Statistical Computing 2

숙제 5
2019년 가을학기
응용통계학과 석사과정 최석준

HW4-3 & HW5-1.

이번에도 MH 알고리즘 돌리는 부분을 위해, 지난번에 짠 코드를 가져다 쓰겠다. 숙제 4 와 비교하면, 기존 버전으로는 hybrid MCMC 한 iteration 마다 터미널에 너무 많은 출력이 나오기 때문에, generate_samples()의 verbose 가 걸리는 IF 문 부분을 약간 고쳐서 진행상황 출력 정보를 좀 줄이도록 수정하였다. 다른 알고리즘 부분 자체는 전혀 바꾼 것이 없다.

[Python]

```
#python 3 file created by Choi, Seokjun

#using Markov chain monte carlo,
#get samples!



import time
from math import log
from random import uniform



class MC_MH:
    def __init__(self, log_target_pdf, log_proposal_pdf, proposal_sampler, data, initial):
        self.log_target_pdf = log_target_pdf #arg (smpl)
        self.log_proposal_pdf = log_proposal_pdf #arg (from_smpl, to_smpl)
        self.proposal_sampler = proposal_sampler #function with argument (smpl)

        self.data = data
        self.initial = initial

        self.MC_sample = [initial]

        self.num_total_iters = 0
        self.num_accept = 0



    def log_r_calculator(self, candid, last):
        log_r = (self.log_target_pdf(candid) - self.log_proposal_pdf(from_smpl=last, to_smpl=candid) - ₩
            self.log_target_pdf(last) + self.log_proposal_pdf(from_smpl=candid, to_smpl=last))
        return log_r

    def sampler(self):
        last = self.MC_sample[-1]
        candid = self.proposal_sampler(last) #기존 state 집어넣게
        unif_sample = uniform(0, 1)
        log_r = self.log_r_calculator(candid, last)
        # print(log(unif_sample), log_r) #for debug
```

```
            if log(unif_sample) < log_r:
                self.MC_sample.append(candid)
                self.num_total_iters += 1
                self.num_accept += 1
            else:
                self.MC_sample.append(last)
                self.num_total_iters += 1

    def generate_samples(self, num_samples, pid=None, verbose=True):
        start_time = time.time()
        for i in range(1, num_samples):
            self.sampler()
            if i%500 == 0 and verbose and pid is not None:
                print("pid:",pid," iteration", i, "/", num_samples)
            elif i%500 == 0 and verbose and pid is None:
                print("iteration", i, "/", num_samples)
        elap_time = time.time()-start_time
        if pid is not None and verbose: #여기 verbose 추가함
            print("pid:",pid, "iteration", num_samples, "/", num_samples, " done! (elapsed time for execution: ", elap_time//60,"min ", elap_time%60,"sec)")
        elif pid is None and verbose: #여기 verbose 추가함
            print("iteration", num_samples, "/", num_samples, " done! (elapsed time for execution: ", elap_time//60,"min ", elap_time%60,"sec)")


    def burnin(self, num_burn_in):
        self.MC_sample = self.MC_sample[num_burn_in-1:]

    def thinning(self, lag):
        self.MC_sample = self.MC_sample[::lag]
```

HW4-3. Using MCMC, get posterior samples of Item Response Model.

기막히게도 지난 코드에서 beta, theta 들의 prior 구현을 빠뜨렸었기 때문에, 이를 수정한 코드를 첨부한다. (prior를 빼먹은 이유... independent MCMC 코드에서 긁어다가 얼렁뚱땅 고쳐서 IRM class를 만들었기 때문이다... 반성...) 수정한 부분은 **볼드**처리 하였다.

다른 부분은 HW4 와 같으므로 추가 설명 및 코멘트를 주요사항만 간략히 하겠다.

세팅은 지난번과 거의 같다. proposal의 sd는 일괄적으로 0.07로 놓고, initial은 unif(-10,10)이다. 8 chain을 동시에 생성하고, 생성된 chain 마다 beta1~beta24 의 traceplot 을 그려서 수렴을 확인한다.

이후 마지막으로 완료된 체인을 골라, beta 들의 histogram 을 그리고 acf 를 확인한다. 그리고 burnin 으로 버릴 수와 thinning 정도를 설정한 후 수행한다. 아래 코드에서는 앞 2 만개를 버리고, 남은 sample 을 200 개마다 1 개씩만 다시 솎아 사용하였다.

[Python]

```
#python 3 file created by Choi, Seokjun

#using Markov chain monte carlo,
#get samples of Item response model's parameters

import re
import time
import os
import multiprocessing as mp

from math import exp, log
from random import normalvariate, uniform
from statistics import mean

import matplotlib.pyplot as plt

from HW4_MC_Core import MC_MH


class IRM_McPost(MC_MH):
    def IRM_proposal_sampler(self, last):
            #sd를 각 parameter마다 다르게 잡을수 있도록 tuple로 받자
            sd = self.proposal_sampler_sd
            return [normalvariate(last[i], sd[i]) for i in range(418+24)]

    def IRM_log_proposal_pdf(self, from_smpl, to_smpl):
        #When we calculate log_r(MH ratio's log value), just canceled.
        #since normal distribution is symmetric.
        #so we do not need implement this term.
        return 0

    def IRM_log_likelihood(self, param_vec):
        theta = param_vec[:self.num_row]
        beta = param_vec[self.num_row:]
        log_likelihood_val = 0
        for i in range(self.num_row):
            for j in range(self.num_col):
                param_sum = theta[i] + beta[j]
```

```
                log_likelihood_val += self.data[i][j]*param_sum - log(1+exp(param_sum))
        return log_likelihood_val

    def IRM_log_prior(self, param_vec):
        #When we calculate log_r(MH ratio's log value), just canceled.
        #so we do not need implement this term.
        # 제출 후 수정
        #<-선희씨의 피드백: 위 주석은.. 아닌거같음 # proposal은 나눠져도, prior가 나눠지진 않는듯...
        log_prior_val = 0
        for param in param_vec:
            log_prior_val += param**2/200 #앞 상수 날리고
        return log_prior_val

    def IRM_log_target(self, param_vec):
        return self.IRM_log_likelihood(param_vec) + self.IRM_log_prior(param_vec)

    def __init__(self, proposal_sampler_sd, data, initial):
        #proposal_sampler_sd : 418+24=442 dim iterable object
        super().__init__(log_target_pdf=None, log_proposal_pdf=None, proposal_sampler=None, data=data, initial=initial)
        # self, log_target_pdf, log_proposal_pdf, proposal_sampler, data, initial
        self.num_row = len(data) #n
        self.num_col = len(data[0]) #p
        # n*p = 418*24

        self.proposal_sampler_sd = proposal_sampler_sd
        self.proposal_sampler = self.IRM_proposal_sampler
        self.log_proposal_pdf = self.IRM_log_proposal_pdf
        self.log_target_pdf = self.IRM_log_target



        #parameters: i: rows(0~417), j:cols(0~23)
        #(theta0, theta1, ..., theta417, beta0, beta1, ..., beta23) : 418+24 dim

    def get_specific_dim_samples(self, dim_idx):
        if dim_idx >= self.num_row+self.num_col:
            raise ValueError("dimension index should be lower than number of dimension. note that index starts at 0")
        return [smpl[dim_idx] for smpl in self.MC_sample]

    def get_acceptance_rate(self):
        return self.num_accept/self.num_total_iters

    def get_sample_mean(self):
        #burnin자르고 / thining 이후 쓸것
        mean_vec = []
        for i in range(self.num_row+self.num_col):
            would_cal_mean = self.get_specific_dim_samples(i)
            mean_vec.append(mean(would_cal_mean))
        return mean_vec

    def show_hist(self, dim_idx, show=True):
        hist_data = self.get_specific_dim_samples(dim_idx)
        plt.ylabel(str(dim_idx)+"th dim")
        plt.hist(hist_data, bins=100)
        if show:
            plt.show()
```

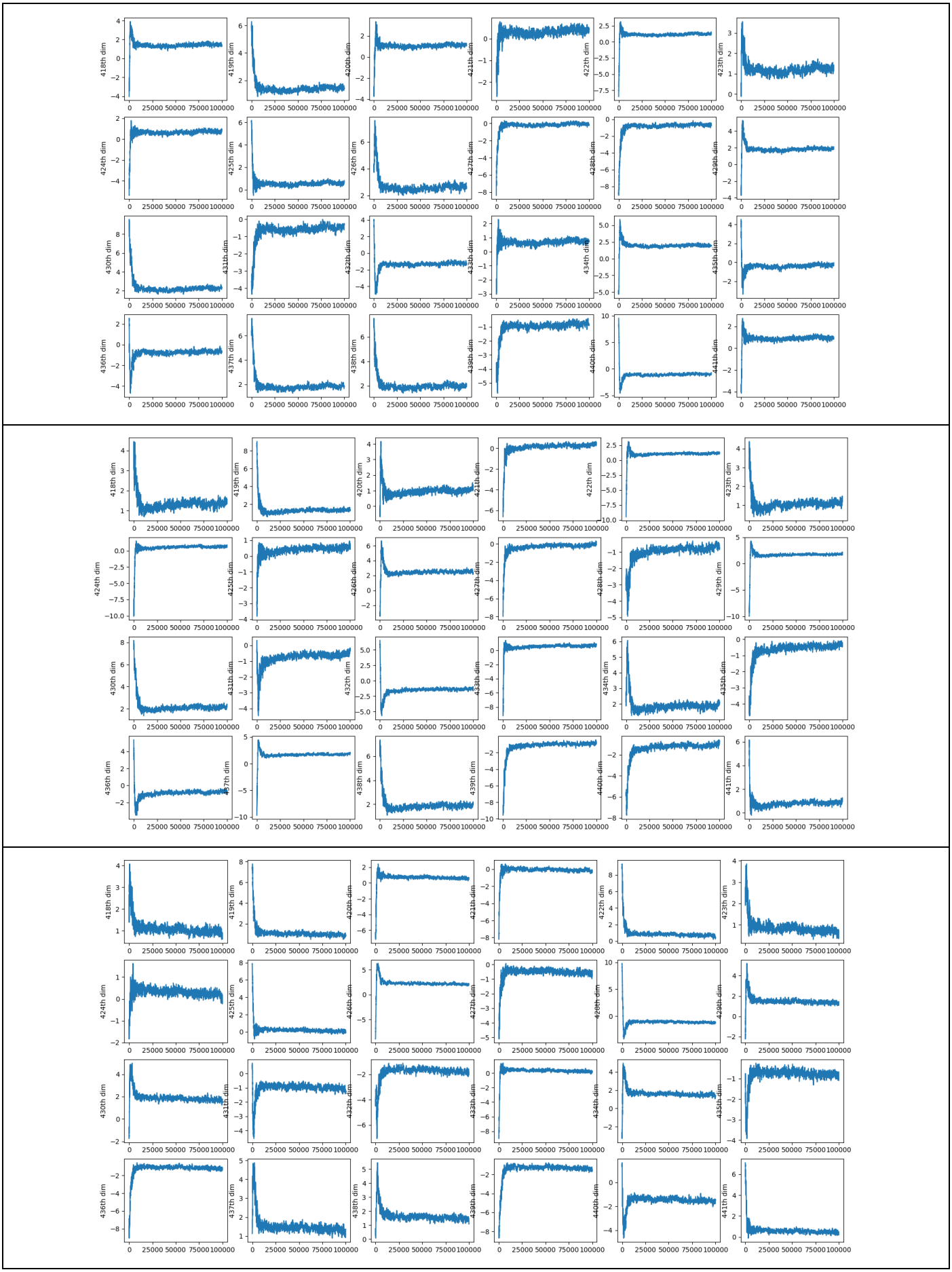
```
def show_traceplot(self, dim_idx, show=True):
    traceplot_data = self.get_specific_dim_samples(dim_idx)
    plt.ylabel(str(dim_idx)+"th dim")
    plt.plot(range(len(traceplot_data)), traceplot_data)
    if show:
        plt.show()

def show_betas_traceplot(self):
    grid_column= 6
    grid_row = 4
    plt.figure(figsize=(5*grid_column, 3*grid_row))
    for i in range(24):
        plt.subplot(grid_row, grid_column, i+1)
        self.show_traceplot(418+i,False)
    plt.show()

def show_betas_hist(self):
    grid_column= 6
    grid_row = 4
    plt.figure(figsize=(5*grid_column, 3*grid_row))
    for i in range(24):
        plt.subplot(grid_row, grid_column, i+1)
        self.show_hist(418+i, False)
    plt.show()

def get_autocorr(self, dim_idx, maxLag):
    y = self.get_specific_dim_samples(dim_idx)
    acf = []
    y_mean = mean(y)
    y = [elem - y_mean   for elem in y]
    n_var = sum([elem**2 for elem in y])
    for k in range(maxLag+1):
        N = len(y)-k
        n_cov_term = 0
        for i in range(N):
            n_cov_term += y[i]*y[i+k]
        acf.append(n_cov_term / n_var)
    return acf

def show_acf(self, dim_idx, maxLag, show=True):
    grid = [i for i in range(maxLag+1)]
    acf = self.get_autocorr(dim_idx, maxLag)
    plt.ylim([-1,1])
    plt.ylabel(str(dim_idx)+"th dim")
    plt.bar(grid, acf, width=0.3)
    plt.axhline(0, color="black", linewidth=0.8)
    if show:
        plt.show()

def show_betas_acf(self, maxLag):
    grid_column= 6
    grid_row = 4
    plt.figure(figsize=(5*grid_column, 3*grid_row))
    for i in range(24):
```

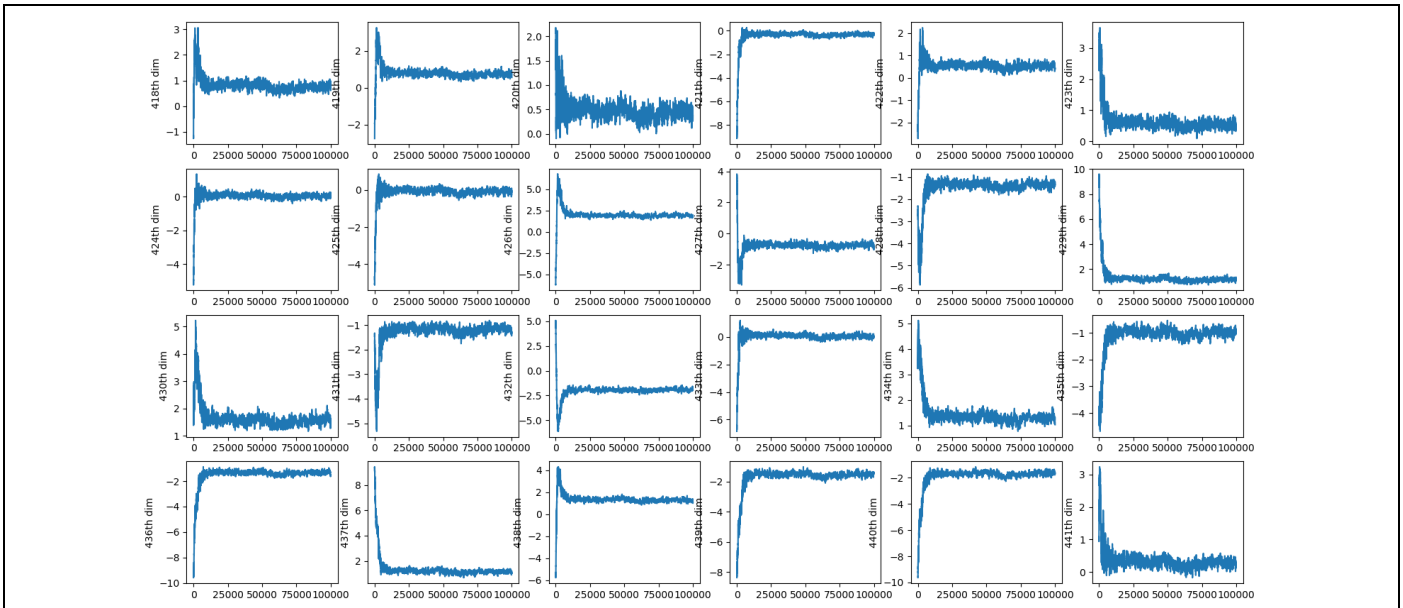
```
                plt.subplot(grid_row, grid_column, i+1)
                self.show_acf(418+i, maxLag, False)
        plt.show()


#####################################################################

#for multiprocessing
def multiproc_1unit_do(result_queue, prop_sd, data, initial, num_iter):
    func_pid = os.getpid()
    print("pid: ", func_pid, "start!")
    UnitMcSampler = IRM_McPost(prop_sd, data, initial)

    UnitMcSampler.generate_samples(num_iter, func_pid)
    # UnitMcSampler.burnin(num_iter//2)
    acc_rate = UnitMcSampler.get_acceptance_rate()
    result_queue.put(UnitMcSampler)
    print("pid: ", func_pid, " acc_rate:",acc_rate)


#ex3
if __name__ == "__main__":

    data = []
    with open("c:/gitProject/statComputing2/HW4/drv.txt","r", encoding="utf8") as f:
        while(True):
            line = f.readline()
            split_line = re.findall(r"₩w", line)
            if not split_line:
                break
            split_line = [int(elem) for elem in split_line]
            data.append(split_line)


    core_num = 8 #띄울 process 수
    num_iter = 100000 #each MCMC chain's
    prop_sd = [0.07 for _ in range(418+24)] #0.07정도에서 acc.rate가 맘에들게나옴

    proc_vec = []
    proc_queue = mp.Queue()

    #여기에서 initial을 만들고 process 등록
    for i in range(core_num):
        unit_initial = [uniform(-10,10) for _ in range(418+24)]

        unit_proc = mp.Process(target = multiproc_1unit_do, args=(proc_queue, prop_sd, data, unit_initial,num_iter))
        proc_vec.append(unit_proc)


    for unit_proc in proc_vec:
        unit_proc.start()

    mp_result_vec = []
    for _ in range(core_num):
        each_result = proc_queue.get()
```

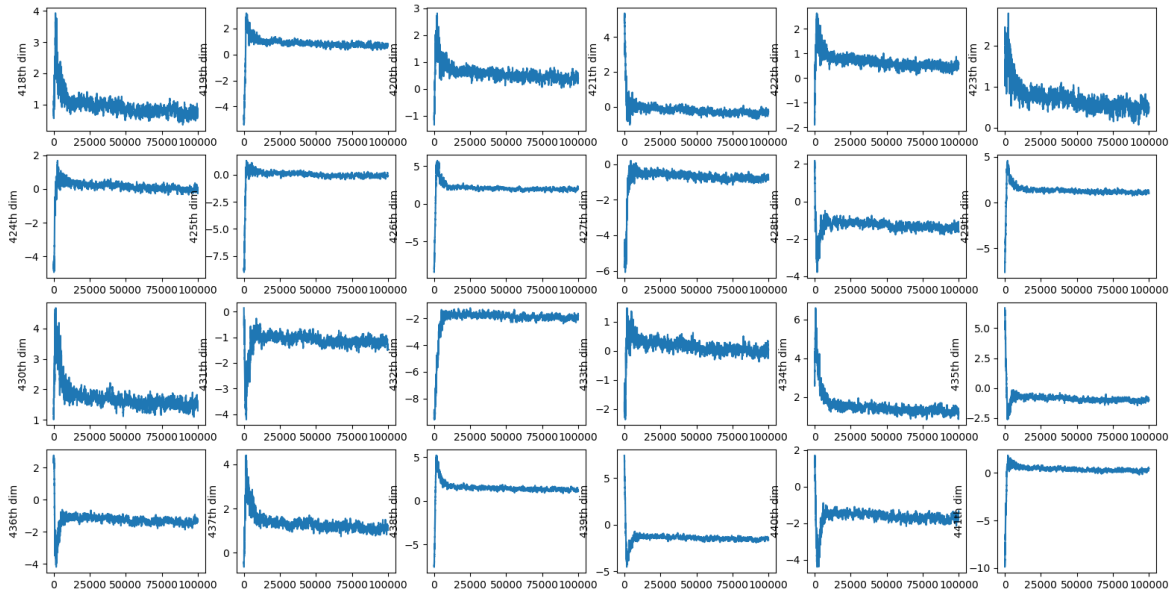
```
        # print("mp_result_vec_object:", each_result)
        mp_result_vec.append(each_result)

    for unit_proc in proc_vec:
        unit_proc.join()
    print("exit.mp")

    #cheak traceplot
    for chain in mp_result_vec:
        chain.show_betas_traceplot()

    ############################
    OurMcSampler = mp_result_vec[-1] #마지막 체인
    print("acc rate: ", OurMcSampler.get_acceptance_rate())
    # OurMcSampler.show_traceplot(0) #theta1
    # OurMcSampler.show_hist(0) #theta1



    OurMcSampler.show_betas_hist()
    OurMcSampler.show_betas_traceplot()
    OurMcSampler.show_betas_acf(200)
    # print("meanvec(before burn-in): ", OurMcSampler.get_sample_mean())



    OurMcSampler.burnin(20000)
    OurMcSampler.thinning(200)
    OurMcSampler.show_betas_traceplot()
    OurMcSampler.show_betas_hist()
    OurMcSampler.show_betas_acf(30)
    print("mean vec(after burn-in): ", OurMcSampler.get_sample_mean())
```

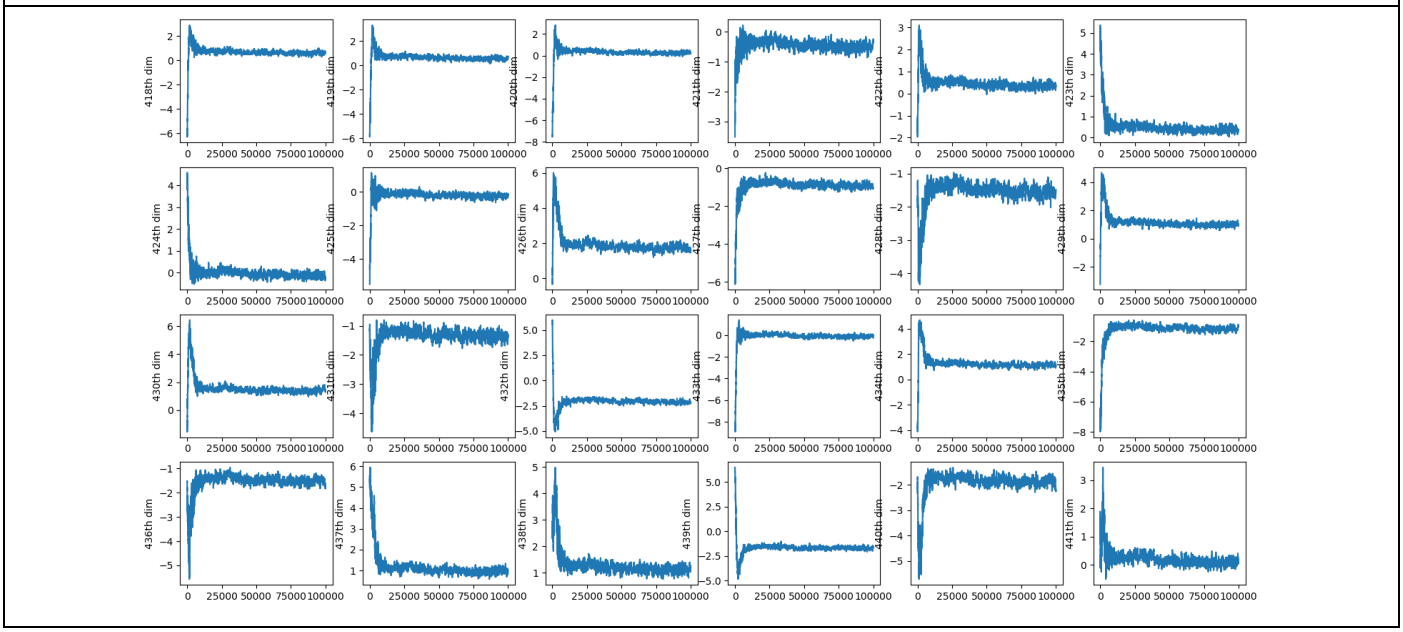
실행하면 다음의 출력을 얻는다. 먼저 멀티프로세싱이 끝난 후 8 개의 chain 의 traceplot 을 각각 그려보는 부분의 결과이다. beta 들만 그렸다.

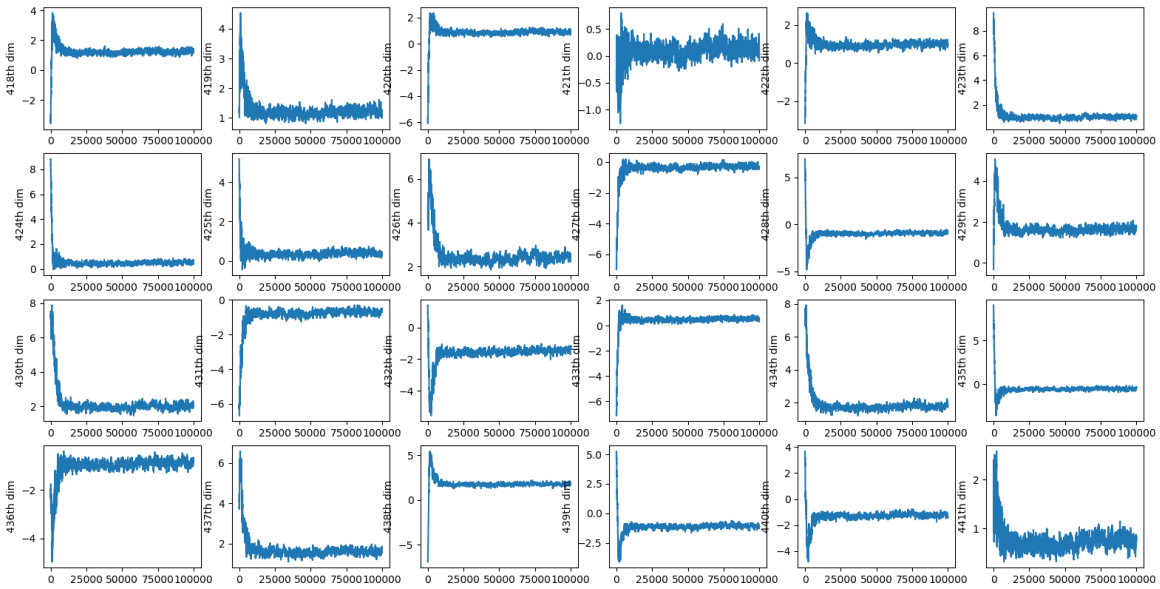
```
[Console]
pid: 12300 iteration 100000 / 100000   done! (elapsed time for execution:   23.0 min   42.45667624473572 sec
pid:   12300   acc_rate: 0.07030070300703008
pid: 9668 iteration 100000 / 100000   done! (elapsed time for execution:   23.0 min   44.26086211204529 sec
pid: 1872 iteration 100000 / 100000   done! (elapsed time for execution:   23.0 min   54.97930550575256 sec
pid:   1872   acc_rate: 0.07052070520705207
pid: 12756   iteration 99500 / 100000
pid: 13060 iteration 100000 / 100000   done! (elapsed time for execution:   23.0 min   54.15328907966614 sec
pid:   13060   acc_rate: 0.06848068480684807
pid: 2116   iteration 98500 / 100000
pid: 12756 iteration 100000 / 100000   done! (elapsed time for execution:   23.0 min   59.30728268623352 sec
pid: 15832 iteration 100000 / 100000   done! (elapsed time for execution:   24.0 min   5.886221647262573 sec
pid:   15832   acc_rate: 0.06859068590685907
pid: 7120 iteration 100000 / 100000   done! (elapsed time for execution:   24.0 min   8.004352807998657 sec
pid:   7120   acc_rate: 0.07070070700707007
pid: 2116 iteration 100000 / 100000   done! (elapsed time for execution:   24.0 min   6.753058910369873 sec
pid:   2116   acc_rate: 0.06711067110671107
```

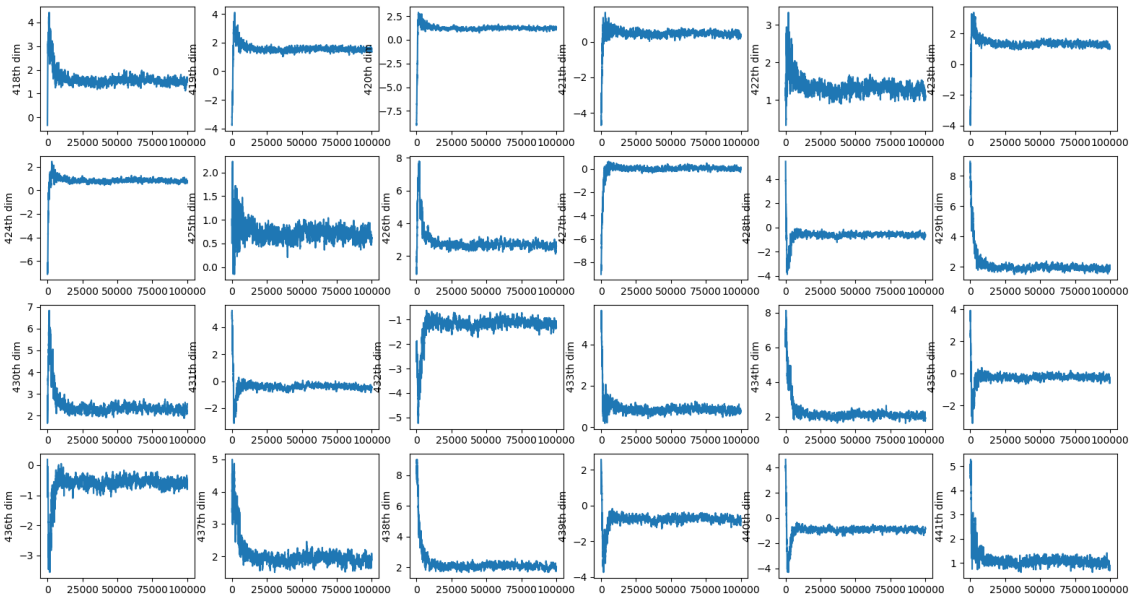

418th dim
419th dim
420th dim
421th dim
422th dim
423th dim
424th dim
425th dim
426th dim
427th dim
428th dim
429th dim
430th dim
431th dim
432th dim
433th dim
434th dim
435th dim
436th dim
437th dim
438th dim
439th dim
440th dim
441th dim
0 25000 50000 75000 100000


418th dim
419th dim
420th dim
421th dim
422th dim
423th dim
424th dim
425th dim
426th dim
427th dim
428th dim
429th dim
430th dim
431th dim
432th dim
433th dim
434th dim
435th dim
436th dim
437th dim
438th dim
439th dim
440th dim
441th dim
0 25000 50000 75000 100000


418th dim
419th dim
420th dim
421th dim
422th dim
423th dim
424th dim
425th dim
426th dim
427th dim
428th dim
429th dim
430th dim
431th dim
432th dim
433th dim
434th dim
435th dim
436th dim
437th dim
438th dim
439th dim
440th dim
441th dim
0 25000 50000 75000 100000


418th dim
419th dim
420th dim
421th dim
422th dim
423th dim
424th dim
425th dim
426th dim
427th dim
428th dim
429th dim
430th dim
431th dim
432th dim
433th dim
434th dim
435th dim
436th dim
437th dim
438th dim
439th dim
440th dim
441th dim
0 25000 50000 75000 100000

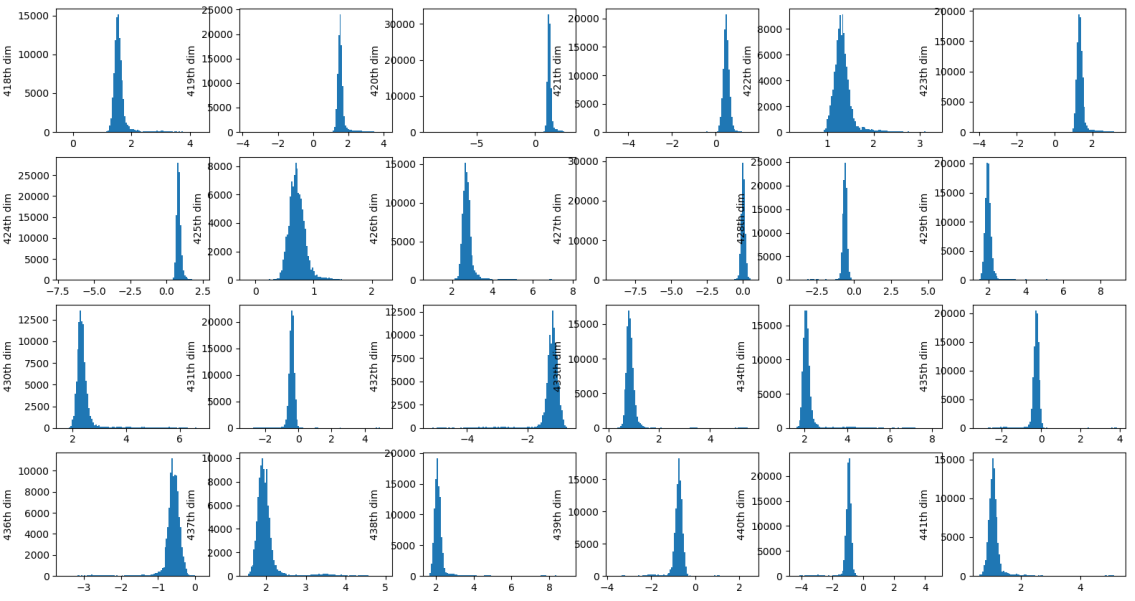


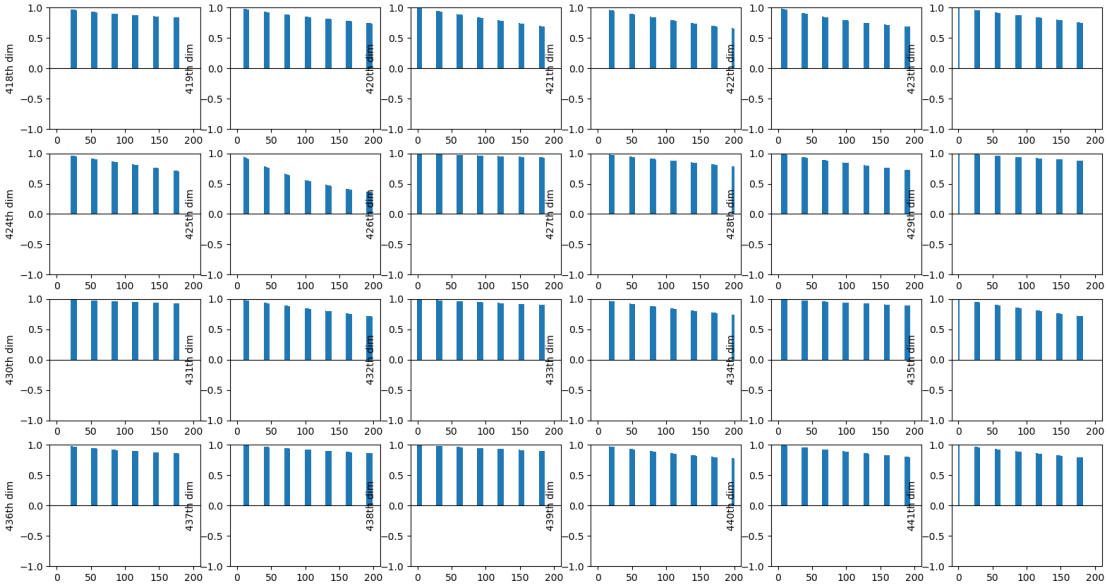


다음으로, 마지막 체인만 골라 beta 들의 histogram 과 acf 를 그려본다.

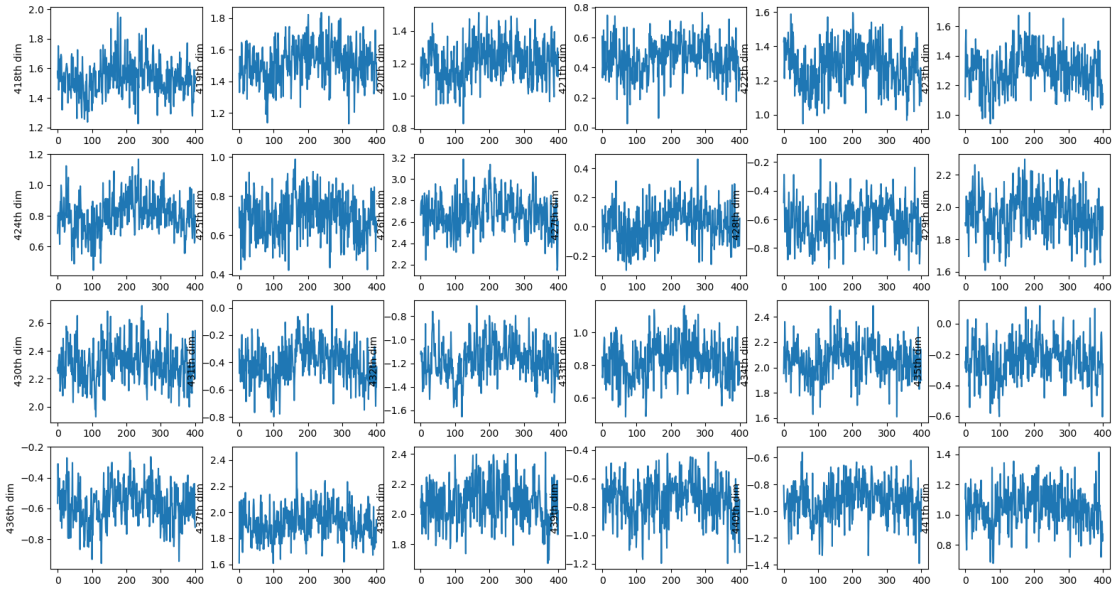
[Console]

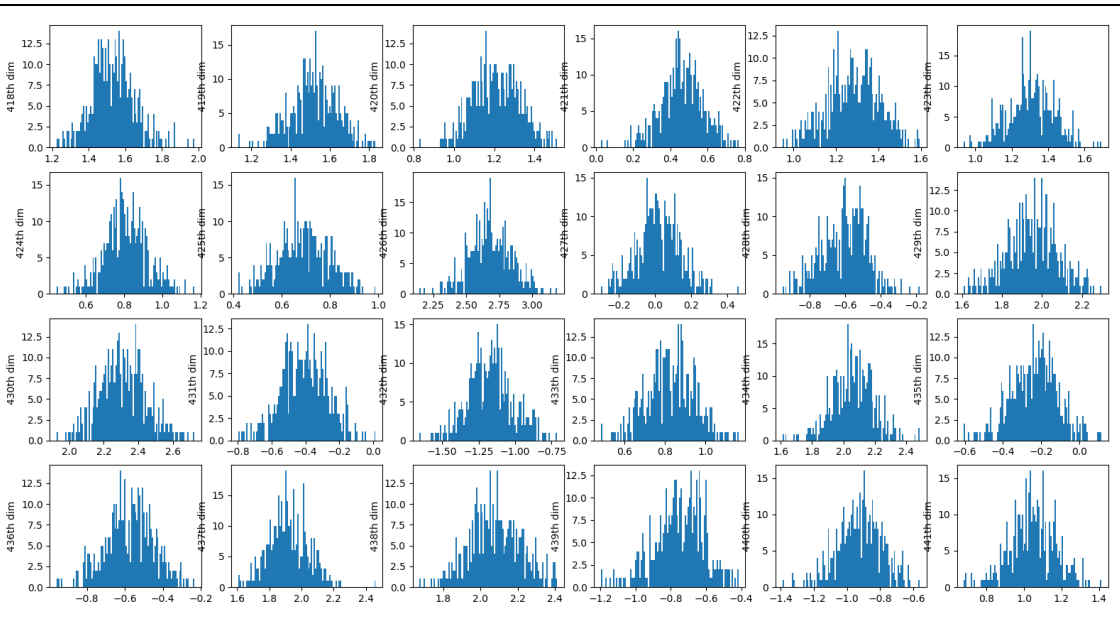
acc rate: 0.06711067110671107

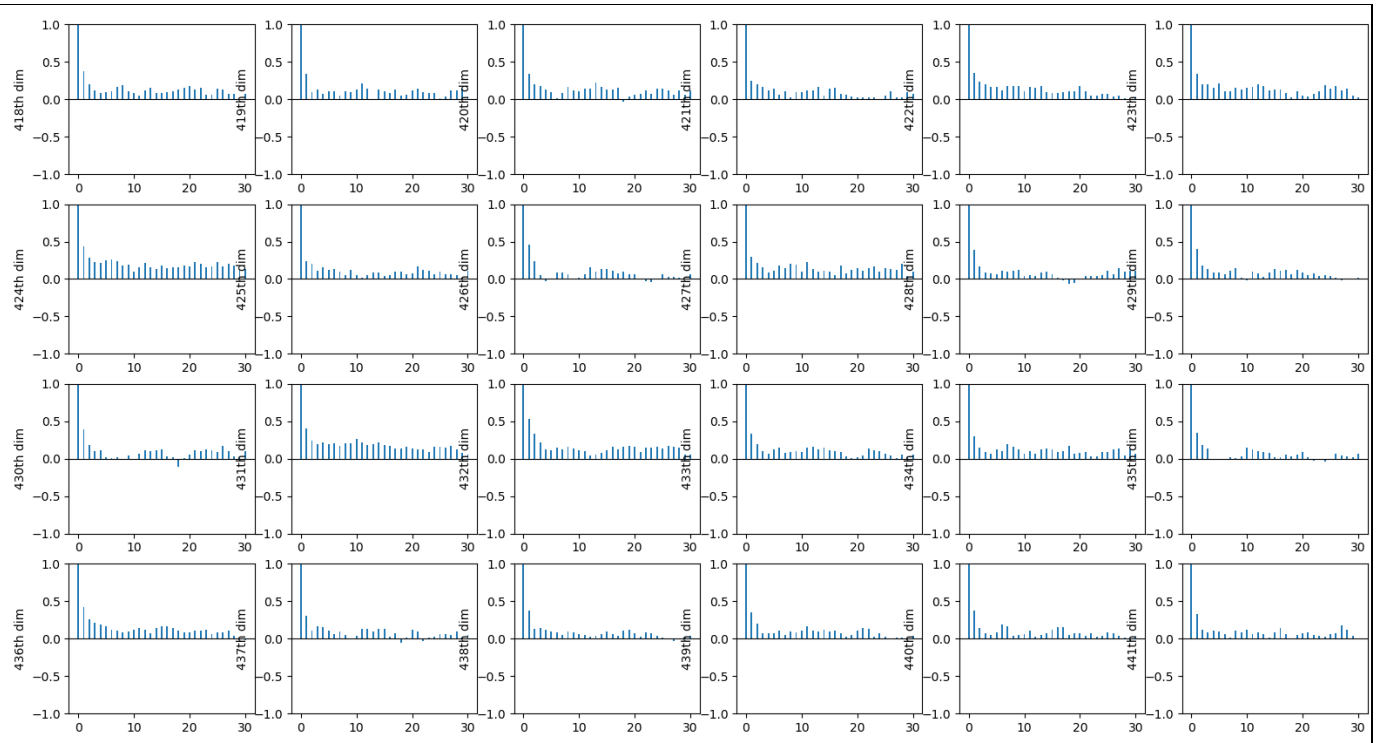




burn-in period 앞 2 만개를 자르고, 200 개당 1 개씩 취하는 thinning 이후의 결과이다.







[Console]

mean vec(after burn-in): [-1.7538818559608262, -0.9035471452325929, -1.2195178462960723, -1.4688274144143252, -1.6549704372918168, -0.7962368898949062, -0.6779298438198984, -1.4718551863155345, -0.719073590881322, -0.8022068163195102, -0.8751424284328339, -1.699439839288686, -1.585094429698931, -0.676718793013507, -0.8319961487271513, -1.0864951401993432, -1.267667260402599, -1.2105470642110916, -0.8466751326893114, -1.3829713161336203, -0.6232772851399466, -1.888699941093978, -0.845853027175302, -1.3889829201315986, -0.9667517014306583, -1.671787656436896, -1.098700518336476, -0.6718479424498346, 0.3506365473102427, -0.4967529068904953, -0.9869646616071845, -1.1271297236341025, -0.7428178624624628, -0.670149170366392, -1.294613246470156, -0.6695764295496353, -0.5669741588630888, -1.005419746011159, -0.9891606408449041, -1.7331847450302706, -0.48841084007127816,
0.02068552093954514, -0.637240696508264, -1.0205965682735656, -1.283424133841699, -1.1081817498905715, -0.9300458897461542, -1.283673948583262, -1.3522381757530764, -0.8417169733225818, -1.083061329779672, -0.9983200208625911, -1.5244462361659874, -1.0172252000390805, -1.4453209009008452, -1.4596011587800055, -1.1187226012852087, -1.506751768850659, -0.7999491286349488, -1.0745308772814341, -1.6255319857433173, -1.3770298930964833, -0.8779600995359214, -0.7010353566406882, -0.8717906793856156, -0.5911559581635285, -0.7753761419199834, -0.8221356271456056, -1.4037153130223035, -1.8375993688889911, -0.806805078360641, -1.3090598369241344, -0.16928238194436607, 0.49788740988347724, -0.1599563534614593, -0.07189390496243012, -0.7727386708500167, -1.1257938749952392, -1.9123724223734484, -1.6308416915908253, -0.8621428171966735, -0.27440266694380266, -0.44605794253850134, -0.9964448954394527, -0.9542367456513691, -0.801777728124834, -0.792200114791767, -0.9090488893889433, -1.2156746190125896, -1.0106383093935418, 0.1584895323532244, -1.812874509236016, -0.5588800468743648, -0.7531817999167986, -0.9984832009495056, -1.3187269971207574, -0.4416360781018561, -2.164987515787303, -1.075207292178373, -0.9563829716318178, -0.78226022467185, -0.6555740406416087, -1.0619841907116516, -0.86434625588321, -0.8661292358709761, -0.6751742475806567, -0.8906802024774573, -0.9498122502412726, -1.4943575355928427, -1.9125214595974394, -0.8980127981252672, -1.146454350749064, -0.526482174024635, -1.3237905013734759, -0.7014782685296193, -1.4093697128407505, -0.8692747670006713, -0.6926069495557025, -0.8275319226006537, -1.2746117064815508, -0.2860072188713959, -0.43588633662822746, -0.5466587054598275, -0.8588618518944665, -1.4195796941628422, -1.090307852324173, -1.0813554651429773, 0.4100538549578799, -0.8084838297933111, -0.8267184211217211, -0.8878828882838, -0.522836749829043, -0.6240303568217157, -1.0404237161201995, -1.1540255514454185, -0.17537019740658158, -1.1152055447339368, -1.0564181582030863, -0.33607844050360386, -1.1528796085389015, -0.6157509690315247, -1.1037911916477479, -1.300460657437036, -0.8822252378424533, -1.8876262633254315, -1.1084914100755396, -0.5362781132205915, -0.8795302908472604, -0.950074514735874, -0.4995700018959542, -0.6911783442522295, -0.6862491289322777, -0.24748170783090642, -0.5121531016591231, -0.4368038356193357, -1.4177929820592585, -1.6678949874890734, 0.6365171580065073, -0.6166758590164841, 0.278903059910566, -1.697578492579279, -1.1565096659535556, -1.4064358622205704, 4.310263883928042, -0.8871683968857671, 0.778282241925252, -0.8388821760317658, 0.18433683744704474, 0.1866945152217589, -0.7913943550420706, 0.8415486595050128, 0.6061867715959669, -0.3677433475825777, 0.4837060732595172, -0.10884484213252053, -1.7941185659028986, -1.3727526074564738, 0.33278396348388173,
-1.460822185487071, -0.7796073430767897, -0.8163924155222653, -0.46668727154480366, -0.14595724358767945, -0.7086935835794731, -2.220144687844661, 0.9480255943352093, -1.4702195287004656, -1.26567279089181, -0.3996540000442728, -0.17946236456573728, -0.816507375474477, 2.0974199271736005, 1.624548467311648, 0.35559621771139993, -1.775900257922393, 1.8062236613459914, -

1.6663483394369898, -1.399279282232009, -1.5069750425891597, -1.301111591422897, -0.9443814904930252, -1.2248045520183362, -1.4128593772327043, -0.759226334828621, 0.7533320402144289, -0.6871887277125416, -1.0913468993781514, -1.118378997744238, -1.4086123793636982, 0.665974552730791,
1.5832809231306817, 0.7917779770174047, -0.4823447734006096, 0.2905136773820349, -1.5941754961445112, -0.7394852050156964, -1.6489530427342336, -1.343543239559759, -1.576260813829228, -0.7236176935604289, -1.4597426082323748, -2.0660806164585988, -0.09881908265678083, -1.929792566049888, -1.3385464112840875, 0.4020660451618559, -0.7774917647550198, -0.5635532026819606, -1.1310459445848176, 0.04029324672980346, 0.2270865740099244, -0.633208224285039, -0.48566767110384473, -0.8927047471283963, -1.1122317511895783, 0.5435735832512452, -0.9583745915223598, 0.390265099263165, 0.569090207912316, -0.40187713226609967, -0.7724348875882432, -0.3287639935324608, -0.5994544236162915, -0.6193469375492283, -1.1374322292665089, -0.2715146875359743, -2.4882352534681673, -0.142583644518755, -0.7138113972667559, -0.3824248512457934, -1.1596670490978158, -0.6788230861305098, -1.074006402970136, 1.409255790832911, 1.5576523571226992, -2.521816150163605, -1.3797640735602263, -0.8847884603078381, -0.835997867061839, -0.5341794193271406, -0.8501191583516585, -0.6164492450217468, -2.640354123716551, -0.19390174884756545, -0.9318350408476168, -0.9439601376588167, -0.6250327822305842, -1.191369354261299, -1.6636705651519688, -1.7151328457637818, -0.607357382882359, -0.9753309133410999, -0.8763610973268077, -1.5277073545115714, -0.8890947676429358, -0.2841127813208886, -1.693506812200289, -0.47746621152192675, -1.3079184183879398, -1.5964856587680802, -2.3671065116439505, -1.3898394009523596, 0.6481767224718884, 0.1772200137381133, 2.275280372119435, 0.537438113026746, -1.2694390580525472, -1.2624941671209886, -1.2164563939839788, -1.6387050123979583, 0.5115248776390883, 5.633911498591798, -0.8361219487881085, -0.655285972244046, -0.12727100733368843, -0.4275582263951657, -1.4952474645713216, -0.6275232835947367, 3.6398259853564654, 0.4560290002383745, -0.6745482223708514, -1.8802572759083056, -1.4259172889046836, 5.0925602706974065, -0.2522537940533032, 0.5090295399309064, -0.05558080405973522, -0.6794064553513025, -0.5049630643423778, 0.37284194786060065, -0.4858288234128476, -0.275698381406929, -0.6119709612492642, -1.6588433375075278, 1.0983215922762606, -0.4507116309333938, 0.2595937327270651, -1.7847475085517852, -0.8261660915253546, -1.348464657274598, 0.22554342518787632, -0.30708846560208264, -0.528023040932079, 1.7905488864857462, -1.7267665198168307, -0.6618784772081457, -0.5440182454770534, 4.330101103862672, 1.8879847166492343, 1.4942789734932025, 3.571017348037011, 0.45924130930726226, 1.2024895108416076, 0.5546113765571974, -1.142374059131885, -0.9323355423946533, 2.1605722484942804, -1.1263015711616513, -0.533433359395582, -0.626216336655007, -0.7324398805564498, -0.8903340705228135, -0.08084093891517319, -1.7369605968983195, -0.7169715892290742, -2.112494105295246, 6.221533147239197, -0.7870635431709453,
-1.2115685330330368, -0.7843252580152219, -1.6345451201346617, 1.115138930826862, -0.8878214534576395, 1.4534777738494744, -2.592737417430858, 3.045831778489255, -0.09481711419860821, -0.3354123005738745, 0.18935597319483202, -1.0875765540645457, -0.4277480619894712, -0.8956092562589024, -1.8804740360727223, -0.42379187655773787, 0.37055994267445425, -0.9275328564045666, -0.24658201841740274, -0.0017752023963924517, 7.102100505122606, -0.7768339205930117, -0.18081940834344087, -1.2437396871945257, -0.7349863379434269, 1.8856220869735143, 0.11136222493574219, 0.6424743381748358, 0.32595296153927267, 0.6478850735933718, 0.45373810399421843, 0.7950127422727881, -0.4686303723991662, -1.2600432521130895, -0.11510139990272879, -1.0146010074169152, -2.000450875319733, -0.903043248096838, 1.6134902975204748, -0.7158029798372992, -0.6390586262216739, 0.5274483846873712, -1.0291499591016449, -0.12353522471683942, -0.132086604740405, -0.5170047050114142, -0.4614133513422481, -2.201265996438695, -0.6577205617257255, -1.9614250367075554, -0.612503631220646, -1.4151741784010887, -0.0753832553196319, -1.1268136352857847, -0.25985860434334496, 0.40419057196722397, -0.35807384183771523, 1.190344478994283, 5.237586618012046, 2.670610821226474, 2.304398612098786, 0.01631614336106527, -1.7900383790591, -1.0430855341111926, 0.3423974309557411, 0.1086707363296274, 0.2260814168522691, -0.4830039525727802, 0.2512179205507691, 0.15451069475756277, 1.5347840200564378, 1.5192421524123454, 1.2099398615739032, 0.4607843768620808, 1.2819682873955427, 1.3041930841665323, 0.8073184538578746, 0.691347250788444, 2.6828741538364986, 0.021546452071861958, -0.6048174682566767, 1.9514983368135617, 2.3098283018602705, -0.40693258223082135, -1.1691751152901873,
0.8276893008210489, 2.060558036682179, -0.23303947714726211, -0.5732552569873389, 1.91939849202629, 2.0780539968812324, -0.7499192205145742, -0.9247839326816437, 1.0460229933319543]

Mean 출력은 theta1~418, beta1~beta24 순서이다.

놀랍게도 (특히 acf 의) 상태가 많이 좋아졌다. 지난주의 기존 코드는 prior 로 improper uniform 을 준 것과 같은 상황임을 생각하면, normal 이 큰 일을 한 것 같다...

HW5-1. Hybrid Gibbs sampling

위의 MC_MH 클래스를 상속해 귀여운 수달 친구들 데이터에 맞게 세팅해서, Gibbs Sampler step 중 MH 를 돌릴 dimension(theta1,2)에 끼워 넣어 사용한다. 전체적으로는 지난 HW4 의 GibbsSampler 클래스의 sampler 부분을, 중간에 MH 알고리즘을 돌리도록 약간 고친 Seal_HybridGibbsSampler 클래스를 사용하는 것 말고는 HW4 의 GibbsSampler 클래스와 코드 구조가 같다.

N 과 alpha 들은 full-conditional distribution 을 이용하고, 그 다음에 theta1, 2 를 MH 로 30000 회 iteration 하여 생성한 후 15000 개를 버리고 남은 sample 들의 mean 값을 이용해 끼워넣는다.

이를 한 Gibbs sampler iteration 이라 할 때, 총 5000 회의 iteration 을 돌려서 5000 쌍의 sample 을 만들었다. 초기값은 $(N, \alpha_1, \dots, \alpha_7, \theta_1, \theta_2)$=(150, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.5, 0.5) 로 두었다.

MH proposal 은 exponential(1000)을 사용했는데, 이는 theta 의 parameter space 가 >0 인 real number 이라 normal 을 쓰기가 나빴기 때문이고, 또 theta posterior 를 볼 때 뒤에 붙는 부분이 exp 꼴이었기 때문이다. 꼴을 맞춰준 덕분에, posterior 와 proposal 이 나눠져서 날아간다. (위 문제와 같은 충격적인 일을 방지하기 위해, 일단 코드는 써 놓고 주석 처리해 두었다.)

이후 각 parameter 에 대해 histogram 과 acf 를 그렸고(아래 결과 있음), 그를 보고 burn-in 부분 cut 과 thinning 을 수행하였다.

[Python]

```
#python 3 file created by Choi, Seokjun

#Hybrid Gibbs sampler on Seal data

import time
from math import log, gamma
from random import seed, betavariate, expovariate
from functools import partial
from statistics import mean

import matplotlib.pyplot as plt
from numpy.random import negative_binomial
#음 negbin...

from HW5_MC_Core import MC_MH


class Seal_MC_MH_onlylast2dim(MC_MH):
    # MH로 돌릴 parameter를 위한 MH class
    #parameter vector order :
    #      [data                    ][param0 1      ] for hyperprior's view
    # 0   1   2   3   4   5   6   7   8         9
    # N   a1 a2 a3 a4 a5 a6 a7 theta1 theta2



    def seal_proposal_sampler(self, last):
        #do not depend on last : indep mcmc
        proposed_thetas = [expovariate(1000) for i in range(2)]
        #문제: theta1>0 theta2>0이어야함 - > exp쓰자(이유:posterior에 gamma - exp kernel꼴이 뒤에 곱해져있음))
        return proposed_thetas
```

```
    def seal_log_proposal_pdf(self, from_smpl, to_smpl):
        #do not depend on from_smpl
        #exp쓰면 target(posterior)에서도 같이 없애버리고 proposal pdf를 구현 안 해도 되나, 코드 직관성을 위해 일단 뒀음
        logval = 0
        # logval = -sum(to_smpl)/1000 #여기가 exp(상쇄 부분)
        return logval

    def seal_log_target_pdf(self, param_vec):
        #exp쓰면 target(posterior)에서도 해당 term을 없애버리고 proposal pdf를 구현 안 해도 되나, 코드 직관성을 위해  일단 뒀음
        thetas = param_vec
        logval = 7*log(gamma(thetas[0]+thetas[1]) / (gamma(thetas[0])+gamma(thetas[1])))
        for alpha in self.data:
            logval += (thetas[0]*log(alpha) + thetas[1]*log(1-alpha))
            # logval -= (thetas[0]+thetas[1])/1000 #여기가 exp (상쇄 부분)
        return logval

    def __init__(self, data, initial):
        #proposal_sampler_sd : 418+24 dim iterable object
        super().__init__(log_target_pdf=None, log_proposal_pdf=None, proposal_sampler=None, data=data, initial=initial)
        # self, log_target_pdf, log_proposal_pdf, proposal_sampler, data, initial
        self.proposal_sampler = self.seal_proposal_sampler
        self.log_proposal_pdf = self.seal_log_proposal_pdf
        self.log_target_pdf = self.seal_log_target_pdf

    def get_thetas_mean(self):
        #burnin자르고 / thining 이후 쓸것
        theta1 = []
        theta2 = []
        for smpl in self.MC_sample:
            theta1.append(smpl[0])
            theta2.append(smpl[1])
        return (mean(theta1), mean(theta2))



class Seal_HybridGibbsSampler:
    #앞 N, alpha1~alpha7은 일반적인 gibbs로, 그리고 theta1~2는 MH로 돌린다

    #다른데이터에도 일반적으로쓰려면 어떻게만들어야할지잘감이안온다
    #방법1: 아예 HW4의 gibbs sampler를 좀더 추상적으로 고쳐야할듯
    # 일반 gibbs + conddist로 업데이트할 dim 과 MH쓸 dim을 밖에서 받고
    # 해당 dim을위한 fullcond iterable object(element:callable, in:now param, out:proposal sample from sampler)를 받고
    # 또 해당 dim을위한 MH update instance를 받고
    # (묶어돌릴애들은 또 어떻게 받나....)
    # 이후 sampler를 아래와같이 오버라이드

    def __init__(self, initial_val, gibbsdim_full_conditional_sampler):
        self.initial = initial_val
        self.up_to_date = list(initial_val)
        self.num_dim = len(initial_val)
        self.full_conditional_sampler = gibbsdim_full_conditional_sampler
        self.samples = [initial_val]

    def sampler(self, num_MCiter):
```

```
        new_sample = [None for _ in range(self.num_dim)]

        # ordinary gibbs using full-conditional distribution
        for dim_idx in range(self.num_dim-2):
            new_val = self.full_conditional_sampler[dim_idx](self.full_conditional_sampler, up_to_date=self.up_to_date)

            new_sample[dim_idx] = new_val
            self.up_to_date[dim_idx] = new_val

        # hybrid part using MH
        MH_object = Seal_MC_MH_onlylast2dim(data=self.up_to_date[1:8], initial=self.up_to_date[8:10])
        #이걸 외부에서 받게만들어야할까? (나중에고치자)

        #pi(theta1,theta2)는 hyperprior이고 alpha가 거기서 뽑혀나오므로
        # alpha를 (실제 전체모델상에서 data는 아니지만 MH부분상에서는 data 역할임) data자리에 집어넣자
        #(코딩방식을 여러가지로 할 수 있는데...
        # param_vec을 떼는 작업을 어느 위치에서 처리하냐의 차이임
        # 다 파라메터로보고 data를 None을 넘긴다음 MC sampler에서 처리하느냐 아니면 그냥 위처럼 넘기느냐..

        MH_object.generate_samples(num_MCiter, verbose=False) #True로 두고 Mcmc 도는걸 구경할수있다(콘솔에 찍느냐 느려지므로 비추천)
        MH_object.burnin(num_MCiter//2) #일괄적으로 절반 자른다
        new_thetas = MH_object.get_thetas_mean()
        new_sample[-2:] = new_thetas
        self.up_to_date[-2:] = new_thetas

        new_sample = tuple(new_sample)
        self.samples.append(new_sample)

    def generate_samples(self, num_samples, num_MCiter=10000):
        start_time = time.time()
        for i in range(1, num_samples):
            self.sampler(num_MCiter=num_MCiter)
            if i%10==0:
                print("in gibbs step iteration", i, "/", num_samples)
        elap_time = time.time()-start_time
        print("in gibbs step iteration", num_samples, "/", num_samples, " done! (elapsed time for execution: ", elap_time//60,"min ", elap_time%60,"sec")


    def get_specific_dim_samples(self, dim_idx):
        if dim_idx >= self.num_dim:
            raise ValueError("dimension index should be lower than number of dimension. note that index starts at 0")
        return [smpl[dim_idx] for smpl in self.samples]

    def get_sample_mean(self):
        #burnin자르고 / thining 이후 쓸것
        mean_vec = []
        for i in range(self.num_dim):
            would_cal_mean = self.get_specific_dim_samples(i)
            mean_vec.append(mean(would_cal_mean))
        return mean_vec

    def show_hist(self):
        grid_column= int(self.num_dim**0.5)
```

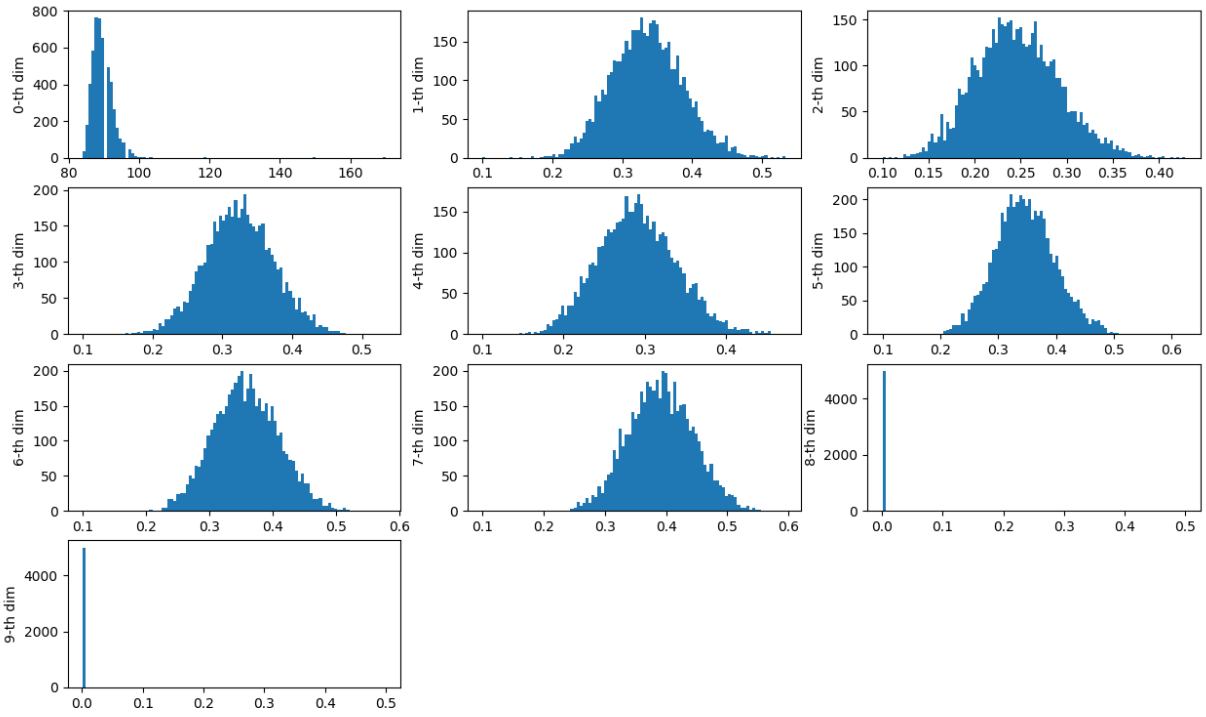
```
        grid_row = int(self.num_dim/grid_column)
        plt.figure(figsize=(5*grid_column, 3*grid_row))
        if grid_column*grid_row < self.num_dim:
            grid_row +=1
        for i in range(self.num_dim):
            plt.subplot(grid_row, grid_column, i+1)
            dim_samples = self.get_specific_dim_samples(i)
            plt.ylabel(str(i)+"-th dim")
            plt.hist(dim_samples, bins=100)
        plt.show()

    def get_autocorr(self, dim_idx, maxLag):
        y = self.get_specific_dim_samples(dim_idx)
        acf = []
        y_mean = mean(y)
        y = [elem - y_mean   for elem in y]
        n_var = sum([elem**2 for elem in y])
        for k in range(maxLag+1):
            N = len(y)-k
            n_cov_term = 0
            for i in range(N):
                n_cov_term += y[i]*y[i+k]
            acf.append(n_cov_term / n_var)
        return acf

    def show_acf(self, maxLag):
        grid_column= int(self.num_dim**0.5)
        grid_row = int(self.num_dim/grid_column)
        if grid_column*grid_row < self.num_dim:
            grid_row +=1
        subplot_grid = [i for i in range(maxLag+1)]
        plt.figure(figsize=(5*grid_column, 3*grid_row))
        for i in range(self.num_dim):
            plt.subplot(grid_row, grid_column, i+1)
            acf = self.get_autocorr(i, maxLag)
            plt.ylabel(str(i)+"-th dim")
            plt.ylim([-1,1])
            plt.bar(subplot_grid, acf, width=0.3)
            plt.axhline(0, color="black", linewidth=0.8)
        plt.show()

    def burnin(self, num_burn_in):
        self.samples = self.samples[num_burn_in-1:]

    def thinning(self, lag):
        self.samples = self.samples[::lag]

class FurSealPupCapRecap_FullCondSampler_with_thetas:
    #parameter vector order :
    # 0   1   2   3   4   5   6   7   8         9
    # N   a1 a2 a3 a4 a5 a6 a7 theta1 theta2
    NumberCaptured = (30,22,29,26,31,32,35)
    NumberNewlyCaught= (30,8,17,7,9,8,5)
    r = sum(NumberNewlyCaught) #84
```

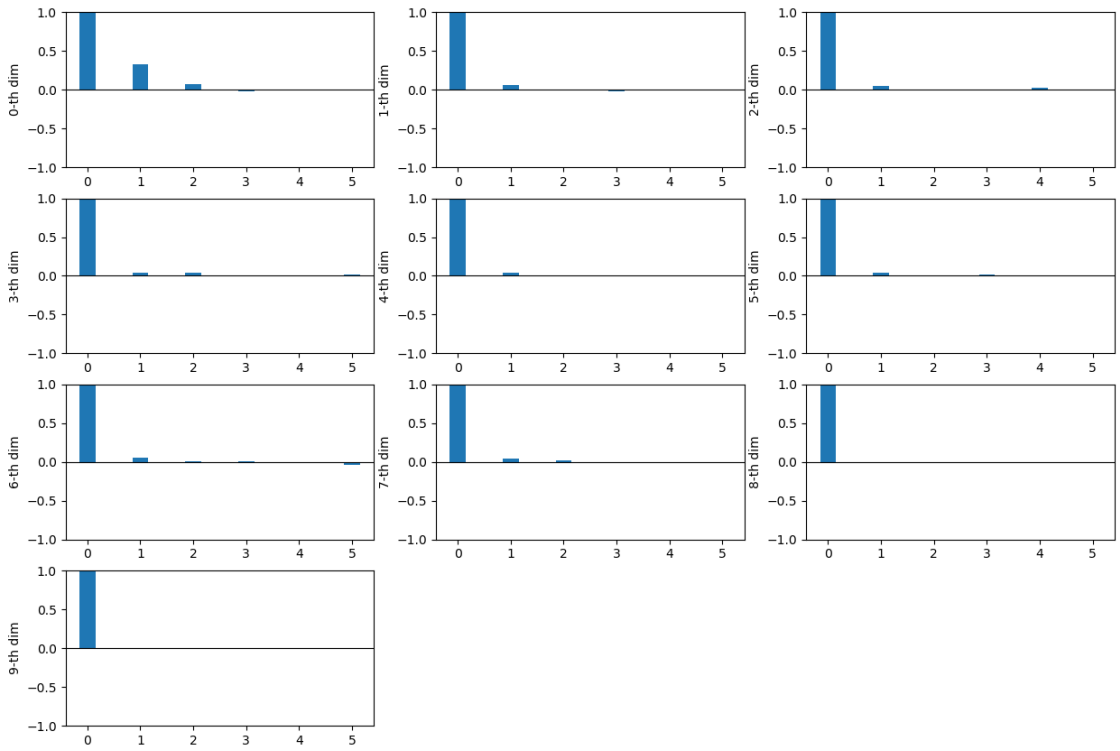
```
    def N(self, up_to_date):
        prod = 1
        for alpha in up_to_date[1:8]:
            prod *= 1-alpha
        # return negative_binomial(self.r+1, 1-prod) + self.r
        return negative_binomial(self.r, 1-prod) + self.r

    def a(self, up_to_date, a_idx):
        c = self.NumberCaptured[a_idx-1]
        alpha = c + up_to_date[8]
        beta = up_to_date[0] - c + up_to_date[9]
        return betavariate(alpha, beta)

    full_cond = [N]
    for i in range(1,8):
        full_cond.append(partial(a, a_idx=i))

    def __getitem__(self, index):
        return self.full_cond[index]

    def __len__(self):
        return len(self.full_cond)


if __name__=="__main__":

    seed(2019-311252)
    #ex1
    Seal_fullcond = FurSealPupCapRecap_FullCondSampler_with_thetas()
    # print(len(Seal_fullcond)) #8
    Seal_initial_values = (150, 0.1,0.1,0.1,0.1,0.1,0.1,0.1, 0.5, 0.5)
    Seal_Gibbs = Seal_HybridGibbsSampler(Seal_initial_values, Seal_fullcond)
    Seal_Gibbs.generate_samples(5000, num_MCiter=30000) #보고서쓸땐 좀 많이돌리자
    Seal_Gibbs.show_hist()
    Seal_Gibbs.show_acf(5)
    # print(Seal_Gibbs.get_sample_mean())

    #자르자
    Seal_Gibbs.burnin(1000)
    Seal_Gibbs.thinning(2)

    print(Seal_Gibbs.get_sample_mean())
    Seal_Gibbs.show_hist()
    Seal_Gibbs.show_acf(5)
```

한 gibbs iteration 마다 3 만회의 MH iteration 이 돌아가기 때문에, 생각보다 느리다.

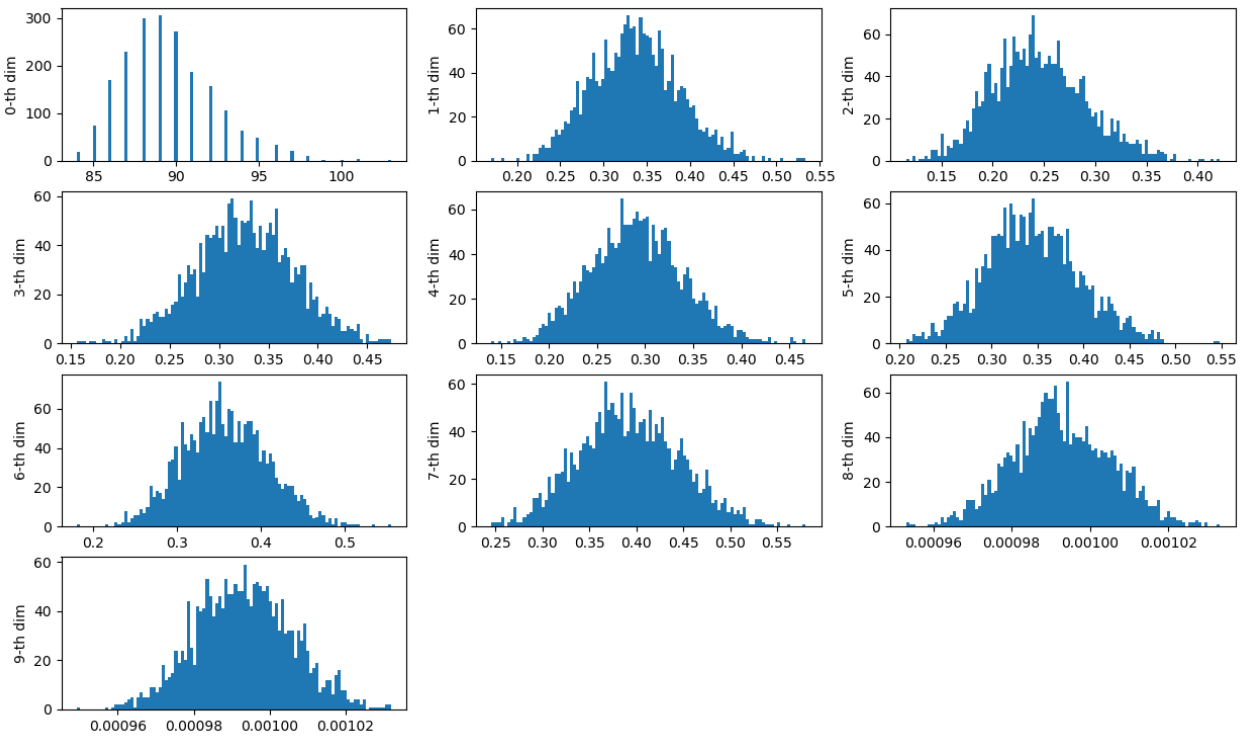
```
[Console]
(생략)
in gibbs step iteration 4990 / 5000
in gibbs step iteration 5000 / 5000   done! (elapsed time for execution:   23.0 min   41.46276831626892 sec
```

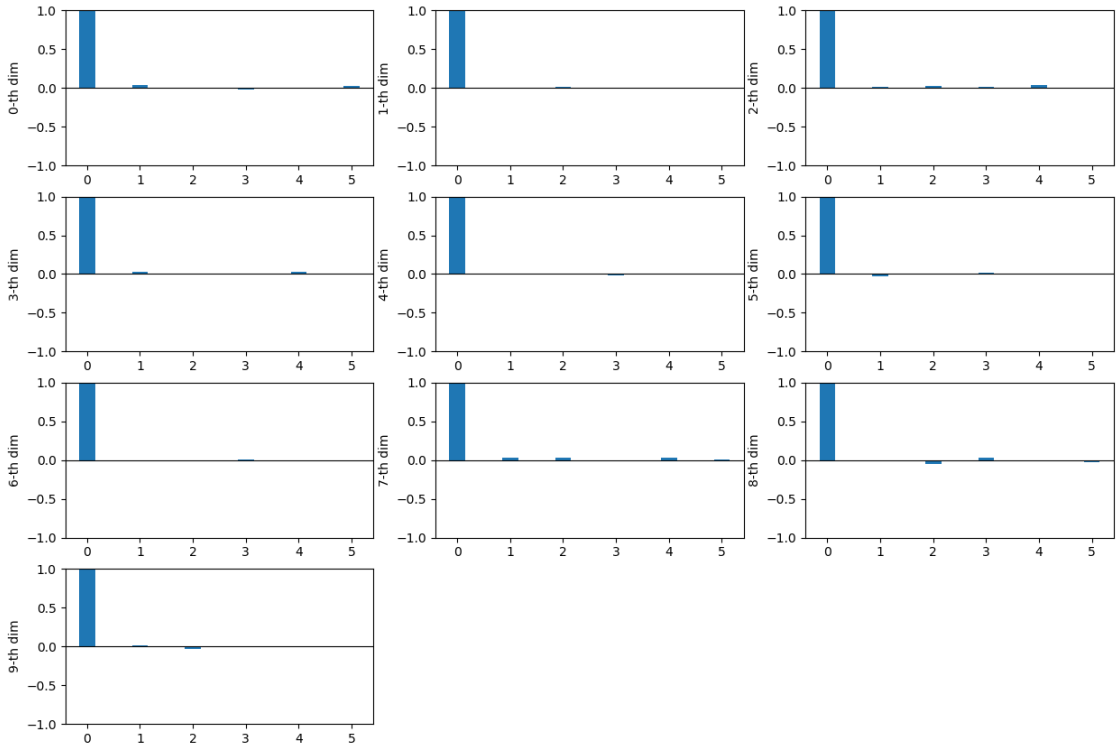
burn-in period 자르기 전, thinning 전의 결과는 다음과 같다. 인덱스는 $N, \alpha_1, \dots, \alpha_7, \theta_1, \theta_2$순이다.





theta 들의 히스토그램이 바 하나처럼 보이는데, 그 이유는 0.5 에서 시작했고, 수렴하며 0 으로 붙었기 때문이다. 이는 burnin period 를 버리면 좀 더 자세히 볼 수 있을 것이다. 또한 acf 의 0^{th} dim 인 N 이 첫 lag 에서 좀 살아있다. 그래서 burn-in 으로 1/5 에 해당하는 1000 개를 잘랐고, 이후 thinning 하여 2 개당 1 개의 sample 만 취했다. 이후 다시 histogram, acf 를 그리고 추가로 각 parameter 별로 mean 도 출력하였다. 결과는 다음과 같다.





[Console]

[89.53623188405797, 0.3367322124432841, 0.2451192784318848, 0.32616502722994023, 0.29067130121312473, 0.3451955196956703, 0.3576071583300228, 0.39252552963170706, 0.0009926652951337143, 0.0009932827543632206]

수렴 전 부분을 잘랐더니 이제 theta 의 히스토그램의 x 축 범위를 줄일 수 있게 되어서 결과가 잘 보인다. 계산된 평균을 보면, 귀여운 친구들이 평균적으로 89.5 마리정도밖에 없어 보인다는 것으로 추정되었다.

HW5-2. Using MALA, generate samples of μ,

with prior $\pi(\mu) \propto unif(-\infty, \infty)$, likelihood $y|\mu \sim N(\mu, 1)$, and data norm.txt,

and posterior $\pi(\mu|y) \propto \exp(-0.5 * \sum_{i=1}^{n}(x_i - \mu)^2)$ and of the derivative $\frac{\partial log\pi(\mu|y)}{\partial \mu} = \sum_{i=1}^{n}(x_i - \mu)$.

(1) 기존 MH algorithm 에서 sampler 와 rejection rule 의 ratio r 을 구하는 방식을 MALA 에 맞게 고쳐주자. 그 다음에는 (2)문제에 맞게 posterior 와 derivative 를 세팅한다. 그리고 (3) traceplot, histogram, acf plot 등을 위해 숙제 4 의 1,2 번 코드, 즉 1 dimension 에 맞는 코드를 재사용한다. 그리고 나서 (4) sample 을 generate 하고 결과를 본다. 수렴 위치가 맞는지 확인하기 위해 initial point 를 unif(-100,100)에서 랜덤으로 뽑은 8 개의 chain 을 만들고 비교한다.

(1), (3)은 MC_MALA_1dim class 정의에서, (2)는 MC_MALA_unif_normal_posterior_sampler class 정의에서, (4)는 main 블록과 multiproc_1unit_do_MALA 함수에서 수행한다.

step size 를 결정하는 σ^2는 일괄적으로 0.01 로 두었다. 각 chain 마다 20 만개 sample 을 만든다.

```
#python 3 file created by Choi, Seokjun

#using Metropolis-Adjusted Langevin Algorithm (MALA),
#get samples!



import time
from math import log
from random import uniform, normalvariate, seed
from statistics import mean
from os import getpid
import multiprocessing as mp

import matplotlib.pyplot as plt



class MC_MALA_1dim:
    def __init__(self, log_target_pdf, derivative_of_log_target, sigma_square, data, initial):
        self.log_target_pdf = log_target_pdf #arg (smpl)
        self.derivative_of_log_target = derivative_of_log_target #arg (smpl)
        #caution : it means grad(log(f(x)))!! (not log(grad(f(x))))

        self.sigma_square = sigma_square
        self.sigma = sigma_square**0.5

        self.data = data
        self.initial = initial

        self.MC_sample = [initial]

        self.num_total_iters = 0
        self.num_accept = 0

    def proposal(self, last):
        discretizer = normalvariate(0,1)
        proposing_state = last + 0.5 * self.sigma_square * self.derivative_of_log_target(last) + self.sigma * discretizer
        return proposing_state
```

```
def log_r_calculator(self, candid, last):
    log_r = (
        self.log_target_pdf(candid) - self.log_target_pdf(last)
        - (last - candid - 0.5*self.sigma_square*self.derivative_of_log_target(candid))**2 / (2*self.sigma_square)
        + (candid - last - 0.5*self.sigma_square*self.derivative_of_log_target(last))**2 / (2*self.sigma_square)
    )
    return log_r

def sampler(self):
    last = self.MC_sample[-1]
    candid = self.proposal(last) #기존 state 집어넣게
    unif_sample = uniform(0, 1)
    log_r = self.log_r_calculator(candid, last)
    # print(candid, log(unif_sample), log_r) #for debug
    if log(unif_sample) < log_r:
        self.MC_sample.append(candid)
        self.num_total_iters += 1
        self.num_accept += 1
    else:
        self.MC_sample.append(last)
        self.num_total_iters += 1

def generate_samples(self, num_samples, pid=None, verbose=True):
    start_time = time.time()
    for i in range(1, num_samples):
        self.sampler()
        if i%10000 == 0 and verbose and pid is not None:
            print("pid:",pid," iteration", i, "/", num_samples)
        elif i%10000 == 0 and verbose and pid is None:
            print("iteration", i, "/", num_samples)
    elap_time = time.time()-start_time
    if pid is not None and verbose: #여기 verbose 추가함
        print("pid:",pid, "iteration", num_samples, "/", num_samples,
        " done! (elapsed time for execution: ", elap_time//60,"min ", elap_time%60,"sec)")
    elif pid is None and verbose: #여기 verbose 추가함
        print("iteration", num_samples, "/", num_samples,
        " done! (elapsed time for execution: ", elap_time//60,"min ", elap_time%60,"sec)")

def burnin(self, num_burn_in):
    self.MC_sample = self.MC_sample[num_burn_in-1:]

def thinning(self, lag):
    self.MC_sample = self.MC_sample[::lag]

def get_autocorr(self, maxLag):
    y = self.MC_sample
    acf = []
    y_mean = mean(y)
    y = [elem - y_mean   for elem in y]
    n_var = sum([elem**2 for elem in y])
    for k in range(maxLag+1):
        N = len(y)-k
        n_cov_term = 0
```

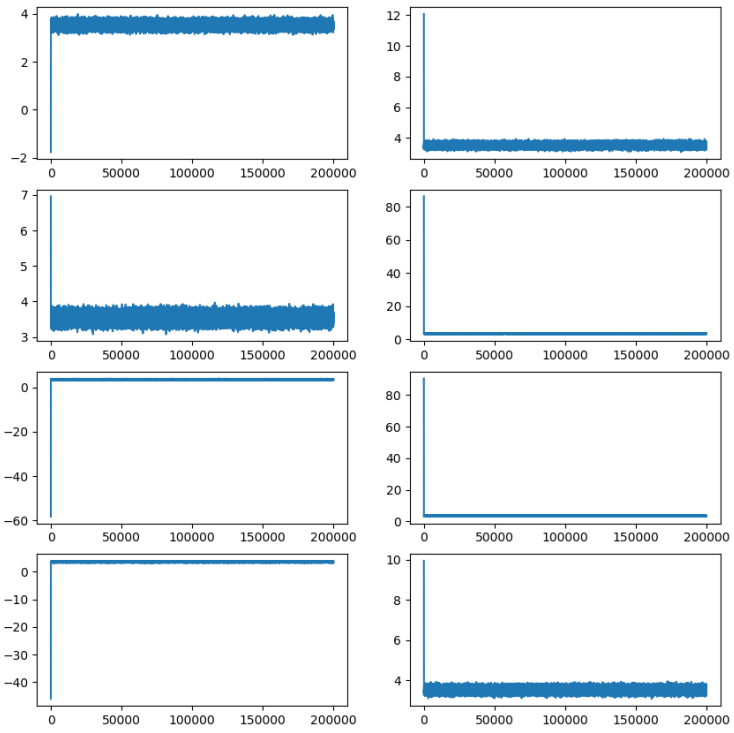
```
                for i in range(N):
                    n_cov_term += y[i]*y[i+k]
                acf.append(n_cov_term / n_var)
            return acf

        def show_traceplot(self, show=True):
            plt.plot(range(len(self.MC_sample)),self.MC_sample)
            if show:
                plt.show()

        def show_hist(self):
            plt.hist(self.MC_sample, bins=100)
            plt.show()

        def show_acf(self, maxLag, show=True):
            grid = [i for i in range(maxLag+1)]
            acf = self.get_autocorr(maxLag)
            plt.ylim([-1,1])
            plt.bar(grid, acf, width=0.3)
            plt.axhline(0, color="black", linewidth=0.8)
            if show:
                plt.show()

        def get_sample_mean(self, startidx=None):
            if startidx is not None:
                return mean(self.MC_sample[startidx:])
            else:
                return mean(self.MC_sample)

        def get_acceptance_rate(self):
            return self.num_accept/self.num_total_iters

class MC_MALA_unif_normal_posterior_sampler(MC_MALA_1dim):
        def UM_log_target_pdf(self, mu):
            log_val = 0
            for data_val in self.data:
                log_val += (data_val - mu)**2
            return -0.5*log_val

        def UM_derivative_of_log_target(self, mu):
            deriv_val = 0
            for data_val in self.data:
                deriv_val += (data_val - mu)
            return deriv_val

        def __init__(self, sigma_square, data, initial):
            super().__init__(self.UM_log_target_pdf,
            self.UM_derivative_of_log_target,
            sigma_square, data, initial)


def multiproc_1unit_do_MALA(result_queue, sigma_square, data, initial, num_iter):
        func_pid = getpid()
        print("pid: ", func_pid, "start!")
```

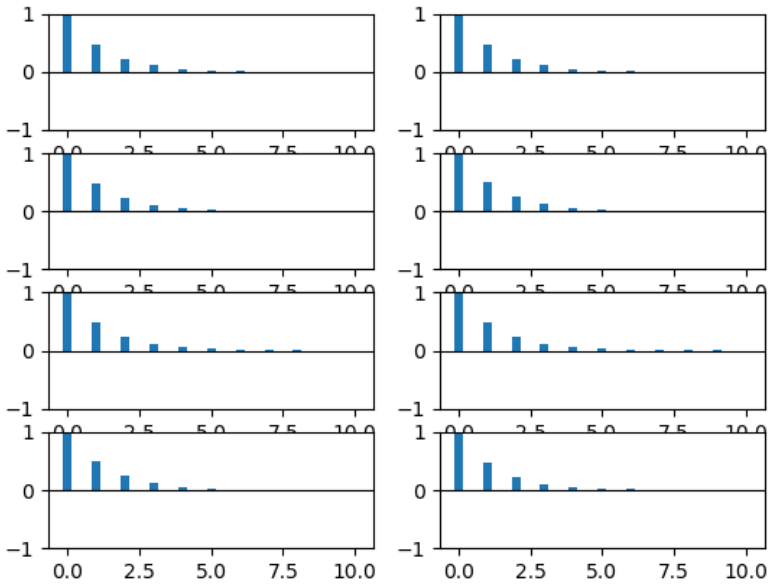
```
    Unit_MALA_Sampler = MC_MALA_unif_normal_posterior_sampler(sigma_square, data, initial)

    Unit_MALA_Sampler.generate_samples(num_iter, func_pid)
    acc_rate = Unit_MALA_Sampler.get_acceptance_rate()
    result_queue.put(Unit_MALA_Sampler)
    print("pid: ", func_pid, " acc_rate:",acc_rate)

if __name__ == "__main__":
    seed(2019311252)
    data = []
    with open("c:/gitProject/statComputing2/HW5/hw5_norm.txt","r", encoding="utf8") as f:
        while(True):
            line = f.readline()
            if not line:
                break
            data.append(float(line))



    core_num = 8 #띄울 process 수
    num_iter = 200000 #each MCMC chain's

    proc_vec = []
    proc_queue = mp.Queue()

    for _ in range(core_num):
        unit_initial = uniform(-100,100) #random으로 뽑자
        unit_sigma_square = 0.01 #그냥 일괄 설정

        unit_proc = mp.Process(target = multiproc_1unit_do_MALA, args=(proc_queue, unit_sigma_square, data, unit_initial, num_iter))
        proc_vec.append(unit_proc)



    for unit_proc in proc_vec:
        unit_proc.start()

    mp_result_vec = []
    for _ in range(core_num):
        each_result = proc_queue.get()
        # print("mp_result_vec_object:", each_result)
        mp_result_vec.append(each_result)

    for unit_proc in proc_vec:
        unit_proc.join()
    print("exit multiprocessing")



    #all traceplot
    grid_column= 2
    grid_row = int(core_num/2+0.5)
    plt.figure(figsize=(5*grid_column, 3*grid_row))
    for i, chain in enumerate(mp_result_vec):
        plt.subplot(grid_row, grid_column, i+1)
        chain.show_traceplot(False)
    plt.show()
```

```
    #all acf plot
    for i, chain in enumerate(mp_result_vec):
        plt.subplot(grid_row, grid_column, i+1)
        chain.show_acf(10, False)
    plt.show()
```

결과는 다음과 같다.

```
[Console]
(생략)
pid: 11520 iteration 200000 / 200000   done! (elapsed time for execution:   0.0 min   13.763561010360718 sec)
pid:   11520   acc_rate: 0.9197295986479932
pid: 10292 iteration 200000 / 200000   done! (elapsed time for execution:   0.0 min   13.925128698348999 sec)
pid:   10292   acc_rate: 0.9212546062730314
pid: 332 iteration 200000 / 200000   done! (elapsed time for execution:   0.0 min   13.988957166671753 sec)
pid:   332   acc_rate: 0.9212146060730304
pid: 732 iteration 200000 / 200000   done! (elapsed time for execution:   0.0 min   14.21235990524292 sec)
pid:   732   acc_rate: 0.9197145985729929
pid: 4764 iteration 200000 / 200000   done! (elapsed time for execution:   0.0 min   14.192414045333862 sec)
pid:   4764   acc_rate: 0.919999599998
pid: 5400 iteration 200000 / 200000   done! (elapsed time for execution:   0.0 min   14.296162843704224 sec)
pid:   5400   acc_rate: 0.9217746088730444
pid: 11364 iteration 200000 / 200000   done! (elapsed time for execution:   0.0 min   14.389884948730469 sec)
pid:   11364   acc_rate: 0.9204646023230116
pid: 3788 iteration 200000 / 200000   done! (elapsed time for execution:   0.0 min   14.432769775390625 sec)
pid:   3788   acc_rate: 0.9222896114480572
```





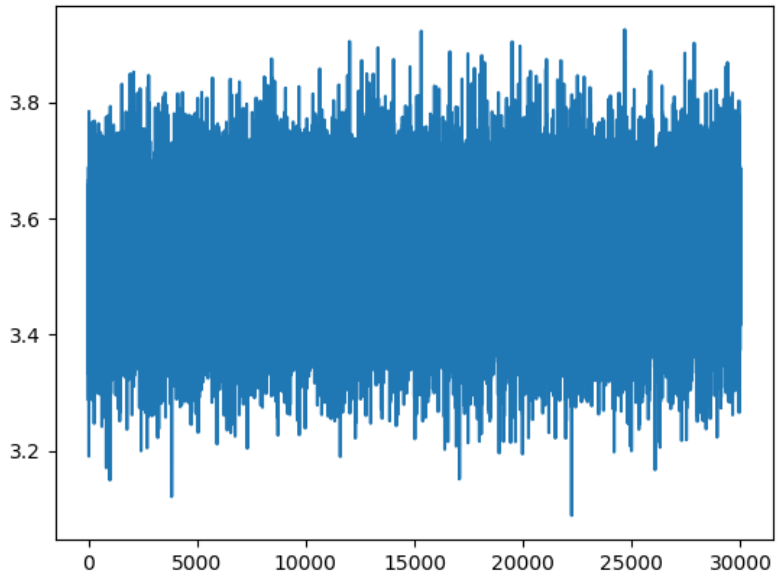
매우 빠르게 실행되고, 마찬가지로 매우 빠르게 수렴한다. 수렴 위치 또한 8 개의 chain 이 모두 같다. acf 를 볼 때, lag 5 정도면 autocorrelation 이 거의 0 에 가까워지므로, 5 개당 1 개의 sample 을 쓰겠다. Burn in period 로는 앞에 수백 개 정도만 버려도 괜찮을 듯하지만, sample 이 넉넉하므로 앞에 5 만개를 버리겠다.

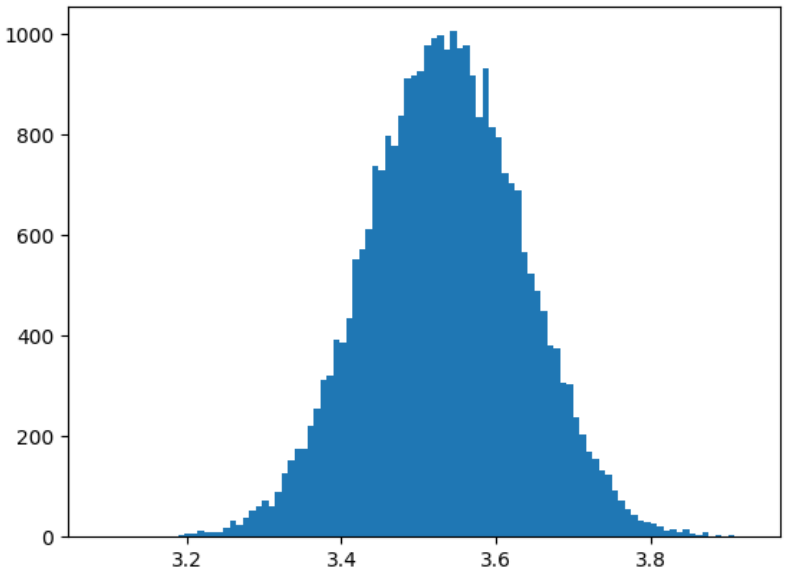
다음 코드를 main 에서 마저 실행한다.

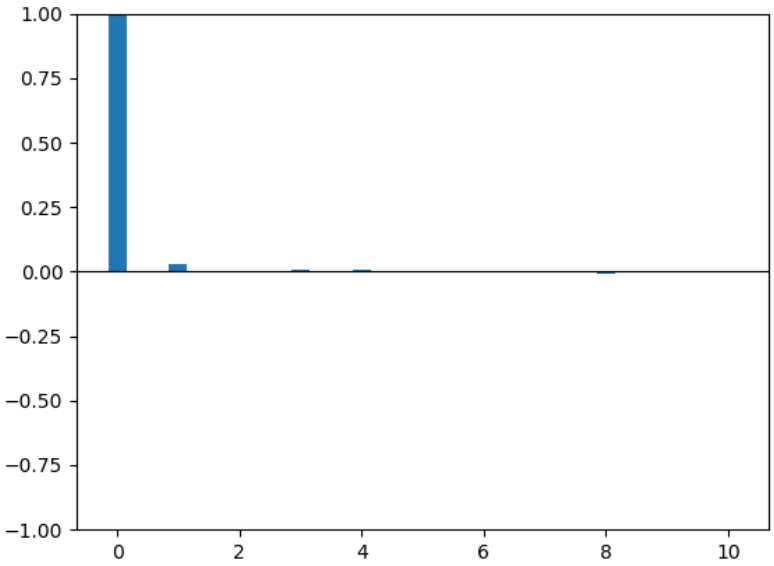
```
[Python]
    #all cut
    for chain in mp_result_vec:
        chain.burnin(50000)
        chain.thinning(5)

    #mean compare
    comp_mean_vec = []
    for chain in mp_result_vec:
        chain_mean = chain.get_sample_mean()
        print(chain_mean)
        comp_mean_vec.append(chain_mean)
    print("max diff: ", max(comp_mean_vec)-min(comp_mean_vec))
    print("**")

    # details of last chain
    OurSampler = mp_result_vec[-1]
    OurSampler.show_traceplot()
    OurSampler.show_hist()
    OurSampler.show_acf(10)
```

Sample 을 버린 후의 sample mean 비교 및 마지막 체인을 가지고 그린 traceplot, histogram, acf 를 첨부한다.

```
[Console]
3.5347360190393795
3.5337054673828194
3.5340144148366144
3.5336286988099945
3.533556832815129
3.5342401799502077
3.5337087759611783
3.5338847948642544
max diff:  0.0011791862242507278
```







8 개 chain 의 최대 차이가 0.0011791862242507278 로 매우 작고, 그래프들 또한 모범적인 모양이다.