Seeds: Random effect logistic regression

Ported to PyMC by Abraham Flaxman and Kyle Foreman, 2/15/2012 from http://www.openbugs.info/Examples/Seeds.html

This example is taken from Table 3 of Crowder (1978), and concerns the proportion of seeds that germinated on each of 21 plates arranged according to a 2 by 2 factorial layout by seed and type of root extract. The data are shown below, where r_i and n_i are the number of germinated and the total number of seeds on the i-th plate, $i = 1, \ldots, N$. These data are also analysed by, for example, Breslow and Clayton (1993).

The model is essentially a random effects logistic, allowing for over-dispersion. If p_i is the probability of germination on the i-th plate, we assume

$$r_i \sim \text{Binomial}(p_i, n_i)$$

$$\text{logit}(p_i) = \alpha_0 + \alpha_1 x_{1,i} + \alpha_2 x_{2,i} + \alpha_{1,2} x_{1,i} x_{2,i} + b_i$$

$$b_i \sim \text{Normal}(0, \tau)$$

where $x_{1,i}, x_{2,i}$ are the seed type and root extract of the i-th plate, and an interaction term $a_{1,2} x_{1,i} x_{2,i}$ is included.

$\alpha_0, \alpha_1, \alpha_2, \alpha_{1,2}, \tau$ are given independent weakly informative priors.

In [1]:
```
import pylab as pl
import pymc as mc
```

In [2]:
```
### data
# germinated seeds
r =  pl.array([10, 23, 23, 26, 17, 5, 53, 55, 32, 46, 10, 8, 10, 8, 23, 0, 3, 22, 15,

# total seeds
n =  pl.array([39, 62, 81, 51, 39, 6, 74, 72, 51, 79, 13, 16, 30, 28, 45, 4, 12, 41, 3

# seed type
x1 = pl.array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])

# root type
x2 = pl.array([0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1])

# number of plates
N =  x1.shape[0]
```

Here is the model in BUGS:

```
model
{
   for( i in 1 : N ) {
      r[i] ~ dbin(p[i],n[i])
      b[i] ~ dnorm(0.0,tau)
      logit(p[i]) <- alpha0 + alpha1 * x1[i] + alpha2 * x2[i] +
         alpha12 * x1[i] * x2[i] + b[i]
   }
   alpha0 ~ dnorm(0.0,1.0E-6)
   alpha1 ~ dnorm(0.0,1.0E-6)
   alpha2 ~ dnorm(0.0,1.0E-6)
   alpha12 ~ dnorm(0.0,1.0E-6)
   tau ~ dgamma(0.001,0.001)
   sigma <- 1 / sqrt(tau)
}
```

In [3]:
```
### hyperpriors
tau = mc.Gamma('tau', 1.e-3, 1.e-3, value=10.)
sigma = mc.Lambda('sigma', lambda tau=tau: tau**-.5)

### parameters
# fixed effects
alpha_0 =  mc.Normal('alpha_0',  0., 1e-6, value=0.)
alpha_1 =  mc.Normal('alpha_1',  0., 1e-6, value=0.)
alpha_2 =  mc.Normal('alpha_2',  0., 1e-6, value=0.)
alpha_12 = mc.Normal('alpha_12', 0., 1e-6, value=0.)

# random effect
b =        mc.Normal('b',        0., tau,  value=np.zeros(N))

# expected parameter
logit_p =  (alpha_0 + alpha_1*x1 + alpha_2*x2 + alpha_12*x1*x2 + b)


### likelihood
@mc.observed
def obs(value=r, n=n, logit_p=logit_p):
    return mc.binomial_like(r, n, mc.invlogit(logit_p))
```

BUGS uses Gibbs steps automatically, so it only takes 10000 steps of MCMC after a 1000 step burn in for this model in their example.

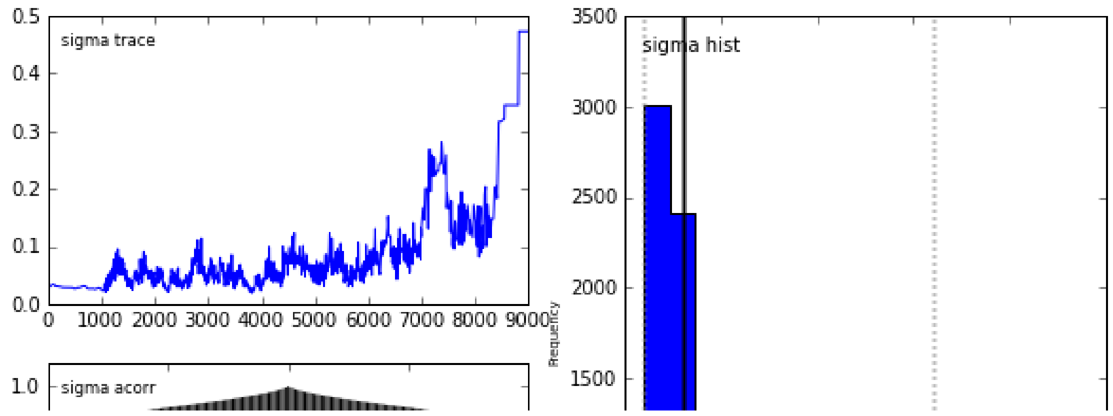
PyMC only uses Gibbs steps if you set them up yourself, and it uses Metropolis steps by default. So 10000 steps go by quickly, but the chain has not converged to the stationary distribution in this time, so predictions based on the samples will be inaccurate.

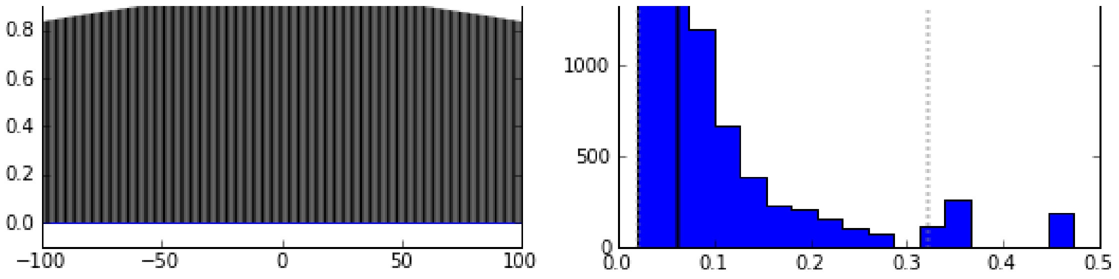
In [4]:
```
m = mc.MCMC([alpha_0, alpha_1, alpha_2, alpha_12, b, tau, sigma, logit_p, obs])

# not long enough to mix
%time m.sample(10000, 1000, progress_bar=False)
```

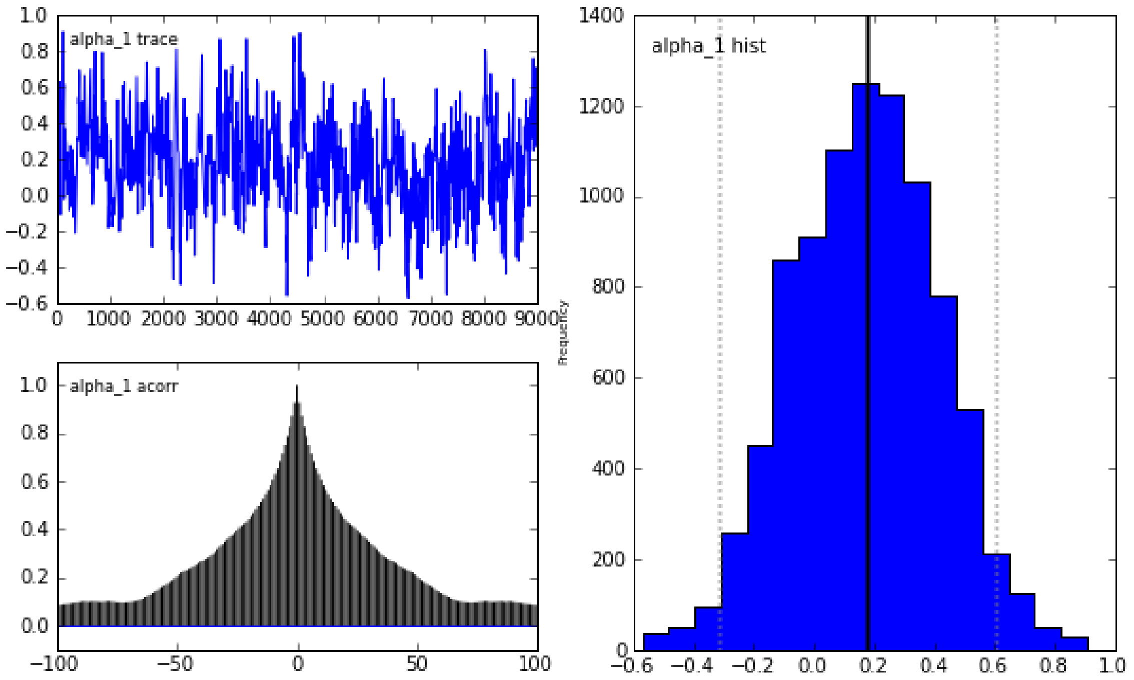
```
CPU times: user 6.75 s, sys: 0.11 s, total: 6.86 s
Wall time: 8.12 s
```

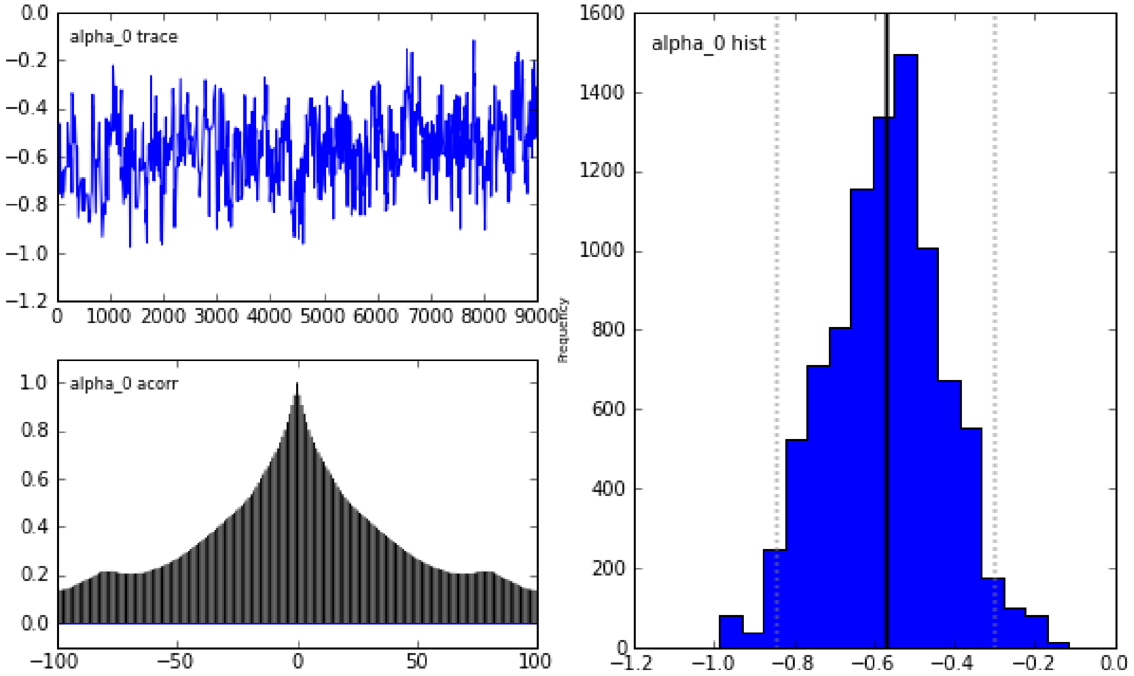
In [5]:
```
mc.Matplot.plot(m)
```

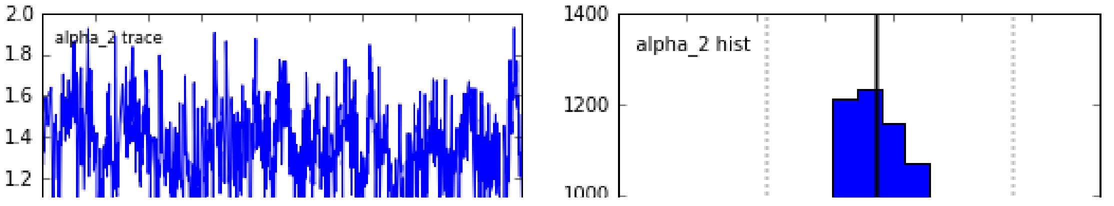
```
Plotting sigma
Plotting alpha_1
Plotting alpha_0
Plotting alpha_2
Plotting tau
Plotting alpha_12
```

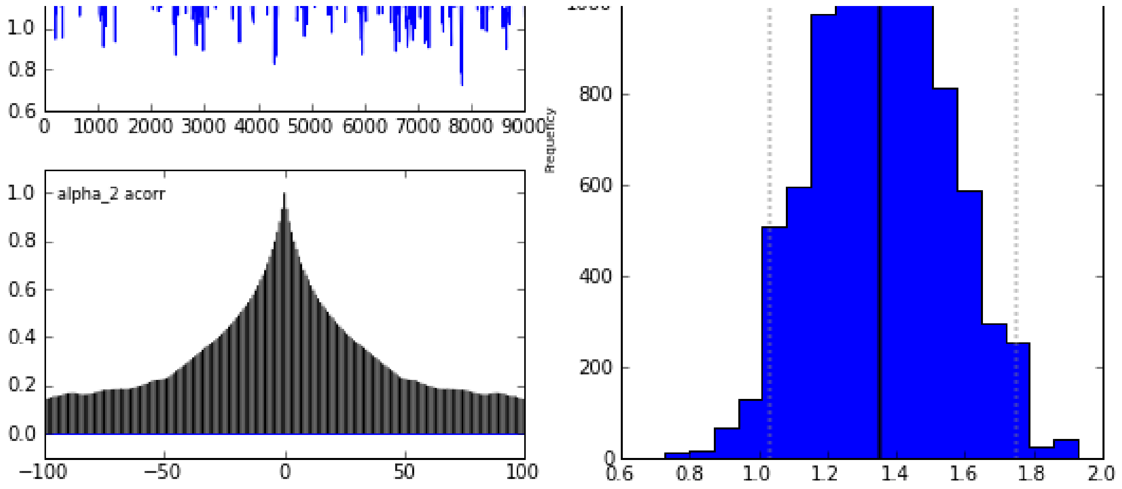


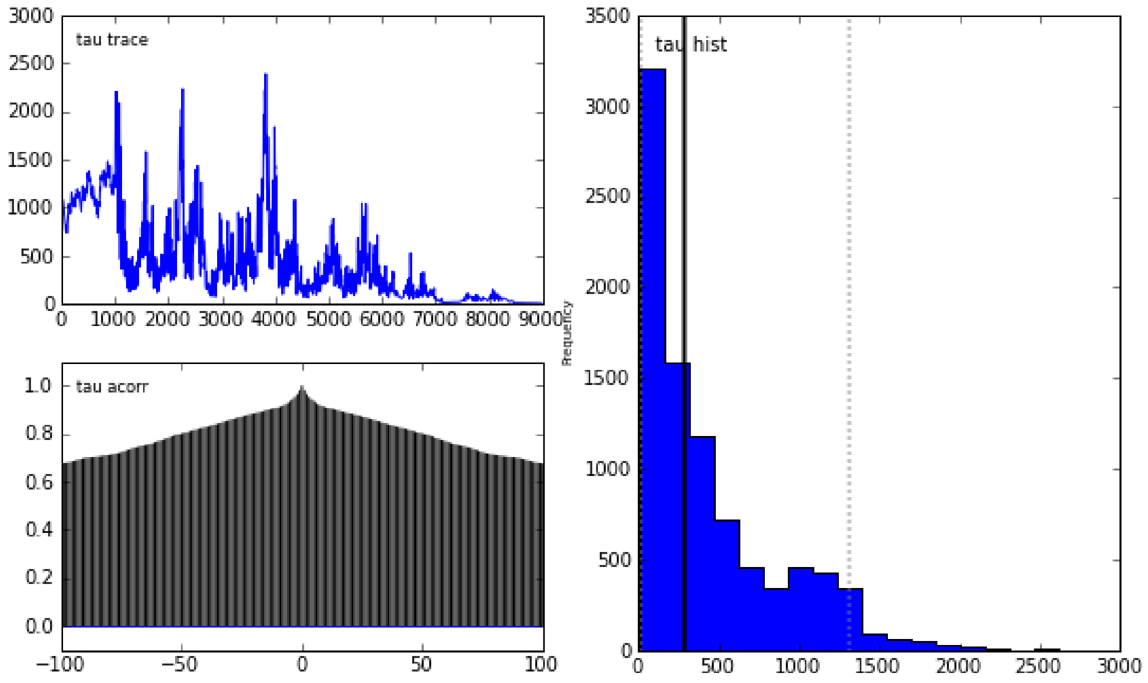


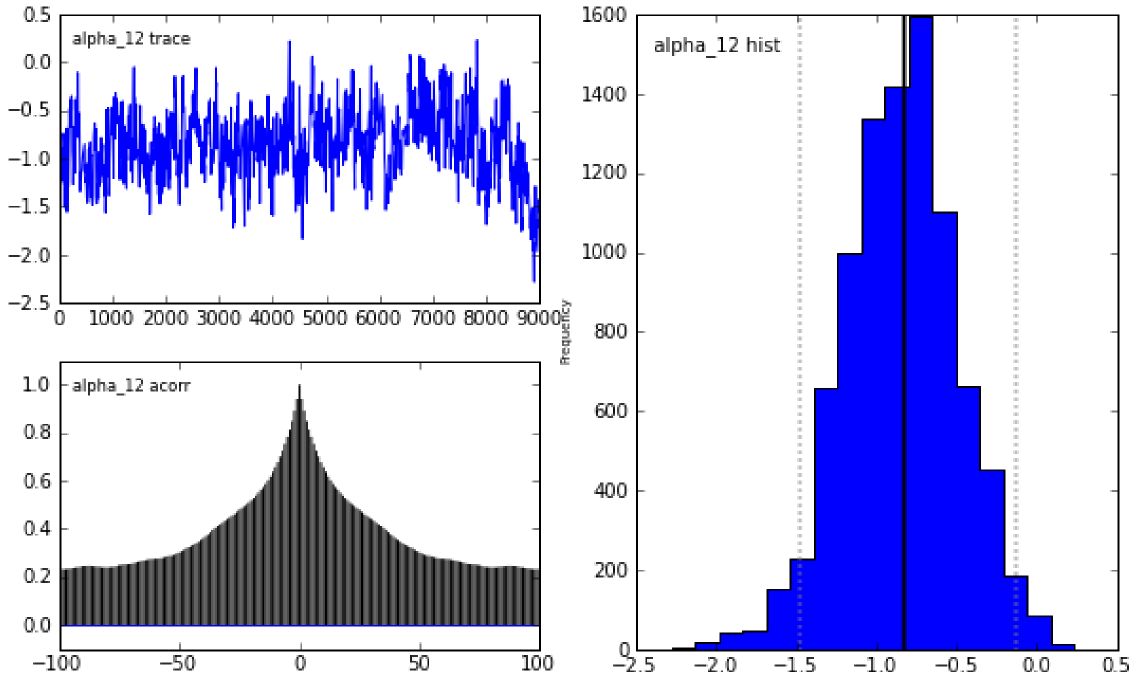

alpha_1 trace
alpha_1 acorr
alpha_1 hist
Frequency


alpha_0 trace
alpha_0 acorr
alpha_0 hist
Frequency


alpha_2 trace
alpha_2 hist


alpha_2 acorr
Frequency


tau trace
tau acorr
tau hist
Frequency


alpha_12 trace
alpha_12 acorr
alpha_12 hist
Frequency

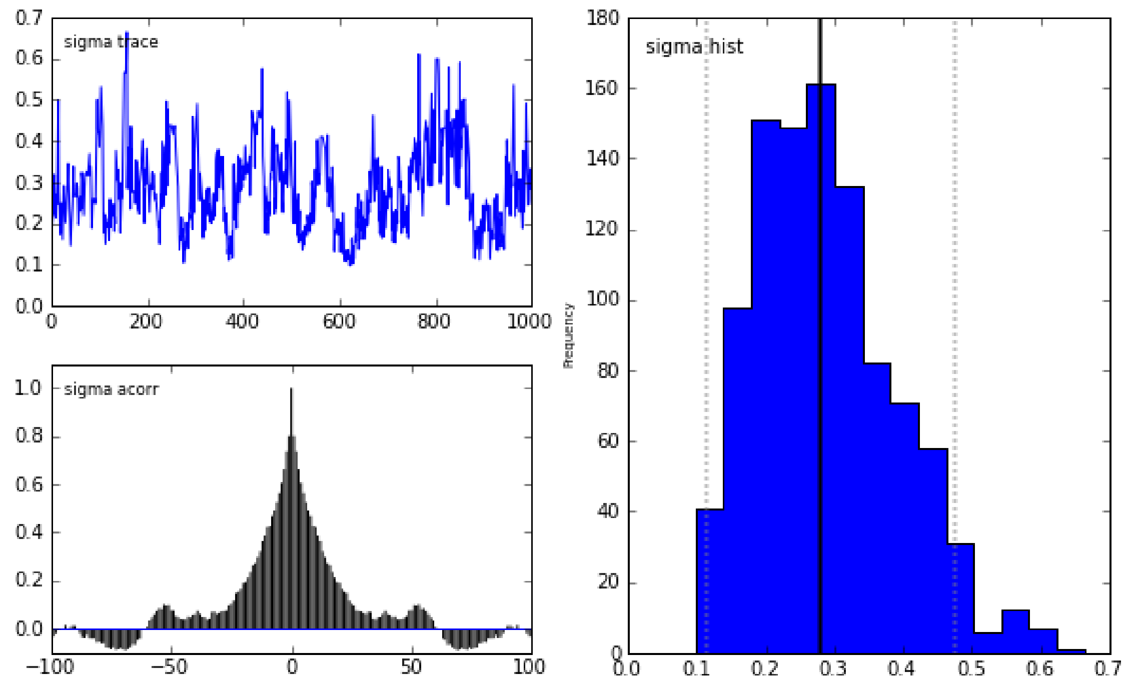
PyMC has a very sophisticated "Adaptive Metropolis" method, which adjusts the proposal distribution for the Metropolis step adaptively. Using this and running for longer provide more reliable results.

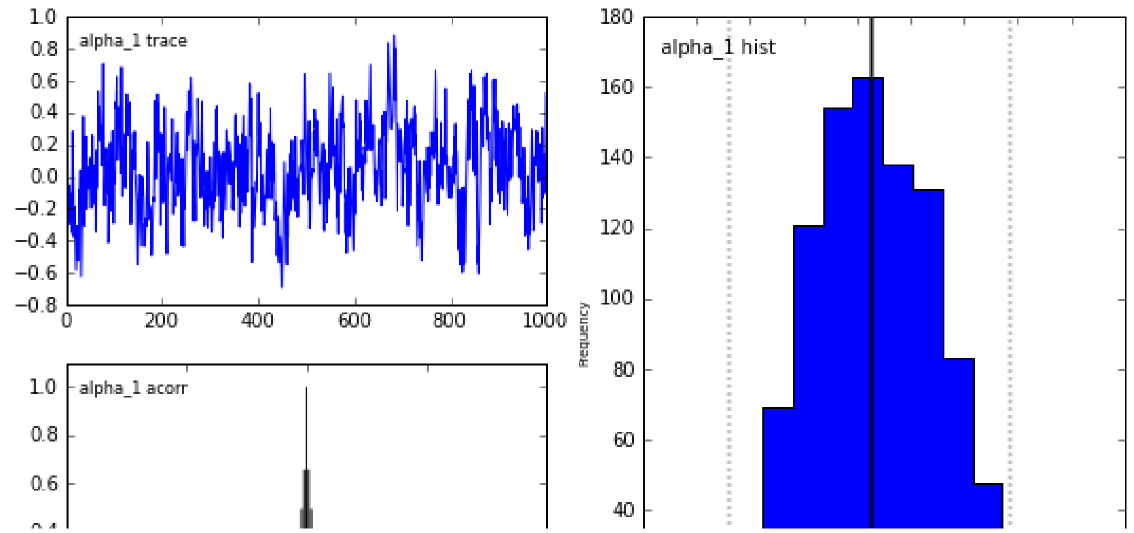
```
In [6]: m = mc.MCMC([alpha_0, alpha_1, alpha_2, alpha_12, b, tau, sigma, logit_p, obs])
        m.use_step_method(mc.AdaptiveMetropolis, b)

        # about long enough, but maybe better initial values would help more
        %time m.sample(20000, 10000, 10, progress_bar=False)
```

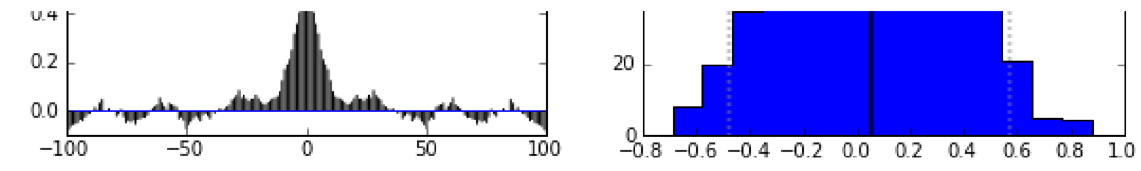
```
CPU times: user 13.35 s, sys: 0.33 s, total: 13.68 s
Wall time: 20.01 s
```

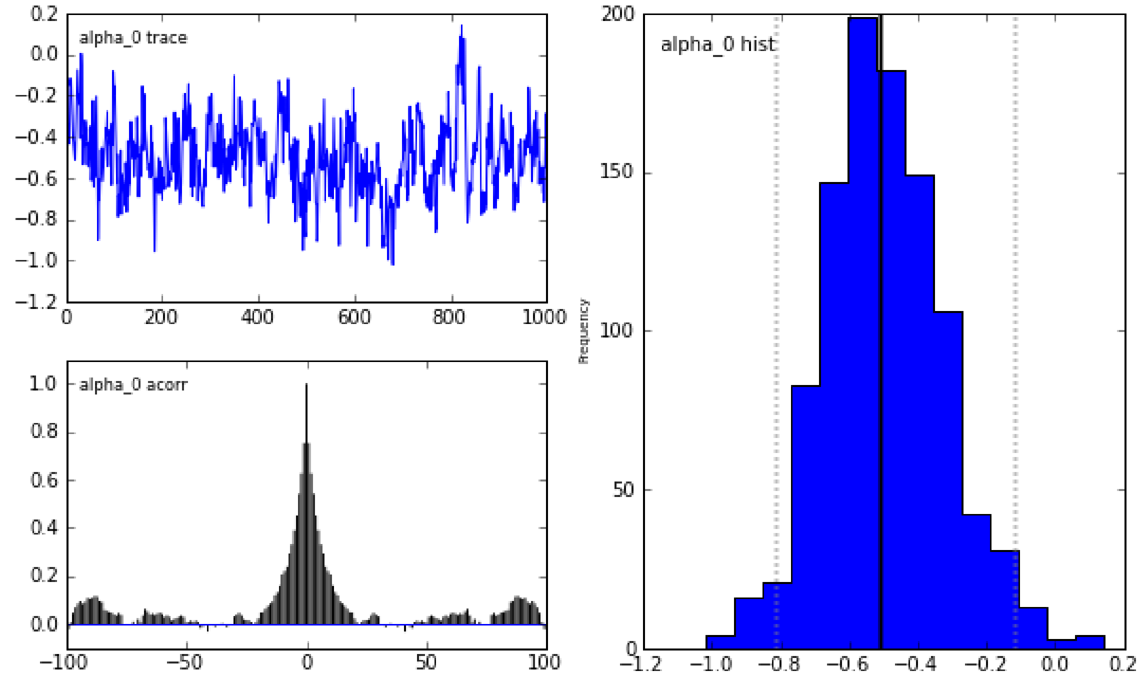
```
In [7]: mc.Matplot.plot(m)
```

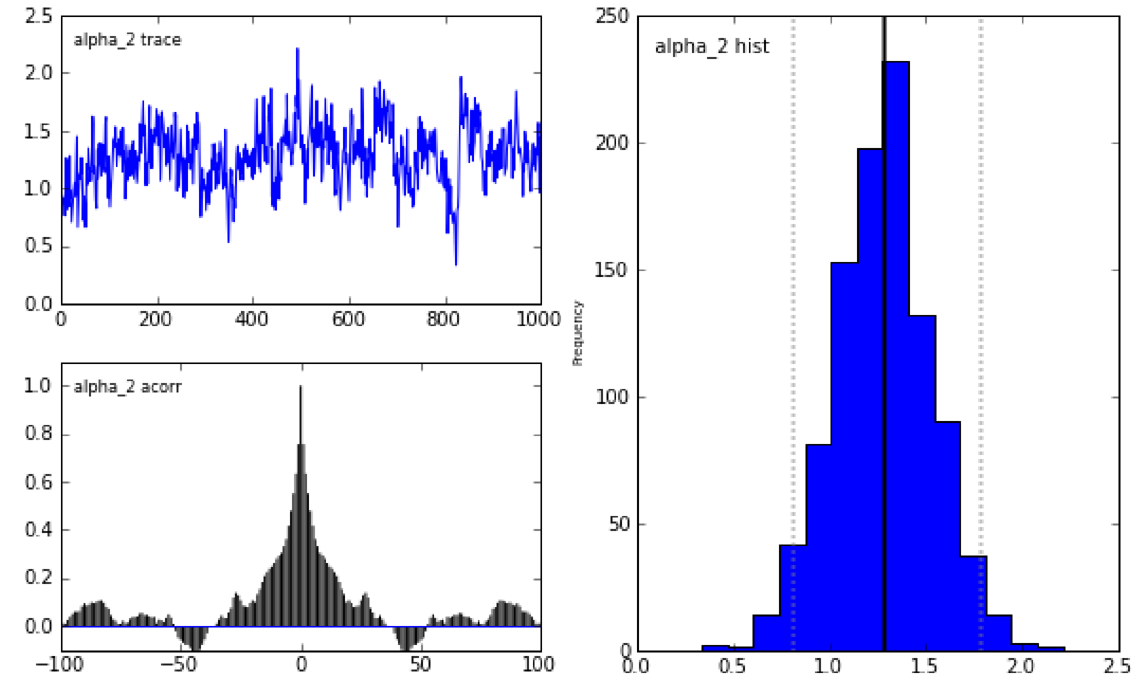
```
Plotting sigma
Plotting alpha_1
Plotting alpha_0
Plotting alpha_2
Plotting tau
Plotting alpha_12
```

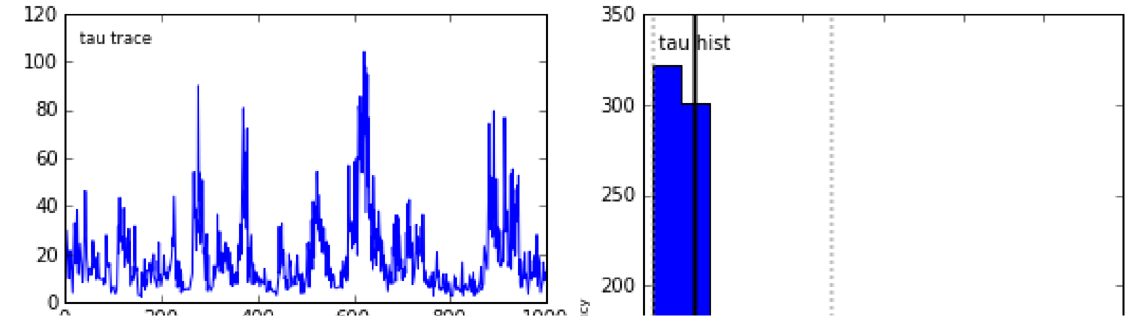


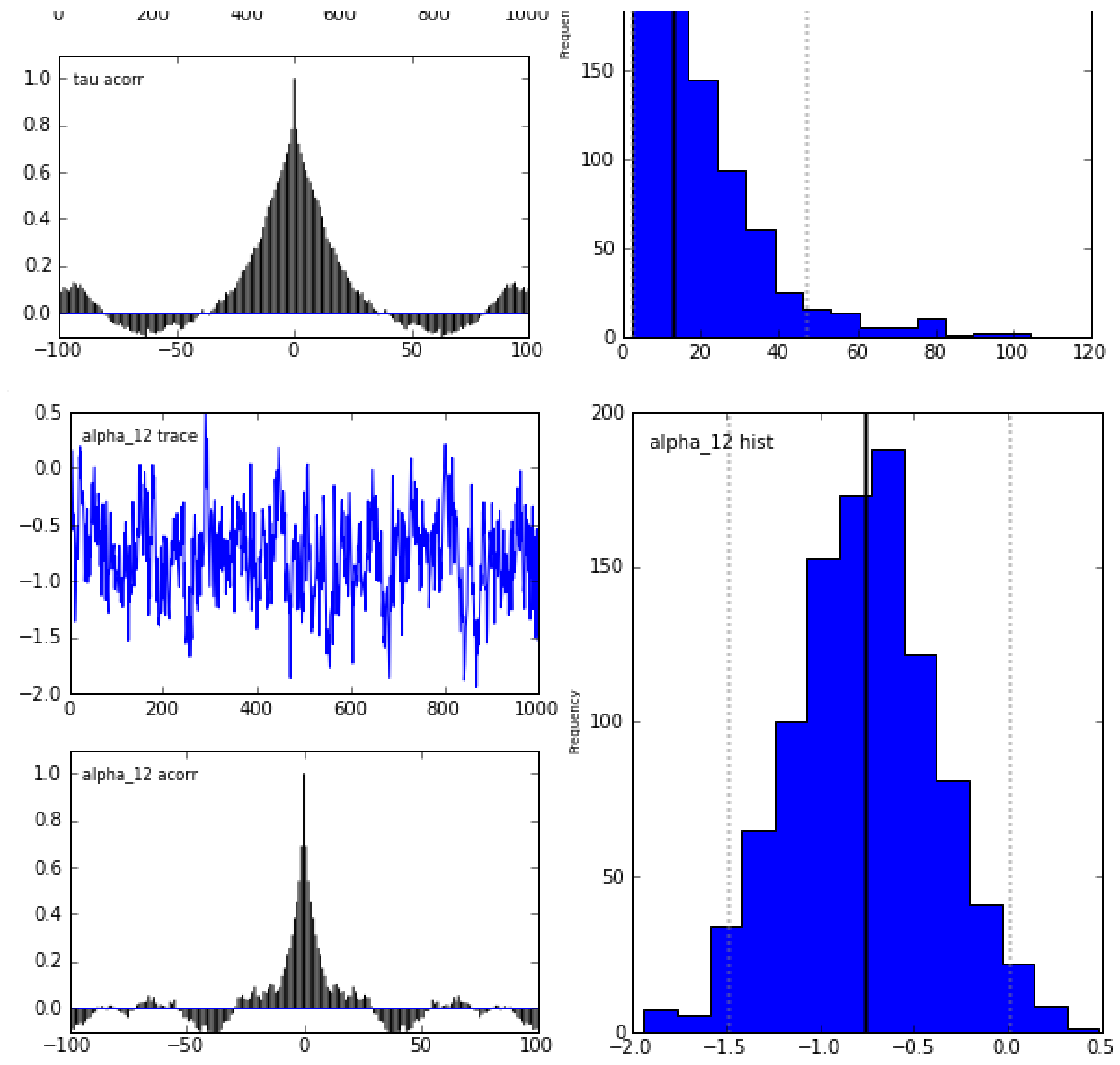





alpha_0 trace
alpha_0 acorr
alpha_0 hist
Frequency


alpha_2 trace
alpha_2 acorr
alpha_2 hist
Frequency


tau trace
tau hist



Starting from carefully chosen initial values and running the chain for even longer yields results that are completely mixed. This is going to give the same answer every time (approximately).

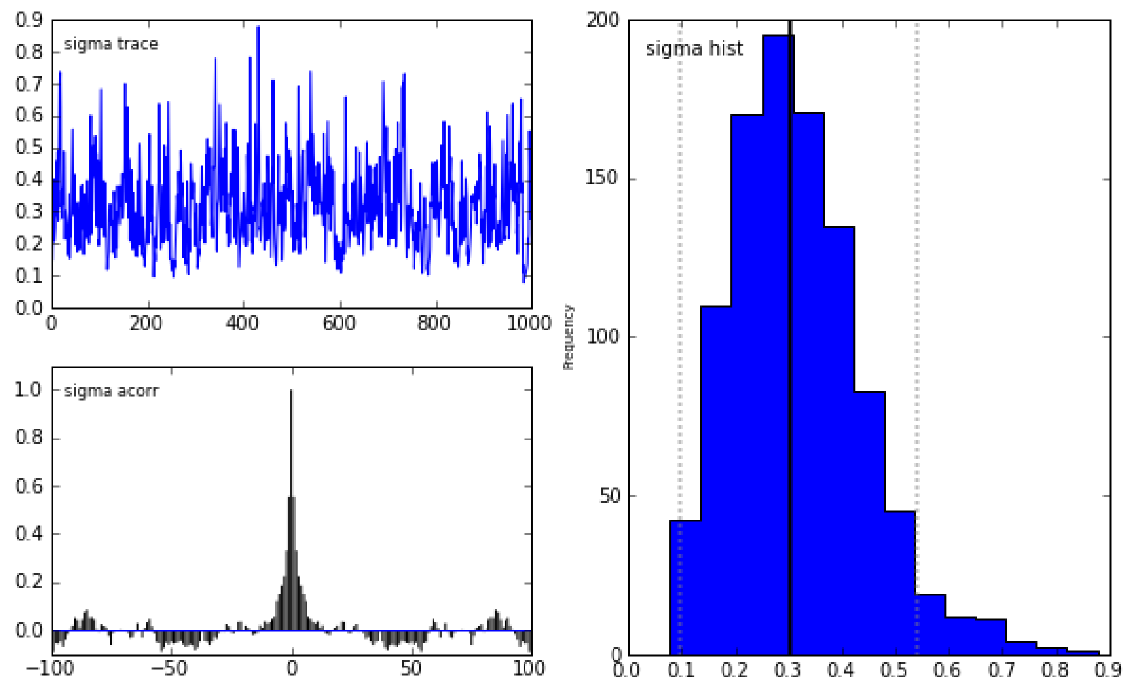
```
In [8]: tau.value=1.
        map = mc.MAP([alpha_0, alpha_1, alpha_2, alpha_12, b, logit_p, obs])
        map.fit(method='fmin_powell')


        m = mc.MCMC([alpha_0, alpha_1, alpha_2, alpha_12, b, tau, sigma, logit_p, obs])
        m.use_step_method(mc.AdaptiveMetropolis, b)

        # a little longer, good initial values, but not Adaptive Metropolis.  Does not converg
        %time m.sample(200000, 100000, 100, progress_bar=False)
```

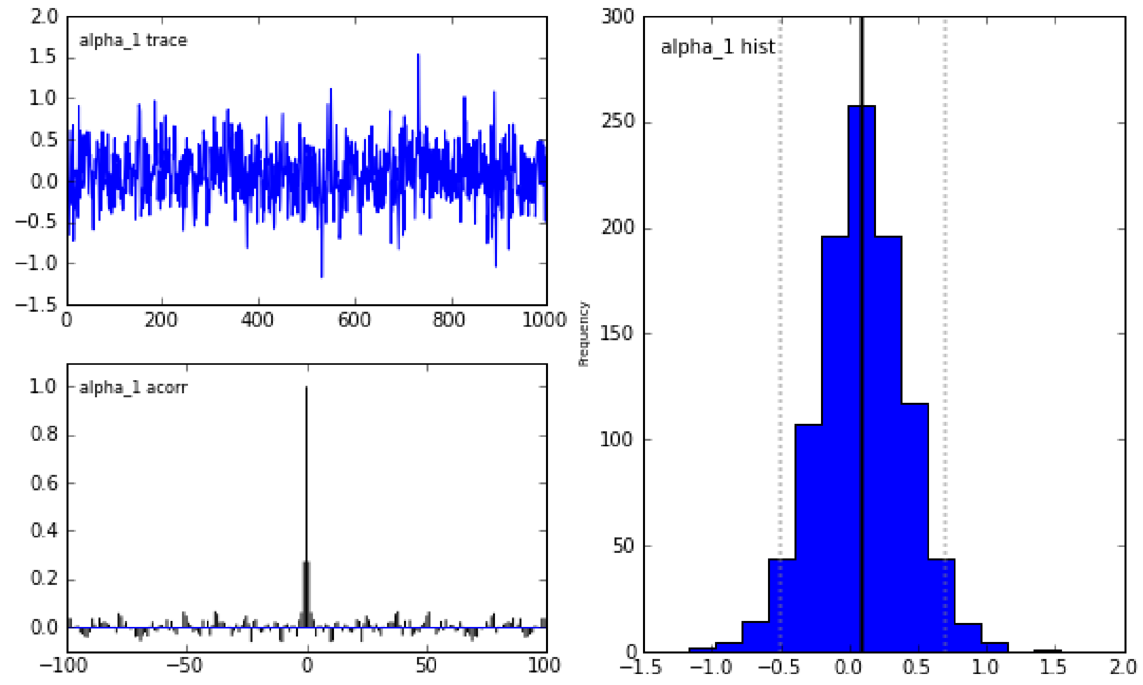
```
CPU times: user 125.07 s, sys: 2.74 s, total: 127.80 s
Wall time: 151.96 s
/home/abie/projects/env_pymc_dev/lib/python2.6/site-packages/pymc/StepMethods.py:1171:
UserWarning:
Covariance was not positive definite and proposal_sd cannot be computed by
Cholesky decomposition. The next jumps will be based on the last
valid covariance matrix. This situation may have arisen because no
jumps were accepted during the last `interval`. One solution is to
increase the interval, or specify an initial covariance matrix with
a smaller variance. For this simulation, each time a similar error
occurs, proposal_sd will be reduced by a factor .9 to reduce the
```

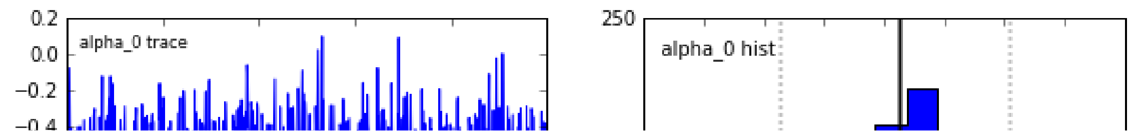
```
jumps and increase the likelihood of accepted jumps.
  warnings.warn(adjustmentwarning)
```

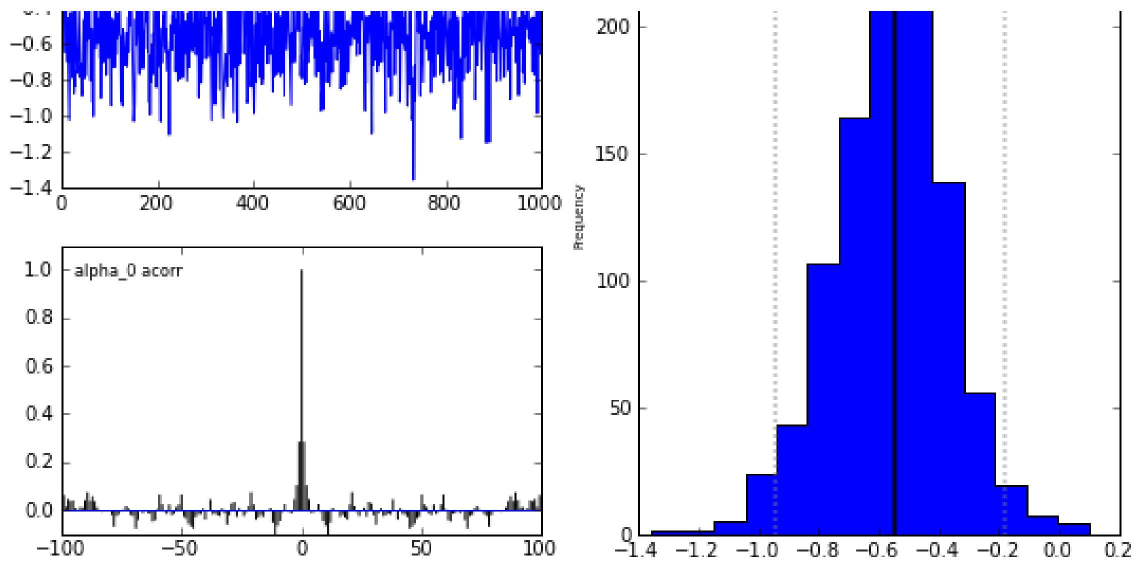
In [9]:
```
mc.Matplot.plot(m)
```

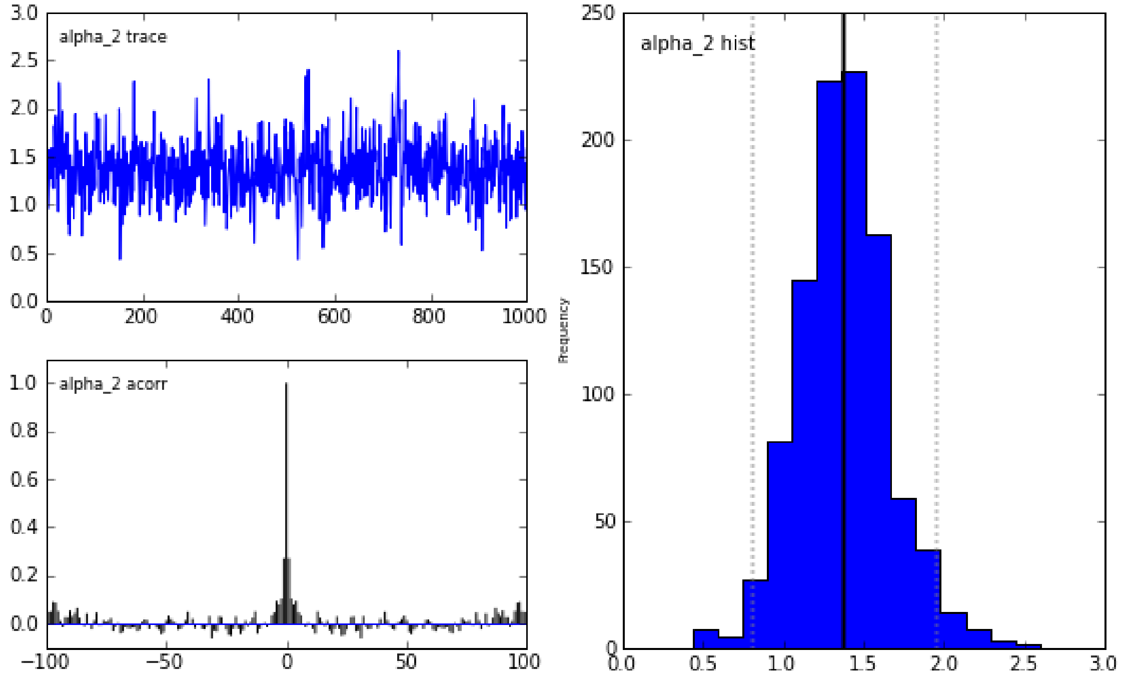
```
Plotting sigma
Plotting alpha_1
Plotting alpha_0
Plotting alpha_2
Plotting tau
Plotting alpha_12
```

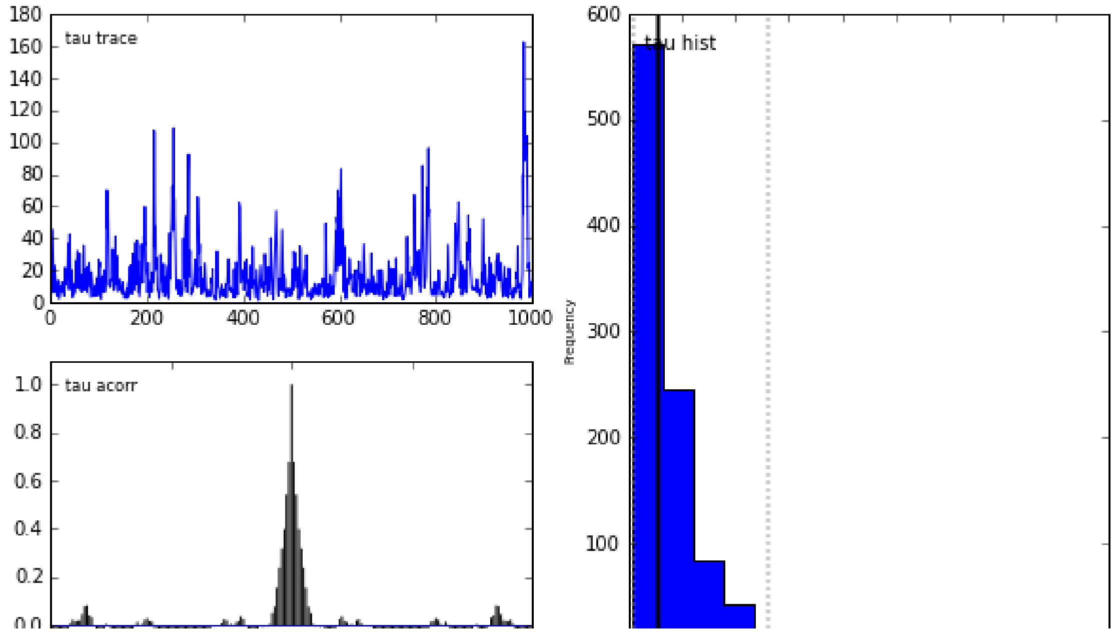


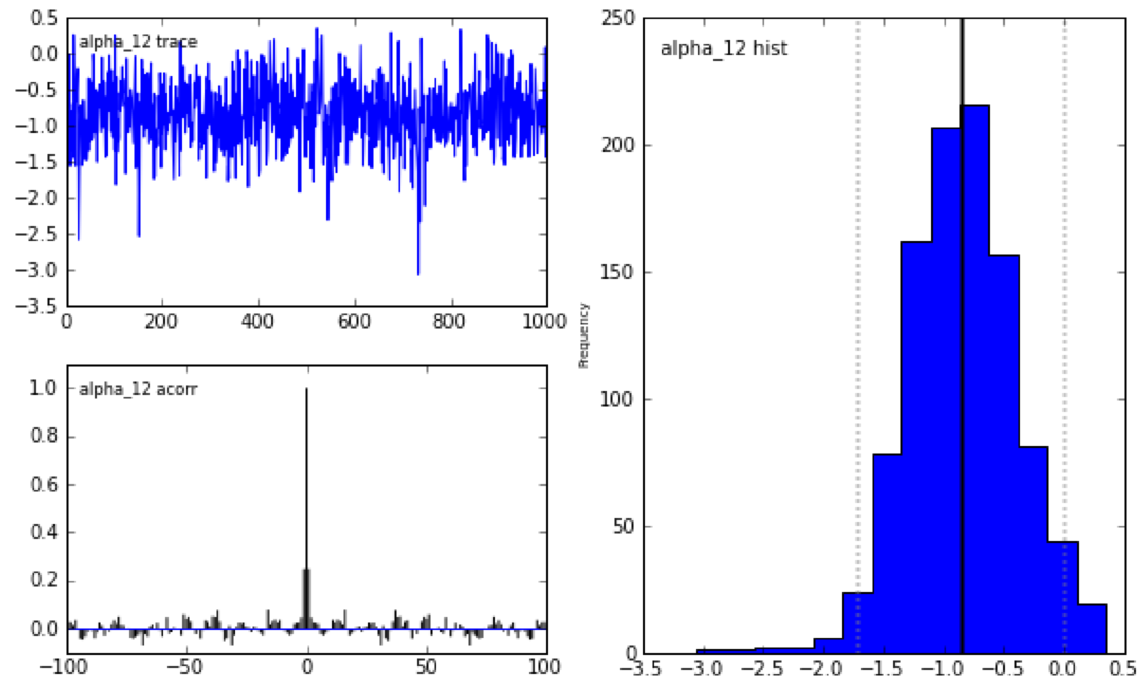





alpha_0 acorr
Frequency


alpha_2 trace
alpha_2 acorr
alpha_2 hist
Frequency


tau trace
tau acorr
tau hist
Frequency



Results

BUGS results:

A burn in of 1000 updates followed by a further 10000 updates gave the following parameter estimates:

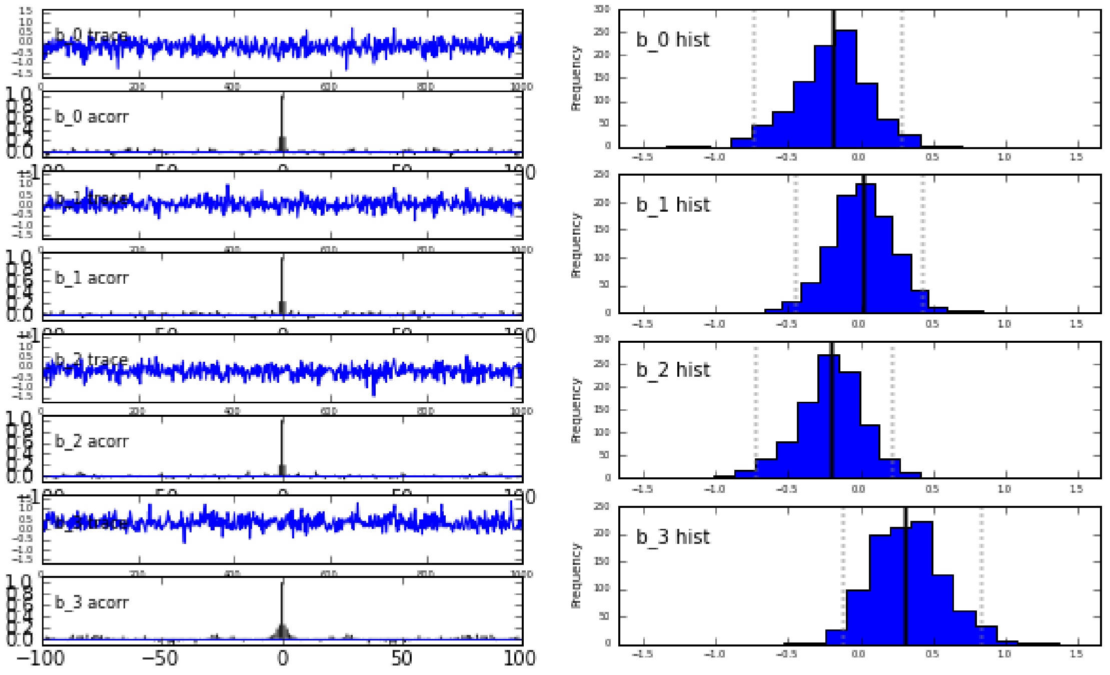
```
                 mean     sd
    alpha_0     -0.55    0.19
    alpha_1      0.08    0.30
    alpha_12    -0.82    0.41
    alpha_2      1.35    0.26
    sigma        0.27    0.15
```

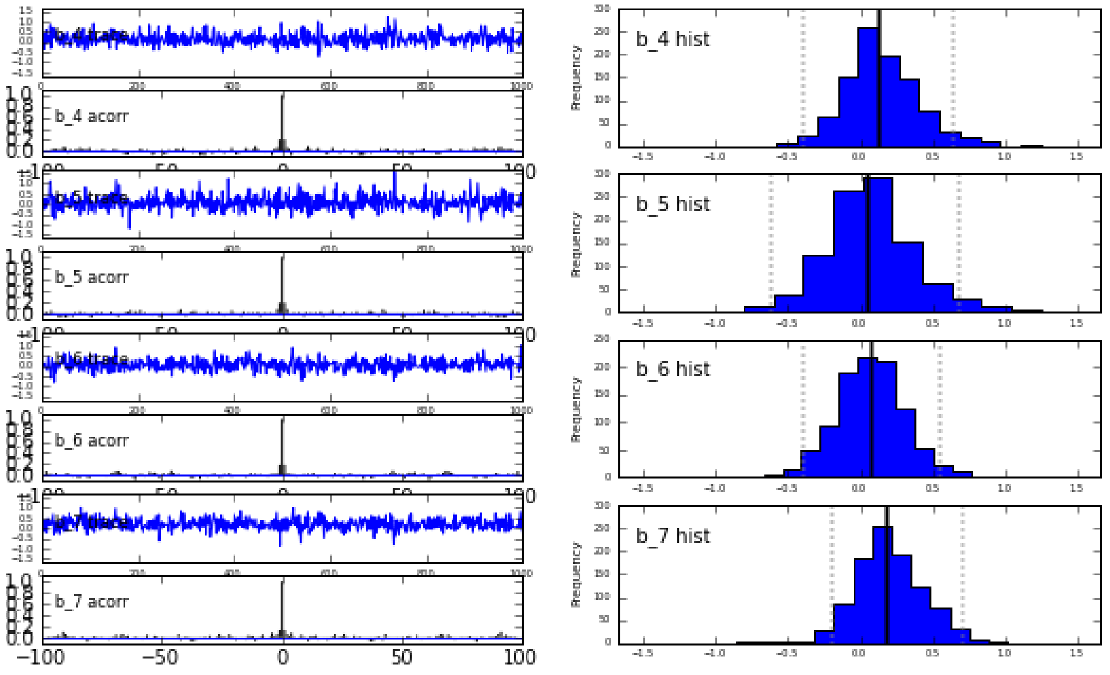
In [10]:
```
for node in [alpha_0, alpha_1, alpha_12, alpha_2, sigma]:
    stats = node.stats()
    print '%10s\t%1.2f \t%.2f' % (node.__name__, stats['mean'], stats['standard deviat
```

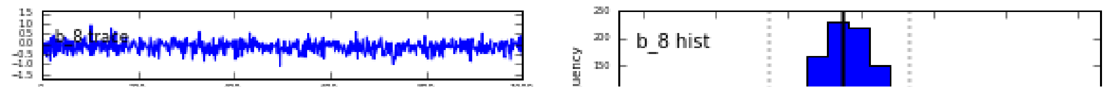
```
   alpha_0      -0.56    0.19
   alpha_1      0.08     0.32
  alpha_12      -0.85    0.45
   alpha_2      1.37     0.29
     sigma      0.32     0.12
```

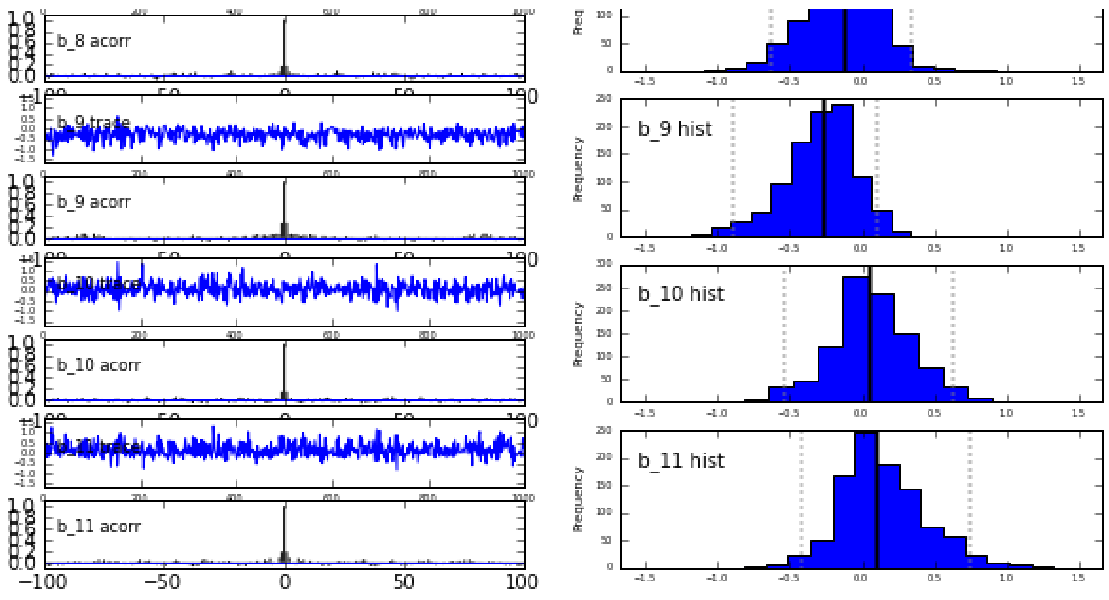
In [11]:
```
mc.Matplot.plot(b)
```

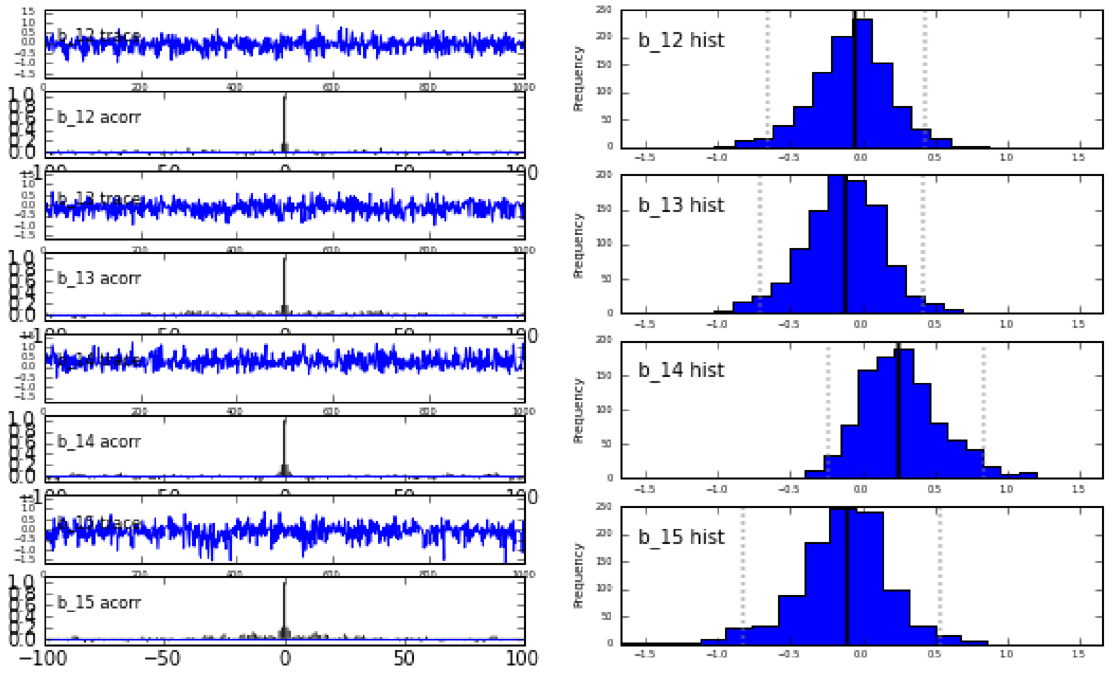
```
Plotting b_0
Plotting b_1
Plotting b_2
Plotting b_3
Plotting b_4
Plotting b_5
Plotting b_6
Plotting b_7
Plotting b_8
Plotting b_9
Plotting b_10
Plotting b_11
Plotting b_12
Plotting b_13
Plotting b_14
Plotting b_15
Plotting b_16
Plotting b_17
Plotting b_18
Plotting b_19
Plotting b_20
```

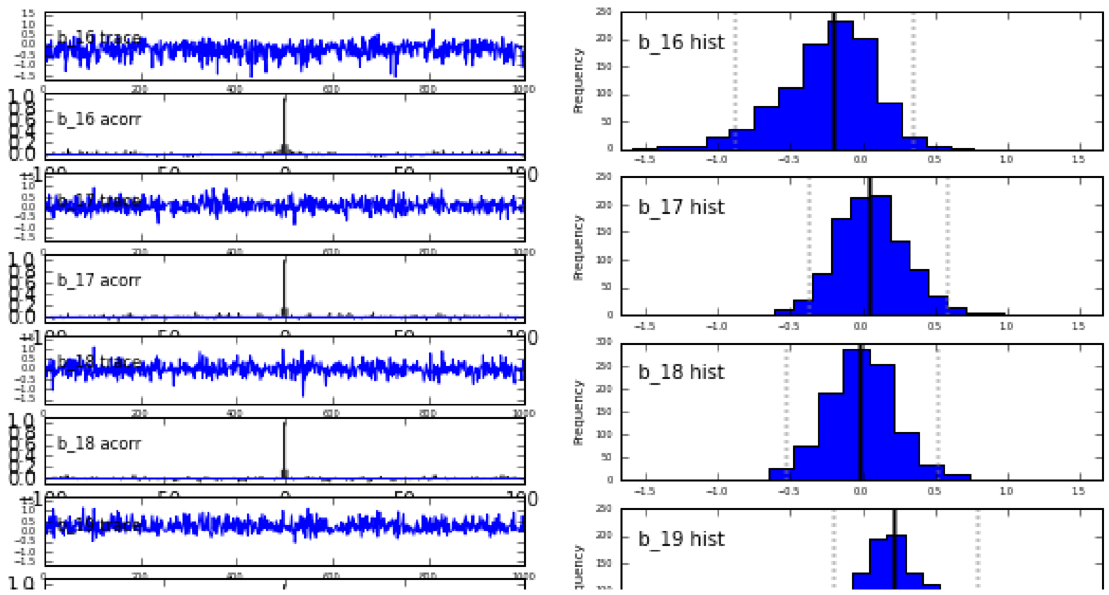


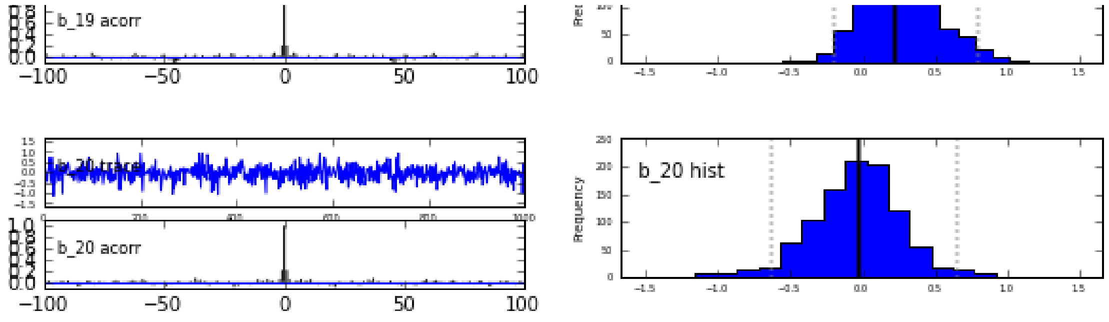





b_8 acorr
b_9 trace
b_9 hist
b_9 acorr
b_10 trace
b_10 hist
b_10 acorr
b_11 trace
b_11 hist
b_11 acorr
Frequency


b_12 trace
b_12 hist
b_12 acorr
b_13 trace
b_13 hist
b_13 acorr
b_14 trace
b_14 hist
b_14 acorr
b_15 trace
b_15 hist
b_15 acorr
Frequency


b_16 trace
b_16 hist
b_16 acorr
b_17 trace
b_17 hist
b_17 acorr
b_18 trace
b_18 hist
b_18 acorr
b_19 trace
b_19 hist
Frequency



Further exploration

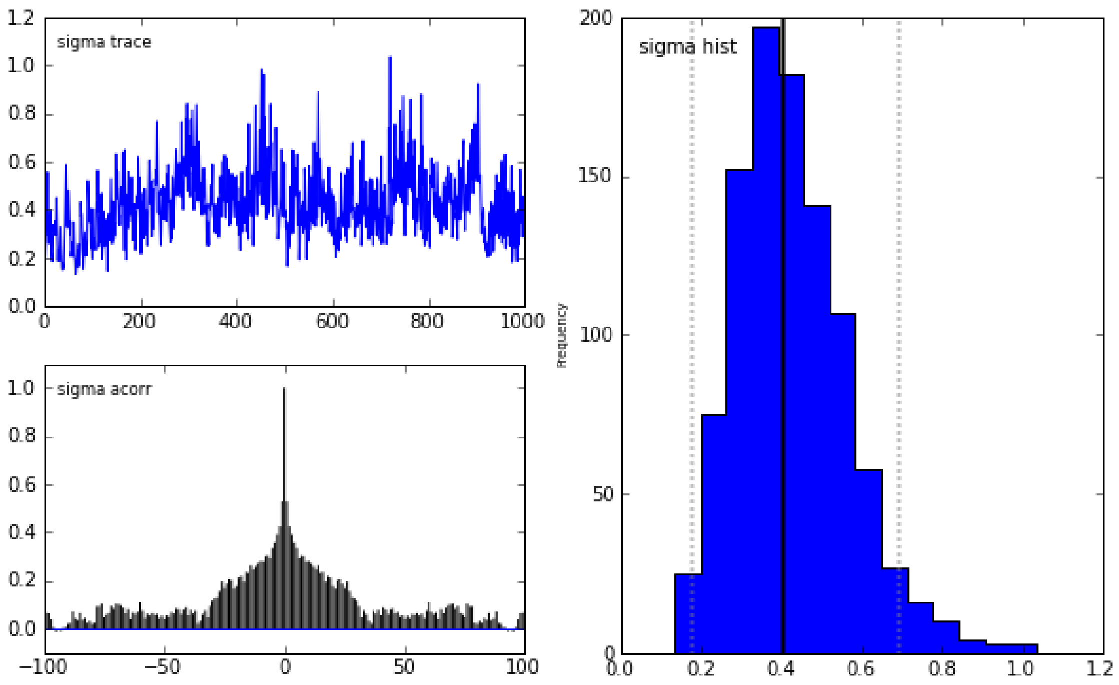
Now that I've seen that it can work, how necessary is the Adaptive Metropolis? Necessary.

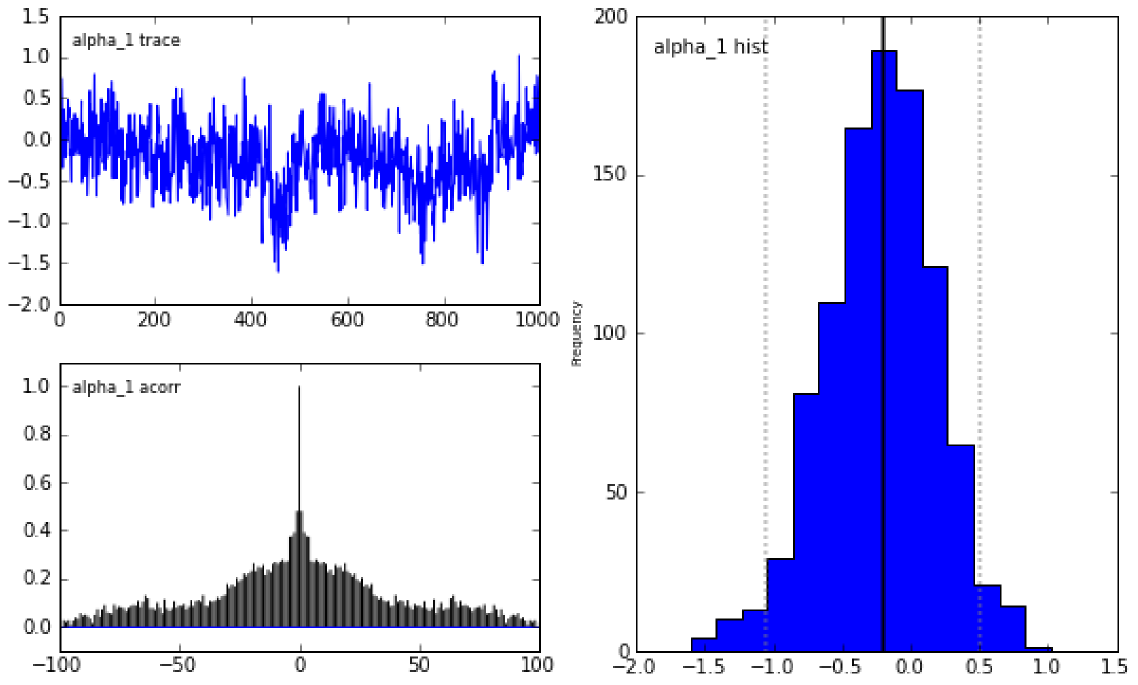
In [12]:
```
tau.value=1.
map = mc.MAP([alpha_0, alpha_1, alpha_2, alpha_12, b, logit_p, obs])
map.fit(method='fmin_powell')


m = mc.MCMC([alpha_0, alpha_1, alpha_2, alpha_12, b, tau, sigma, logit_p, obs])

%time m.sample(200000, 100000, 100, progress_bar=False)
```

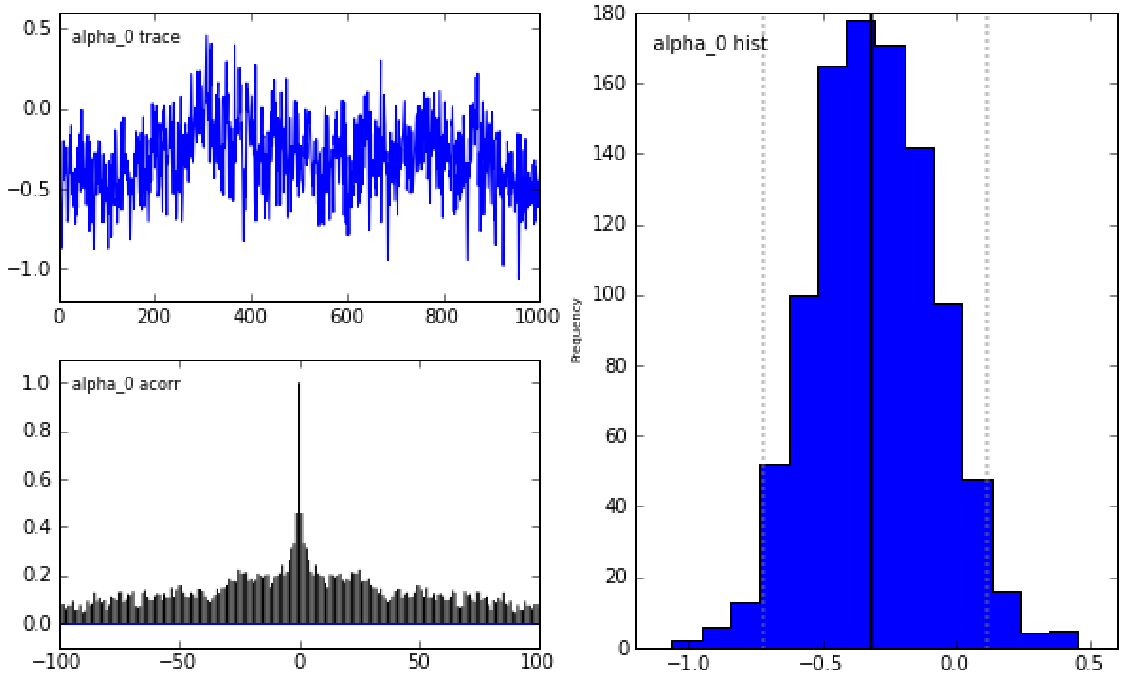
```
CPU times: user 122.45 s, sys: 0.70 s, total: 123.15 s
Wall time: 127.73 s
```

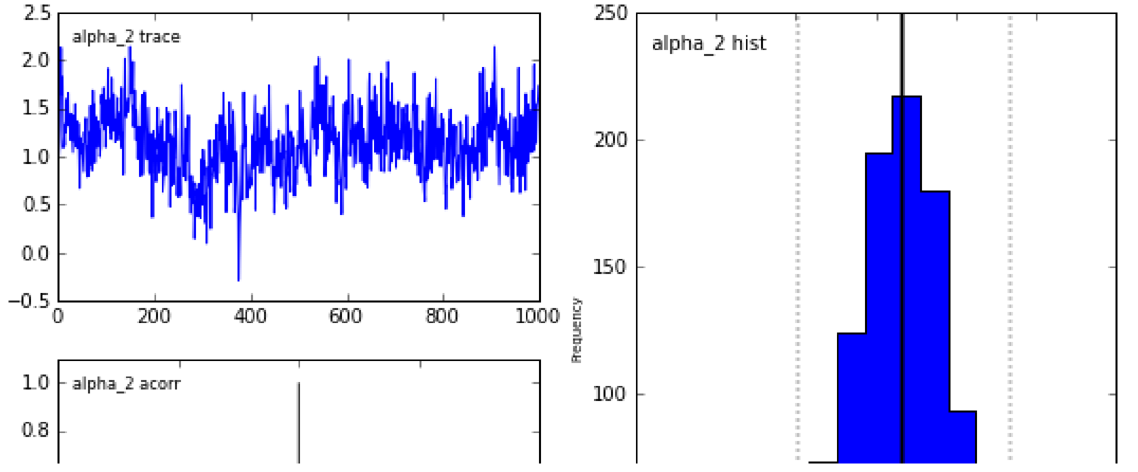
In [13]:
```
mc.Matplot.plot(m)
```

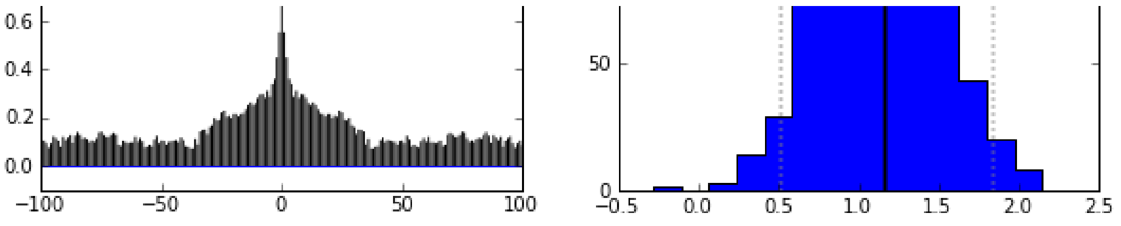
```
Plotting sigma
Plotting alpha_1
Plotting alpha_0
Plotting alpha_2
Plotting tau
Plotting alpha_12
```

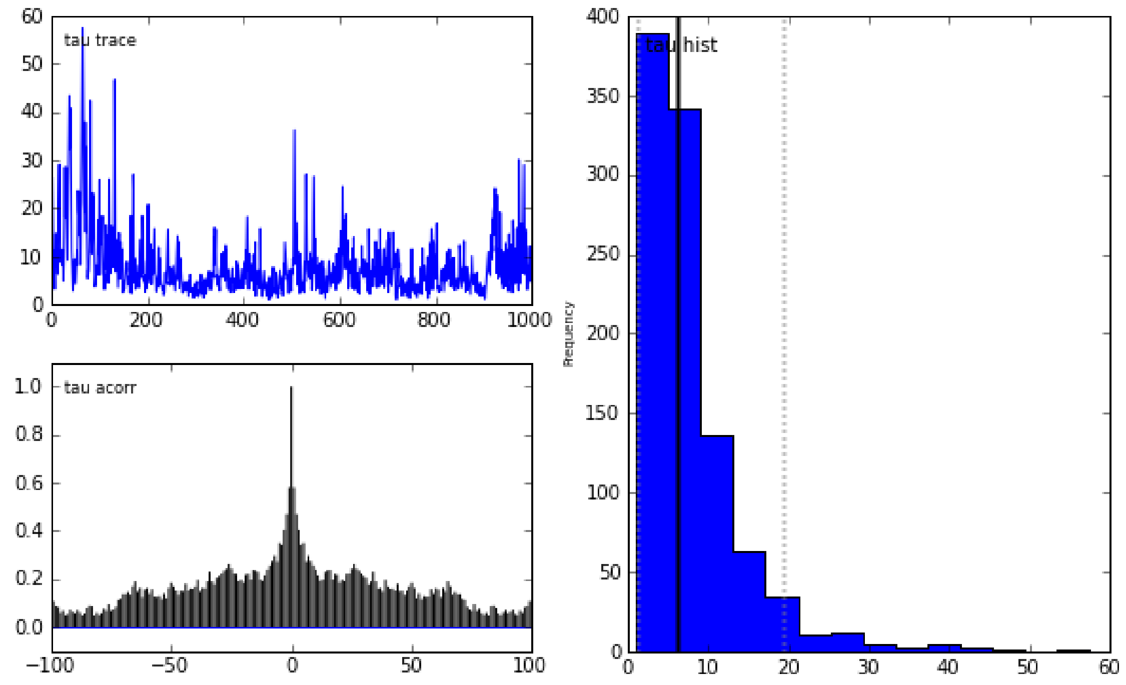


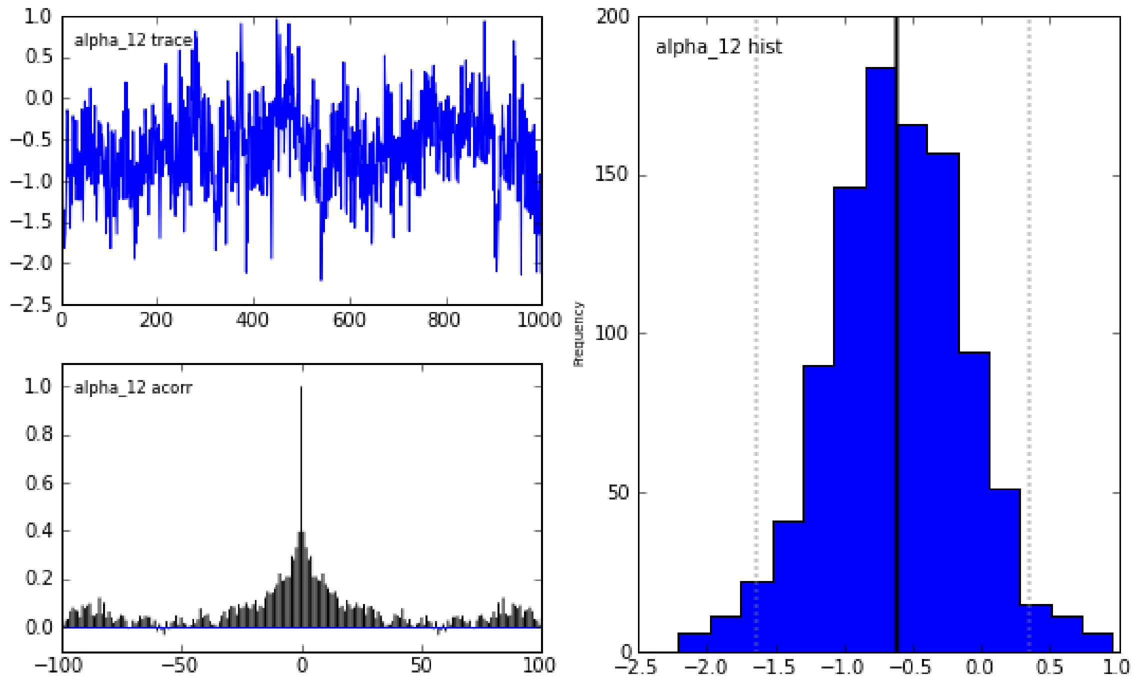

alpha_1 trace
alpha_1 acorr
alpha_1 hist
Frequency


alpha_0 trace
alpha_0 acorr
alpha_0 hist
Frequency


alpha_2 trace
alpha_2 acorr
alpha_2 hist
Frequency







And how necessary are the carefully chosen initial values?

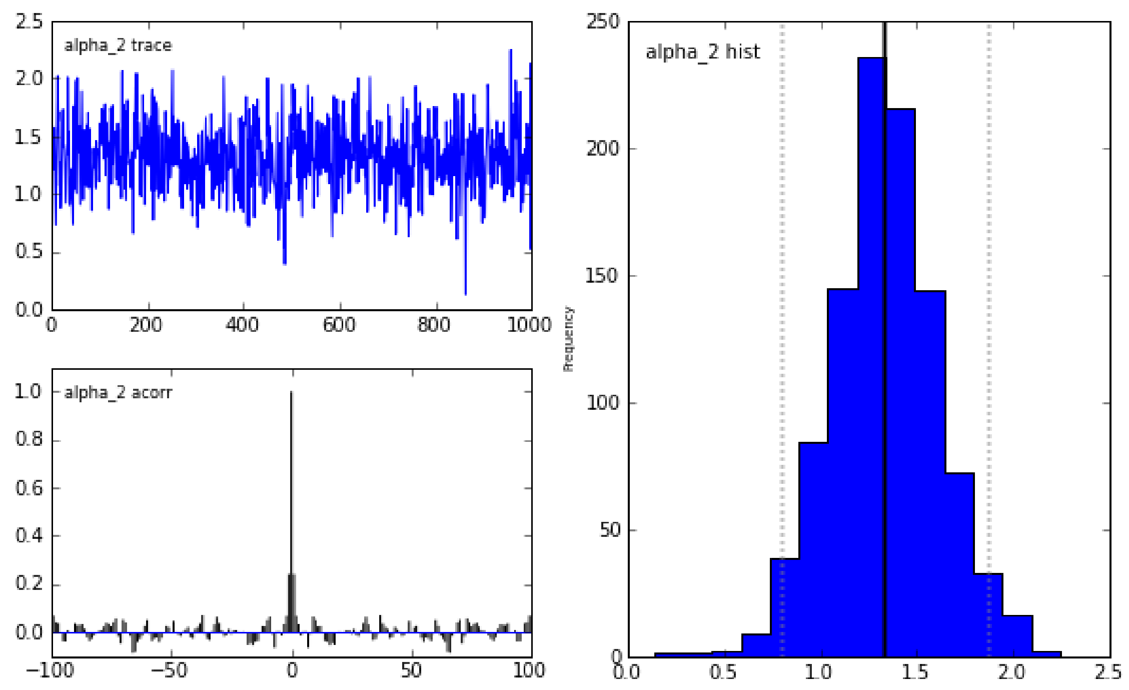
```
In [14]: ### hyperpriors
         tau = mc.Gamma('tau', 1.e-3, 1.e-3, value=10.)
         sigma = mc.Lambda('sigma', lambda tau=tau: tau**-.5)

         ### parameters
```

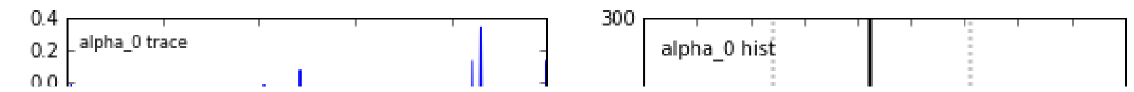
```
# fixed effects
alpha_0 =  mc.Normal('alpha_0',  0., 1e-6, value=0.)
alpha_1 =  mc.Normal('alpha_1',  0., 1e-6, value=0.)
alpha_2 =  mc.Normal('alpha_2',  0., 1e-6, value=0.)
alpha_12 = mc.Normal('alpha_12', 0., 1e-6, value=0.)

# random effect
b =        mc.Normal('b',        0., tau,  value=np.zeros(N))

# expected parameter
logit_p =  (alpha_0 + alpha_1*x1 + alpha_2*x2 + alpha_12*x1*x2 + b)


### likelihood
@mc.observed
def obs(value=r, n=n, logit_p=logit_p):
    return mc.binomial_like(r, n, mc.invlogit(logit_p))

m = mc.MCMC([alpha_0, alpha_1, alpha_2, alpha_12, b, tau, sigma, logit_p, obs])
m.use_step_method(mc.AdaptiveMetropolis, b)

# a little longer, but not special initial values, does it converge?
%time m.sample(200000, 100000, 100, progress_bar=False)
```

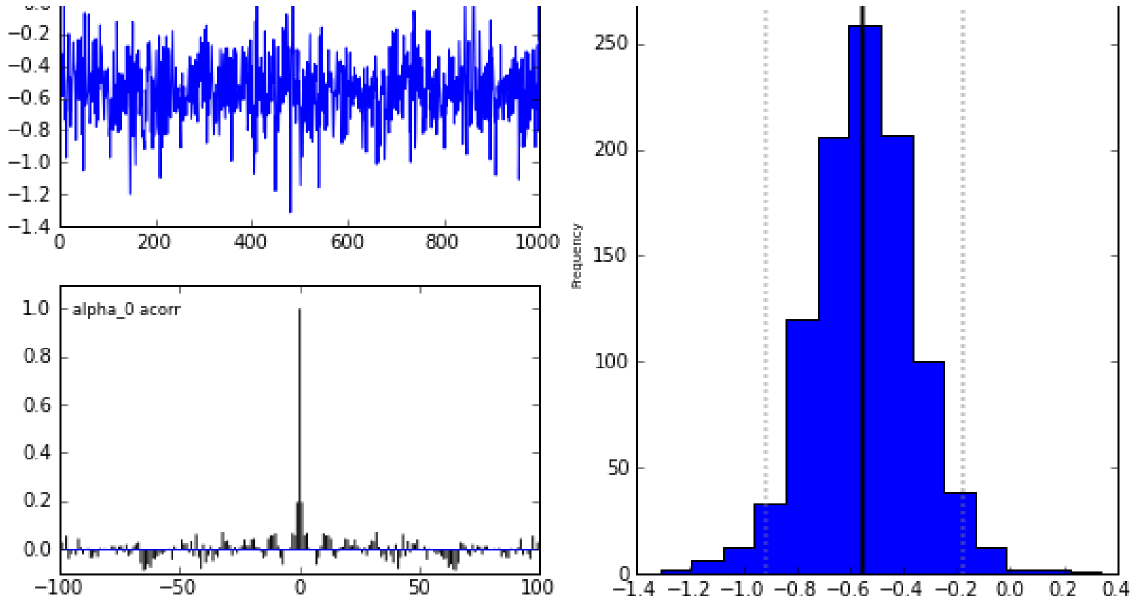
```
CPU times: user 123.14 s, sys: 0.64 s, total: 123.77 s
Wall time: 128.74 s
```

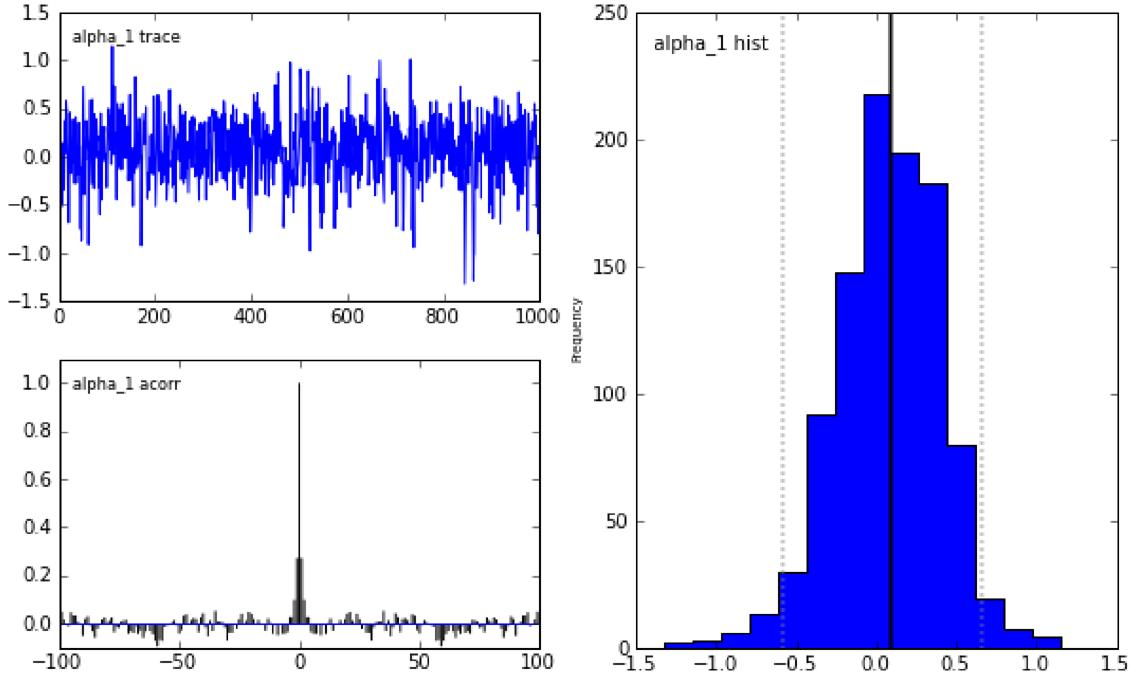
In [15]:
```
mc.Matplot.plot(m)
```

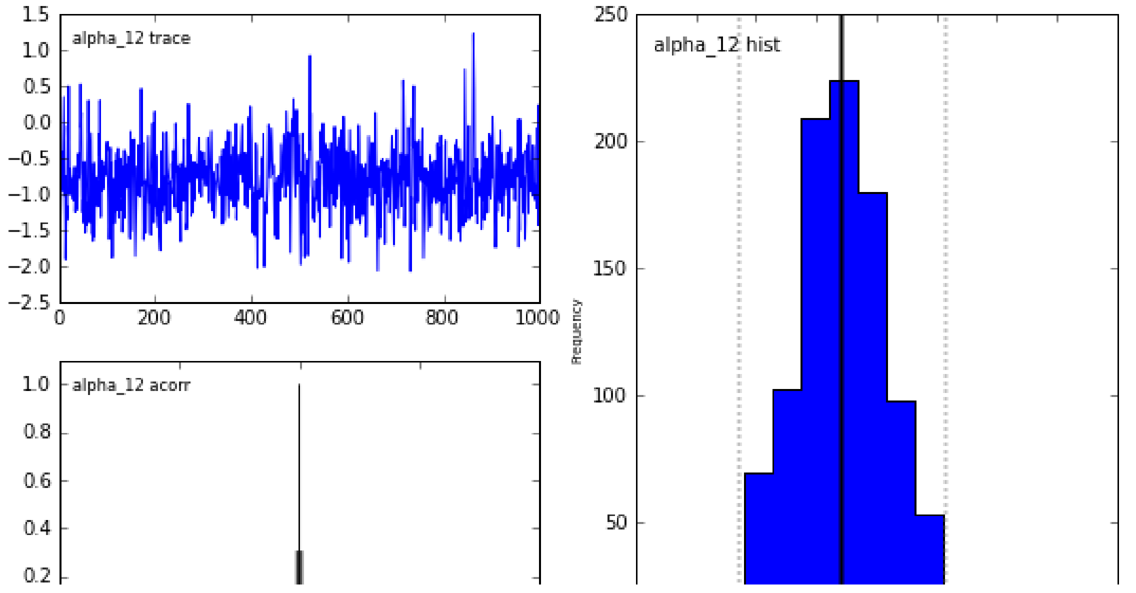
```
Plotting alpha_2
Plotting alpha_0
Plotting alpha_1
Plotting alpha_12
Plotting sigma
Plotting tau
```

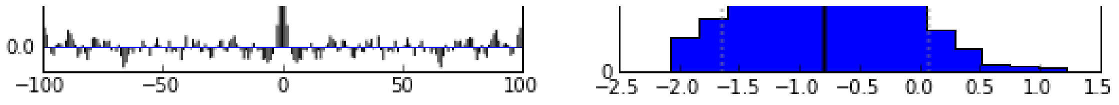


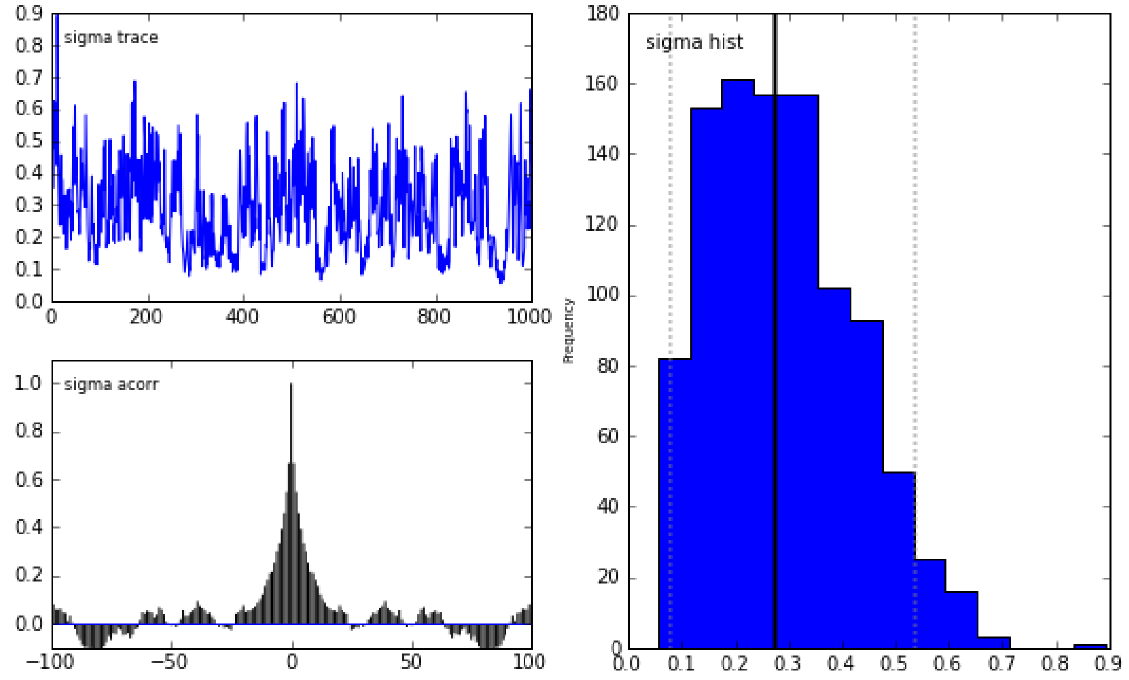


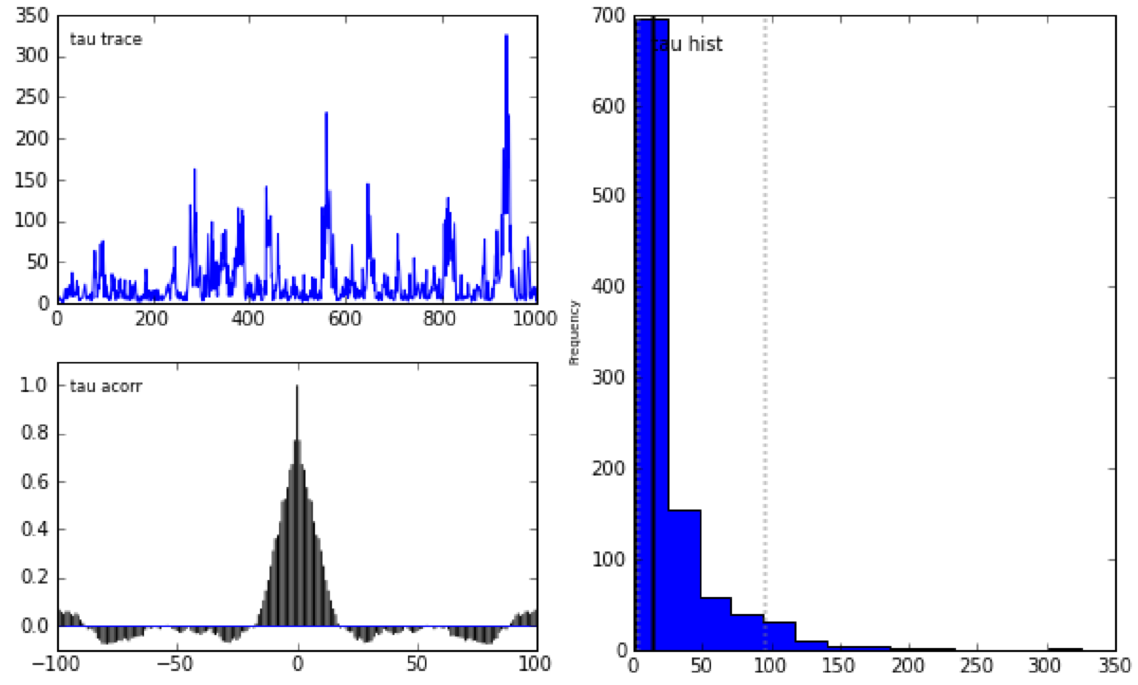

alpha_0 acorr
Frequency


alpha_1 trace
alpha_1 acorr
alpha_1 hist
Frequency


alpha_12 trace
alpha_12 acorr
alpha_12 hist
Frequency




sigma trace
sigma acorr
sigma hist
Frequency


tau trace
tau acorr
tau hist
Frequency

```
In [ ]:
```