):
  break
```

```
 inittime = time.time()
```

Here we begin a loop over the MCMC, where each interval (100 iterations) will plot stuff. These lines of code are dedicated to finding how far in the calculation we are, so that we can label the plots by their percentage completed in the calculation.

Once it finds the starting point, it notes the initial starting time for future calculations of approximately how long the run will take.

```
 if (i == 0):
  pos, prob, state = sampler.run_mcmc(p0, savenum)
 elif (i == nburn/savenum):
# remove burn-in period samples
  print("sampler reset")
  sampler.reset()
  pos, prob, state = sampler.run_mcmc(pos, savenum, lnprob0=prob, rstate0
 else:
  pos, prob, state = sampler.run_mcmc(pos, savenum, lnprob0=prob, rstate0
 f = open('maximumprob.dat', 'a')
```

Calculate the initial positions MCMC, then continue with new positions (or reset the sampler for the specified burn-in period).

```
 if (continuerun==False):
  samples = sampler.flatchain
  chain = sampler.chain
  lnlike_flat = sampler.flatlnprobability
 else:
  samples = np.append(intflatchain,sampler.flatchain,axis=0)
  chain = np.append(intchain,sampler.chain,axis=1)
  lnlike_flat = np.append(intlikeflat, sampler.flatlnprobability)
```

If continuing a run, load the relevant parameters.

```
# get names of params, then plot PDFs
 tnames = np.array([])
 for t in range(ntemp):
  tna = "T" + str(t+1)
  tnames=np.append(tnames,tna)

 gasnames=np.array(['H2O', 'CO2', 'CO', 'CH4', 'NH3', 'TiO', 'VO'])
 restnames = np.array(['Gamma', 'Mass', 'Radius', 'b'])
 paramnames = np.append(tnames, gasnames)
 paramnames = np.append(paramnames, restnames)
 plot_pdf_kde(ndim,paramnames,samples)
```

```
# work out percentage complete for plot names
 percent = ((i+1)*100 / interval)
```

Get the names of the temperature points, the gas names and the rest of the parameter names, for use in subsequent plotting. Plot the PDFs. Calculate the percentage completed so far.

```
# triangle plot

 figure=triangle.corner(samples,
 quantiles=[0.16,0.5,0.84])

 figure.savefig(str(percent)+ '%'+"triangle.pdf")
 plt.close()

# plot the walker positions over interations

 plot_walkers_iter(ndim, paramnames, chain)

 autocorr_time = sampler.get_autocorr_time()
```

Plot the triangle plot and save, similarly for the walkers plot. Calculate the autocorrelation time (a measure of how convergence time).

```
# get the maximum likelihood value and parameter values
 f.write('\n\tMean acceptance: %d \n' % (sp.mean(sampler.acceptance_fract
 ML = sp.unravel_index(lnlike_flat.argmax(), lnlike_flat.shape)
 par_ML = samples[ML]
# calculate the parameter medians and +/- 1 sigma uncertainties
# for a normal distribution 68% of scores between +/- 1 sigma
# this corresponds to percentiles of 16 (-1 sig) and 84 (+1 sig)

 f.write('\n\tAutocorrelation time: \n%s' % autocorr_time)
 f.write('\n\tMaximum likelihood parameters = \n%s\nML index = %s \nML va
# f.write('Median, Upper, Lower = \n%s\n' % vmr)
 f.write('PERCENTAGE COMPLETE: %s%% \n' % percent)
```

Write the autocorrelation time, maximum likeilhood parameters, and percentage complete to file.

```
# plot spectra and temperature profile if in retrieval

 plot_spectra(percent, samples, runname, y, yerr, vflag, gflag, ntemp, nv
```

```
# save run details to load later
 save_object('state.bin',state)
 save_object('pos.bin', pos)
 save_object('prob.bin', prob)
 save_object('start.bin', i+start+1)
 save_object('chain.bin', chain)
 save_object('flatchain.bin', samples)
 save_object('likeflat.bin', lnlike_flat)
```

Plot spectra and save relevant parameters.

```
# Calculate Gelman-Rubin statistic  (should be distributed < 1.1 for conv
# Take last 50% of chain.
# W is the mean of the variances of each chain.
# chainit is iterations of chain
 chainlen = len(chain[0,:,0])/2
 wchain = chain[:,chainlen:,:]

 W = np.mean(np.var(wchain, axis=1),axis=0)

# B is the variance of the chain means multipied by n because each chain
# (length of chain = 2n, therefore n is 50% of chain length)

# B = 0.0
# for i in range(nwalkers):
#  B = B + ( np.mean(chain[i,:,:]) - np.mean(chain,axis=0) )**2

 B = (chainlen / (nwalkers-1.0)) * np.var(np.mean(wchain,axis=1),axis=0)
# Variance of stationary distribution is weighted average
# of W and B:

 var = (1.0-(1.0/chainlen))*W + (1.0/chainlen)*B

# because of overdispersion of the starting values,
# this overestimates the true variance, but it is unbiased if
# the starting distribution equals the stationary distribution
# (if starting values were not overdispersed)

# The potential scale reduction factor is
 R = np.sqrt(var/W)

# When R is high (>1.1 or 1.2) then we should run our chains out
# longer to improve convergence to the stationary distribution

 f.write('GELMAN-RUBIN STATISTIC: %s\n' % R)
```

Calculate and write the Gelman-Rubin statistic, a measure of covergence for R < 1.1.

```
 totaltime = (time.time() - inittime) / (60*60)
 f.write('TIME TAKEN (HOURS): %s \n' % totaltime)
 sumtime += totaltime
 averagetime = (100.0/percent * sumtime)
 f.write('AVERAGE TIME TO GO (HOURS): %s \n' % averagetime)
 averagetime = averagetime/24.0
 f.write('AVERAGE TIME TO GO (DAYS): %s \n' % averagetime)
 f.close()

print("Done.")

#                              END OF MCMC
###############################################################################
```

Write the approximate time taken and time to go, and then loop into the next 100 iterations. After completing all iterations, the MCMC is finished.

2.3.4 Changes to the EMCEE Package

```
        if self.runtime_sortingfn is not None:
            p, idx = self.runtime_sortingfn(p)
        np.savetxt('position.txt',p)

        # Run the log-probability calculations (optionally in parallel).
        results = list(M(self.lnprobfn, [p[i] for i in range(len(p))]))
```

Only one alteration was made to the emcee package itself. In the ensemble.py code, just before the log-probability calculations are called, we save the positions of the walkers in a .txt file. This is so that we can deduce which walker we are calculating, so that we can decide which directory to calculate in.

2.3.5 Changes to the NEMESIS Codes

The *EMCEE.f editions have only a few differences to the regular nemesis*.f codes, we'll discuss those changes here.

```
      subroutine nemesisdiscEMCEE(runname, specsize, mcntemp,
     1 mcnvmr, ith, intemp, invmr, vflag, gflag,
     2 inmass, inrad, MCMCspec)
```

The 'programs' have been altered to subroutines, as shown above.

```
C     Set measurement vector and source vector lengths here.
      include '../radtran/includes/arrdef.f'
      include '../radtran/includes/planrad.f'
      include '../radtran/includes/emcee.f'
      INCLUDE 'arraylen.f'
```

'emcee.f' has been added to includes for passing the mass and radius.

```
c     RG EMCEE INPUTS
c 999 = nconv number of wavelengths, must be exact for python to not read
c and change the shape of the array
      character*100 runname
      integer arrsize, specsize,ith, vflag, gflag
      real MCMCspec(specsize), xfac
      real intemp(mcntemp), invmr(mcnvmr)
      real inrad, inmass
      integer mcntemp, mcnvmr
      character*3 sith
      character*255 path, path1, oldpath, path2
      logical countexist

cf2py intent(in) runname, specsize,mcntemp,mcnvmr
cf2py intent(in) intemp,invmr,ith
cf2py intent(in) inmass, inrad, vflag, gflag
cf2py intent(out) MCMCspec
cf2py depend(mcntemp) intemp
cf2py depend(mcnvmr) invmr
cf2py depend(specsize) MCMCspec

C     ************************************************
```

The additional parameter definitions, as well as their 'intents' (i.e., if they are passed from python or to python), and dependencies (which dictate the sizes of the arrays).

```
C     ************************************************
C     ***************    CODE ****************************
C     ************************************************

C RG find free directory to do calculation

      CALL getcwd(path1)

      if (ith.lt.10)then
```

```
        write( sith, '(i1)' ) ith
       elseif (ith.ge.10.and.ith.lt.100)then
        write( sith, '(i2)' ) ith
       else
        write( sith, '(i3)' ) ith
       endif
       print*, path, ith, sith
       CALL chdir(TRIM(path1)//'/'//TRIM(sith))
       CALL getcwd(path)
       print*, ith, sith, path, path1

c RG assign array elements. This intermediate way of passing is due to
c being unable to pass the allocatable arrays (necessary for the F77-Py
c interface so that dimensions match during passing) in a common block.

       VMRflag = vflag
       GRAVflag = gflag
       MCMCmass = inmass
       MCMCrad = inrad



       DO i=1, mcnvmr
        MCMCvmr(i) = 10**(invmr(i))
       ENDDO

       DO i=1, mcntemp
        MCMCtemp(i) = intemp(i)
       ENDDO
```

In the first section, we find the new directory to calculate in and switch to that directory. We then reassign array elements. This intermediate way of passing is to being unable to pass the allocatable arrays (necesary for the F77-Py interface so that dimensions match during passing) in a common block.

```
C      set up a priori of x and its covariance
       CALL readaprioriMCMC(runname,lin,lpre,xlat,npro,nvar,varident,
      1   varparam,jsurf,jalb,jtan,jpre,jrad,jlogg,nx,xa,sa,lx)
```

Readapriori.f is adjusted here to overwrite the temperatures read in to the forward model. Subprofretg.f has also been adjusted for the VMRs.

2.3.6 The Checklist

Okay, so you've read through the manual (yes all of it!) and there's a whole bunch of information and tidbits you've probably forgotten. From your base code, go through this checklist to initiate a run:

1. Setting up the run
   (a) Setup NEMESIS files and set niter = -1. Ensure H_2 and He are the last VMRs in the .ref files.
   (b) Multiply the directories into 300 subdirectories (multidirec.csh)
   (c) Compile the .comp file to create a .so file.(optional - only needs to be done once)
   (d) Copy over the relevant base code and shared object to the top-level directory (nemesisdiscEMCEE.py, nemesisdiscEMCEE.so)
2. Creating the N-d Gaussian Ball
   (a) Enter runname, specify the number of rows skipped for .ref and .spx, and the VMRs up to H_2 / He.
   (b) Ensure your pressure and temperature points begin and end with the same points as your a priori. (pres0, tempp0)
   (c) Enter a priori values for mass, radius, γ, b, the standard VMRs for H_2 and He. (mp0, rp0 etc)
   (d) Adjust flags to capture the relevant parameters you wish to retrieve.(tflag, gflag etc)
   (e) Enter your number of walkers, iterations, and burn-in period. (nwalkers, niter, nburn)
   (f) Edit your initial positions so that the concatenation only contains the parameters you wish to retrieve. (p0)
3. Controlling the speed of execution
   (a) Decide if you want to continue the run or start afresh. (continuerun)
   (b) Enter the number of cores (threads) you wish to use.
   (c) Decide how often you wish the sampling to be saved to file / plotted. (savenum)
4. Fine-tuning
   (a) Edit 'gasnames', 'restnames' to reflect the different parameters in the run for plotting.

(b) Adjust the upper and lower limits for priors (mass, radius, VMRs, temp etc)

(c) Include only the appropriate priors in lnprior.

(d) Adjust the plots to your liking.

5. The Final Check

(a) Set threads=1 and model=1 and begin the run (ipython nemesisPTEMCEE.py > tmp.dat &). Wait until it completes two rounds of savenum iterations, to ensure nothing has gone awry in the setup.

(b) Fix any outstanding issues.

(c) Reset threads and model to their appropriate value / function.

(d) Strap yourself in.

Bibliography

Foreman-Mackey, D., Hogg, D. W., Lang, D., & Goodman, J. 2013, PASP, 125, 306

Jullion, A., & Lambert, P. 2007, Computational Statistics And Data Analysis, 51, 2542

Lang, S., & Brezger, A. 2004, Journal of Computational and Graphical Statistics, 13, 183

Line, M. R., Teske, J., Burningham, B., Fortney, J. J., & Marley, M. S. 2015, ApJ, 807, 183