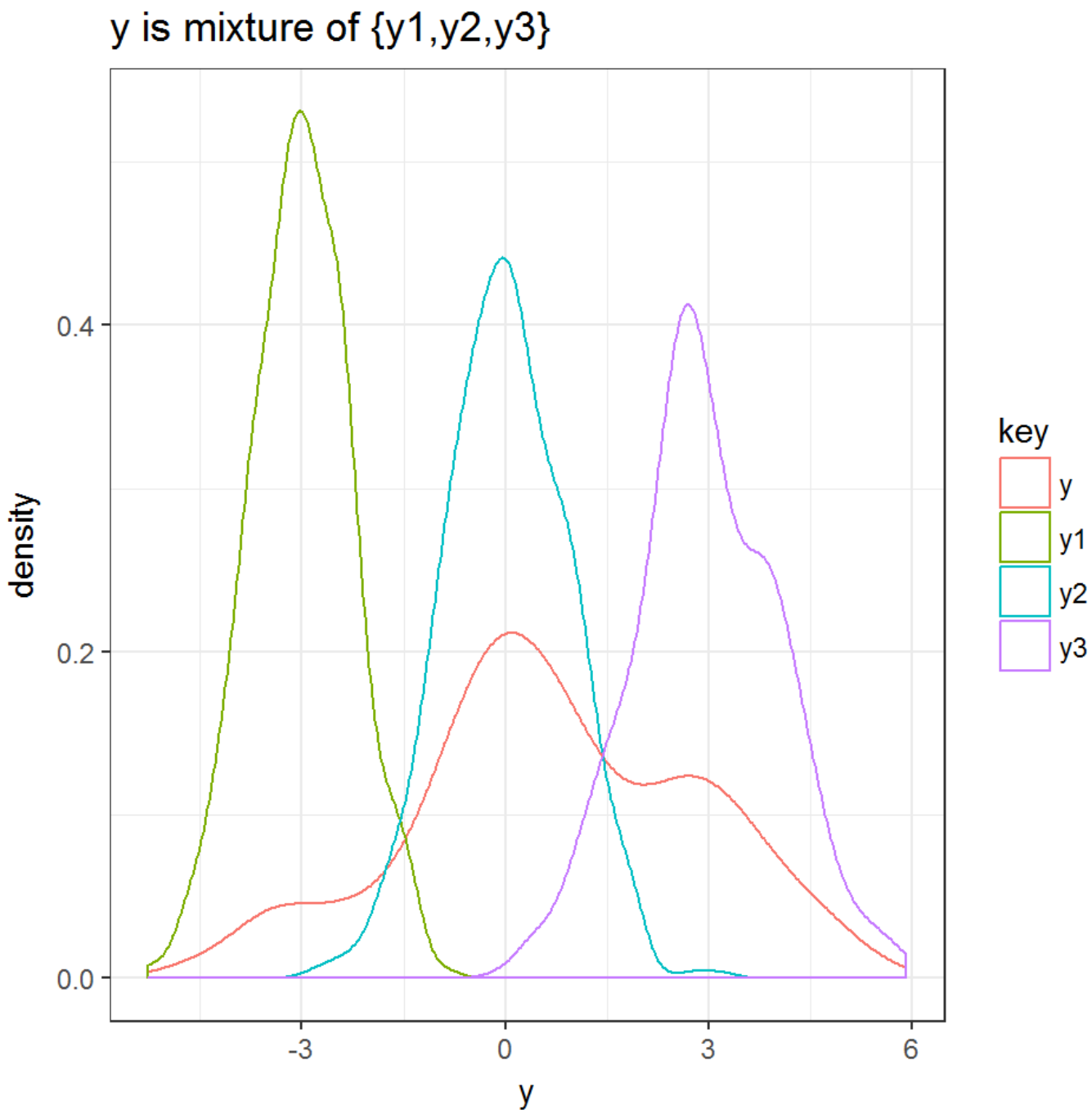
HMC for a Simple Dirichlet Process Model via Stan

In [1]:

```
import pystan
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
```

We will mimic Dirichlet process based Gaussian mixture model using Stan. Since Stan doesn't provide the Dirichlet Process prior, we mimic it via finite mixture model. The stick-breaking process is achived inside of the stan code.

The data y is a mixture of y_1, y_2 and y_3, where $y_1 \sim \mathcal{N}(-3, 0.5^2)$, $y_2 \sim \mathcal{N}(0, 0.75^2)$, and $y_3 \sim \mathcal{N}(3, 1^2)$, and the mixing rate is $\pi = (0.1, 0.5, 0.4)$.



Write Model in Stan

A Stan model requires at least three blocks, for each of data, parameters, and the model. The data block specifies the types and dimensions of the data that will be used for sampling, and the parameter block specifies the relevant parameters. The distribution statement goes in the model block.

If no prior is defined, Stan uses default priors with the specifications uniform(-infinity, +infinity). You can restrict priors using upper or lower when declaring the parameters (i.e. lower = 0> to make sure a parameter is positive).

In [2]:

```
# simple DP example, via Truncated Stick-breaking process

model = """
data{
  int<lower=0> C;//maximum num of cludter
  int<lower=0> N;//data num
  real y[N];
}

parameters {
  real mu_cl[C]; //cluster mean
  real <lower=0,upper=1> v[C];
  real<lower=0> sigma_cl[C]; // error scale
  real<lower=0> alpha; // hyper prior DP(alpha,base)
}

transformed parameters{
  simplex [C] pi;
  pi[1] = v[1];

  for(j in 2:(C-1)){
      pi[j]= v[j]*(1-v[j-1])*pi[j-1]/v[j-1];
  }
  pi[C]=1-sum(pi[1:(C-1)]); // to make a simplex.
}

model {
  real a = 1.0;
  real b = 1.0;
  real ps[C];
  sigma_cl ~ inv_gamma(a,b);
  mu_cl ~ normal(0,10);
  alpha ~ gamma(6,1);
  v ~ beta(1,alpha);

  for(i in 1:N){
    for(c in 1:C){
      ps[c]=log(pi[c])+normal_lpdf(y[i]|mu_cl[c],sigma_cl[c]);
    }
    target += log_sum_exp(ps);
  }

}
"""
```

HMC Sampling

In [3]:

```
# Read in data
data = np.loadtxt('dat.dat',delimiter=' ',skiprows=1)

C = 10 # truncation point for stick-breaking process
y = data[:,1]
print(len(y))

# Put data in a dictionary
stanData = {'C': C, 'N': len(y), 'y': y}
```

5000

In [4]:

```
# Compile the model
sm = pystan.StanModel(model_code=model)

# Train the model and generate samples
fit = sm.sampling(data=stanData, iter=1000, chains=1, warmup=500, thin=1, seed=101)

print("Training finished ...")
```

```
INFO:pystan:COMPILING THE C++ CODE FOR MODEL anon_model_ea5cd205a19b4eb290f029dcf59d
07fb NOW.
WARNING:pystan:Rhat above 1.1 or below 0.9 indicates that the chains very likely hav
e not mixed
WARNING:pystan:500 of 500 iterations saturated the maximum tree depth of 10 (100 %)
WARNING:pystan:Run again with max_treedepth larger than 10 to avoid saturation

Training finished ...
```

Results

In [5]:

```
print(fit)
```

```
Inference for Stan model: anon_model_ea5cd205a19b4eb290f029dcf59d07fb.
1 chains, each with iter=1000; warmup=500; thin=1;
post-warmup draws per chain=500, total post-warmup draws=500.

                mean se_mean     sd   2.5%    25%    50%    75%  97.5%  n_eff   Rhat
mu_cl[1]         2.0  3.5e-7 7.9e-6    2.0    2.0    2.0    2.0    2.0    515    1.0
mu_cl[2]        0.21    0.26   0.58  -0.94  -0.17   0.33   0.67   1.08      5   1.21
mu_cl[3]         3.0  5.4e-7 1.2e-5    3.0    3.0    3.0    3.0    3.0    478    1.0
mu_cl[4]         1.0  3.7e-6 7.9e-5    1.0    1.0    1.0    1.0    1.0    464    1.0
mu_cl[5]       -1.56    0.76   1.25  -3.56  -2.53   -1.6  -0.69   0.61      3   2.64
mu_cl[6]       -0.82    0.23   0.76  -2.06  -1.42  -0.88  -0.15   0.62     11   1.18
mu_cl[7]       -1.56    0.21   0.55  -2.65  -1.92  -1.59  -1.18   -0.6      6    1.0
mu_cl[8]        1.89    0.33   0.79   0.62   1.21   1.88   2.44   3.54      6   1.52
mu_cl[9]        0.56    0.14   0.37  -0.22   0.31   0.54   0.86   1.19      7   1.02
mu_cl[10]       2.22    0.18   0.36   1.62   1.96   2.18   2.37   3.03      4    1.8
v[1]            0.51  6.4e-4 6.9e-3    0.5   0.51   0.51   0.52   0.53    114   1.01
v[2]          5.1e-4  9.5e-5 4.9e-4 1.9e-5 1.5e-4 4.1e-4 7.0e-4 1.7e-3     27    1.1
v[3]            0.78  8.0e-4 8.6e-3   0.76   0.77   0.78   0.79    0.8    116    1.0
v[4]             1.0  1.8e-4 1.8e-3   0.99    1.0    1.0    1.0    1.0    102   1.03
v[5]             0.3    0.04   0.08    0.1   0.23   0.33   0.36   0.41      4   1.41
v[6]            0.76    0.01   0.03   0.71   0.74   0.76   0.78   0.82      6   1.48
v[7]            0.87    0.05    0.1   0.64   0.82   0.89   0.96   0.97      4   1.49
v[8]            0.64     0.1   0.16   0.41    0.5   0.62   0.78   0.89      3   2.85
v[9]             0.2    0.05    0.1   0.03   0.11    0.2   0.29   0.39      4   2.47
v[10]           0.32    0.06   0.14   0.07   0.21   0.35   0.42   0.58      6   1.27
sigma_cl[1]  3.9e-4  5.6e-7 6.7e-6 3.8e-4 3.9e-4 3.9e-4 3.9e-4 4.0e-4    142    1.0
sigma_cl[2]    1.73     0.5   1.08   0.44   0.73   1.59   2.48   4.14      5   1.25
sigma_cl[3]  5.3e-4  1.3e-6 1.2e-5 5.0e-4 5.2e-4 5.2e-4 5.3e-4 5.5e-4     74    1.0
sigma_cl[4]  1.9e-3  6.9e-6 8.2e-5 1.7e-3 1.8e-3 1.9e-3 1.9e-3 2.1e-3    139   1.01
sigma_cl[5]    0.43    0.02   0.06   0.33   0.39   0.42   0.46   0.55     14   1.08
sigma_cl[6]    2.13    0.23   0.53   1.21   1.78    2.1    2.5   3.25      5   1.15
sigma_cl[7]    5.09    1.46   3.17   1.74   2.61   4.05   6.83   12.8      5   1.31
sigma_cl[8]     0.9    0.11    0.4   0.33   0.61   0.81   1.16   1.82     12    1.0
sigma_cl[9]    2.27    0.59   1.11   0.61   1.36   2.29   2.99   4.39      4   1.64
sigma_cl[10]   2.79    0.09    0.4   2.15   2.49   2.76   3.04   3.76     19    1.0
alpha          1.01    0.05    0.2   0.65   0.87   1.01   1.11   1.47     17    1.2
pi[1]          0.51  6.4e-4 6.9e-3    0.5   0.51   0.51   0.52   0.53    114   1.01
pi[2]        2.5e-4  4.7e-5 2.4e-4 9.5e-6 7.2e-5 2.0e-4 3.4e-4 8.4e-4     27    1.1
pi[3]          0.38  6.1e-4 6.9e-3   0.37   0.38   0.38   0.38   0.39    130   1.01
pi[4]          0.11  4.1e-4 4.4e-3    0.1    0.1   0.11   0.11   0.12    114    1.0
pi[5]        5.6e-5  6.4e-6 6.2e-5 1.9e-6 1.5e-5 3.5e-5 7.1e-5 2.3e-4     95    1.0
pi[6]        1.0e-4  1.4e-5 1.0e-4 2.9e-6 2.6e-5 7.0e-5 1.5e-4 3.7e-4     58   1.07
pi[7]        2.7e-5  2.8e-6 2.8e-5 8.3e-7 7.3e-6 1.9e-5 3.7e-5 9.7e-5     97   1.01
pi[8]        2.4e-6  6.5e-7 3.1e-6 3.7e-8 4.2e-7 1.3e-6 3.4e-6 1.1e-5     23   1.08
pi[9]        2.9e-7  1.2e-7 4.8e-7 2.6e-9 3.3e-8 1.0e-7 3.2e-7 1.9e-6     16   1.12
pi[10]       1.7e-6  9.6e-7 2.9e-6 7.7e-9 1.0e-7 5.4e-7 2.0e-6 1.0e-5      9    1.3
lp__          2.3e4    0.19   2.28  2.3e4  2.3e4  2.3e4  2.3e4  2.3e4    141    1.0

Samples were drawn using NUTS at Mon May 31 14:04:12 2021.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

Converged lp__ allows greater confidence that the whole sampling process has converged, but the value itself isn't particularly important.

n_eff is the effective sample size, which because of correlation between samples, can be significantly lower than the nominal amount of samples generated. The effect of autocorrelation can be mitigated by thinning the Markov chains.

Rhat is the Gelman-Rubin convergence statistic, a measure of Markov chain convergence, and corresponds to the scale factor of variance reduction that could be observed if sampling were allowed to continue forever. So if Rhat is approximately 1, you would expect to see no decrease in sampling variance regardless of how long you continue to iterate, and so the Markov chain is likely (but not guaranteed) to have converged.

In [6]:

```
# Extracting traces
pi = fit['pi']
mu = fit['mu_cl']
lp = fit['lp__']
```

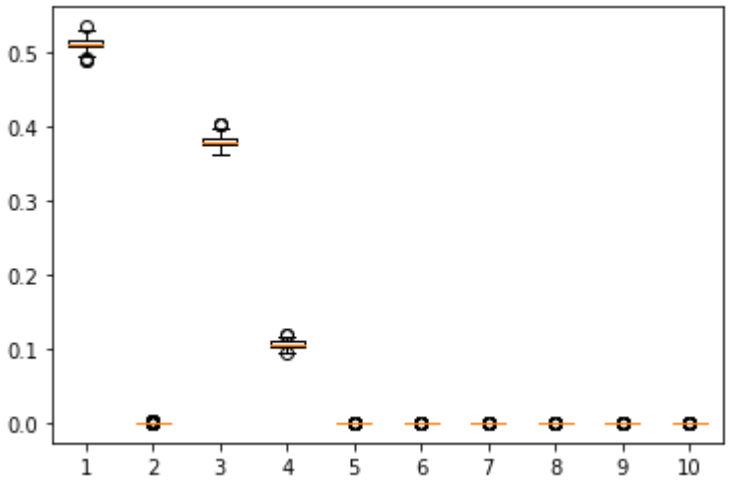
Plotting Posteriors

In [7]:

```
# Define a function that plots the trace and posterior distribution for a given parameter
def plot_trace(param, param_name='parameter'):
    """Plot the trace and posterior of a parameter."""

    # Summary statistics
    mean = np.mean(param)
    median = np.median(param)
    cred_min, cred_max = np.percentile(param, 2.5), np.percentile(param, 97.5)

    # Plotting
    plt.subplot(2,1,1)
    plt.plot(param)
    plt.xlabel('samples')
    plt.ylabel(param_name)
    plt.axhline(mean, color='r', lw=2, linestyle='--')
    plt.axhline(median, color='c', lw=2, linestyle='--')
    plt.axhline(cred_min, linestyle=':', color='k', alpha=0.2)
    plt.axhline(cred_max, linestyle=':', color='k', alpha=0.2)
    plt.title('Trace and Posterior Distribution for {}'.format(param_name))

    plt.subplot(2,1,2)
    plt.hist(param, 30, density=True); sns.kdeplot(param, shade=True)
    plt.xlabel(param_name)
    plt.ylabel('density')
    plt.axvline(mean, color='r', lw=2, linestyle='--',label='mean')
    plt.axvline(median, color='c', lw=2, linestyle='--',label='median')
    plt.axvline(cred_min, linestyle=':', color='k', alpha=0.2, label=r'95\% CI')
    plt.axvline(cred_max, linestyle=':', color='k', alpha=0.2)

    plt.gcf().tight_layout()
    plt.legend()
```

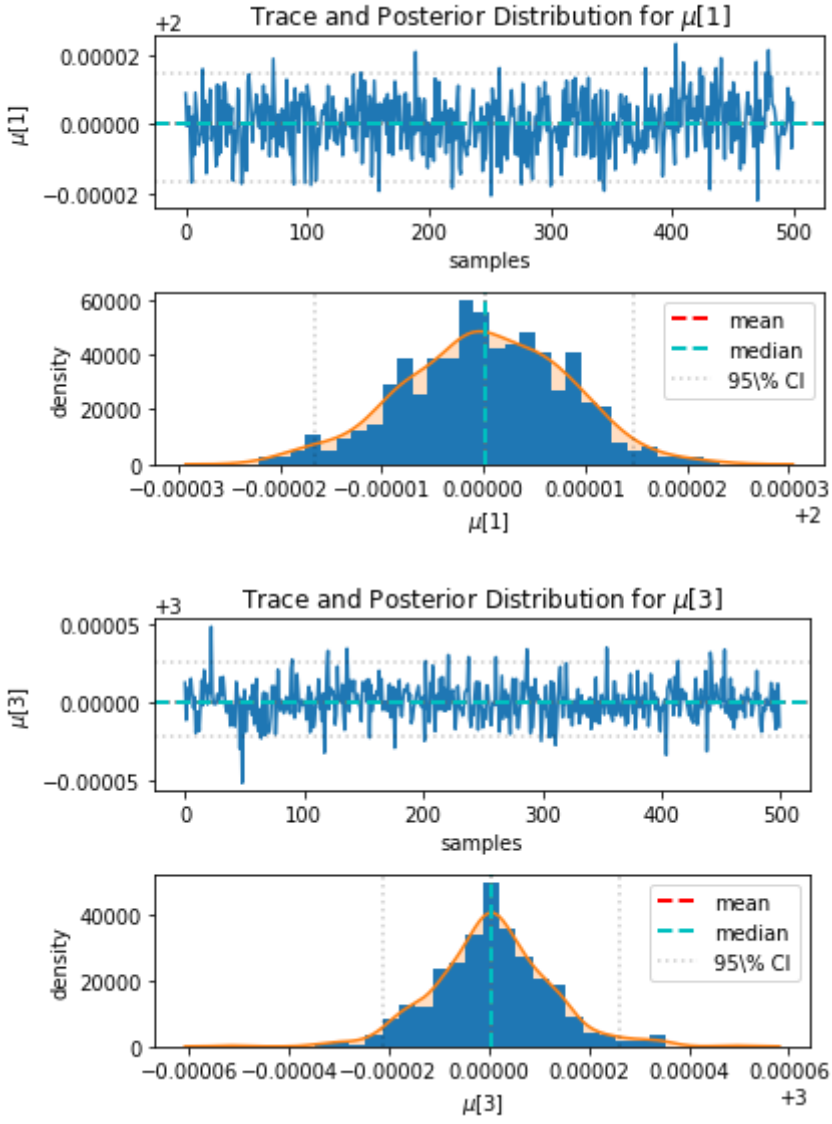
In [8]:

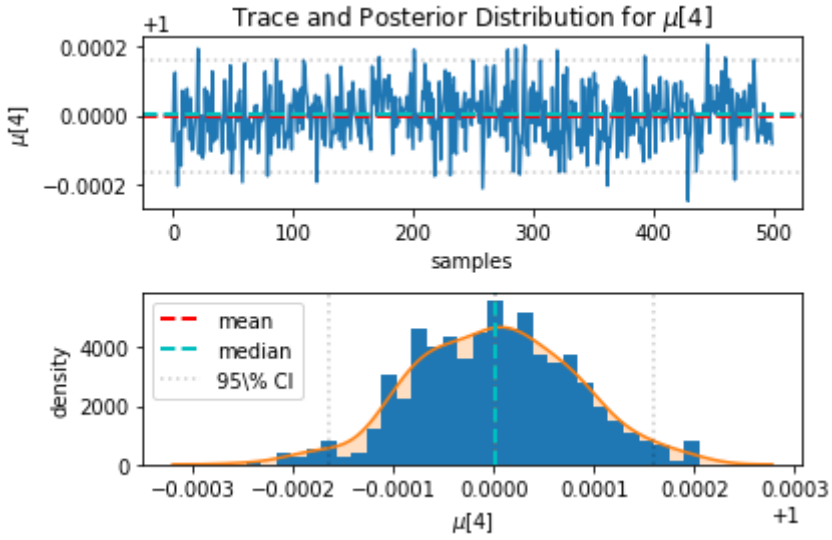
```
# Detect number of clusters
plt.boxplot(pi)
plt.show()
```

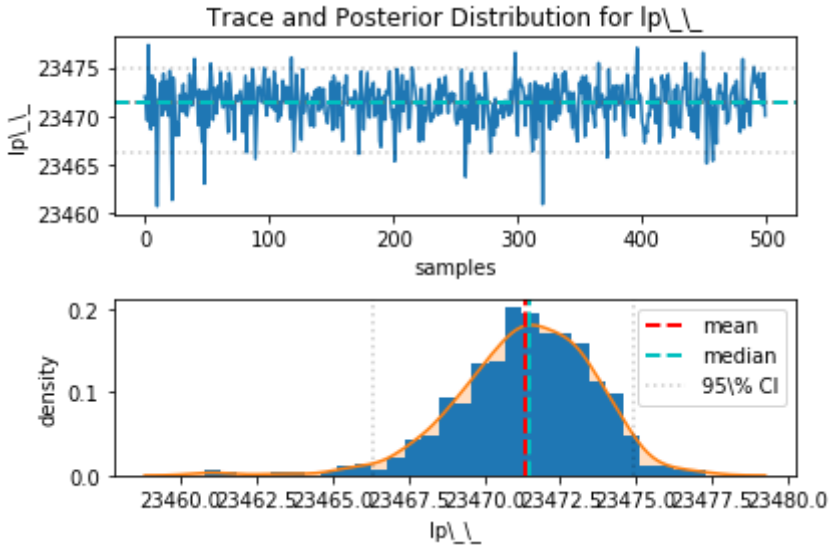


In [9]:

```
plot_trace(mu[:,0], r'$\mu[1]$')
plt.show()
plot_trace(mu[:,2], r'$\mu[3]$')
plt.show()
plot_trace(mu[:,3], r'$\mu[4]$')
plt.show()
plot_trace(lp, r'lp\_\_')
plt.show()
```




Trace and Posterior Distribution for μ[4]
+1
0.0002
0.0000
−0.0002
μ[4]
0
100
200
300
400
500
samples
mean
median
95\% CI
density
4000
2000
0
−0.0003
−0.0002
−0.0001
0.0000
0.0001
0.0002
0.0003
μ[4]
+1


Trace and Posterior Distribution for lp__
23475
23470
23465
23460
lp__
0
100
200
300
400
500
samples
0.2
0.1
0.0
density
mean
median
95\% CI
23460.0
23462.5
23465.0
23467.5
23470.0
23472.5
23475.0
23477.5
23480.0
lp__

In [ ]: