The PyMC MCMC python package

MCMC Coffee - Vitacura, December 7, 2017

Jan Bolmer

Outline

PyMC - Version 2.3.6

Purpose

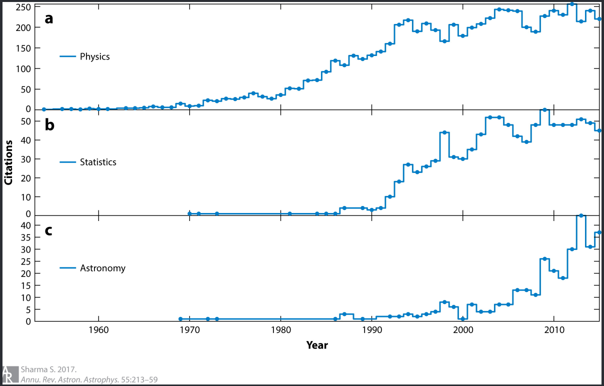
`https://pymc-devs.github.io/pymc/`

> PyMC *is a python module that implements Bayesian statistical models and fitting algorithms, including Markov chain Monte Carlo. Its flexibility and extensibility make it applicable to a large suite of problems. Along with core sampling functionality,* PyMC *includes methods for summarizing output, plotting, goodness-of-fit and convergence diagnostics.*

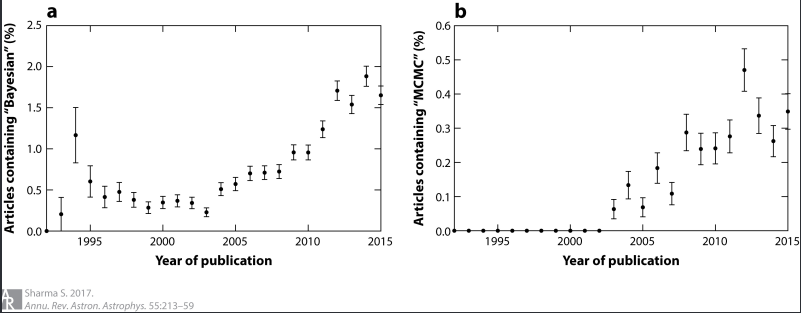
MCMC, Bayesian Statistics

- Metropolis-Hastings algorithm 1970 + increase in computational power

Citations of Metropolis et al. (1953)



MCMC, Bayesian Statistics



- Problems with correlations and degeneracies between parameters ⇒ development of many new algorithms (Gibbs, nested sampling etc.)
- Challenge: express problem within the Bayesian framework; choose the appropriate MCMC method (i.e. Python package) to solve it

Marcov Chain Monte Carlo, Bayesian Statistics

class of algorithms used to efficiently sample posterior distributions

Monte Carlo:
Generation of random Numbers (sample from a distribution)

Marcov Chain:
chain of numbers, with each number depending on the previous number

$$\theta_{t+1} = \text{Normal}\left(\theta_t, \sigma\right)$$

Marcov Chain Monte Carlo, Bayesian Statistics

class of algorithms used to efficiently sample posterior distributions

Monte Carlo:

Generation of random Numbers (sample from a distribution)

Marcov Chain:

chain of numbers, with each number depending on the previous number

$$\theta_{t+1} = \text{Normal}\left(\theta_t, \sigma\right)$$

Bayesian Statistics: We are interested in the Probability/Posterior Distribution of a (set of) parameter(s) θ, which we want to sample

$$P\left(\theta|D, M\right) = \frac{P\left(D|\theta, M\right) P\left(\theta\right)}{P\left(D\right)} \text{ (Bayes Theorem)}$$

Metropolis-Hastings Algorithm

Algorithm to decide weather a new value should be accepted or not, e.g. the Metropolis Hastings Algorithm

$$\theta_{t+1} = \text{Normal}\left(\theta_t, \sigma\right)$$

$$a = \frac{P\left(\theta_{t+1} | D, M\right)}{P\left(\theta_t | D, M\right)} \overset{\text{Bayes Theorem}}{=} \frac{\frac{P(D|\theta_{t+1}, M) P(\theta_{t+1})}{P(D)}}{\frac{P(D|\theta_t, M) P(\theta_t)}{P(D)}} = \frac{\mathscr{L}\left(\theta_{t+1}\right) P\left(\theta_{t+1}\right)}{\mathscr{L}\left(\theta_t\right) P\left(\theta_t\right)}$$

Metropolis-Hastings Algorithm

Algorithm to decide weather a new value should be accepted or not, e.g. the Metropolis Hastings Algorithm

$$a = \frac{P(\theta_{t+1}|D, M)}{P(\theta_t|D, M)} \overset{\text{Bayes Theorem}}{=} \frac{\frac{P(D|\theta_{t+1}, M)P(\theta_{t+1})}{P(D)}}{\frac{P(D|\theta_t, M)P(\theta_t)}{P(D)}} = \frac{\mathscr{L}(\theta_{t+1})P(\theta_{t+1})}{\mathscr{L}(\theta_t)P(\theta_t)}$$

$$P(D) = \int_\theta P(x, \theta)\, d\theta$$

hard to compute!

Likelihood function (assumption of Gaussian errors):

$$\mathscr{L}(\theta) = \prod_i l_i(\theta) = \prod_i \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{(x_i-\mu)^2}{2\sigma_i^2}}$$

Metropolis-Hastings Algorithm

Algorithm to decide weather a new value should be accepted or not, e.g. the Metropolis Hastings Algorithm

$$a = \frac{P\left(\theta_{t+1}|D, M\right)}{P\left(\theta_t|D, M\right)} \overset{\text{Bayes Theorem}}{=} \frac{\frac{P(D|\theta_{t+1}, M)P(\theta_{t+1})}{P(D)}}{\frac{P(D|\theta_t, M)P(\theta_t)}{P(D)}} = \frac{\mathscr{L}\left(\theta_{t+1}\right)P\left(\theta_{t+1}\right)}{\mathscr{L}\left(\theta_t\right)P\left(\theta_t\right)}$$

$$\theta_{t+1} = \begin{cases} \theta_{t+1}, & \text{if } a > 1 \\ \theta_t, & \text{otherwise} \end{cases}$$

(Animation!)

PyMC - Version 2.3.6

Features

- Includes a large suite of well-documented statistical distributions

PyMC - Version 2.3.6

Features

- Includes a large suite of well-documented statistical distributions
- Creates summaries including tables and plots (Trace, Posterior Distribution, quantiles etc.)

PyMC - Version 2.3.6

Features

- Includes a large suite of well-documented statistical distributions
- Creates summaries including tables and plots (Trace, Posterior Distribution, quantiles etc.)
- Traces can be saved to the disk as plain text, Python pickles, SQLite or MySQL database, or hdf5 archives (⇒ powerful in combination with `pandas`)

PyMC - Version 2.3.6

Features

- Includes a large suite of well-documented statistical distributions
- Creates summaries including tables and plots (Trace, Posterior Distribution, quantiles etc.)
- Traces can be saved to the disk as plain text, Python pickles, SQLite or MySQL database, or hdf5 archives (⇒ powerful in combination with `pandas`)
- Several convergence diagnostics are available

PyMC - Version 2.3.6

Features

- Includes a large suite of well-documented statistical distributions
- Creates summaries including tables and plots (Trace, Posterior Distribution, quantiles etc.)
- Traces can be saved to the disk as plain text, Python pickles, SQLite or MySQL database, or hdf5 archives (⇒ powerful in combination with `pandas`)
- Several convergence diagnostics are available
- Extensible: easily incorporates custom step methods and unusual probability distributions. MCMC loops can be embedded in larger programs, and results can be analyzed with the full power of Python

PyMC - Version 2.3.6

Comparison with other packages

- `emcee`: extremely lightweight, easy to use, affine-invariant ensemble sampling (developed by astronomers)

PyMC - Version 2.3.6

Comparison with other packages

- `emcee`: extremely lightweight, easy to use, affine-invariant ensemble sampling (developed by astronomers)
- `PyMC`: more features than emcee, including built-in support for efficient sampling of common prior distributions. Metropolis-Hasting ([1]). Version 3 is independent of `fortran`, includes Gibbs-Sampling; not fully stable yet.

PyMC - Version 2.3.6

Comparison with other packages

- `emcee`: extremely lightweight, easy to use, affine-invariant ensemble sampling (developed by astronomers)
- `PyMC`: more features than emcee, including built-in support for efficient sampling of common prior distributions. Metropolis-Hasting ([1]). Version 3 is independent of `fortran`, includes Gibbs-Sampling; not fully stable yet.
- `PyStan`: official Python wrapper of the Stan Probabilistic programming language, which is implemented in `C++`. Uses a No U-Turn Sampler, which is more sophisticated than classic Metropolis-Hastings or Gibbs sampling ([1]). Requires writing non-python code, harder to learn.

PyMC - Version 2.3.6

Comparison with other packages

- `emcee`: extremely lightweight, easy to use, affine-invariant ensemble sampling (developed by astronomers)
- `PyMC`: more features than emcee, including built-in support for efficient sampling of common prior distributions. Metropolis-Hasting ([1]). Version 3 is independent of `fortran`, includes Gibbs-Sampling; not fully stable yet.
- `PyStan`: official Python wrapper of the Stan Probabilistic programming language, which is implemented in `C++`. Uses a No U-Turn Sampler, which is more sophisticated than classic Metropolis-Hastings or Gibbs sampling ([1]). Requires writing non-python code, harder to learn.
- `MultiNest`: nested sampling techniques, which are superior for parameter spaces with strong and non-linear correlations. Written in `fortran` and `C`, python wrapper available: `http://johannesbuchner.github.com/PyMultiNest/`

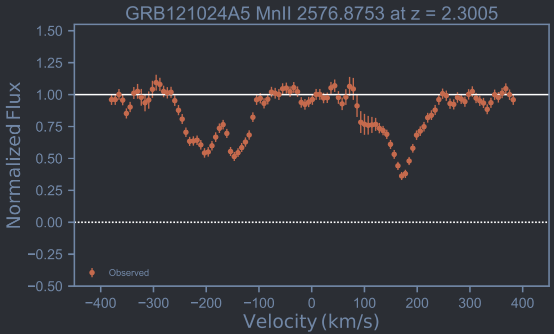
Outline

1. PyMC, MCMC & Bayesian Statistics
2. Absorption Line Fitting
   2.1 Absorption Lines in GRB afterglow spectra
   2.2 The Voigt Profile
3. Implementation in PyMC

Absorption Line Fitting

Fitting N Voigt profiles to GRB afterglow spectra

- Voigt Profile(s): (N, b, z) + Continuum, Background
- Popular codes: `VPFIT`, `autoVP`, `FITLYMAN/MIDAS` (χ^2-based)
- Problems: non-detections, saturated lines, computationally expansive when fitting multiple components (b, z - fixed)



The Model

Voigt Profile in Velocity Space

$$F_{\text{model}} = F_{\text{cont}} \cdot \prod_{i=1}^{n_{\text{voigt}}} \cdot e^{-\tau_i}$$

$$\tau_i = \frac{\pi e^2}{m_e c} f_{ij} \lambda_{ij} N_i \cdot \phi \left(v - v_{0_i}, \frac{b_i}{\sqrt{2}}, \Gamma \right)$$

```
def add_abs_velo(v, N, b, gamma, f, l0):
    # Add an absorption line in velocity space
    A = (((np.pi*e**2)/(m_e*c))*f*l0*1E-13) * (10**N)
    tau = A * voigt(v,b/np.sqrt(2.0),gamma)
return np.exp(-tau)
```

The Model

Voigt Profile in Velocity Space

```
def add_abs_velo(v, N, b, gamma, f, l0):
    # Add an absorption line in velocity space
    A = (((np.pi*e**2)/(m_e*c))*f*l0*1E-13) * (10**N)
    tau = A * voigt(v,b/np.sqrt(2.0),gamma)
return np.exp(-tau)

from scipy.special import wofz #Faddeeva function
def voigt(x, sigma, gamma):
    #gamma: HWHM of the Lorentzian profile
    #sigma: the standard deviation of the Gaussian profile
    z = (x + 1j*gamma) / (sigma * np.sqrt(2.0))
    V = wofz(z).real / (sigma * np.sqrt(2.0*np.pi))
    return V
```

Outline

Implementation in PyMC

General Structure

```
import pymc, numpy, scipy, matplotlib
def model(velocity, flux, flux_err, *args, **kwargs):
    def priors(): #Priors on unknown parameters
        return priors
    def physical_model(priors):
        return model
    data = pymc.Normal('y_val',mu=physical_model, tau=1.0/(
         flux_err**2),value=flux,observed=True) #likelihood
    return locals()

def mcmc(model, velocity, flux, flux_err):
    MDL = pymc.MCMC(model(velocity, flux, flux_err))
    MDL.sample(iterations=20000, burn_in=15000)

    return fit_parameters
```

Defining the Priors, The Stochastic Class

Variables whose values are not determined by its parents

```
for i in range(1, nvoigts+1): #Automatic (Iteratively,
     Containers)
    v0 = pymc.Uniform('v0'+str(i),lower=-400,upper=400,
                       doc='v0'+str(i))
    N  = pymc.Normal('N'+str(i),mu=15.0,tau=1.0/(10**2),
                       doc='A'+str(i)
    b  = pymc.Normal('b'+str(i),mu=15.0,tau=1.0/(10**2),
                       doc='b'+str(i))
    vars_dic['b'+str(i)] = b # etc.; Add to dictionary
@pymc.stochastic(dtype=float) #Decorator
def BG(value=1.0, mu=1.0, dev=0.05, doc='BG'):
    if 0.90 <= value < 1.10:
        return gauss(value, mu, sig)
    else:
        return -np.inf
```

Python Decorators

Modifying functions without rewriting code

```
def add(x, y):
    return x + y
def sub(x, y):
    return x - y
def timer(func):
    def f(x, y):
        before = time()
        rv = func(x, y)
        after = time()
        print after-before
        return rv
    return f
add = timer(add)
sub = timer(sub)
```

Python Decorators

Modifying functions without rewriting code

```
def timer(func):
    def f(*args,**kwargs):
        before = time()
        rv = func(*args,**kwargs)
        after = time()
        print after-before
        return rv
    return f
@timer #"Decorate" the add function with the timer function
def add(x, y):
    return x + y
@timer
def sub(x, y):
    return x - y
```

The Physical Model, The Deterministic class

A variable that is entirely determined by its parents

```
@pymc.deterministic(plot=False) #Deterministic Decorator
def multVoigt(vv,BG,f,gamma,l0,nvoigts,vars_dic):

    gauss_k = Gaussian1DKernel(stddev=RES/fwhmsig*ps,mode="
        oversample")
    flux = np.ones(len(vv))*BG
    for i in range(1, nvoigts + 1):
        v = vv-vars_dic["v0"+str(i)]
        flux *= add_abs_velo(v, vars_dic["N"+str(i)],
            vars_dic["b"+str(i)], gamma, f, l0)
    return np.convolve(flux, gauss_k, mode='same')

y_val = pymc.Normal('y_val',mu=multVoigt,tau=tau,value=fluxv,
    observed=True) #Data with Gaussian errors
return locals()
```

Start the MCMC - The MCMC Class

```
def mcmc(grb, redshift, line, velocity, fluxv, fluxv_err,
     grb_name, gamma, nvoigts, iterations, burn_in, RES,
     velo_range, para_dic):

    MDL = pymc.MCMC(mult_voigts(velocity,fluxv,fluxv_err,
        gamma,nvoigts,RES,CSV_LST, velo_range, para_dic),
        db='pickle',dbname='velo_fit.pickle')

    MDL.db
    MDL.sample(iterations, burn_in)
    MDL.db.close()

    y_min = MDL.stats()['multVoigt']['quantiles'][2.5]
    y_max = MDL.stats()['multVoigt']['quantiles'][97.5]
    y_fit = MDL.stats()['multVoigt']['mean']

    return y_min, y_max, y_fit
```

Start the MCMC - The MCMC Class

```
def mcmc(*args, **kwargs):

    MDL = pymc.MCMC(mult_voigts(*args, **kwargs))

    MDL.use_step_method(pymc.Metropolis, MDL.a, proposal_sd
        =0.05, proposal_distribution='Normal')
    MDL.use_step_method(pymc.Metropolis, MDL.v0, proposal_sd=
        velo_range/2.0, proposal_distribution='Normal')
    MDL.use_step_method(pymc.AdaptiveMetropolis, [MDL.N, MDL.
        b], scales={MDL.N:1.0, MDL.b:1.0})
    MDL.sample(iterations, burn_in)

    y_fit = MDL.stats()['multVoigt']['mean']
    N1_mean = MDL.stats()['N1']['mean']

    return y_fit
```

Additional Features

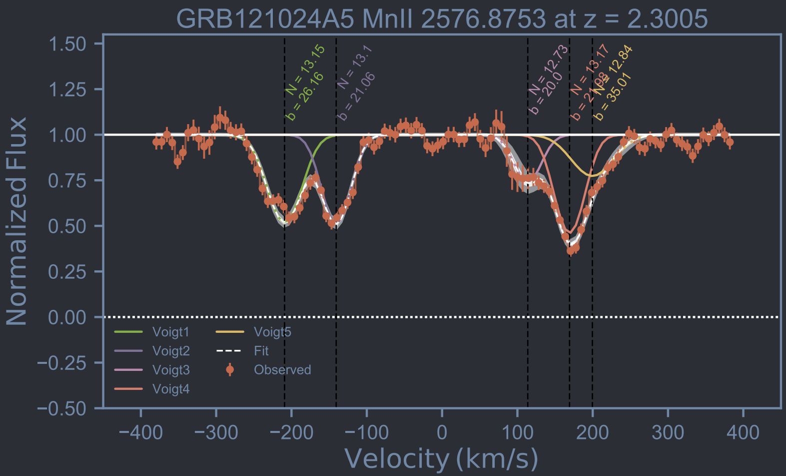
- The Potential Class: add probability terms to existing models (Indicator function)

$$\mathscr{L}(\theta) P(\theta) \cdot I(|v01 - v02| > 5)$$

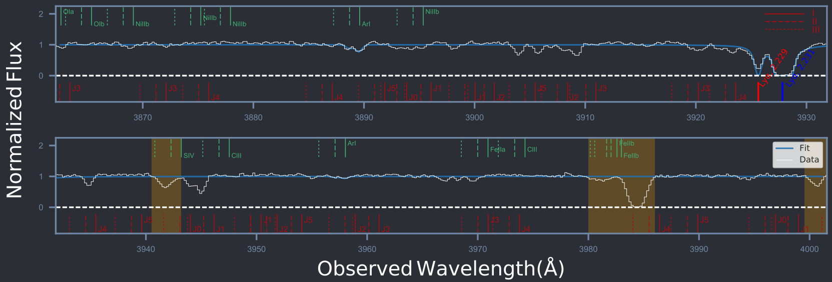
```
@pymc.potential
def I(v01, v02):
    if math.abs(v01-v02) > 5:
        return 1.0
    else:
        return -np.inf
```

- Also: Graphing Models, User-defined step methods etc. (I didn't look into this yet)

Results


GRB121024A5 MnII 2576.8753 at z = 2.3005
Normalized Flux
Velocity (km/s)
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−400
−300
−200
−100
0
100
200
300
400
N = 13.15
b = 26.16
N = 13.1
b = 21.06
N = 12.73
b = 20.0
N = 13.17
N = 12.84
b = 35.01
Voigt1
Voigt2
Voigt3
Voigt4
Voigt5
Fit
Observed

Results



Normalized Flux
Observed Wavelength(Å)
Fit
Data
3870
3880
3890
3900
3910
3920
3930
3940
3950
3960
3970
3980
3990
4000

Bibliography

Thanks for your attention!

[1] http://jakevdp.github.io/blog/2014/06/14/frequentism-and-bayesianism-4-bayesian-in-python/

[2] Sanjib Sharma. *Markov Chain Monte Carlo Methods for Bayesian Data Analysis in Astronomy*. Annual Review of Astronomy and Astrophysics, 2017.