Statistical Computing 2

숙제 4
2019년 가을학기
응용통계학과 석사과정 최석준

1.2.3. using MCMC, do below.

1,2,3 번은 공통적으로 MCMC 문제이므로, MH-MCMC 알고리즘을 추상적으로 구현한 아래 코드를 가져다 각 문제에 맞게 적용하며 반복적으로 쓸 것이다.

핵심 알고리즘은 MC_MH.sampler()와 그를 반복하는 MC_MH.generate_samples()에 구현되어 있다. 아예 log version 으로 짜서 뒤에서 각 경우마다 따로 overflow 및 underflow 를 잡을 필요를 줄였다. 또한 sample 의 dimension 에 영향을 받지 않는다.

사용 시 인스턴스를 __init__(파이썬 class 문법의 생성자)의 argument 에 맞게 넣거나, 혹은 상속하여 __init__이 받는 property 를 직접 method 로 override 하여 정의한다. 이후 해당 인스턴스에서 generate_samples() method 를 작동시킨다. 이후 필요에 맞게 burni()n 이나 thinning()을 호출하여 자르고 솎는다.

구체적인 용례는 아래의 각 문제 적용 케이스에 있다.

프로그래밍적인 추가 노트

1. argument 에대해 주석을 제대로 써놓아야하지만 안 써놓았기 때문에... (다른시험좀마저보고...쓰겠습니다...) 주의할 점은 __init__의 self 뒤 3 종으로 모두 함수를 받는다는 것이다. (심지어 argument 형식이 좀 까다로운데 아래에서 예시를 보고 맞추어 정의된 것을 넘겨야 한다.) 참고로 Python 에서 함수는 1 급 객체이므로, 굳이 C 계열처럼 포인터를 넘기지 않고도 다른 object 와 같이 변수에 넣어서 넘겨버릴 수 있다.

2. Python 이 private-public 접근제어를 허용하지 않으므로, 모든 method 와 property 에 접근 가능하여 더 정리가 안된다. 커뮤니티 관례적으로는 _을 붙이나, 일단은 그냥 내버려두었다.

[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)
        print()
        elap_time = time.time()-start_time
        if pid is not None:
            print("pid:",pid, "iteration", num_samples, "/", num_samples, " done! (elapsed time for execution: ", elap_time//60,"min ",
elap_time%60,"sec")
        else:
            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]
```

1. 2. Using Independent MCMC, with proposal=prior and given data, get samples from each posterior.

1. Poisson likelihood and lognormal(4,0.5) prior, obs={8,3,4,3,1,7,2,6,2,7}

2. Normal likelihood and Cauchy(0,1) prior, obs=norm.txt

위 MC_MH 를 상속하여 작업한다. 1 dim case 에서 사용할 추가 메소드를 MC_MH_1dim_withUtil 에서 정의하고, 이를 다시 상속받아 각 문제에 맞는 세팅을 한다.

해 둔 모습은 구체적으로 아래 EX1, EX2 class definition 에서 볼 수 있다.

참고로, Independent MCMC 를 이용하므로, MCMC rejection rule 을 계산할 때 posterior 의 prior term 과 proposal 이 각각 나눠지게 되므로, 해당 부분을 implementation 할 필요가 없다. 따라서 코드에서는 경우에 따라 적당한 값(0 혹은 1 등 결과의 유효성을 해치지 않는 값)을 return 하는 method 를 끼워 넣고 사용한다.

2 번을 위해 cauchy sampler 가 필요한데, python 에는 rcauchy()처럼 미리 짜여 있는 sampler 가 없으므로 (scipy 에 있나?) 과거 숙제 코드에서 가져온 후 callable 로 만들었다. 또한 Autocorrelation plot 을 보여주는 acf()같은 애도 없어서, 만들었다.

[Python]

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

#using "independent" Markov chain monte carlo,
#draw posterior samples



from math import log, exp, pi, factorial, tan
from random import seed, normalvariate, uniform
from statistics import mean
import multiprocessing as mp
import os

import matplotlib.pyplot as plt

from HW4_MC_Core import MC_MH

class MC_MH_1dim_withUtil(MC_MH):
    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):
        plt.plot(range(len(self.MC_sample)),self.MC_sample)
```

```
        plt.show()

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

    def show_acf(self, maxLag):
        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)
        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 EX1(MC_MH_1dim_withUtil):
    def EX1_proposal_sampler(self, last):
        #do not depend last
        param_mu=4
        param_sigma=0.5
        return exp(normalvariate(log(param_mu), param_sigma))

    def EX1_log_proposal_pdf(self, from_smpl, to_smpl):
        # Indep MCMC: do not depend on from_smpl
        # prior를 proposal로 쓰기때문에, r 계산시 proposal과 target의 prior가 나눠져서 날아감. 할필요가없음
        #(거꾸로 여길 구현하면, prior도 구현해야함)
        # param_mu=4
        # param_sigma=0.5
        # x=to_smpl
        # const = 1/(x*param_sigma*((2*pi)**0.5))
        # ker = exp(-(log(x)-param_mu)**2/(2*(param_sigma**2)))
        # return log(const*ker)
        return 0

    def EX1_log_likelihood(self, param_lambda):
        log_likelihood_val = 0
        for val in self.data:
            log_likelihood_val += log((param_lambda**val)*exp(-param_lambda)/factorial(val))
        return log_likelihood_val

    def EX1_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.
        return 0
```

```
    def EX1_log_target(self, param_vec):
        return self.EX1_log_likelihood(param_vec) + self.EX1_log_prior(param_vec)


    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.EX1_proposal_sampler
        self.log_proposal_pdf = self.EX1_log_proposal_pdf
        self.log_target_pdf = self.EX1_log_target


#from HW2,(좀 개조하긴함. __call__부분)
class CauchySampler:
    def __init__(self, param_loc, param_scale):
        if param_scale <= 0:
            raise ValueError("scale parameter should be >0")
        self.param_loc = param_loc
        self.param_scale = param_scale

    def sampler(self):
        unif_sample = uniform(0,1)
        return (self.param_scale * tan(pi * (unif_sample - 0.5)) + self.param_loc)

    def get_sample(self, number_of_smpl):
        result = []
        for _ in range(0, number_of_smpl):
            result.append(self.sampler())
        return result

    def __call__(self):
        return self.sampler()


class EX2(MC_MH_1dim_withUtil):
    def EX2_proposal_sampler(self, last):
        #do not depend last
        param_loc=0
        param_sigma=1
        EX2_proposal_sampler = CauchySampler(param_loc, param_sigma)
        return EX2_proposal_sampler()

    def EX2_log_proposal_pdf(self, from_smpl, to_smpl):
        # Indep MCMC: do not depend on from_smpl
        # prior를 proposal로 쓰기때문에, r 계산시 proposal과 target의 prior가 나눠져서 날아감. 할필요가없음
        #(거꾸로 여길 구현하면, prior도 구현해야함)
        return 0

    def EX2_log_likelihood(self, param_mu):
        param_sigma = 1
        log_likelihood_val = 0
        for val in self.data:
            const = 1/(param_sigma*((2*pi)**0.5))
```

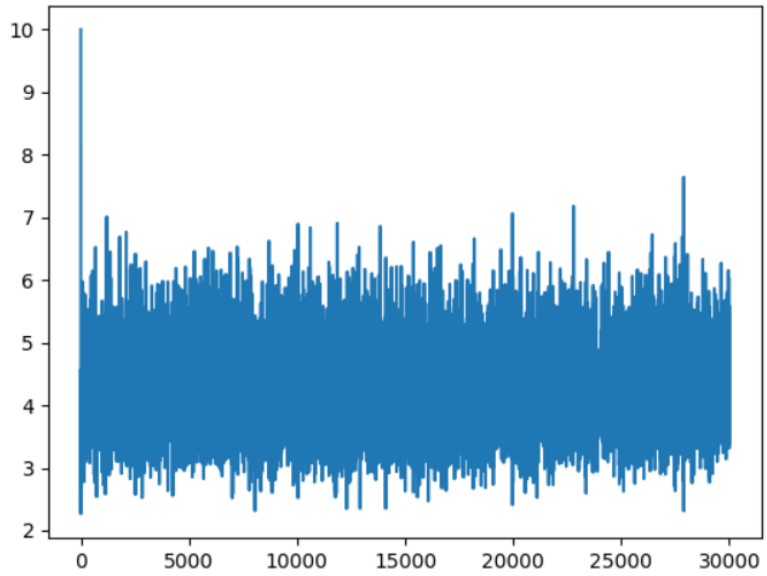
```
            ker = exp(-(val-param_mu)**2/(2*(param_sigma**2)))
            pdfval = const*ker
            if pdfval==0: #cauchy가 너무 꼬리가길어서 튄점에서 underflow나서 0나옴 log(0)==끔찍
                pdfval = 0.00000000000000001 #python 64bit minimum value(>0)(before floating point expression)
            log_likelihood_val += log(pdfval)
        return log_likelihood_val

    def EX2_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.
        return 0

    def EX2_log_target(self, param_vec):
        return self.EX2_log_likelihood(param_vec) + self.EX2_log_prior(param_vec)



    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.EX2_proposal_sampler
        self.log_proposal_pdf = self.EX2_log_proposal_pdf
        self.log_target_pdf = self.EX2_log_target
```

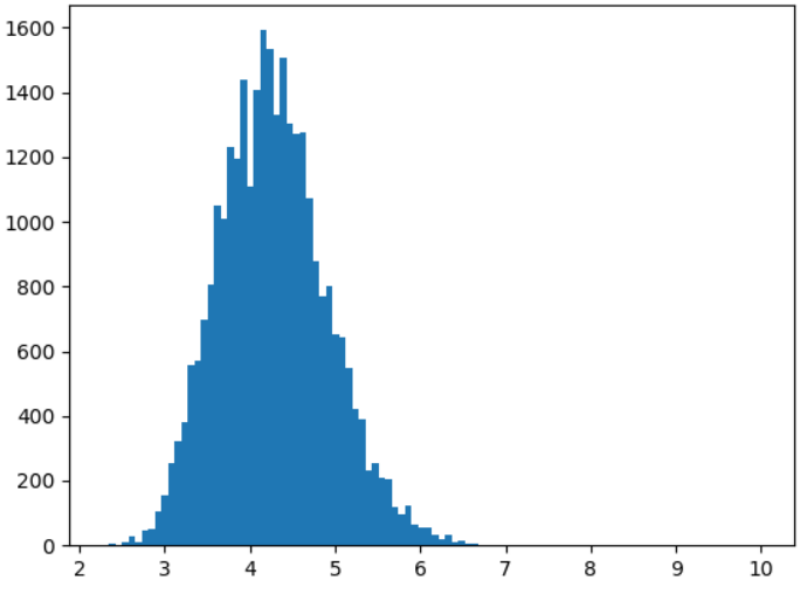
이제 직접 돌리자. 먼저 1 번 세팅에서 한번 돌려보자. 10 을 initial point 로 (임의로) 잡고, 3 만개를 만들었다. (이 수는 generate_samples 의 argument 를 조절하면 된다.)

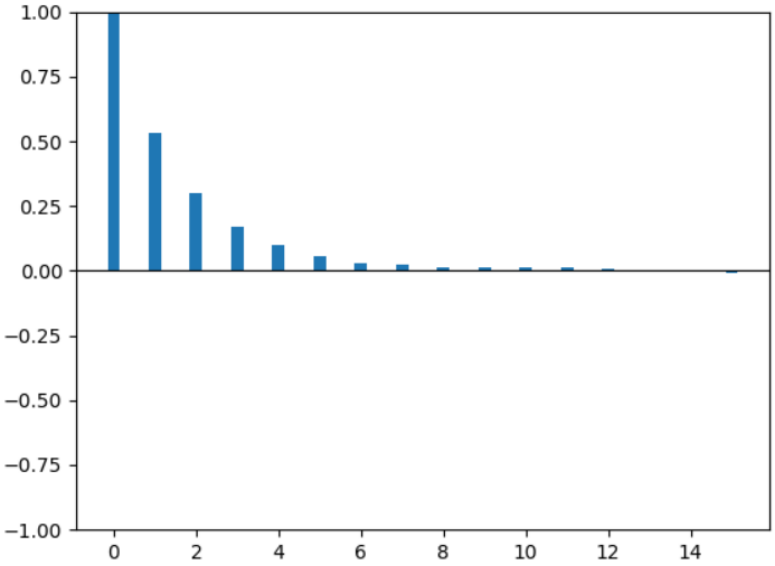
```
[Python]
if __name__ == "__main__":
    seed(2019311252)
    #ex1
    ex1_data = (8,3,4,3,1,7,2,6,2,7)

    Pois_Lognorm_model = EX1(ex1_data, 10)
    Pois_Lognorm_model.generate_samples(30000)
    print("acceptance rate: ", Pois_Lognorm_model.get_acceptance_rate())
    Pois_Lognorm_model.show_traceplot()
    Pois_Lognorm_model.show_hist()
    Pois_Lognorm_model.show_acf(15)
```

결과는 다음과 같다. 마지막 세 줄이 차례로 그림을 출력하고, 순서대로 traceplot, histogram, acf plot 이다.

```
[Console]
iteration 500 / 30000
iteration 1000 / 30000
(...생략)
iteration 30000 / 30000   done! (elapsed time for execution:   0.0 min   0.47867465019226074 sec
acceptance rate:   0.3587119570652355
```





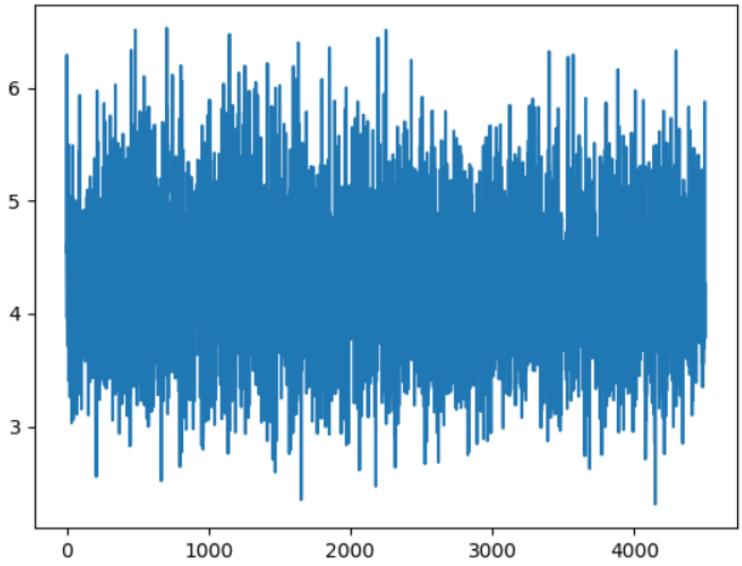


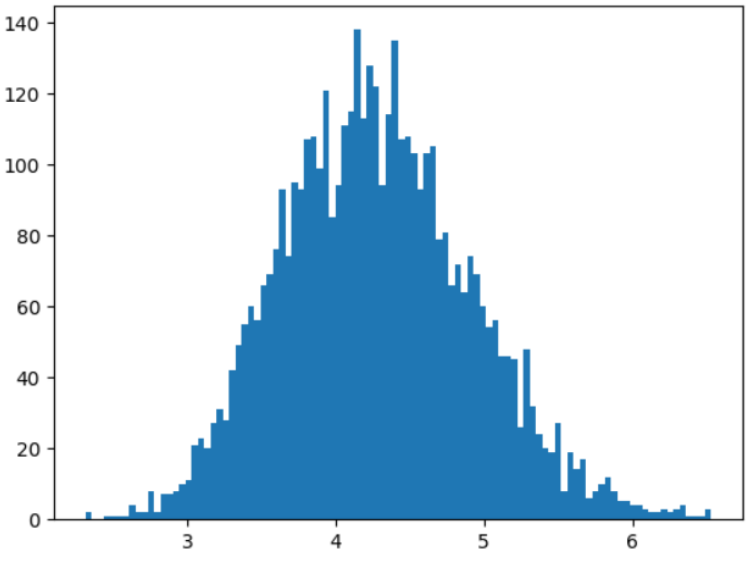
Acceptance rate 0.358 로는 양호한 범위 내에 있는 것으로 보인다. Traceplot 을 보면 매우 빠르게 수렴하는 것을 알 수 있고, 따라서 burnin period 를 짧게 잡아도 괜찮을 듯하다. 하지만 acf plot 을 볼 때, autocorrelation 이 좀 맘에 안 드는 속도로 떨어지므로, thinning 을 하자. 앞 3 천개를 자른 후 남은 샘플을 6 개마다 하나씩 쓰겠다.

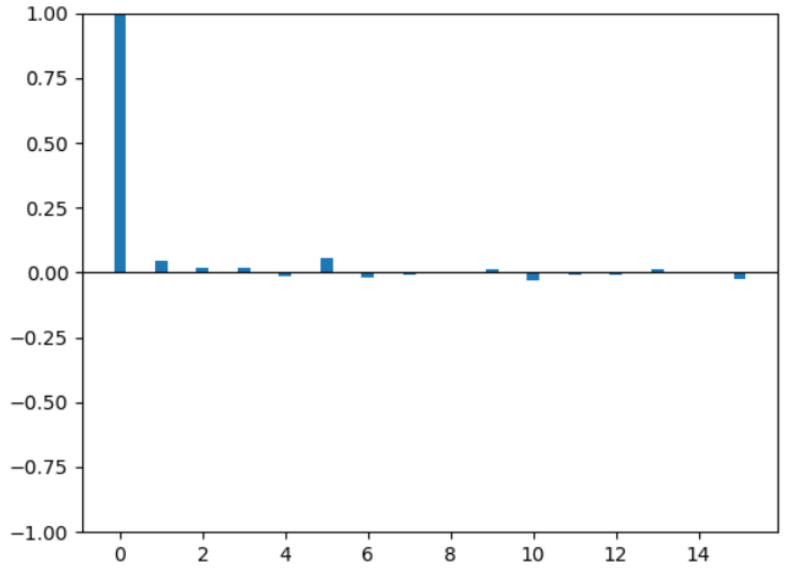
위의 main 블록에서 계속하여 작업한다. (Python 이므로 indent 를 매우 주의해야 한다.)

```
[Python]
    Pois_Lognorm_model.burnin(3000) #짧게잘라도될듯
    Pois_Lognorm_model.thinning(6)
    Pois_Lognorm_model.show_traceplot()
    Pois_Lognorm_model.show_hist()
    Pois_Lognorm_model.show_acf(15)
```

마지막 세 줄은 다시 method 이름에 맞는 그림을 각각 출력한다.







이제 acf 가 마음에 든다.

Histogram 을 보면 약 4.2~3 근처가 평균임을 알 수가 있다. 하지만 아직 여기가 정말 제대로 된 수렴점인지 아닌지는 확신할 수가 없다. Initial point 를 바꾸어 가며 chain 을 여러 개 만들어 평균을 비교해보자.

병렬처리를 위해 다음 세팅을 한다. 각 process 마다 한 chain 을 생성할 것이다. burnin 및 thinning 을 각각 traceplot 을 보며 설정해주기 귀찮으므로(...) 각 chain 마다 30000 개 생성 후, 그냥 일괄적으로 앞 절반을 버리겠다. thinning 은 하지 않겠다.

아래 함수를 전역에 선언한다. (python multiprocessing module 특성 상 선언 위치를 무조건 지켜줘야 한다.)

```
[Python]
#for multiprocessing
#each proc's details are adjusted after see 1 test chain. (see main part's code first)
def multiproc_1unit_do_EX1(result_queue, data, initial, num_iter):
    func_pid = os.getpid()
    print("pid: ", func_pid, "start!")
    UnitMcSampler = EX1(data, initial)

    UnitMcSampler.generate_samples(num_iter, func_pid)
    UnitMcSampler.burnin(num_iter//2) #걍 반 자르자 파라메터로 또 받기 귀찮
    acc_rate = UnitMcSampler.get_acceptance_rate()
    result_queue.put(UnitMcSampler)
    print("pid: ", func_pid, " acc_rate:",acc_rate)
```

이제 프로그램의 main 부분에서 정말 MCMC 를 돌릴 것이다. 체인마다 각각의 initial point 로 unif(0,10)에서 랜덤으로 뽑힌 값을 사용한다.

(indent 에 주의하라! 마찬가지로 python multiprocessing module 특성 상 이를 if __name__=="__main__"으로 둘러싸여진 main part 에 꼭 두어야 한다.)

```
[Python]
    ########verify convergence with random initial points
    core_num = 8 #띄울 process 수
    EX1_num_iter = 30000 #each MCMC chain's

    EX1_proc_vec = []
    EX1_proc_queue = mp.Queue()
```

```
    for _ in range(core_num):
        unit_initial = uniform(0,100) #random으로 뽑자
        unit_proc = mp.Process(target = multiproc_1unit_do_EX1, args=(EX1_proc_queue, ex1_data, unit_initial, EX1_num_iter))
        EX1_proc_vec.append(unit_proc)


    for unit_proc in EX1_proc_vec:
        unit_proc.start()

    EX1_mp_result_vec = []
    for _ in range(core_num):
        each_result = EX1_proc_queue.get()
        # print("EX1_mp_result_vec_object:", each_result)
        EX1_mp_result_vec.append(each_result)

    for unit_proc in EX1_proc_vec:
        unit_proc.join()
    print("exit multiprocessing")

    print("**EX1: compare mean vec**")
    EX1_comp_mean_vec = []
    for chain in EX1_mp_result_vec:
        chain_mean = chain.get_sample_mean()
        print(chain_mean)
        EX1_comp_mean_vec.append(chain_mean)
    print("max diff: ", max(EX1_comp_mean_vec)-min(EX1_comp_mean_vec))
    print("**")
```

출력은 다음과 같다. (pid 는 process id 로, os 가 알아서 배정하며, 떄문에 돌릴 때문에 값이 다르다.) 변수명상 core_num 은 컴퓨터 운영체제 위에 띄울 process 수인데, 여러 개를 띄우면 똑똑한 os 가 알아서 cpu 코어를 배정해 줄 것이다. cpu 의 코어 수를 고려하며, 시스템 리소스 모니터의 cpu 사용률을 보며 적당히 설정하자. (아래 코드에선 8 로 설정하였다.)

편의를 위해서 core_num 개만큼 chain 을 만들겠다. (더 만들게 할 수도 있는데... ...기다림을 즐길 수 없다...)

```
[Console]
pid:   948 start!
pid:   2812 start!
pid:   9540 start!
pid:   3316 start!
pid:   15284 start!
pid:   10848 start!
pid:   12344 start!
pid:   1224 start!
pid: 9540   iteration 500 / 30000
pid: 10848   iteration 500 / 30000
pid: 3316   iteration 500 / 30000
pid: 2812   iteration 500 / 30000
pid: 948   iteration 500 / 30000
pid: 15284   iteration 500 / 30000
pid: 12344   iteration 500 / 30000
pid: 1224   iteration 500 / 30000
pid: 3316   iteration 1000 / 30000
```

```
(생략)
pid: 10848 iteration 30000 / 30000   done! (elapsed time for execution:   0.0 min   1.468111276626587 sec
pid: 3316   iteration 29500 / 30000
pid:   10848   acc_rate: 0.3569785659521984
(생략)
pid: 948 iteration 30000 / 30000   done! (elapsed time for execution:   0.0 min   1.4940414428710938 sec
pid: 15284   iteration 28500 / 30000
pid: 1224 iteration 30000 / 30000   done! (elapsed time for execution:   0.0 min   1.493046760559082 sec
pid: 9540   iteration 29500 / 30000
pid: 2812   iteration 29500 / 30000
pid:   948   acc_rate: 0.36081202706756893
pid:   1224   acc_rate: 0.3610453681789393
pid: 12344 iteration 30000 / 30000   done! (elapsed time for execution:   0.0 min   1.5030193328857422 sec
pid: 3316 iteration 30000 / 30000   done! (elapsed time for execution:   0.0 min   1.5030186176300049 sec
pid:   12344   acc_rate: 0.3624120804026801
pid:   3316   acc_rate: 0.36291209706990235
pid: 2812 iteration 30000 / 30000   done! (elapsed time for execution:   0.0 min   1.5209715366363525 sec
pid:   2812   acc_rate: 0.3600786692889763
pid: 9540 iteration 30000 / 30000   done! (elapsed time for execution:   0.0 min   1.5259573459625244 sec
pid:   9540   acc_rate: 0.35861195373179106
pid: 15284   iteration 29000 / 30000
pid: 15284   iteration 29500 / 30000
pid: 15284 iteration 30000 / 30000   done! (elapsed time for execution:   0.0 min   1.5947744846343994 sec
pid:   15284   acc_rate: 0.3629787659588653
exit multiprocessing
**EX1: compare mean vec**
4.265154582431186
4.2789953099199725
4.278176430065
4.263834389454827
4.271748969075559
4.260749700087412
4.277693298090785
4.2854430155279095
max diff:   0.0246933154404978
```

1 개 chain 돌릴 때 0.5 초 걸렸는데 8 개 돌리는데 1.6 초 걸렸으니 나름 이득을 보았다. (1 초....)

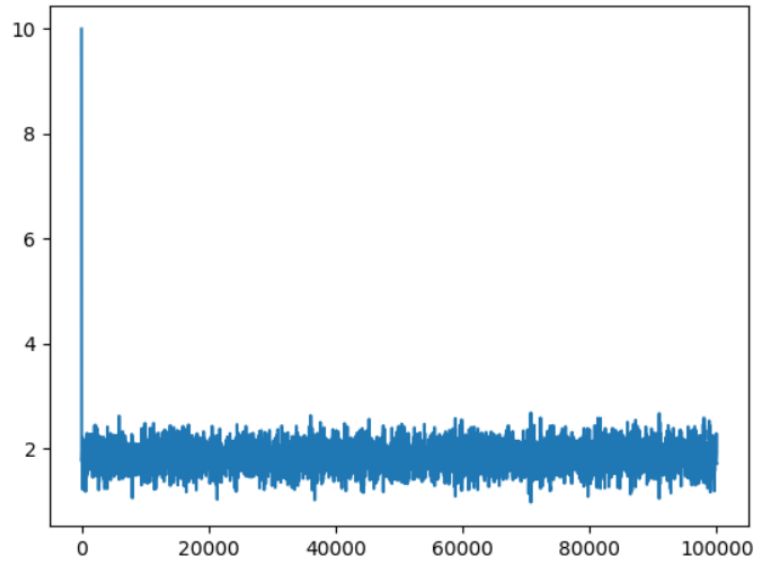
출력의 마지막 부분에서 각 chain 의 burin 버린 후 sample mean 값을 각각 출력하고, 그 최고 차이를 구해 보여주고 있다. 제일 크게 차이나는 경우가 0.0246933154404978 정도의 차이를 보여주므로, 얼추 수렴점이 여기가 맞다는 것으로 생각해볼 만하다.

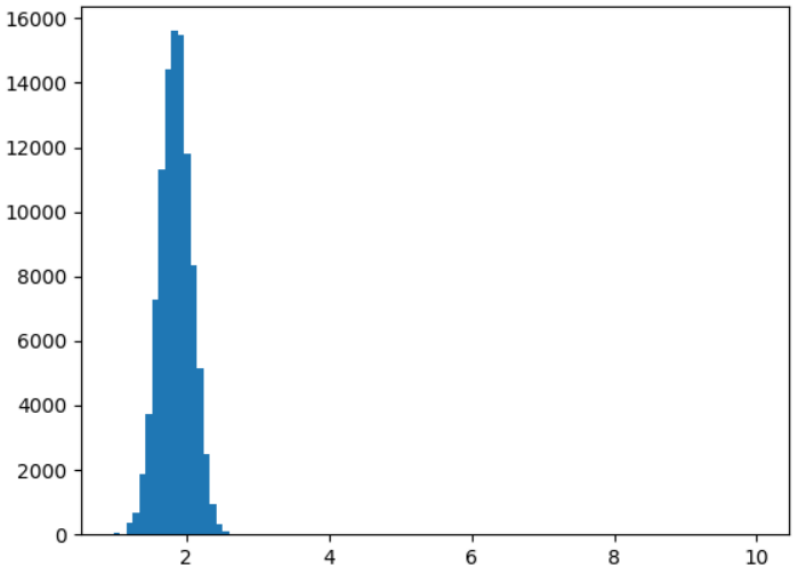
이제 2 번을 돌리자. 구체적인 세팅은 위 EX2 를 선언하는 class 에서 다 해 두었으므로, 그냥 인스턴스만 찍어서 돌리면 된다. 다음 코드를 main 에서 실행하여 먼저 한 chain 의 결과를 보자. Initial value 는 또 임의로 10 으로 설정하였다. 10 만개 만들자.

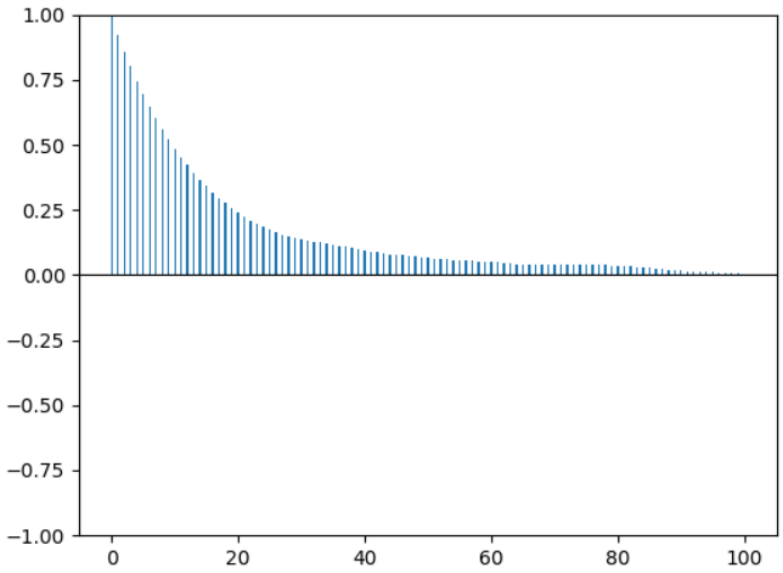
```
[Python]
ex2_data = (2.983, 1.309, 0.957, 2.16, 0.801, 1.747, -0.274, 1.071, 2.094, 2.215,
            2.255, 3.366, 1.028, 3.572, 2.236, 4.009, 1.619, 1.354, 1.415, 1.937)
    Norm_Cauchy_model = EX2(ex2_data, 10)
    Norm_Cauchy_model.generate_samples(num_samples=100000)
    print("acceptance rate: ", Norm_Cauchy_model.get_acceptance_rate())
    Norm_Cauchy_model.show_traceplot()
    Norm_Cauchy_model.show_hist()
    Norm_Cauchy_model.show_acf(100) #음 문제가많군
```

결과는 다음과 같다.

```
[Console]
iteration 500 / 100000
(생략)
iteration 99500 / 100000
iteration 100000 / 100000   done! (elapsed time for execution:   0.0 min   6.255975008010864 sec
acceptance rate:   0.052260522605226054
```



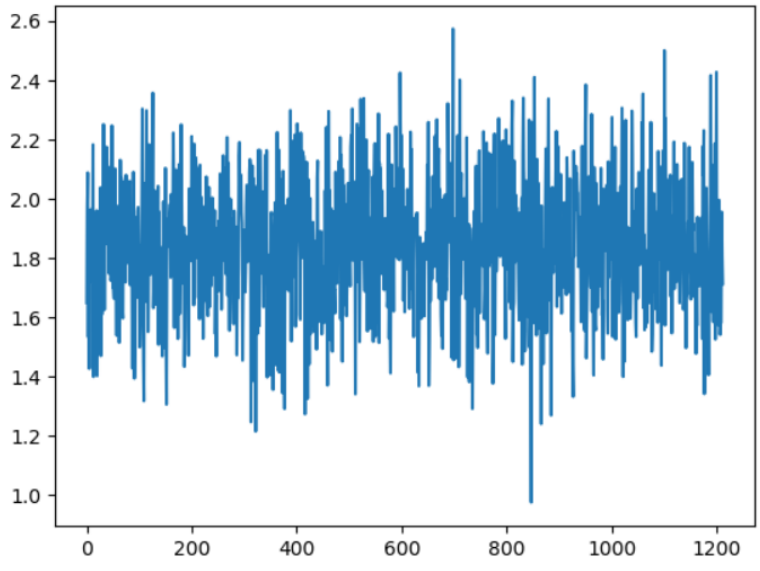


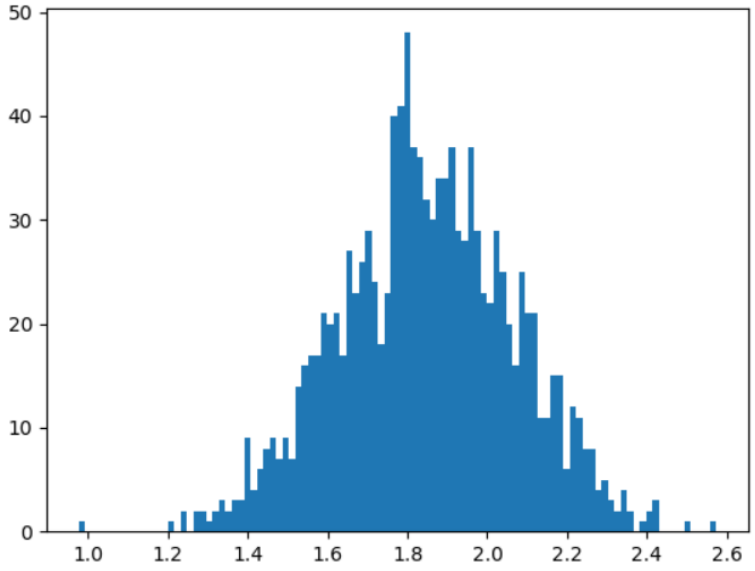


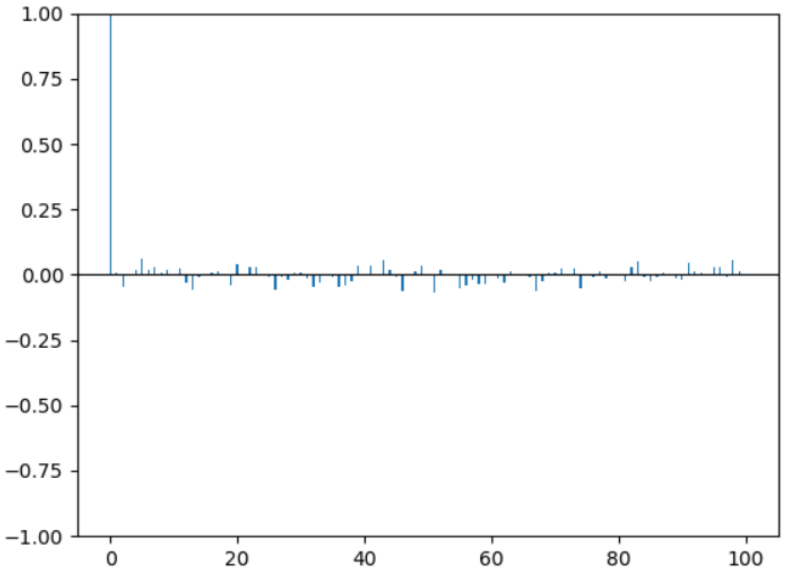
마지막 acf plot 을 보면, dependency 가 살아있는 lag 이 엄청나게 길다! burnin 으로 3 천개 버리고, 과감하게 thinning 을 해서 80 개마다 1 개씩만 쓰겠다. 다음 코드를 실행하자.

```
[Python]
    Norm_Cauchy_model.burnin(3000)
    Norm_Cauchy_model.thinning(80) #과감하게 자릅시다
    Norm_Cauchy_model.show_traceplot()
    Norm_Cauchy_model.show_hist()
    Norm_Cauchy_model.show_acf(100)
```

결과는 다음과 같다.







많이 버려서 sample 이 1200 개정도밖에 안 남았기는 했지만, histogram 과 acf 를 보면 아주 만족스러우므로, 아까워하지 말고 버리는게 나은 것 같다. 아마 유효 sample 수를 기준으로 보면 크게 손해보진 않았을 것이다.

이번에도 마찬가지로, 히스토그램에서 보이는 1.8 정도가 수렴점이 맞는지 여러 번 돌려 확인해보자. 각 chain 당 10 만개를 만들 것이고, initial point 로는 unif(0,100)에서 랜덤으로 뽑힌 값을 사용한다. 각 체인의 traceplot 을 또 각각 보기 귀찮으므로, 돌린 후 burnin 으로 절반 버리고, thinning 을 80 개마다 1 개 뽑는 정도로 하겠다.

다음을 전역에 선언하고,

```
[Python]
def multiproc_1unit_do_EX2(result_queue, data, initial, num_iter):
    func_pid = os.getpid()
    print("pid: ", func_pid, "start!")
    UnitMcSampler = EX2(data, initial)

    UnitMcSampler.generate_samples(num_iter, func_pid)
    UnitMcSampler.burnin(num_iter//2) #여기도..반자르자..
    UnitMcSampler.thinning(80) #하나돌려서그려보니까 autocorr이 잘 안죽음
    acc_rate = UnitMcSampler.get_acceptance_rate()
    result_queue.put(UnitMcSampler)
    print("pid: ", func_pid, " acc_rate:",acc_rate)
```

main 에서 아래 코드를 돌린다.

```
[Python]
    ########verify convergence with random initial points
    core_num = 8 #띄울 process 수
    EX2_num_iter = 100000 #each MCMC chain's

    EX2_proc_vec = []
    EX2_proc_queue = mp.Queue()

    for _ in range(core_num):
        unit_initial = uniform(0,100) #random으로 뽑자
        unit_proc = mp.Process(target = multiproc_1unit_do_EX2, args=(EX2_proc_queue, ex2_data, unit_initial, EX2_num_iter))
        EX2_proc_vec.append(unit_proc)



    for unit_proc in EX2_proc_vec:
        unit_proc.start()

    EX2_mp_result_vec = []
    for _ in range(core_num):
        each_result = EX2_proc_queue.get()
        # print("EX2_mp_result_vec_object:", each_result)
        EX2_mp_result_vec.append(each_result)

    for unit_proc in EX2_proc_vec:
        unit_proc.join()
    print("exit multiprocessing")

    print("**EX2: compare mean vec**")
    EX2_comp_mean_vec = []
    for chain in EX2_mp_result_vec:
        chain_mean = chain.get_sample_mean()
        print(chain_mean)
        EX2_comp_mean_vec.append(chain_mean)
    print("max diff: ", max(EX2_comp_mean_vec)-min(EX2_comp_mean_vec))
    print("**")
```

결과로, 다음 출력을 얻는다.

```
[Console]
(생략)
pid: 16500 iteration 100000 / 100000   done! (elapsed time for execution:   0.0 min   15.010056495666504 sec
pid:   16500   acc_rate: 0.052980529805298056
pid: 16172   iteration 99000 / 100000
pid: 15656   iteration 97000 / 100000
pid: 11516   iteration 98000 / 100000
(생략)
pid: 4132 iteration 100000 / 100000   done! (elapsed time for execution:   0.0 min   15.145333766937256 sec
pid:   4132   acc_rate: 0.05189051890518905
pid: 1168   iteration 98000 / 100000
pid: 1844   iteration 98000 / 100000
pid: 15656   iteration 98000 / 100000
pid: 16172 iteration 100000 / 100000   done! (elapsed time for execution:   0.0 min   15.203914165496826 sec
```

```
pid:   16172   acc_rate: 0.05161051610516105
pid: 18408   iteration 98000 / 100000
pid: 11516   iteration 99000 / 100000
(생략)
pid: 11516 iteration 100000 / 100000   done! (elapsed time for execution:   0.0 min   15.296074628829956 sec
pid:   11516   acc_rate: 0.05134051340513405
(생략)
pid: 1168 iteration 100000 / 100000   done! (elapsed time for execution:   0.0 min   15.42382550239563 sec
pid:   1168   acc_rate: 0.05127051270512705

pid: 1844 iteration 100000 / 100000   done! (elapsed time for execution:   0.0 min   15.430808544158936 sec
pid:   1844   acc_rate: 0.052050520505205054
pid: 15656 iteration 100000 / 100000   done! (elapsed time for execution:   0.0 min   15.426560163497925 sec
pid: 18408 iteration 100000 / 100000   done! (elapsed time for execution:   0.0 min   15.348454475402832 sec
pid:   15656   acc_rate: 0.05341053410534105
pid:   18408   acc_rate: 0.05212052120521205
exit multiprocessing
**EX2: compare mean vec**
1.846570975223058
1.8589557592604278
1.8682294251372538
1.8499513207628464
1.8536817893788475
1.8579172174439833
1.8445872802787493
1.827357551364237
max diff:   0.04087187377301671
**
```

한 체인 생성시 6.25 초 걸렸던 것이 8 chain 에는 15.35 초 걸렸으므로, 열심히 몇 분 들여 짠 병렬처리 코드의 이득을 약 30 초정도 보았다. (1 초보단 보람이 있다...)

결과의 마지막 부분은 마찬가지로 각 chain 에서 생성된 sample 에서 구한 sample mean 값을 보여준다. 최고로 차이나는 경우가 0.04 정도이므로, 여기가 제대로 된 수렴점이 맞다고 할 수 있겠다.

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

계속하여 MCMC를 이용하는 문제이므로, 앞 문제에서 소개한 MC_MH class를 여기에서도 가져다 사용한다. 문제 상황 세팅 및 뒤에서 쓸 메소드들 몇 종을 상속한 클래스에 overriding 혹은 새로 정의하고 시작하자. 또한 몇몇 분석용 method도 정의한다. (유틸리티(?) 함수를 분리할까 했지만... 일단은 한꺼번에 넣어두었다.)

[Python]

```
#python 3 file created by Choi, Seokjun
#using Markov chain monte carlo,
#get samples of Item response model's parameters
#요구사항 :
    # beta_i 는 posterior mean을 보일 것
    # theta_i는 histogram 그릴 것


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.
```

```
        return 0

    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 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()
```

이번에는 먼저 여러 체인을 한꺼번에 생성한 후, 수렴결과를 비교한 후 그 중 한 체인을 보겠다. 다음은 global 에 선언할 병렬처리용 함수이다.

```
[Python]
#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)
```

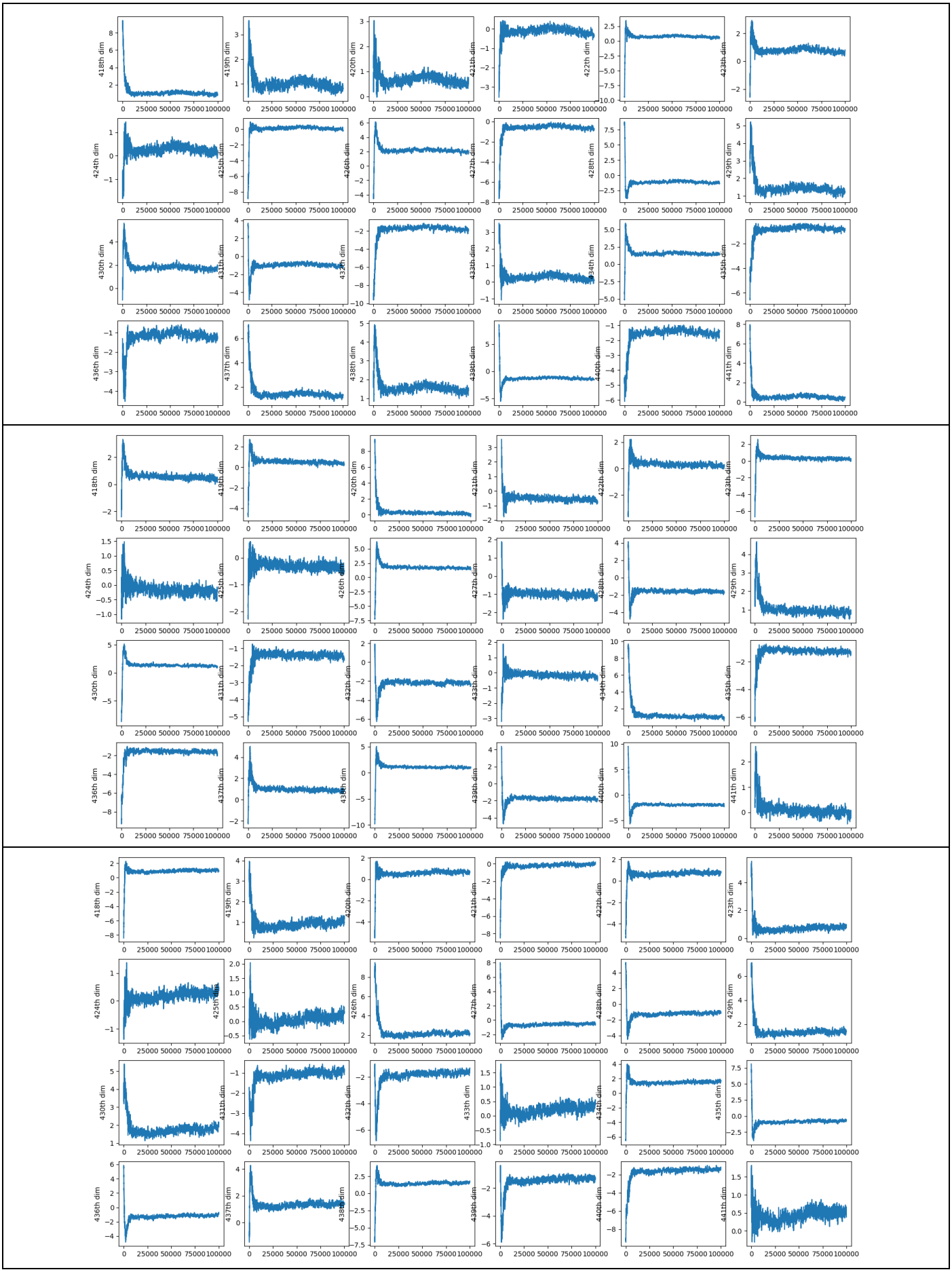
그 다음으로는 메인블록으로, 이번에도 chain 8 개를 동시에 만들 것이다. proposal distribution(normal)의 standard deviation 은 모든 parameter 에 대해 0.07 로 두었다. (parameter 마다 각각 조정할 수 있게 만들어 두기는 했지만 parameter 수가 400 개가 넘어서 튜닝 자체가 힘들다... 0.07 은 여러 값을 넣어보며 실험해본 후 acceptance ratio 와 traceplot 의 수렴 모습을 보고 선택한 값이다.) 각 체인의 initial value 로는 모든 parameter dimension 에 대해 랜덤으로 뽑는데, 구체적으로는 uniform(-20,20)에서 값을 뽑아 사용하였다. 각 체인마다 10 만개의 sample 을 만들고 나서 8 개의 chain 마다 beta1~beta24 (코드상 parameter index 로는 418~441 번째. index 가 0 부터 시작한다)의 traceplot 을 출력한다. (theta 들도 보긴 해야겠지만..생략한다.) 아래의 코드를 돌린다.

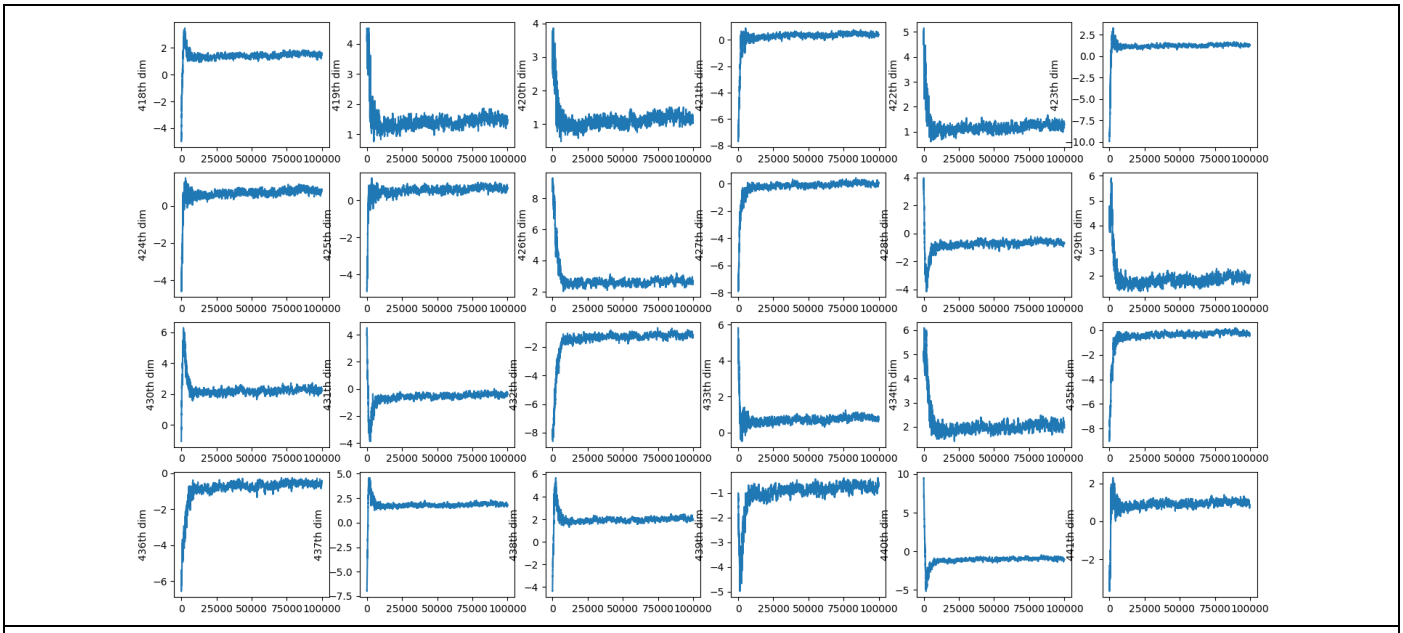
```
[Python]

#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)
```

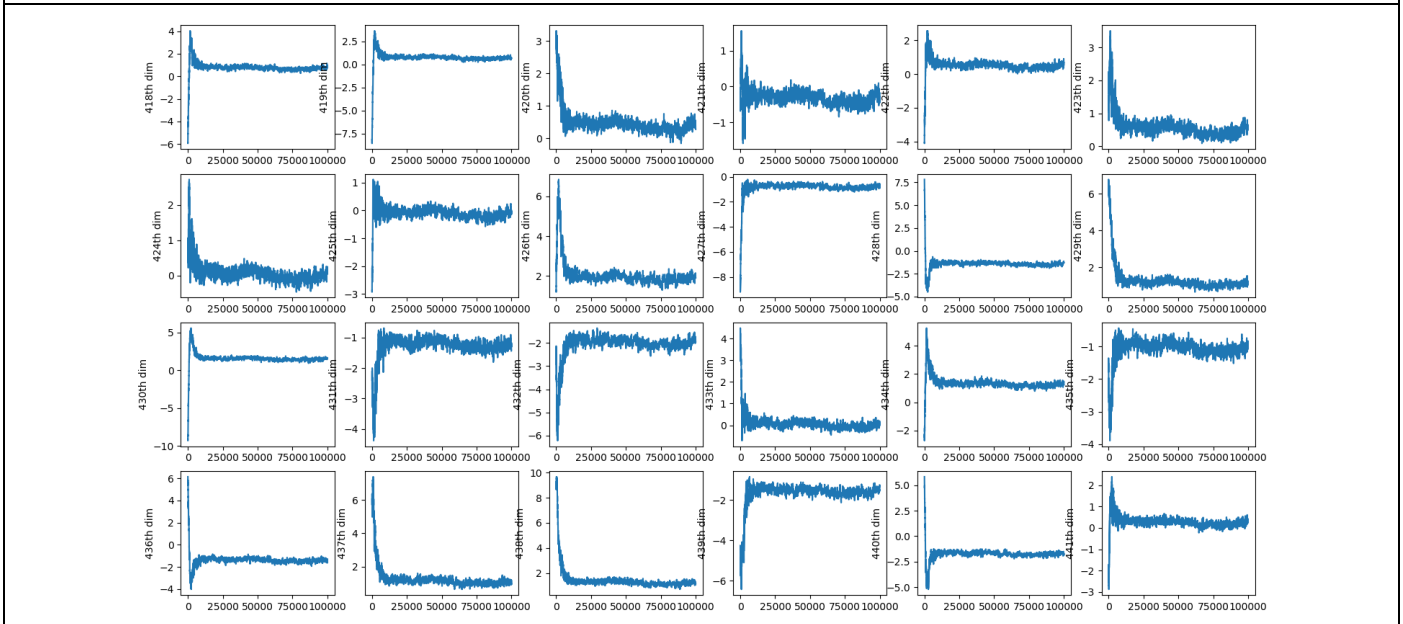
```
        mp_result_vec.append(each_result)

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

    #cheak traceplot
    for chain in mp_result_vec:
        chain.show_betas_traceplot()
    #음 그냥 맞춰주는 조합이...여기저기있나봄.....
    #아님 분포가 modal이 많거나...(빠져서못나오나?)
```

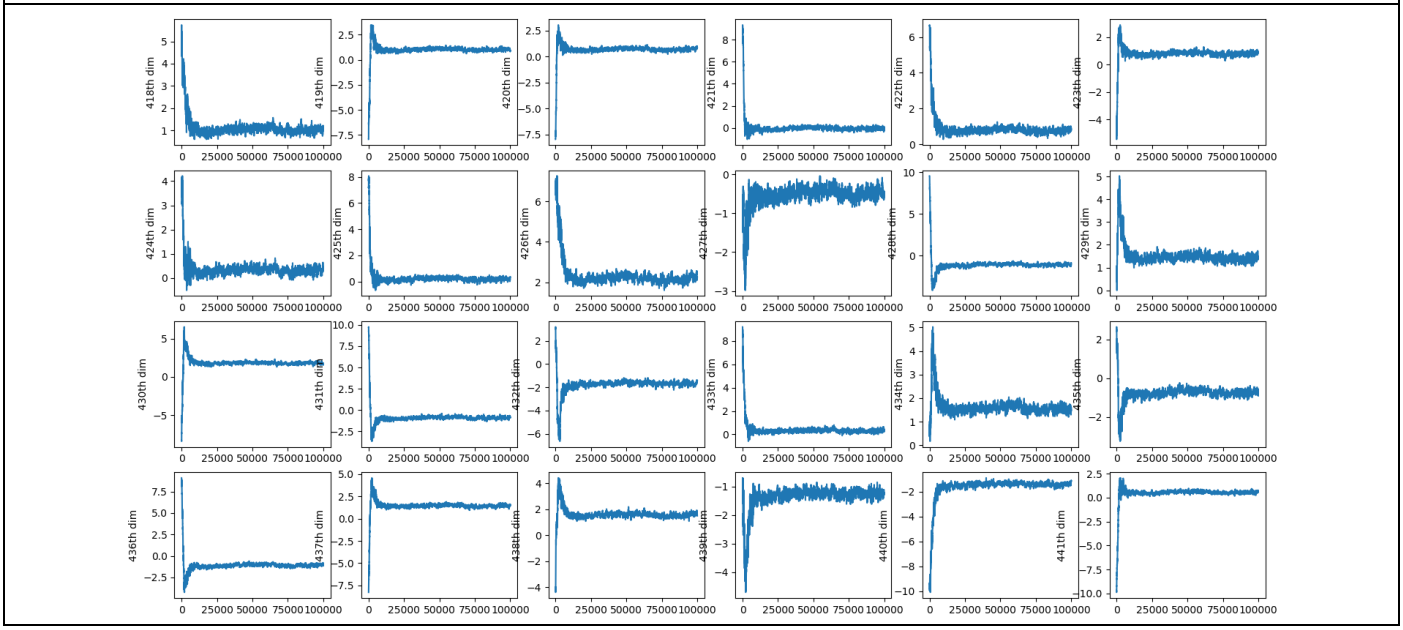
실행하면 다음의 출력을 얻는다.

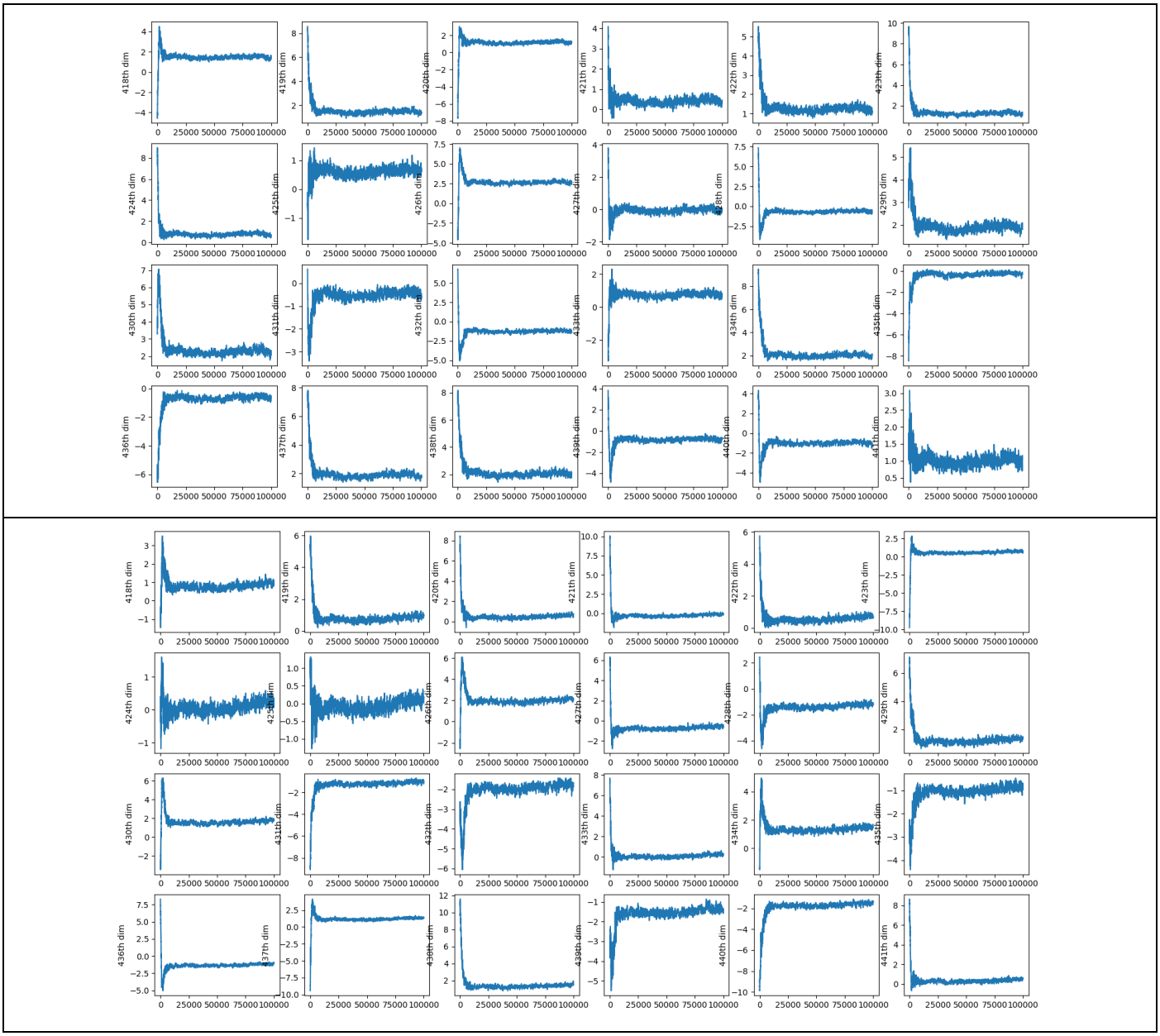
```
[Console]
pid:   7548 start!
pid:   11404 start!
pid:   13956 start!
pid:   8060 start!
pid:   13492 start!
pid:   15580 start!
pid:   8600 start!
pid:   7144 start!
pid: 7548   iteration 500 / 100000
pid: 11404   iteration 500 / 100000
(생략)
(매우긴시간이필요합니다)
pid: 8060   iteration 96500 / 100000
pid: 11404 iteration 100000 / 100000   done! (elapsed time for execution:   23.0 min   17.480346202850342 sec
pid:   11404   acc_rate: 0.06775067750677506
pid: 7144   iteration 96000 / 100000
(생략)
pid: 8600 iteration 100000 / 100000   done! (elapsed time for execution:   23.0 min   21.40987730026245 sec
pid:   8600   acc_rate: 0.06812068120681207
(생략)
pid: 8060   iteration 98500 / 100000
pid: 7144   iteration 98000 / 100000
pid: 7548 iteration 100000 / 100000   done! (elapsed time for execution:   23.0 min   41.80066251754761 sec
pid: 7144   iteration 99000 / 100000
pid: 13492 iteration 100000 / 100000   done! (elapsed time for execution:   23.0 min   51.3707480430603 sec
pid:   13492   acc_rate: 0.06777067770677707
pid: 15580   iteration 96000 / 100000
pid: 8060 iteration 100000 / 100000   done! (elapsed time for execution:   23.0 min   54.128042221069336 sec
pid:   8060   acc_rate: 0.07072070720707208
(생략)
pid: 13956   iteration 98000 / 100000
pid: 7144 iteration 100000 / 100000   done! (elapsed time for execution:   23.0 min   57.039127588272095 sec
pid:   7144   acc_rate: 0.07036070360703607
(생략)
pid: 13956 iteration 100000 / 100000   done! (elapsed time for execution:   24.0 min   14.422696352005005 sec
pid:   13956   acc_rate: 0.07087070870708707
pid: 15580   iteration 99000 / 100000
pid: 15580   iteration 99500 / 100000
pid: 15580 iteration 100000 / 100000   done! (elapsed time for execution:   24.0 min   25.51966142654419 sec
pid:   15580   acc_rate: 0.06805068050680507
exit.mp
```


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



나름 감동적인 결과를 얻었다. acceptance rate 는 0.06~0.08 사이에 있어 약간 낮은 편이긴 한데, 442 개의 차원을 고려하면 나쁜 편은 아니다. 또한 traceplot 을 보면, 비록 beta 24 개만 보았지만, 수렴점을 모두 다 잘 찾아간 것처럼 보인다. 엄밀하게는 다른 차원(theta)들도 봐야하지만, 그래프 400 개를 그려서 붙여 넣는 것은 생략하겠다. Code 를 이용한다면, 결과로 나온 각 IRM_McPost 클래스의 object 에 구현해둔 show_traceplot()이란 method 에, 각 dimension index(0~441)을 넣어서 모든 차원의 결과를 볼 수 있다. (beta 는 show_betas_traceplot() 등 show_betas 시리즈 메소드를 만들었지만... theta 에 대해서까지 만들어 두지는 않았다.)

그리고 실행시간적으로도 매우 감동적이다. 실험에 따르면, 가진 컴퓨터에서(intel i7-8700, 3.2Ghz, 6 물리코어, 12 논리프로세서(하이퍼스레딩), 16GB RAM) 10 만개의 sample 을 생성하는 한 chain 을 만드는데에 약 16~18 분정도가 걸린다. 하지만 병렬처리로 1.5 배정도의 시간인 25 분동안 8 개의 chain 을 만들었으므로, 이번 한번 작업에서 약 103 분의 이득을 보았다. parameter 바꿔가며 실험해본 시간을 생각해보면, 병렬처리 부분의 코딩 시간을 충분히 뽑았다!

추가적으로 자세히 보며 할 일을 마저 하기 위해, 위 8 개 chain 중 마지막으로 완료된 chain(위 결과 그래프상 마지막 체인, pid 15580)을 대표로 가지고 와서 작업하겠다. 먼저 histogram 과 acf plot 이다.

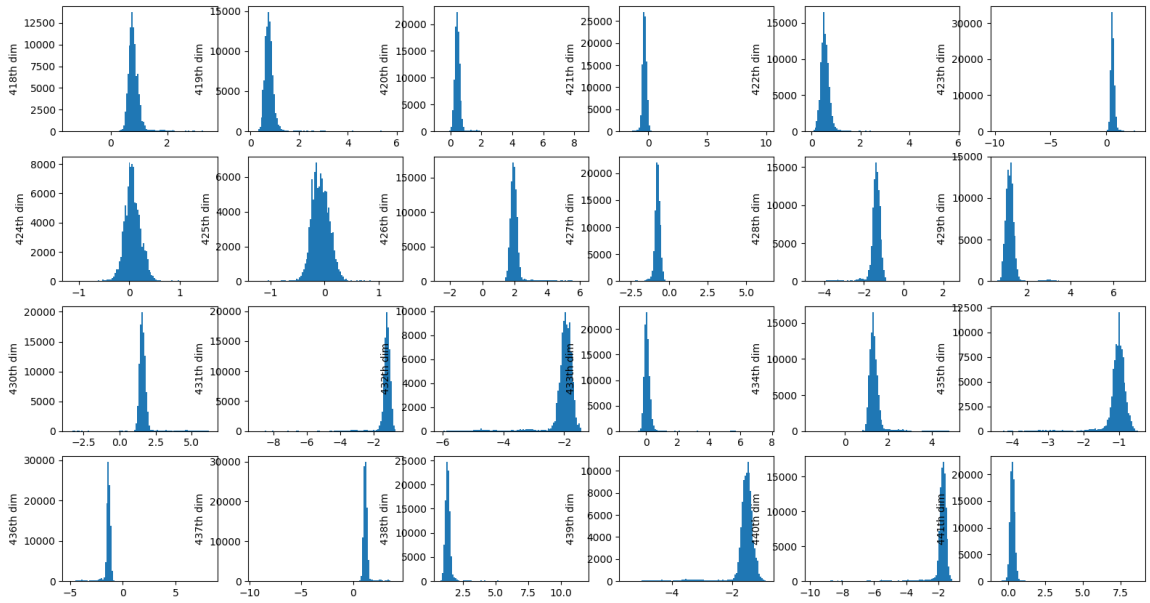
```
[Python]
OurMcSampler = mp_result_vec[-1] #마지막 체인
```

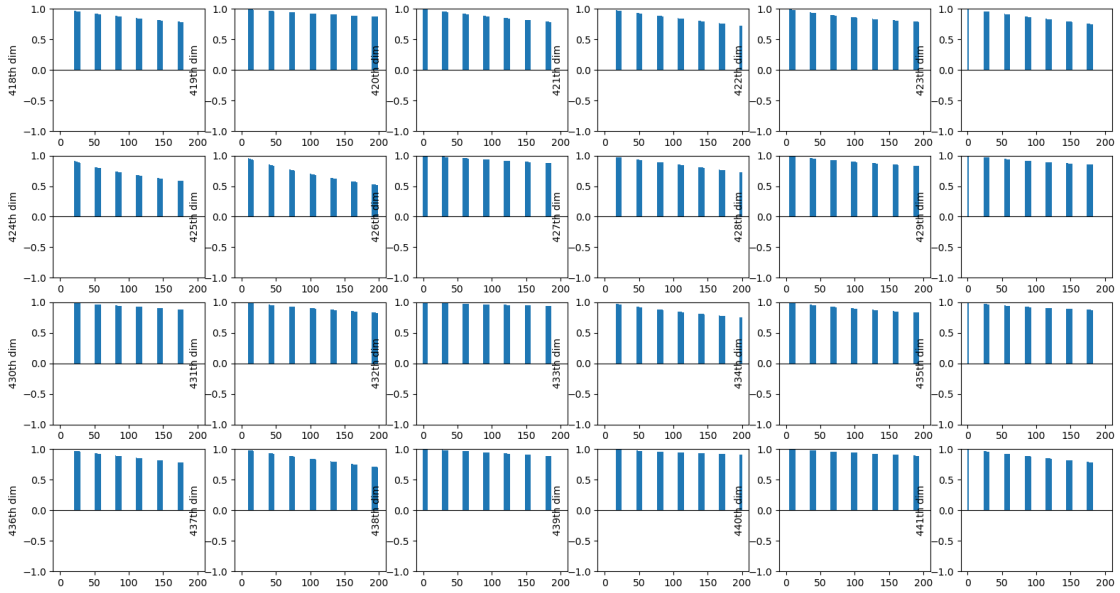
```
print("acc rate: ", OurMcSampler.get_acceptance_rate())

OurMcSampler.show_betas_hist()
OurMcSampler.show_betas_traceplot()
OurMcSampler.show_betas_acf(200)
```

출력은 다음과 같다. (중간에 traceplot 도 확인용으로 다시 한번 더 나오는데, 위에 있으므로 생략하였다.) 참고로 200 lag 를 그려서 show_betas_acf 실행이 좀 오래걸린다.

[Console]

acc rate: 0.06805068050680507





아직 sample 을 자르기 전이라 histogram 이 끝에서 튀고 있는 것은 그렇다 치더라도, acf 가 2 가지 이유로 매우 끔찍하다. 첫번째 이유는 python matplotlib 의 버그로 bar chart 가 어이없게 bar 끼리 붙어서 나왔다는 점이고, (x 축 요소가 많은데 차트 자체가 작아서 그런 듯하다. 고칠수 있을 것 같은데 더 이상 만지기 힘들어서(...) 고치지 않았다...), 두번째 이유는 acf 가 200 lag 이 되도록 안 죽는다는 것이다.

때문에 과감하게 thinning 을 하겠다. 위의 traceplot 을 보며 burnin 부분으로 앞 10000 개를 버린 후, 남은 90000 개를 200 개당 1 개의 sample 만 쓰겠다. 마음같아서는 chain 을 더 돌리고(generate_samples()를 추가로

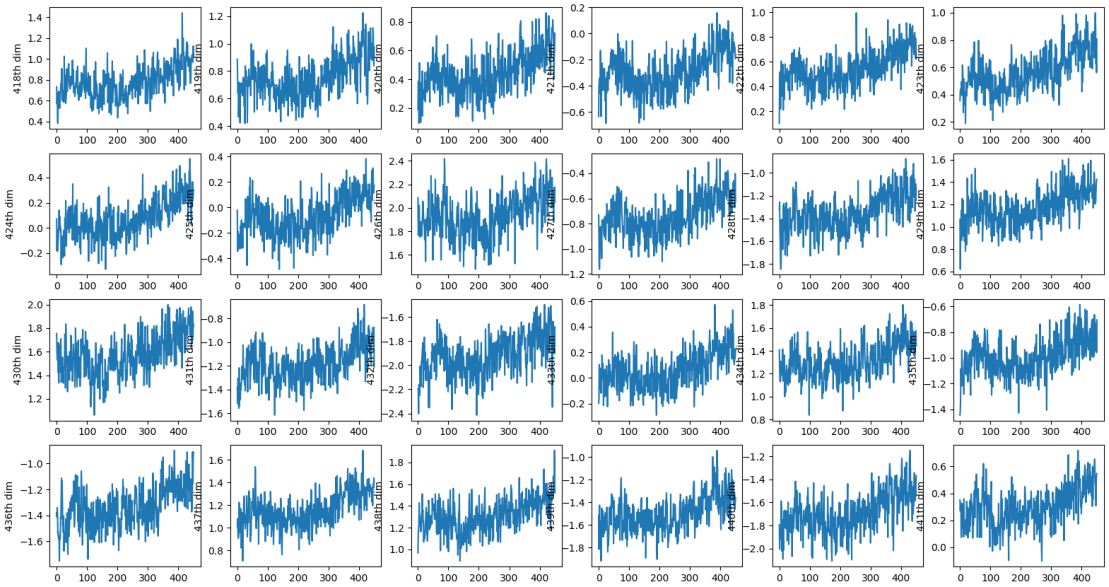
호출해서, 코드 자체는 원한다면 뒤에 이어서 계속 돌릴 수 있게 만들었다.) thinning lag 를 더 늘이고 싶지만, 보고서를 내일쯤 써야 되는 상황이 될 것 같으므로 그러지 않았다.

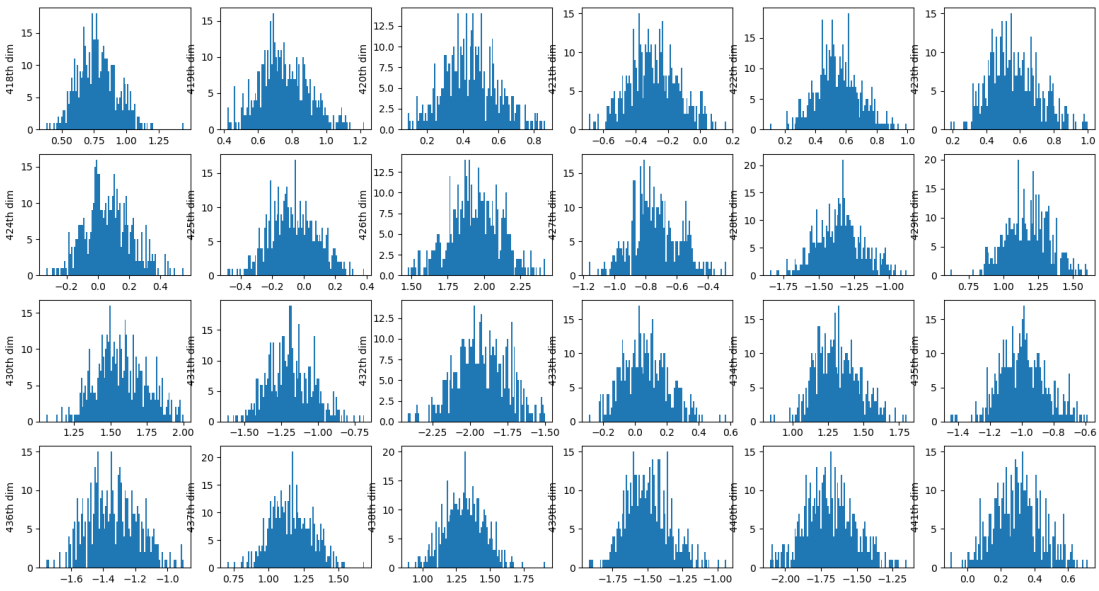
또한, 전체 442 parameter 에 대해 평균을 구해 출력하겠다.

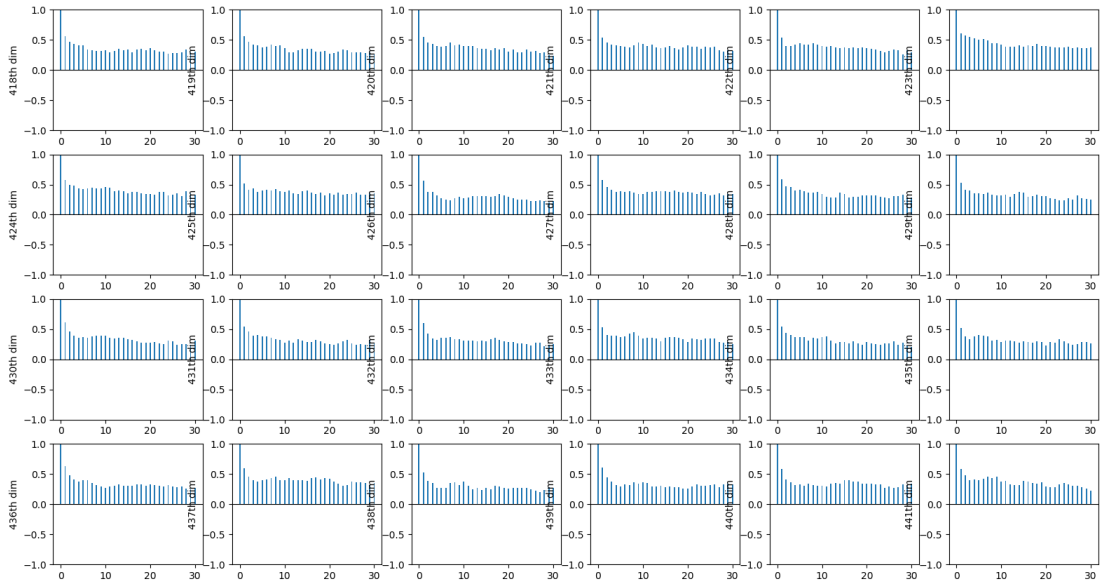
다음 코드를 돌린다.

```
[Python]
    OurMcSampler.burnin(10000)
    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())
```

결과는 다음과 같다.







[Console]

mean vec(after burn-in): [-0.9175841693812258, -0.19484989263555, -0.3700642293607025, -0.5679182574885994, -0.798262759084158, 0.10703873140094625, -0.15935315403984004, -0.7516444421371031, -0.0656120048869592, 0.050867030654701675, -0.00014401438555153274, -1.0484476501417759, -0.973289267936736, -0.007854137504678458, -0.02967364528096604, -0.37039482352283815, -0.27610545872573955, -0.6350893734888607, -0.048437236434159245, -0.5527025323818257, 0.026291165098709275, -1.0731254879679695, -0.12140971515149938, -0.5916234349481349, -0.34233234732535633, -0.9644678379787414, -0.21469839340355168, -0.10478925657593853, 1.1214556628050392, 0.1101277589733591, -0.1020729835674771, -0.33100309975663783, -0.11662745775331368, -0.11240477674248563, -0.45528787163873935, -0.19783470534223968, 0.23252367182142072, -0.37284634968982433, -0.2793537528833217, -1.1103675961056019, 0.31952405972909426, 0.968396966643534, -0.02907449275337615, -0.13462452310843356, -0.4222032776327231, -0.1741955321769312, -0.04208126966239297, -0.6184831180684848, -0.4519037521196953, 0.028509647764339945, -0.27349606651637, -0.2931356213445719, -0.6798513024705471, -0.2954700195531399, -0.600709698270848, -0.7176000098734748, -0.6124951759193789, -0.9722520968417028, 0.052012967971949466, -0.1734302919084778, -0.929596751161505, -0.6956221030257012, -0.18041863342349812, 0.2585425145926409, -0.3877142498106583, 0.20895523688916326, -0.12480638502648643, -0.08317744036831774, -0.3486105621463107, -1.0748570088299187, -0.02310007152095035, -0.4380306726679224, 0.5628088387471052, 0.9607981834687149, 0.5299642935135669, 0.5936858672578293, -5.797750714299438e-05, -0.3415927338292753, -1.3915086311820961, -0.8705420222188424, -0.05312771856953195, 0.2841168247998708, 0.08554641407709448, -0.2894071359435203, -0.18288045218419824, 0.04879512441943608, -0.12293958705695796, -0.09684795568507185, -0.5910273199732465, -0.28854862664960623, 0.9145920166666988, -1.143391570188998, 0.13285742009268153, 0.06955627340490797, -0.33571228351470284, -0.37209559335159986, 0.2306744958845548, -1.3903164804496944, -0.1034016490622222, -0.26141357711144786, -0.3431509268865336, -0.08734615415409933, -0.12238399689447253, -0.04022216601776613, 0.04257615972971472, 0.16289725021109025, -0.2541346265244472, -0.1794776529248798, -1.204327919388623, -1.072545446818709, -0.17435601383078939, -0.388272358881127, 0.16982713098348992, -0.445067023798632, 0.02628022734562588, -0.6105225395864302, -0.008954145661489657, 0.09696096293942062, -0.2718206549247276, -0.4536416307729773, 0.40817991729639846, 0.19100316574057663, 0.03607372473550179, -0.006253150261621077, -0.6949846172744084, -0.3705616222794969, -0.4610443788054614, 1.5564104786393576, -0.31957617237285874, -0.14350682483375715, -0.03885047097101819, 0.5283221771688926, -0.18516869587136275, -0.4523745638200093, -0.013712338314551285, 0.7260836336480422, -0.4090515894414833, -0.2748540725006565, 0.36375847053065224, -0.23849845112518137, 0.1355452415072199, -0.13335267435113335, -0.5728186084993526, -0.2792514285238286, -1.096347802391291, -0.43158799833242845, 0.21176480932292083, -0.2194098336760143, -0.27047879532561137, 0.04327613642041055, 0.17594169524850295, 0.001327235502530709, 0.4428314111262025, 0.19360849974184124, 0.3500265016674753, -0.4799706207212163, -0.7348008639673527, 1.6616296797513395, 0.353424604842088, 0.7426803483165769, -0.8272072411698332, -0.5090033044682473, -0.7694886972055216, 3.1423912342010127, -0.1675788272360347, 1.795914003598521, -0.3497568602560795, 0.7605647562741977, 0.8361884550498024, -0.015584194010669693, 1.4749081479072514, 1.3748979226537168, 0.26897298307399814, 1.3593086495766735, 0.6384948790047776, -1.0600603696900353, -0.9084758664752004, 0.6287231694141232, -0.7510484098702732, -0.044511071256866944, -0.3972168390060615, 0.28308478216270494, 0.3857086070225466, -0.16739568430056417, -1.135806068984595, 1.4143176087237765, -0.6061527045743349, -0.6914942184074402, 0.4683103052918473, 0.6692847904194387, 0.12252225874078299, 2.644556177872203, 2.6210188793178784, 1.295480585429681, -0.8152749080519615, 3.007602230901751, -0.9878220625888688, -0.6800644516102932, -0.6207447999359381, -0.5526727328071411, -0.1794334314139486, -0.5268269660594409, -0.3688254774995617, -0.09671381453809708, 1.519590906989838, 0.13716794393678258, -0.23149442204853915, -0.3084965702024252, -0.5776539862737277, 1.4101787331320965, 2.3691822195232466,

1.8473440394726668, 0.4697723176800307, 0.9805302074668123, -0.8007698949580349, -0.17915221851395488, -0.9415398124260046, -0.6268203119170288, -0.7227025137077681, -0.0007595520156818256, -0.7514886017148869, -1.57876583417698, 0.9321430944760146, -0.9444902461272037, -0.5765461897738574, 1.3384807965424155, 0.03537662624391094, 0.17950088095180103, -0.24105693193782976, 0.8912662496951467, 1.052891028716697, 0.4326411922898387, 0.09857791946932623, 0.16108309849746538, -0.30562425925431125, 1.1864439647762657, 0.04365702525422807, 1.0489028838572478, 1.4366487669614463, 0.41040076721600743, -0.0031427133905070647, 0.43210902506224186, 0.175592272792924, 0.1458811901328985, -0.330244122584674, 0.34912912102094146, -2.4147771002332385, 0.685868192038252, 0.01583911383350293, 0.44094079107277595, -0.24549361466589065, -0.005574811802154161, -0.09377529393690971, 1.9933930896264247, 2.9305636422336434, -1.3300466179134902, -0.569219883265292, -0.03650327950151677, -0.11988169323455822, 0.0675086741185647, -0.013207290421570896, -0.02390916711312537, -1.9646834342055832, 0.5918630566314113, -0.3260692060263787, -0.054697328075682033, 0.0010375427366984629, -0.4313934832965667, -0.6838051699324252, -0.6929178231389932, 0.13032016568550206, -0.29030390267818174, -0.2707481484435603, -0.7349362369121608, -0.0357720644403578, 0.634753829611787, -0.7239874132208501, 0.3095802029074921, -0.5607150564779507, -0.7834403563520393, -1.722916667509335, -0.44587133936086726, 1.522837346438668, 0.8032932776532814, 3.2556908233398403, 0.9608420298508296, -0.3185207788885682, -0.4920751907950162, -0.5395431683053155, -0.6485368013705393, 1.4859558919889173, 4.180959010234503, -0.30628795006277515, 0.19156572149308332, 0.5229714681250711, 0.2908118912695665, -0.7095066945339475, -0.0714749127652145, 3.5977412973378007, 1.1659859491437101, 0.26896247902737136, -1.4008612125907105, -0.8065575982271165, 6.69172307969294, 0.5958827449315589, 1.4267126465010656, 1.0806190068358226, -0.19057559461205434, 0.13021944436849942, 1.0411783236905776, 0.21943879285010967, 0.4600360748734794, 0.2601293717456082, -0.7558033283002172, 2.1328631258204047, 0.5064339198064101, 1.035383364142559, -1.1650802249397472, 0.03497787708248117, -0.7762278382432405, 1.1316216646539772, 0.4152650984668573, 0.37361235223659717, 1.8099704660920792, -0.7830952361773956, -0.048746827674173904, 0.14495940287817685, 4.7288831797189195, 3.236797802450014, 1.8732226156234117, 3.909046129444791, 1.019601769416297, 2.453014911697049, 1.1146550680550784, -0.7165566046577477, -0.19266226261456884, 3.312392470771874, -0.336605454683406, 0.0633597111992233, 0.025761409813370682, -0.127716098092384, -0.19513619183060912, 0.5261528644457342, -0.9191541307880124, 0.22232898963441028, -0.9200797414528867, 3.447634378889967, 0.10223085547864197, -0.20417599185821375, -0.01308838038458119, -0.8294361626134061, 1.9070677029765069, -0.173473565475968, 2.3673348227924262, -1.964357331541197, 4.406518994577825, 0.6091559528356909, 0.5573261360081115, 0.9477305343297097, -0.4924896732014704, 0.21876294908918728, -0.2659654272444711, -1.0593301270365787, 0.5605996110976783, 0.6074032113270992, -0.3860942370180875, 0.850705155946009, 0.4796350399657199, 6.203120458245919, 0.07786456035810571, 0.23558149869492592, -0.5964368800018262, 0.008609227692142526, 2.13462219828955, 0.7821882237628919, 1.3186820508334143, 1.080075957918571, 1.4049105880540813, 1.2019338272324884, 1.330227630962786, 0.31269334688283823, -0.5674618609713789, 0.6857199647647361, -0.08355578684145713, -1.4271670843314692, -0.1268123770873984, 1.8222303526619674, 0.16846727436588899, -0.002933848970441063, 1.312713934317403, -0.02129253578060944, 0.6437135723449596, 0.6794778430808195, 0.1761105860412836, 0.2819540482748723, -1.5092166220411483, 0.1394367888312547, -0.8540330297982022, 0.2595066998215814, -0.6274511581350863, 0.5318153088077022, -0.30245260828875487, 0.5982203296742572, 0.983041559894092, 0.7244298091744366, 1.6299424900744626, 4.944233487666921, 4.654813736440404, 3.161550748636102, 0.6544515964916454, -1.032719268607085, -0.17616242284999348, 1.1890799292120806, 0.6534349205436893, 1.2293525877379683, 0.32326820069943024, 0.607389645685753, 1.3570118824642712, 0.7799844943537803, 0.7526981301023412,
0.44400412124264915, -0.2882910248086651, 0.5471196115122376, 0.5677637961953343, 0.068191586042498, -0.06455706690088656, 1.93723985058106, -0.7359797198106298, -1.3509467938372008, 1.177032988567247, 1.573035400533702, -1.1868380062145811, -1.9219504058668917, 0.06556002320117932, 1.3142346757700127, -0.9884097123454693, -1.3364105949914789, 1.160580010867791, 1.312232878709542, -1.497502243496379, -1.6879946299729613, 0.3015514376878106]

여전히 문제가 있긴 있는 것을 확인할 수 있다.

acf 부터 보면, 무려 200 개당 1 개를 쓰는 과감한 thinning 을 통해 이전보다는 많이 나아졌지만. 여전히 불만족스럽게 높다. 더 문제는 30lag 에서도 여전히 죽을 기미를 안 보인다는 것인데, 이를 다시 말하면 200*30 = 6000 개 이상의 sample 마다 1 개씩 쓰는 대국적인(?) thinning 을 해야 autocorrelation 이 잡힐 것이란 이야기이다.

traceplot 도 잘라 놓고 나서 좁은 y 축과 확대된 x 축에서 보니, 위의 높은 correlation 의 결과처럼, 아직 어떤 경향성이 있음을 같이 보여주고 있다. 이 또한 sample 을 더 생성한 후 acf 를 맘에 들 정도까지 죽일수 있는 정도까지 sample 을 잘라 써야 해결될 문제로 보인다.

따라서 시뮬레이션을 더 돌려야 하나, 일단은 여기까지만 하겠다. (슈퍼컴퓨터가 있으면 6000 만개정도 더...)

그 외에, beta 들의 histogram 과 mean vector 는 위와 같다.

4. 5. Using Gibbs sampler,

4. With FurSealPup Cap-Recap model and given data, get posterior samples of N and each alpha.

5. Do Bayesian simple regression using data index.txt and get posterior samples of each beta and sigma square.

위 문제들처럼 두 문제가 같은 알고리즘을 사용하므로, Gibbs sampler 를 추상적으로 구현한 뒤 이것저것 utility 를 붙인 아래 클래스를 먼저 선언 후 공통으로 사용하겠다. 핵심 알고리즘은 GibbsSampler.sampler()에 구현되어 있다.

생성자가 full_conditional_sampler 로 받는 argument 는 iterable 하고 python 의 대괄호 [index] 표현으로 접근가능한 객체로, 안에 각 parameter 순서대로, 현재 parameter 조합을 받아 자기 index 에 맞는 새 sample 을 생성하는 해당 conditional distribution 을 따르는 sampler 가 들어있어야 한다.

[Python]

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

#Gibbs sampler

import time
from math import log
from random import seed, betavariate, normalvariate, gammavariate
from functools import partial
from statistics import mean
import re


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


class GibbsSampler:
    def __init__(self, initial_val, full_conditional_sampler):
        if not len(initial_val)==len(full_conditional_sampler):
            raise ValueError("number of initial_val's dimension should be equal to number of conditional densities.")
        self.initial = initial_val
        self.up_to_date = list(initial_val)
        self.num_dim = len(initial_val)
        self.full_conditional_sampler = full_conditional_sampler
        self.samples = [initial_val]

    def sampler(self):
        new_sample = [None for _ in range(self.num_dim)]
        for dim_idx in range(self.num_dim):
            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
        new_sample = tuple(new_sample)
        self.samples.append(new_sample)

    def generate_samples(self, num_samples):
```

```
        start_time = time.time()
        for i in range(1, num_samples):
            self.sampler()
            if i%10000==0:
                print("iteration", i, "/", num_samples)
        elap_time = time.time()-start_time
        print("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):
            subplot_idx=str(grid_row)+str(grid_column)+str(i+1)
            plt.subplot(subplot_idx)
            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):
            subplot_idx=str(grid_row)+str(grid_column)+str(i+1)
            plt.subplot(subplot_idx)
            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]
```

4 번문제를 먼저 하자. 다음과 같이 full conditional distribution 을 세팅한다. global 에서 다 선언하고 리스트와 같은 편리한 iterable 객체에 넣으면 되도록 만들었지만, 코드 정리를 위해 클래스로 만들고 __getitem__과 __len__을 구현하여 [] 표현을 사용할 수 있는 프로토콜에 맞추어 만들었다.

α_i들은 생긴 모양이 index 만 바꿔주면 똑같이 생겼으므로, index 에 대한 함수로 놓고 partial 함수를 이용해 index 를 각 값으로 고정하여 full_cond 를 만든다.

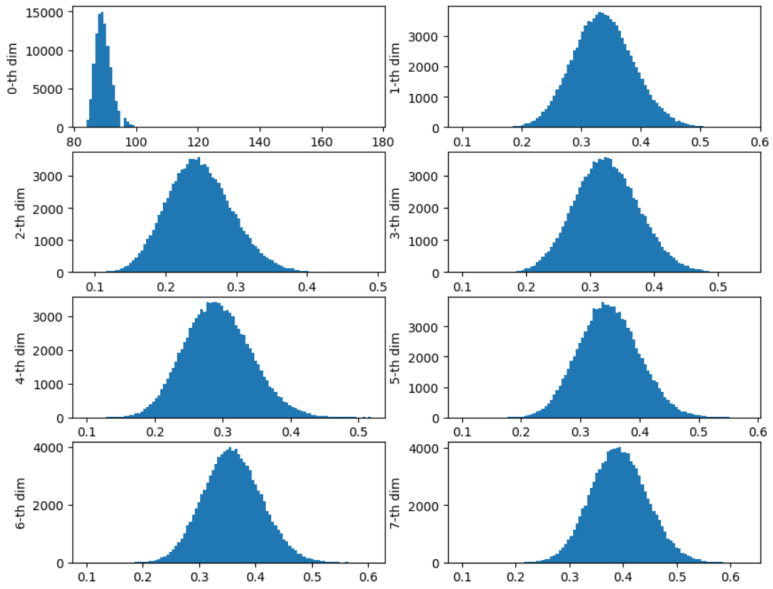
```
[Python]
class FurSealPupCapRecap_FullCondSampler:
    #parameter vector order :
    # 0   1   2   3   4   5   6   7
    # N   a1 a2 a3 a4 a5 a6 a7
    NumberCaptured = (30,22,29,26,31,32,35)
    NumberNewlyCaught= (30,8,17,7,9,8,5)
    r = sum(NumberNewlyCaught) #84

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

    def a(self, up_to_date, a_idx):
        c = self.NumberCaptured[a_idx-1]
        alpha = c + 0.5
        beta = up_to_date[0] - c + 0.5
        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)
```

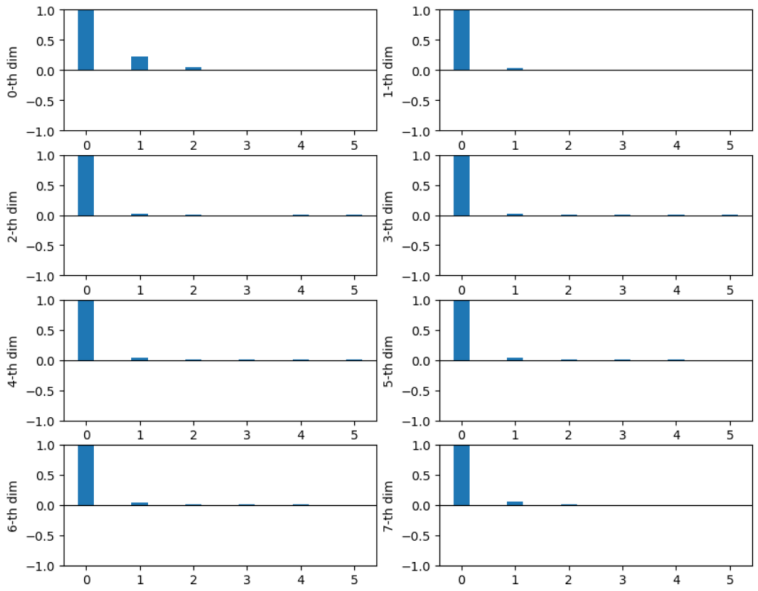
이제 정말 gibbs sampler 를 돌리자. 초기값은 임의로 (N, α_i ($i = 1,2,3,4,5,6,7$))(150, 0.1,0.1,0.1,0.1,0.1,0.1,0.1)로 설정하였다. Sample 수는, 10 만개 만들겠다. 다음 코드를 실행하여 먼저 보자.

```
[Python]
if __name__=="__main__":
    #ex4
    Seal_fullcond = FurSealPupCapRecap_FullCondSampler()
    # print(len(Seal_fullcond)) #8
    Seal_initial_values = (150, 0.1,0.1,0.1,0.1,0.1,0.1,0.1)
    Seal_Gibbs = GibbsSampler(Seal_initial_values, Seal_fullcond)
    Seal_Gibbs.generate_samples(100000)
    Seal_Gibbs.show_hist()
    Seal_Gibbs.show_acf(5)
```

결과는 다음과 같다. 출력 파라메터 순서는 $(N, \alpha_1, \ldots, \alpha_7)$이다. 그래프는 순서대로 sample histogram, acf 이다.

```
[Console]
iteration 10000 / 100000
(생략)
iteration 100000 / 100000    done! (elapsed time for execution:    0.0 min    2.785384178161621 sec
```





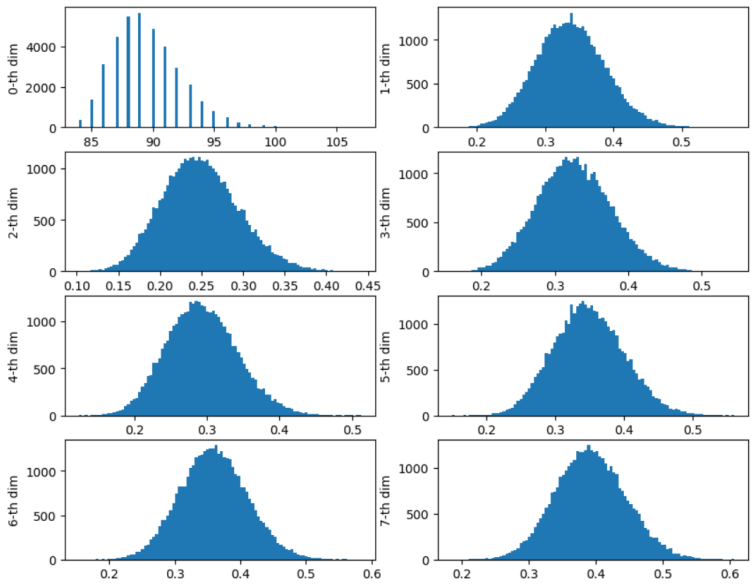
Gibbs sampler 에도 수렴이 보장된 것과는 별개로, 수렴 이전의 sample 을 버리는 burn-in period 가 필요하다. 각 dimension 의 traceplot 을 보고 잘라야 할 것 같은데, 수업시간에는 굳이 그릴 필요 없다는 소리를 들은 것

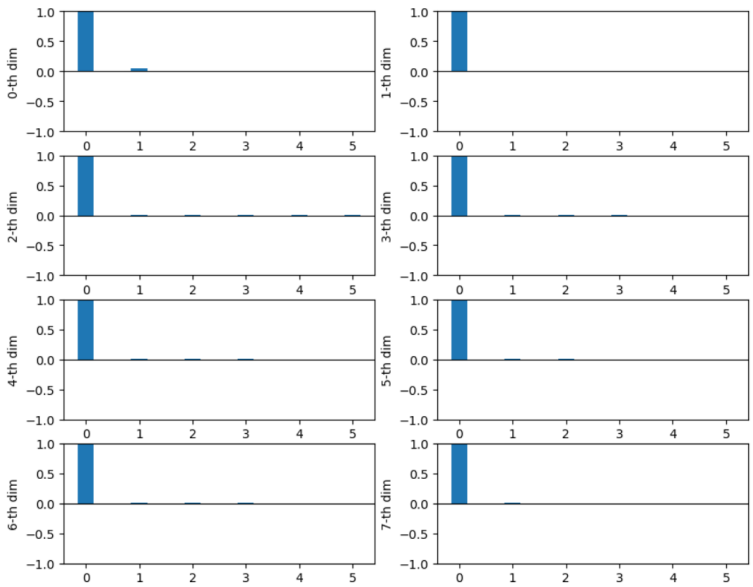
같아서(?) 그냥 충분할 정도로 자르겠다. 앞 25000 개(1/4 에 해당)을 자르고, acf plot 에서 0-th dim 이라 축이 달려있는 N 이 첫 lag 에선 좀 살아있으므로, sample 을 2 개당 하나만 쓰겠다. 이후 mean 을 구하자. 다음 코드를 main 에서 돌리자.

```
[Python]
#음 좀 자르자
    Seal_Gibbs.burnin(25000)
    Seal_Gibbs.thinning(2)

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

결과는 다음과 같다. 콘솔 출력은 $(N, \alpha_1, \dots, \alpha_7)$ 의 mean 이다. 그래프 결과 또한 $(N, \alpha_1, \dots, \alpha_7)$ 순서대로 각각 sample histogram, acf 이다.

```
[Console]
[89.50086664355617, 0.33750692828122475, 0.2490924480393129, 0.32637261744240387, 0.2932305532779302, 0.34863356830840414, 0.3592964501495169, 0.3930099337189183]
```





귀여운 친구들이 이 지역에 (평균적으로) 89.5 마리밖에 없다고 추정된 것을 확인할 수 있다.

이제 5 번을 진행하자. 다음과 같이 full conditional distribution 을 class 형태로 선언한다. Hyperprior parameter 중 따로 언급이 없었던 것들은 적절히 설정하였다.

[Python]

```
class BayesianSimpleReg_FullCondSampler:
    #regression model
    #population
    # y ~ Normal(b0+b1*x, sigma^2)
    #hyperprior
    # b_j ~ Normal(mu_j, tau_j^2), j=0,1
    # sigma^2 ~ Inv.Gamma(a,b)
    #parameter of hyperprior
    # tau_j=10^2, a=0.001, b=0.001, mu0=?, mu1=?

    #parameter vector order :
    # 0   1   2
    # b0 b1 sigma**2

    def __init__(self, data_x, data_y):
        self.x = data_x
        self.y = data_y
        self.n = len(data_x)
        self.mu0 = 0
        self.mu1 = 0
        self.tau0 = 10
        self.tau1 = 10
        self.invGam_a = 0.001
        self.invGam_b = 0.001

    def b0(self, up_to_date):
        cond_post_precision = self.n/up_to_date[2] + 1/(self.tau0**2)
        cond_post_mu_upperpart = sum([self.y[i]-up_to_date[1]*self.x[i] for I in range(self.n)])/up_to_date[2] + self.mu0/(self.tau0**2)
        return normalvariate(cond_post_mu_upperpart/cond_post_precision, 1/cond_post_precision)

    def b1(self, up_to_date):
        cond_post_precision = sum([self.x[i]**2 for I in range(self.n)])/up_to_date[2] + 1/(self.tau1**2)
        cond_post_mu_upperpart = sum([(self.y[i]-up_to_date[0])*self.x[i] for I in range(self.n)])/up_to_date[2] + self.mu1/(self.tau1**2)
        return normalvariate(cond_post_mu_upperpart/cond_post_precision, 1/cond_post_precision)

    def inv_gamma_generator(self, param_a, param_rate):
        return 1/gammavariate(param_a, 1/param_rate)

    def sigma_square(self, up_to_date):
        #inv_gamma(a,b)
        scale = self.n/2 + self.invGam_a
        rate = sum([((self.y[i]-up_to_date[0]-up_to_date[1]*self.x[i])**2)/2 for I in range(self.n)]) + self.invGam_b
        return self.inv_gamma_generator(scale, rate)

    full_cond = [b0, b1, sigma_square]

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

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

다음은 이제 정말 실행하는 부분이다. 앞부분은 데이터를 불러오는 부분(커서 직접 코드에 넣기가 좀 그랬다)이고, 바로 위에서 정의한 setting class 의 instance 를 만든 후 이 문제 맨 위에서 정의한 Gibbs sampler 에 집어넣어 sampler instance 를 만든다. 초기값은 임의로 $(\beta_0, \beta_1, \sigma^2) = (0,0,1)$로 설정하였다. 10 만개 sample 을 생성하자.

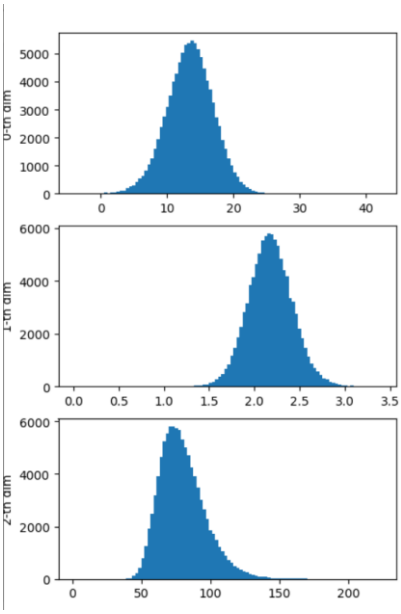
[Python]

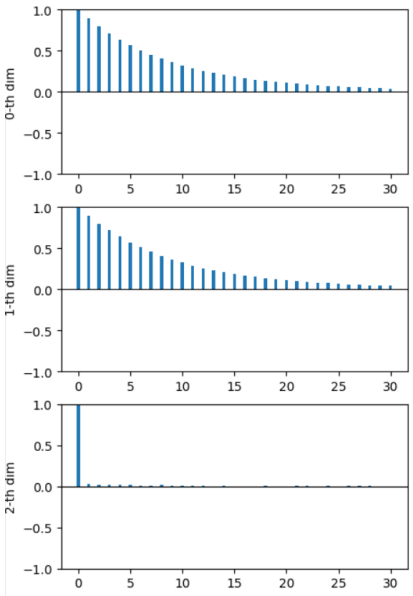
```
#ex5
teen_birth = []
poverty = []
with open("c:/gitProject/statComputing2/HW4/index.txt","r", encoding="utf8") as f:
    f.readline() #header 한번 밀어야함
    while(True):
        line = f.readline()
        split_line = re.findall(r"[₩w.]+", line)
        if not split_line:
            break
        teen_birth.append(float(split_line[5]))
        poverty.append(float(split_line[1]))

Reg_fullcond = BayesianSimpleReg_FullCondSampler(poverty, teen_birth)
# print(len(Reg_fullcond)) #3
Reg_initial_value = (0,0,1)
Reg_Gibbs = GibbsSampler(Reg_initial_value, Reg_fullcond)
Reg_Gibbs.generate_samples(100000)
Reg_Gibbs.show_hist()
Reg_Gibbs.show_acf(30)
```

다음 출력을 얻는다. 파라메터 순서는 $\beta_0, \beta_1, \sigma^2$이고, histogram 과 acf 이다.

[Console]

```
iteration 10000 / 100000
(생략)
iteration 90000 / 100000
iteration 100000 / 100000    done! (elapsed time for execution:    0.0 min    6.5831944942474365 sec
```





앞 두 beta parameter 들에 대해 길게 autocorrelation 이 살아 있는 것을 볼 수 있다. thinning 하자.

Burnin period 로 충분히 50000 개 앞 자르고, 이후 남은 sample 중 30 개중 1 개만 쓰겠다.

다음 코드를 실행하자.

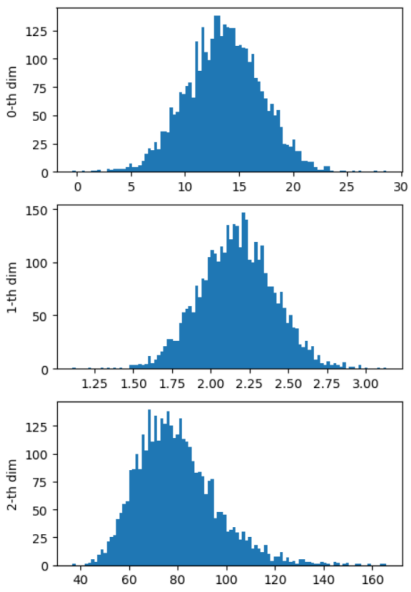
[Python]

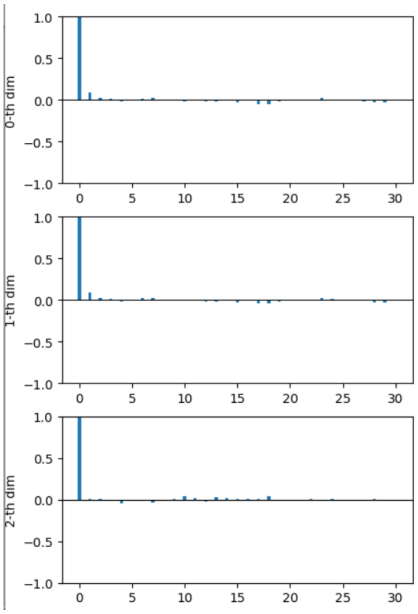
```
#여기도좀자르자-_-
Reg_Gibbs.burnin(25000)
Reg_Gibbs.thinning(20)

print(Reg_Gibbs.get_sample_mean())
Reg_Gibbs.show_hist()
Reg_Gibbs.show_acf(30)
```

출력은 다음과 같다. 콘솔 결과는 남은 sample 을 가지고 구한 $\beta_0, \beta_1, \sigma^2$의 sample mean 이며, 그래프 또한 해당 순서로 그린 histogram 과 acf 이다.

[Console]

[13.597905258276938, 2.168738075318048, 79.67579981886391]





앞 burn-in 부분을 잘라서, histogram 이 model 에서 예상할 수 있는 결과와 같이 훨씬 normal 처럼 보이게 되었다. 또한 autocorrelation 도 첫 lag 에서 많이 죽어서 양호해진 것을 볼 수 있다.