PSI Numerical Methods 2023, homework assignment for MCMC Week

In [1]:
```
using CairoMakie
using LinearAlgebra
using AffineInvariantMCMC
using Statistics
using Printf
```

We will start by defining the same data and functions that you're probably bored of by now!

In [2]:
```
# Example data set from  arxiv:1008.4686, table 1 (https://arxiv.org/abs/1008.4686
# You can also refer to that paper for more background, equations, etc.
alldata = [201. 592 61; 244 401 25; 47  583 38; 287 402 15; 203 495 21; 58  173 15
           202 504 14; 198 510 30; 158 416 16; 165 393 14; 201 442 25; 157 317 52;
           166 400 34; 160 337 31; 186 423 42; 125 334 26; 218 533 16; 146 344 22
# The first 5 data points are outliers; for this we'll just use the "good" data po
x    = alldata[6:end, 1]
y    = alldata[6:end, 2]
# this is the standard deviation (uncertainty) on the y measurements, also known a
yerr = alldata[6:end, 3];
```

In [3]:
```
function log_likelihood_one(params, x, y, yerr)
    """This function computes the log-likelihood of a data set with coordinates
    (x_i,y_i) and Gaussian uncertainties on y_i of yerr_i (aka sigma_i)

    The model is a straight line, so the model's predicted y values are
        y_pred_i = b + m x_i.

    params = (b,m) are the parameters (scalars)
    x,y,yerr are arrays (aka vectors)

    Return value is a scalar log-likelihood.
    """
    # unpack the parameters
    b,m = params
    # compute the vector y_pred, the model predictions for the y measurements
    y_pred = b .+ m .* x
    # compute the log-likelihoods for the individual data points
    # (the quantity inside the sum in the text above)
    loglikes = log.(1 ./ (sqrt(2*π) .* yerr)) .- 0.5 .*(y - y_pred).^2 ./ yerr.^2
    # the log-likelihood for the whole vector of measurements is the sum of indivi
    loglike = sum(loglikes)
    return loglike
end;
```

And here's the "vanilla" Metropolis-Hasting Markov Chain Monte Carlo from the MCMC-filled-in notebook. Please feel free to use your own if you prefer it!

In [4]:
```
function mcmc(logprob_func, logprob_args,
              propose_func, propose_args,
              initial_pos, nsteps)
    """
    MCMC: Markov Chain Monte Carlo.  Draw samples from the *logprob_func* probabil
    using proposed moves generated by the function *propose_func*.

    * logprob_func: a function that returns the log-probability at a given value o
```

```
              It will get called like this:
        lnp = logprob_func(params, logprob_args)
    * logprob_args: extra arguments to pass to logprob_func.
    * propose_func: a function that proposes to jump to a new point in parameter sp
              It will get called like this:
        p_new = propose_func(p, propose_args)
    * propose_args: extra arguments to pass to propose_func.
    * initial_pos: initial position in parameter space (list/array)
    * nsteps: integer number of MCMC steps to take

    Returns  (chain, faccept)
    * chain: size Nsteps x P, MCMC samples
    * faccept: float: fraction of proposed jumps that were accepted
    """
    p = initial_pos
    logprob = logprob_func(p, logprob_args)
    chain = zeros(Float64, (nsteps, length(p)))
    naccept = 0
    for i in 1:nsteps
        # propose a new position in parameter space
        p_new = propose_func(p, propose_args)
        # compute probability at new position
        logprob_new = logprob_func(p_new, logprob_args)
        # decide whether to jump to the new position
        if exp(logprob_new - logprob) > rand()
            p = p_new
            logprob = logprob_new
            naccept += 1
        end
        # save the position
        chain[i,:] = p
    end
    return chain, naccept/nsteps
end;
```

Here is the proposal distribution function for the simple (uncorrelated) Gaussian. The samples from this function will lie in an axis-aligned ellipse.

In [5]:
```
function propose_gaussian(p, stdevs)
    """
    A Gaussian proposal distribution for mcmc.
    *p*: the point in parameter space to jump from
    *stdevs*: standard deviations for each dimension in the parameter space.
    """
    return p .+ randn(length(p)) .* stdevs
end;
```

I'm asking you to act like Bayesians, so instead of the log-likelihood we're going to use the log-posterior (but with a flat prior...).

In [6]:
```
function log_posterior_one(params, args)
    (x, y, yerr) = args
    loglike = log_likelihood_one(params, x, y, yerr)
    # Improper, flat priors on params!
    logprior = 0.
    return loglike + logprior
end;
```

Part 1. Metropolis-Hastings MCMC, with uncorrelated Gaussian jumps and bad jump sizes!

In [7]:
```
# initial B,M
initial_pos = [0., 1.0]
# proposal distribution: jump sizes for B,M
jump_sizes = [1., 0.1]

# How many MCMC steps to take
nsteps = 5000
# How many samples to throw away as "burn-in"
nburn = 1000

# Run MCMC!
wholechain,accept = mcmc(log_posterior_one, (x,y,yerr),
                         propose_gaussian, jump_sizes,
                         initial_pos, nsteps)
println("Fraction of moves accepted:", accept)
chain = wholechain[nburn+1:end, :]
size(chain)
```

```
Fraction of moves accepted:0.3644
```

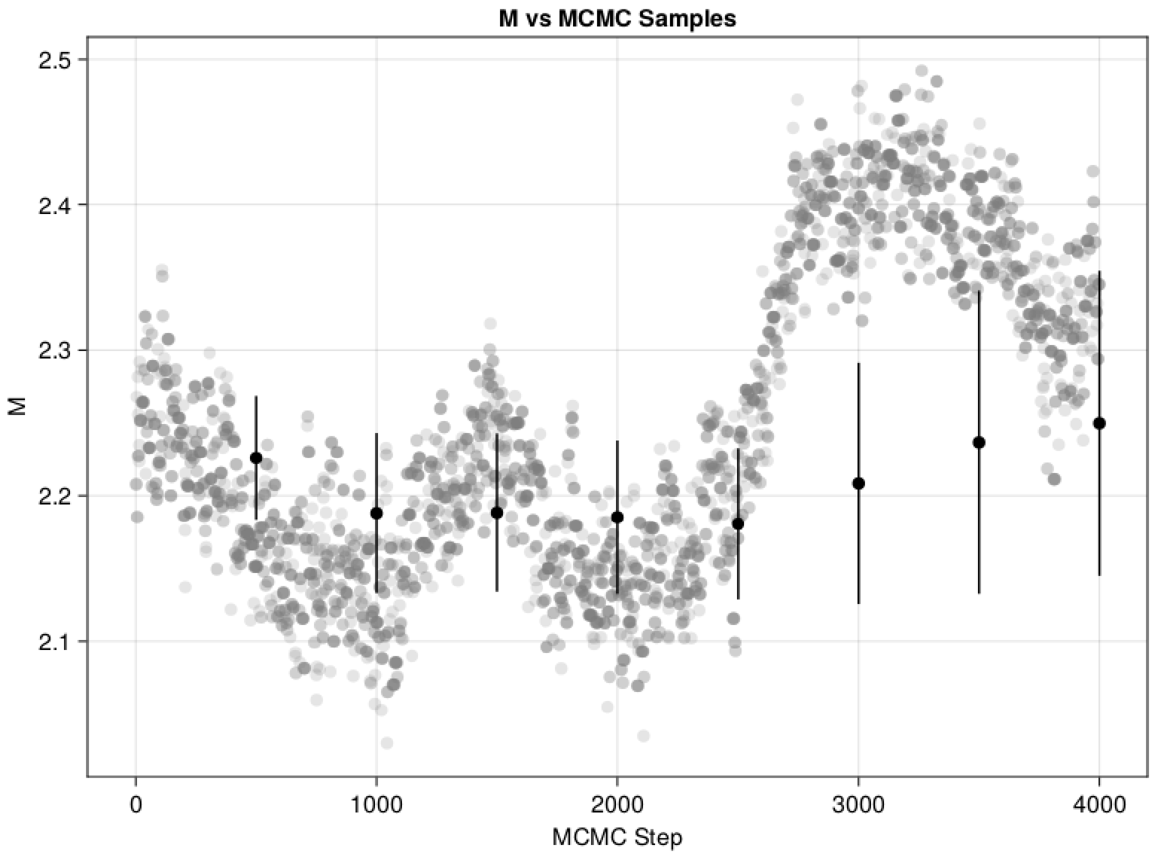
Out[7]:
```
(4000, 2)
```

Now, let's have a look at the convergence. Looking at the samples for just one of `B` or `M`, let's compute the mean and standard deviation of all the samples up to different lengths of the chain. (Eg, the mean and standard deviation after 1000 steps, then 2000 steps, then 3000 steps.) If our chain is converging, we should see that the mean and variance do not change much as we take more samples.

In [8]:
```
# Measure the mean and standard deviation of array `x` up to index `steps`, `2*step
function running_mean_std(x, steps)
    n = size(x,1)
    # the list of steps that we're going to measure the mean and std up to.
    ss = steps:steps:n
    mn = zeros(size(ss,1))
    st = zeros(size(ss,1))
    for i in 1:size(ss,1)
        mn[i] = mean(x[1:ss[i]])
        st[i] =  std(x[1:ss[i]])
    end
    return ss,mn,st
end;
```

In [9]:
```
# Plot the parameter values in the chain!
f = Figure()
ax = Axis(f[1, 1], xlabel="MCMC Step", ylabel="M", title="M vs MCMC Samples")
scatter!(chain[:,2], color=(:grey, 0.2), label="Samples")

# Let's measure the running mean and variance after different chain lengths
runsteps = 500
ss,mn,st = running_mean_std(chain[:,2], runsteps)

scatter!(ss, mn, color=:black, label="Mean + Std up to this sample")
errorbars!(ss, mn, st)
#Legend(f[1,2],ax)
f
```

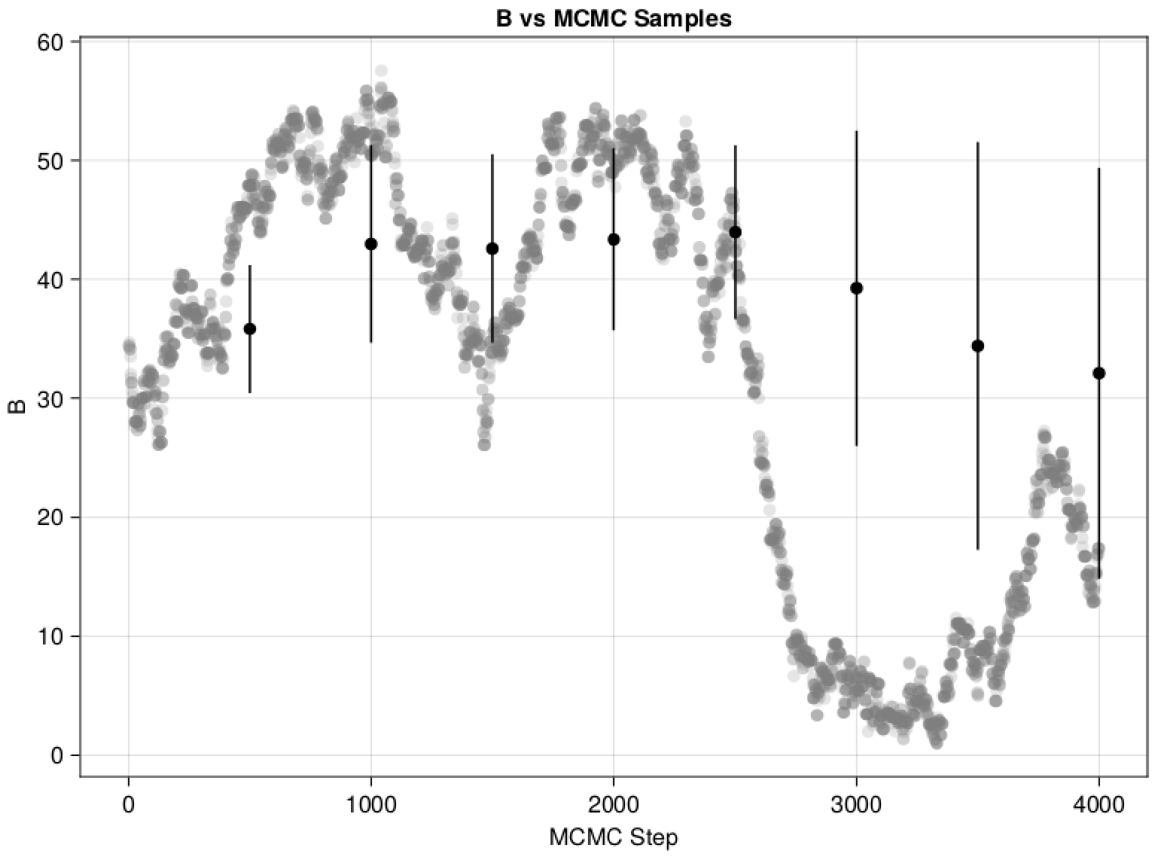
Out[9]:



In [10]:

```
# Plot the parameter values in the chain!
f = Figure()
ax = Axis(f[1, 1], xlabel="MCMC Step", ylabel="B", title="B vs MCMC Samples")
scatter!(chain[:,1], color=(:grey, 0.2), label="Samples")

# Let's measure the running mean and variance after different chain lengths
runsteps = 500
ss,mn,st = running_mean_std(chain[:,1], runsteps)

scatter!(ss, mn, color=:black, label="Mean + Std up to this sample")
errorbars!(ss, mn, st)
#Legend(f[1,2],ax)
f
```

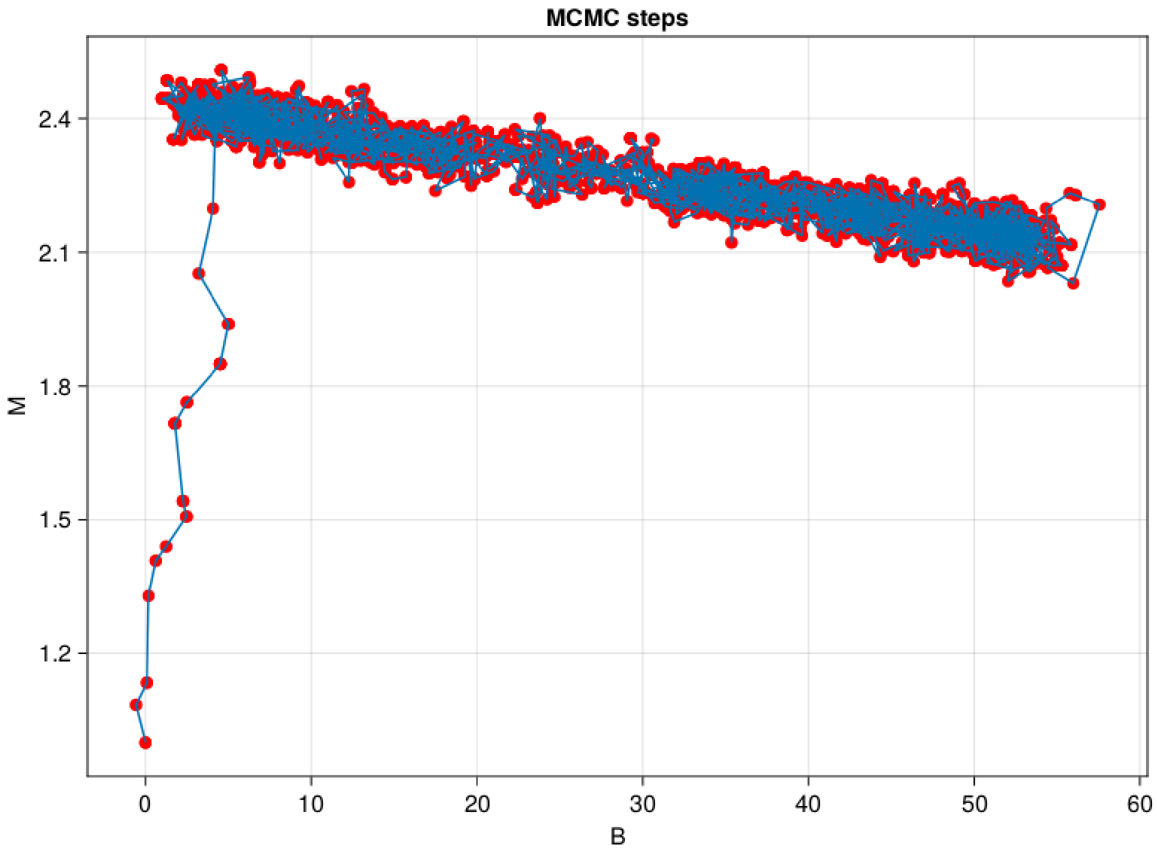
Out[10]:



In [13]:

```
# Let's have a look at the resulting samples!
f = Figure()
Axis(f[1, 1], xlabel="B", ylabel="M", title="MCMC steps")
scatter!(wholechain[:,1], wholechain[:,2], color=:red)
lines!(wholechain[:,1], wholechain[:,2])
f
```

Out[13]:



Does this chain look like it has converged?

Not that much for both `B` and `M`..

One way we would use the samples from our MCMC is to report the values, say in a paper we are writing.

Looking at the numbers in the `mn` and `st` arrays from above (the mean and standard deviation of the first 500 samples, first 1000 samples, first 1500, etc), please write out what you would report for the mean and uncertainty of `M`, in the form $X \pm Y$. For example, to print them out for the first 500 samples, you could use the code below. Please write out those values for all the entries in the `mn` and `st` arrays. If you were writing a paper, you would just use the last entry. Are you happy with that value, or do you think that if you ran the chain for longer, it would settle on a different final value?

In [14]:
```
ss1,mn1,st1 = running_mean_std(chain[:,2], runsteps);
ss2,mn2,st2 = running_mean_std(chain[:,1], runsteps);
```

In [15]:
```
[@printf("The mean and variance after for m in step %.2f is %.2f +/- %.2f \n", ss1
```

```
The mean and variance after for m in step 500.00 is 2.23 +/- 0.04
The mean and variance after for m in step 1000.00 is 2.19 +/- 0.06
The mean and variance after for m in step 1500.00 is 2.19 +/- 0.05
The mean and variance after for m in step 2000.00 is 2.19 +/- 0.05
The mean and variance after for m in step 2500.00 is 2.18 +/- 0.05
The mean and variance after for m in step 3000.00 is 2.21 +/- 0.08
The mean and variance after for m in step 3500.00 is 2.24 +/- 0.10
The mean and variance after for m in step 4000.00 is 2.25 +/- 0.10
```

Answer:

There is a little fluctuation in the mean and somewhat large fluctuation (around 10%) of variance as we can see here. Since the jumps are uncorrelated there might be a very small chance of the value of `mn` and `st` to be settle down. So, if we run the chain the longer it might fluctuate within a small range. It can be compared to the behavior of optimizer function fluctuation near maxima e.g. Brownian motion..

Part 2. Metropolis-Hastings MCMC, with uncorrelated Gaussian jumps and good jump sizes!

Please repeat the steps above, but with `jump_sizes` that you found in the tutorial session that resulted in good acceptance fractions. (If you need to, you can also check later on in the `MCMC-filled-in` notebook!)

With better step sizes, what do you observe about how much the running mean and standard deviation changes? Does it look like the chain has converged after 5000 steps?

Answer:

With better `jump_sizes` it is clear that convergence has improved a lot. It looks like the chain has converged after 5000 steps. The codes for this short experiment is provided below. .

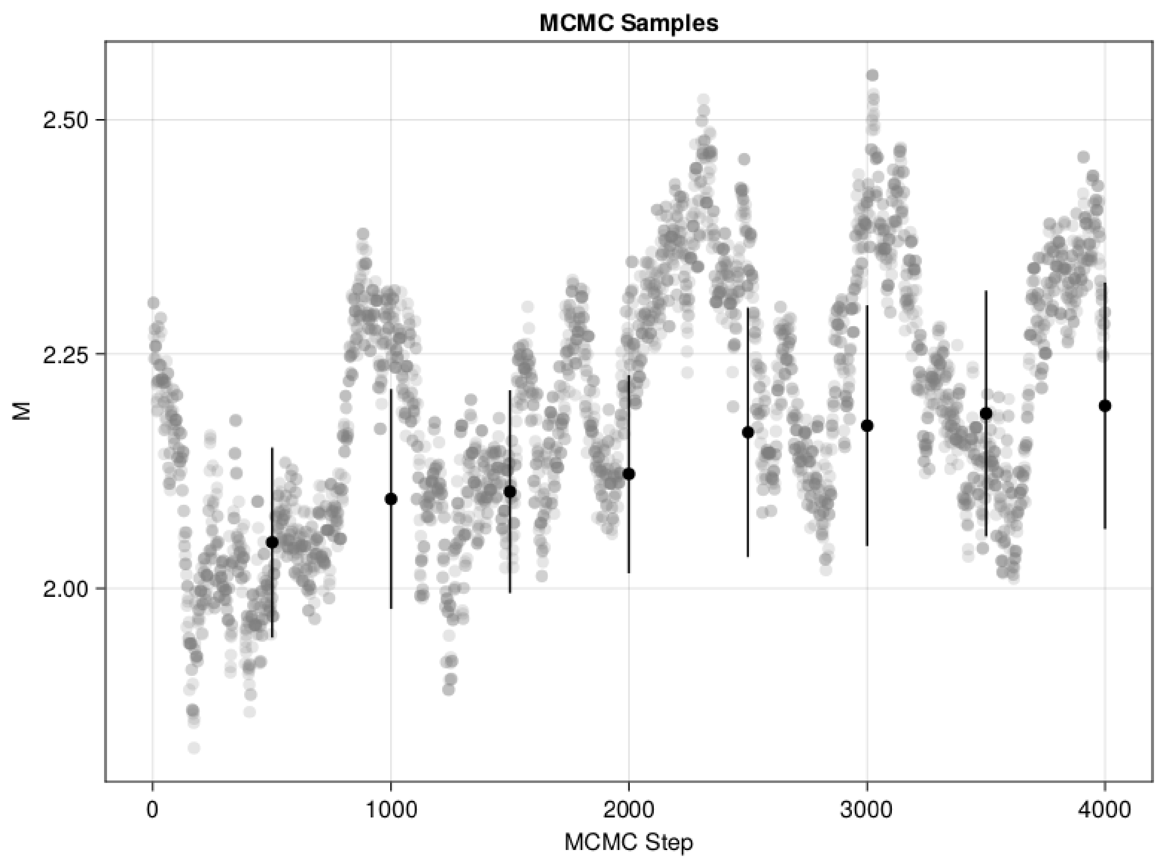
In [16]:
```
# initial B,M
initial_pos = [0., 1.0]
# proposal distribution: jump sizes for B,M
# better jump size
jump_sizes = [3., 0.05]

# How many MCMC steps to take
nsteps = 5000
# How many samples to throw away as "burn-in"
nburn = 1000

# Run MCMC!
wholechain,accept = mcmc(log_posterior_one, (x,y,yerr),
                         propose_gaussian, jump_sizes,
                         initial_pos, nsteps)
println("Fraction of moves accepted:", accept)
chain = wholechain[nburn+1:end, :]
size(chain)
```

```
Fraction of moves accepted:0.552
```

Out[16]:
```
(4000, 2)
```

In [17]:
```
# Plot the parameter values in the chain!
f = Figure()
ax = Axis(f[1, 1], xlabel="MCMC Step", ylabel="M", title="MCMC Samples")
scatter!(chain[:,2], color=(:grey, 0.2), label="Samples")

# Let's measure the running mean and variance after different chain lengths
runsteps = 500
ss,mn,st = running_mean_std(chain[:,2], runsteps)

scatter!(ss, mn, color=:black, label="Mean + Std up to this sample")
errorbars!(ss, mn, st)
#Legend(f[1,2],ax)
f
```

Out[17]:



In [18]:

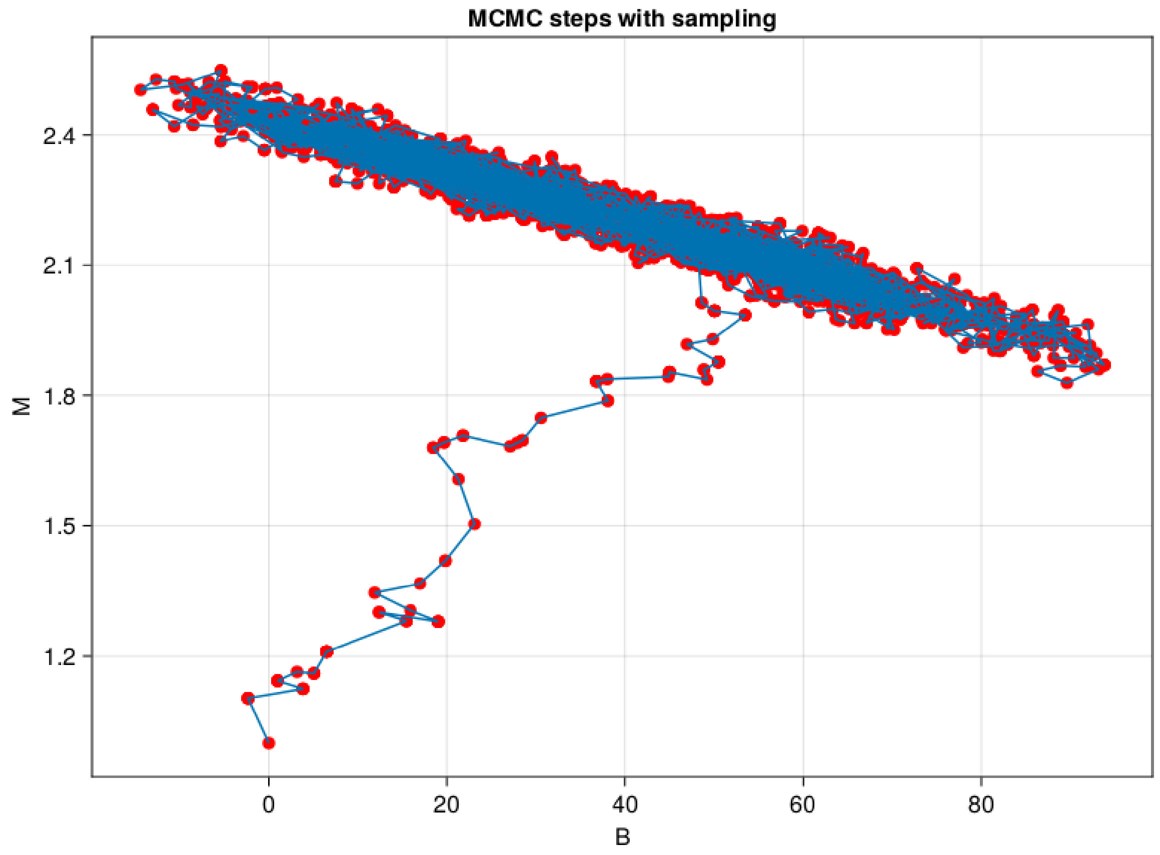
```
[@printf("The mean and variance after step %.2f is %.2f +/- %.2f \n", ss[i] ,mn[i]
```

```
The mean and variance after step 500.00 is 2.05 +/- 0.10
The mean and variance after step 1000.00 is 2.10 +/- 0.12
The mean and variance after step 1500.00 is 2.10 +/- 0.11
The mean and variance after step 2000.00 is 2.12 +/- 0.11
The mean and variance after step 2500.00 is 2.17 +/- 0.13
The mean and variance after step 3000.00 is 2.17 +/- 0.13
The mean and variance after step 3500.00 is 2.19 +/- 0.13
The mean and variance after step 4000.00 is 2.19 +/- 0.13
```

In [19]:

```
# Let's have a look at the resulting samples!
f = Figure()
Axis(f[1, 1], xlabel="B", ylabel="M", title="MCMC steps with sampling")
scatter!(wholechain[:,1], wholechain[:,2], color=:red)
lines!(wholechain[:,1], wholechain[:,2])
f
```

Out[19]:



Part 3. Metropolis-Hastings MCMC, with correlated Gaussian jumps and good jump sizes!

In the first two parts, we are using the `propose_gaussian` function to propose the jumps to take. That function takes an array of two standard deviations, one for the `B` direction and one for the `M` direction. It draws two random Gaussian values and multiplies one by the `B` step size and the other by the `M` step size. This results in a proposal ellipse that is axis-aligned in the `B`, `M` plane.

In this part, I want you to write a proposal function that proposes jumps in the `B,M` plane that are Gaussian distributed but with a covariance between the `B` and `M` values, so that the proposals are inclined in the `B,M` plane, approximately matching the shape of the likelihood contours.

We are going to *cheat* a bit by using the `chain` computed above to define our proposal distribution!

In [21]:
```
# We will take the "chain" computed above and compute its covariance using the cov
C = cov(chain)
```

Out[21]:
```
2×2 Matrix{Float64}:
 487.886    -2.82214
  -2.82214   0.0173188
```

This is the covariance matrix between `M` and `B`. The diagonals are the variances of the individual variables -- we know that `B` has a larger spread of allowed values, so its variance is larger than the variance of `M`. The covariance term, which is symmetric, is negative -- the variables are *anti*-correlated.

Please complete the `propose_gaussian_cov` function below, and confirm that it works by drawing 1000 samples centered on point `(0,0)` and confirming that the covariance is approximately equal to the desired covariance (C), and plot the samples and confirm that they are strongly anti-correlated.

Answer:

Key idea here is to find the principal direction of gaussian distribution and we want to make choice such that on that two direction we choose the points independently. This will lead to the point choice in the original direction with the expected covariance. So the apparent rotation we need to can be shown mathematically.

The probability of the jump relative with the terrain for the next jump related to the following quantity (given present location is at `B` = 0, `M` = 0 and the next location is (b, m)):

$$M = exp\left(-\frac{1}{2}[b \quad m]\begin{bmatrix} c_{11} & c_{12} \\ c_{12} & c_{22} \end{bmatrix}\begin{bmatrix} b \\ m \end{bmatrix}\right)$$

If we diagonalize the C covariance matrix in the middle we can write it in the following form :

$$M = exp\left(-\frac{1}{2}[b \quad m]\begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix}\begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}\begin{bmatrix} u_{11} & u_{12} \\ u_{21} & c_{22} \end{bmatrix}^T\begin{bmatrix} b \\ m \end{bmatrix}\right)$$

This is possible because cause C is symmetric matrix in the present context; hence λ_1, λ_2 are real. Now we rotate the vector $[b \quad m]$ in the following way:

$$\begin{aligned} M = exp\Bigg(-\frac{1}{2}[b \quad m]\begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix}\begin{bmatrix} \sqrt{\lambda_1} & 0 \\ 0 & \sqrt{\lambda_2} \end{bmatrix}\begin{bmatrix} u_{11} & u_{12} \\ u_{21} & c_{22} \end{bmatrix}^T\begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix}\Bigg[\\ = exp\left(-\frac{1}{2}[b' \quad m']\begin{bmatrix} b' \\ m' \end{bmatrix}\right) \\ = exp(-\frac{1}{2}b'^2)exp(-\frac{1}{2}m'^2) \end{aligned}$$

where we define

$$\begin{bmatrix} b \\ m \end{bmatrix} = \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix}\begin{bmatrix} \sqrt{\lambda_1} & 0 \\ 0 & \sqrt{\lambda_2} \end{bmatrix}\begin{bmatrix} u_{11} & u_{12} \\ u_{21} & c_{22} \end{bmatrix}^T\begin{bmatrix} b' \\ m' \end{bmatrix}$$

Finally we can use this new transformation matrix for the next step location. This can be understood as we transform regular coordinate to the principal directions and perform separate scaling for jumps in different directions and then return back to the original coordinates.

In [22]:
```
function propose_gaussian_cov(p, cov)
    """
    A Gaussian proposal distribution for mcmc, that draws samples with covariance =

    That is, it draws samples from the multivariate Gaussian.
    *p*: the point in parameter space to jump from
    *cov*: the covariance between the points
    """

    # Hint: You will probably want to draw two independent regular Gaussian samples
    g_sample = randn(length(p))

    ### FILL ME IN!
    ## do something with g_sample...
    ###################################################
    V,U=eigen(C)
    steps =  U*diagm(sqrt.(V))*U' * g_sample
    ###################################################
    return p .+ steps # ....
end;
```

(Hint: "sample from gaussian covariance" is a pretty good Google search phrase! Another hint: you have the *covariance* but you want to multiply your independent Gaussian samples by something that is like the *matrix square root* of the covariance!)

In [23]:
```
# To test your `propose_gaussian_cov` function, you can try drawing 1,000 samples f
pp = Float64[]
for i in 1:1000
    append!(pp, propose_gaussian_cov([0.,0.], C))
end
pp = reshape(pp, 2,:)';
```

Compute the covariance of the samples generated by your function. Do they match the desired covariance, C?

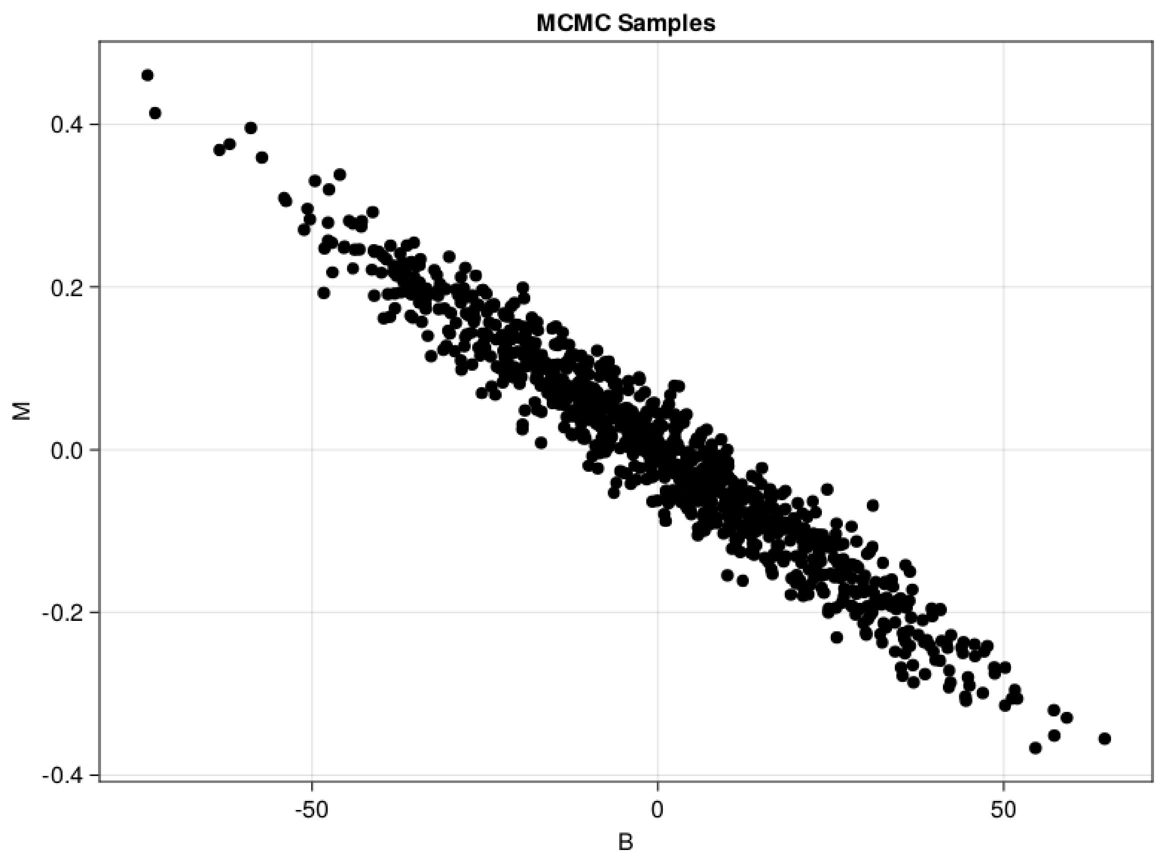
In [24]:
```
cov(pp)
```

Out[24]:
```
2×2 Matrix{Float64}:
 499.684    -2.89677
  -2.89677   0.017849
```

**Plot the samples generated by your function. Are they an inclined ellipse with the expected larger scatter in B than in M?

In [25]:
```
f = Figure()
Axis(f[1, 1], xlabel="B", ylabel="M", title="MCMC Samples")
plot!(pp[:,1], pp[:,2])
f
```

Out[25]:



FINALLY, let's run the MCMC algorithm using your new proposal function (and the target covariance matrix)!

In [35]:

```
# initial B,M
initial_pos = [0., 1.0]
# Run MCMC!
wholechain,accept = mcmc(log_posterior_one, (x,y,yerr),
                         propose_gaussian_cov, C,
                         initial_pos, nsteps)
println("Fraction of moves accepted:", accept)
chain = wholechain[nburn+1:end, :]
size(chain)
```

```
Fraction of moves accepted:0.5228
```

Out[35]:

```
(4000, 2)
```

Repeat the plots we made in Part 1 and compare the results. Does it look like this chain is converged after 5,000 steps? Did the burn-in take the same amount of steps, or was it maybe slower or faster?

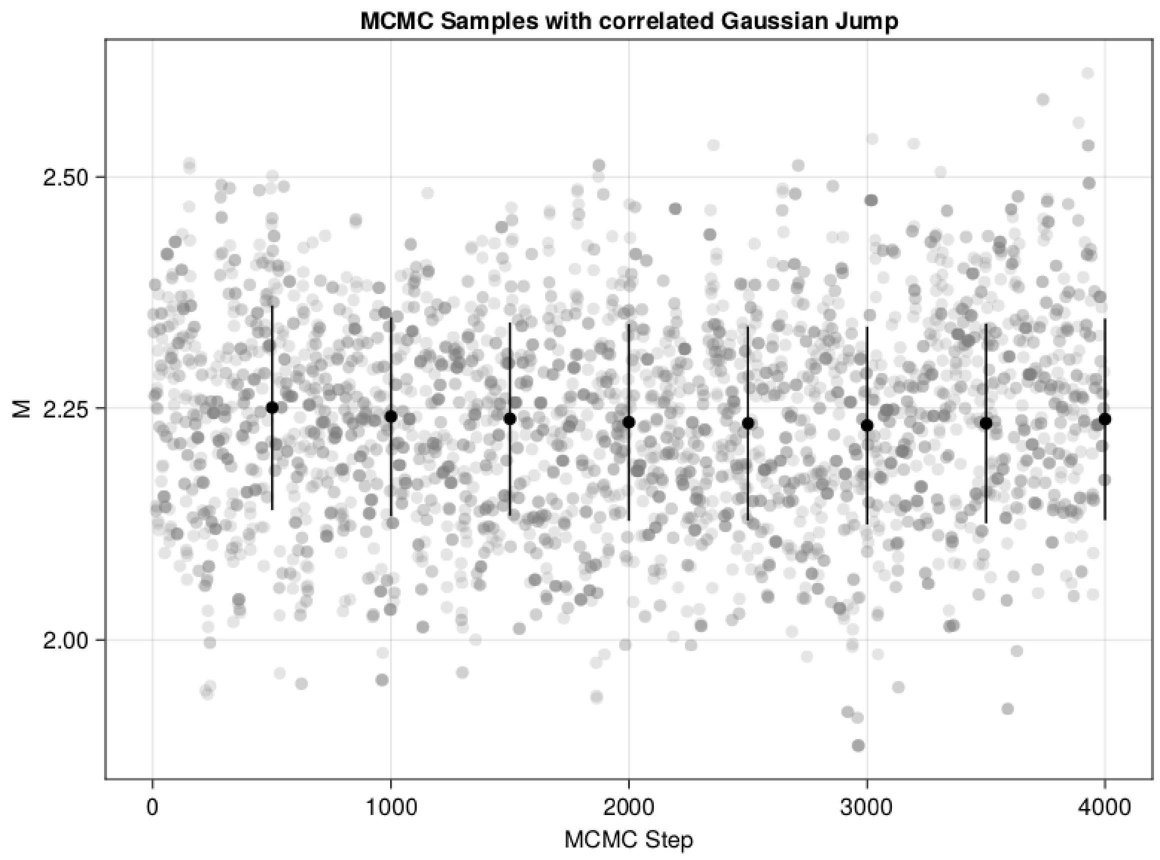
Answer:

In [36]:

```
# Plot the parameter values in the chain!
f = Figure()
ax = Axis(f[1, 1], xlabel="MCMC Step", ylabel="M", title="MCMC Samples with correl
scatter!(chain[:,2], color=(:grey, 0.2), label="Samples")

# Let's measure the running mean and variance after different chain lengths
runsteps = 500
ss,mn,st = running_mean_std(chain[:,2], runsteps)

scatter!(ss, mn, color=:black, label="Mean + Std up to this sample")
```

```
errorbars!(ss, mn, st)
#Legend(f[1,2],ax)
f
```

Out[36]:



In [37]:

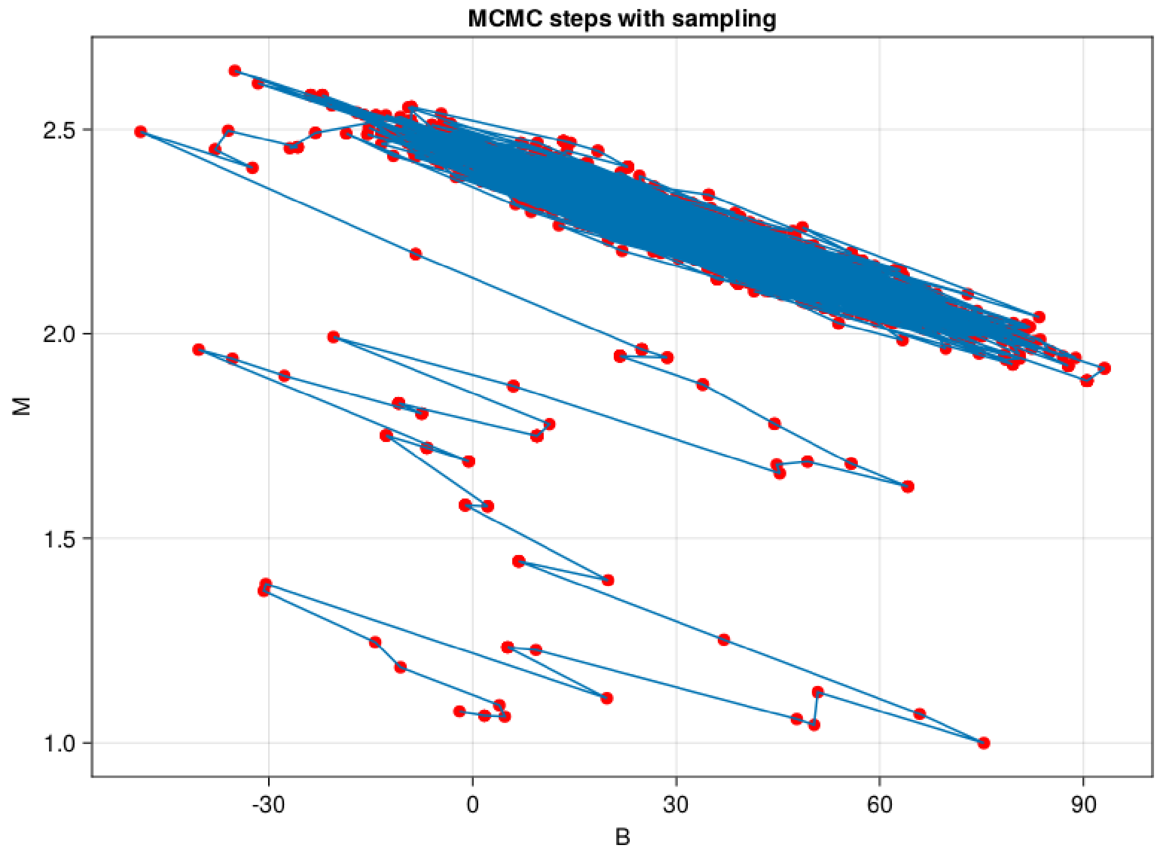
```
[@printf("The mean and variance after step %.2f is %.2f +/- %.2f \n", ss[i] ,mn[i]
```

```
The mean and variance after step 500.00 is 2.25 +/- 0.11
The mean and variance after step 1000.00 is 2.24 +/- 0.11
The mean and variance after step 1500.00 is 2.24 +/- 0.10
The mean and variance after step 2000.00 is 2.23 +/- 0.11
The mean and variance after step 2500.00 is 2.23 +/- 0.10
The mean and variance after step 3000.00 is 2.23 +/- 0.11
The mean and variance after step 3500.00 is 2.23 +/- 0.11
The mean and variance after step 4000.00 is 2.24 +/- 0.11
```

In [38]:

```
# Let's have a look at the resulting samples!
f = Figure()
Axis(f[1, 1], xlabel="B", ylabel="M", title="MCMC steps with sampling")
scatter!(wholechain[:,1], wholechain[:,2], color=:red)
lines!(wholechain[:,1], wholechain[:,2])
f
```

Out[38]:



It took small amount of time to reach the desired location, from that we can say that slower amount of burn-in (more step is neded here) took place than previous calculation..

Part 4. The Affine-invariant sampler.

In this part, we'll use the affine-invariant ensemble sampler.

Since the ensemble sampler moves a *group* of *walkers* at once, we have to be a bit careful how we compare it to regular MCMC. Probably the fairest comparison is if you make the total number of samples the same. That is, run the ensemble sampler for `n_ensemble = n_mcmc / n_walkers` where `n_mcmc` was the number of steps you took in the regular MCMC chain.

You may see, however, that the burn-in takes longer. So you might want to either initialize the ensemble closer to the correct answer, or run for more burn-in steps.

In [70]:
```
# We need to tell the sampler a number of things:
# - how many dimensions are being sampled
numdims = 2
# - how many walkers we want
numwalkers = 25
# In the example code below, it runs an initial "burn-in" round of sampling, then 
# burn-in samples to take.
nburn_aff = nburn ÷ numwalkers
# - how many steps to take (after burn-in)
nsteps_aff = (nsteps - nburn) ÷ numwalkers

# We also need to give the walkers some initial positions.  NOTE that you can't gi
# Here, we're just drawing uniform numbers of between 0 and 1.
x0 = rand(numdims, numwalkers)
```

```
### FILL ME IN -- maybe you want to initialize the walkers around some better poin
####################################
x0[1,:] .+= 0.0;
x0[2,:] .+= 1.0;
###################################
# for i ∈ 1:numwalkers
#     data = propose_gaussian_cov(x0[:,i], cov)
#     @show data,x0[:,i];
# end
```

In [71]:
```
# We need to pass to the sampler a function that takes *only* the vector of parame
# also needs the (x,y,yerr) values.  So to make this work, we need a "wrapper" func
# and pass them to log_likelihood_one.

ll_func(bm) = log_posterior_one(bm, (x, y, yerr))

# Here we go, let's call the AffineInvariant MCMC's "sample" function -- this is th
chain2, _ = AffineInvariantMCMC.sample(ll_func, numwalkers, x0, nburn_aff, 1)

# And here's the "real" run.
# Notice that it's passing in the end of the burn-in chain as the initial position
chain, llvals = AffineInvariantMCMC.sample(ll_func, numwalkers, chain2[:, :, end],

# And that's it!  Now "chain" contains our samples!
println("Chain:", size(chain))
```

```
Chain:(2, 25, 160)
```

In [72]:
```
# We can "flatten" that list of samples so that it's a (N x 2) array like for regu
flatchain2, flat_llvals = AffineInvariantMCMC.flattenmcmcarray(chain2, llvals);
chain2 = flatchain2'

flatchain, flat_llvals = AffineInvariantMCMC.flattenmcmcarray(chain, llvals);
chain = flatchain'
```

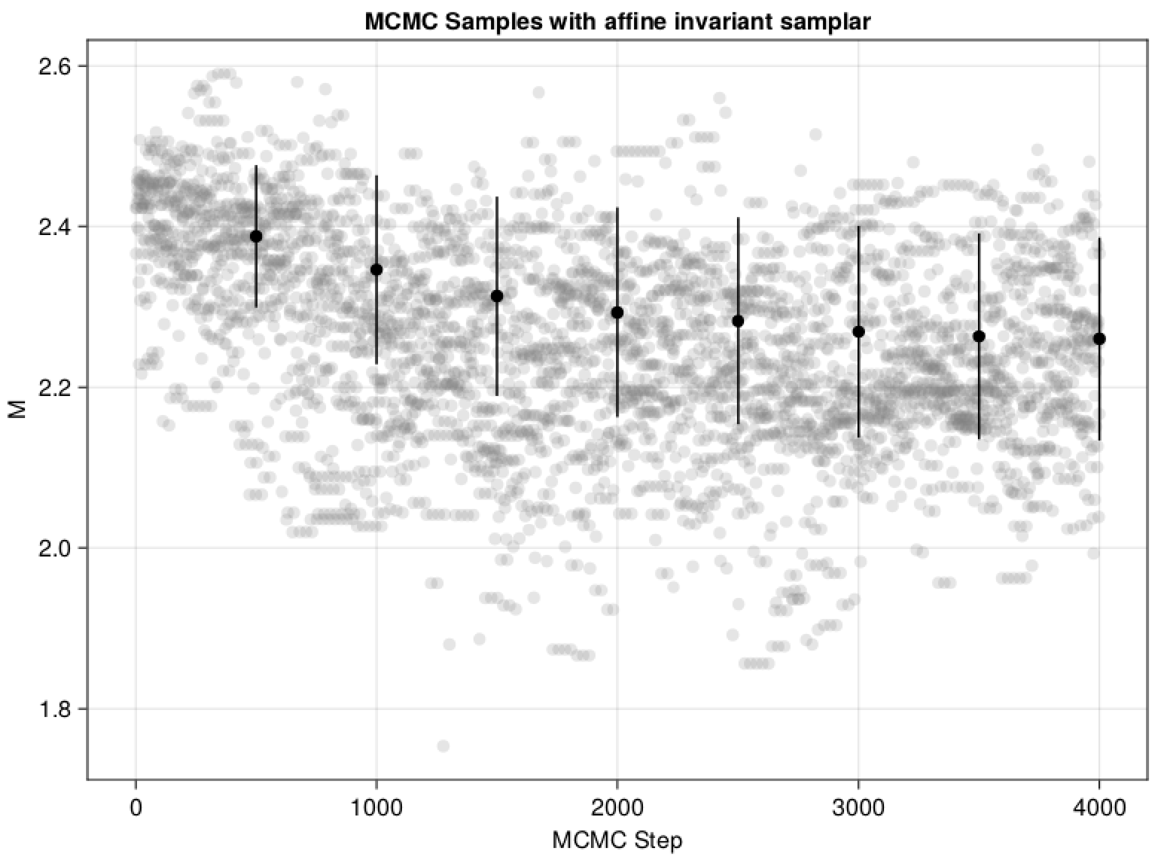
Out[72]:
```
4000×2 adjoint(::Matrix{Float64}) with eltype Float64:
  3.89023   2.42236
  2.92155   2.36623
  3.34588   2.42764
  3.03017   2.42298
 -1.14409   2.45916
  0.51762   2.45338
 -0.179328  2.39858
  2.14131   2.44438
  5.6219    2.4351
 -0.886651  2.46738
 -6.52205   2.45672
  6.134     2.33028
 -1.56221   2.395
  ⋮
 31.6577    2.26442
 29.5862    2.27934
 53.5655    2.10011
 41.4981    2.12659
 12.9289    2.37584
 32.1598    2.2755
 35.6818    2.23144
 26.0347    2.28215
 66.7096    2.03867
 35.457     2.25389
 44.5627    2.16643
 13.647     2.36634
```

Now please repeat the plots and the $M = mean \pm std$ printing from Part 1. Does it look like the ensemble sampler is converging faster? Does the burn-in look faster or slower?

Answer:

In [73]:
```
# Plot the parameter values in the chain!
f = Figure()
ax = Axis(f[1, 1], xlabel="MCMC Step", ylabel="M", title="MCMC Samples with affine
scatter!(chain[:,2], color=(:grey, 0.2), label="Samples")

# Let's measure the running mean and variance after different chain lengths
runsteps = 500
ss,mn,st = running_mean_std(chain[:,2], runsteps)

scatter!(ss, mn, color=:black, label="Mean + Std up to this sample")
errorbars!(ss, mn, st)
#Legend(f[1,2],ax)
f
```

Out[73]:



In [74]:
```
[@printf("The mean and variance after step %.2f is %.2f +/- %.2f \n", ss[i] ,mn[i]
```

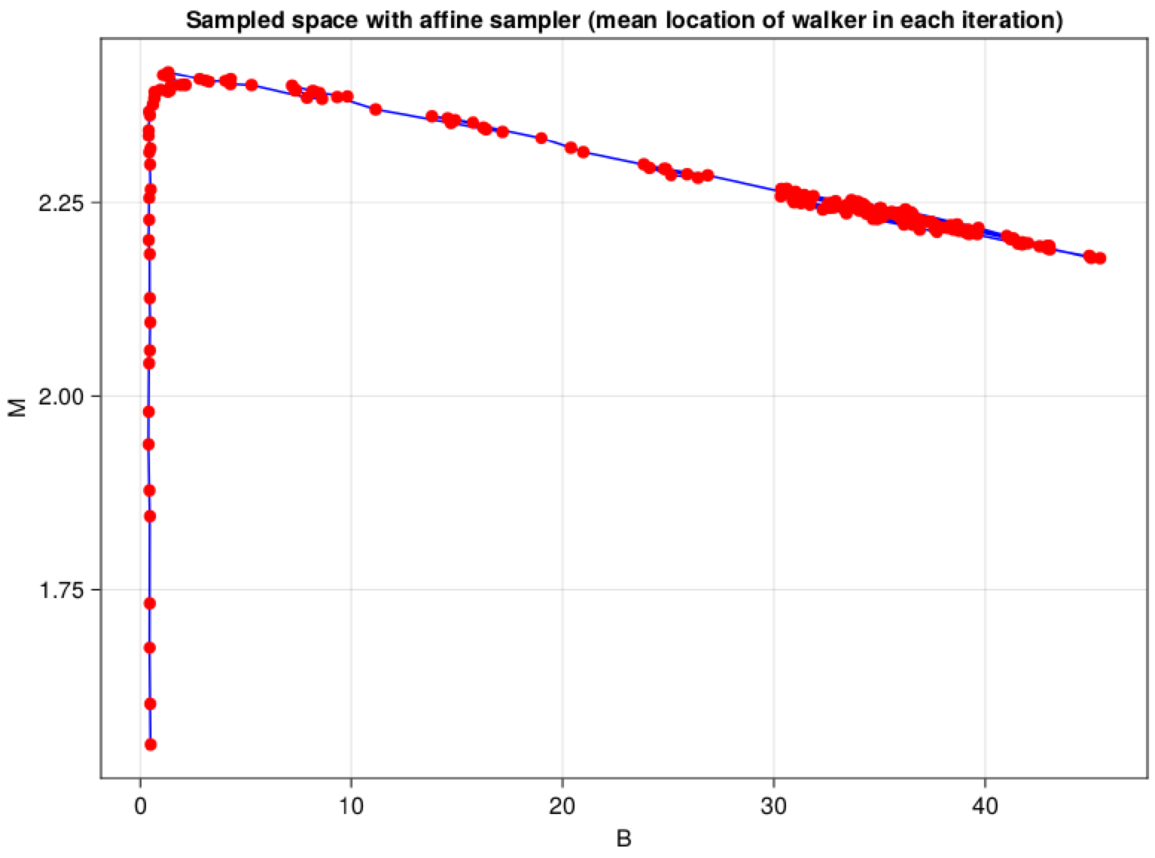
```
The mean and variance after step 500.00 is 2.39 +/- 0.09
The mean and variance after step 1000.00 is 2.35 +/- 0.12
The mean and variance after step 1500.00 is 2.31 +/- 0.12
The mean and variance after step 2000.00 is 2.29 +/- 0.13
The mean and variance after step 2500.00 is 2.28 +/- 0.13
The mean and variance after step 3000.00 is 2.27 +/- 0.13
The mean and variance after step 3500.00 is 2.26 +/- 0.13
The mean and variance after step 4000.00 is 2.26 +/- 0.13
```

It seems it is converging faster than other methods. However, if the initial location is far away from the higher probability region the burn-in slows down shown in the figure below..

In [126…

```
# Let's have a look at the resulting samples!
f = Figure()
Axis(f[1, 1], xlabel="B", ylabel="M", title="Sampled space with affine sampler (mea
cc = vcat(chain2, chain)
mean_location_M = Float64[]
mean_location_B = Float64[]
for i in 1:25:(size(cc,1))
    push!(mean_location_M, mean(cc[i:(i+numwalkers-1),1]))
    push!(mean_location_B, mean(cc[i:(i+numwalkers-1),2]))
    # scatter!(x0[1,:], x0[2,:], color=:red)
end
lines!(mean_location_M,mean_location_B, color=:blue)
scatter!(mean_location_M,mean_location_B, color=:red)
f
```

Out[126]:



Try changing the number of walkers. Try 10, or 100. How do the burn-in and convergence speed look? Of 10, 20, or 100 walkers, which one looks best to you?

In [177…

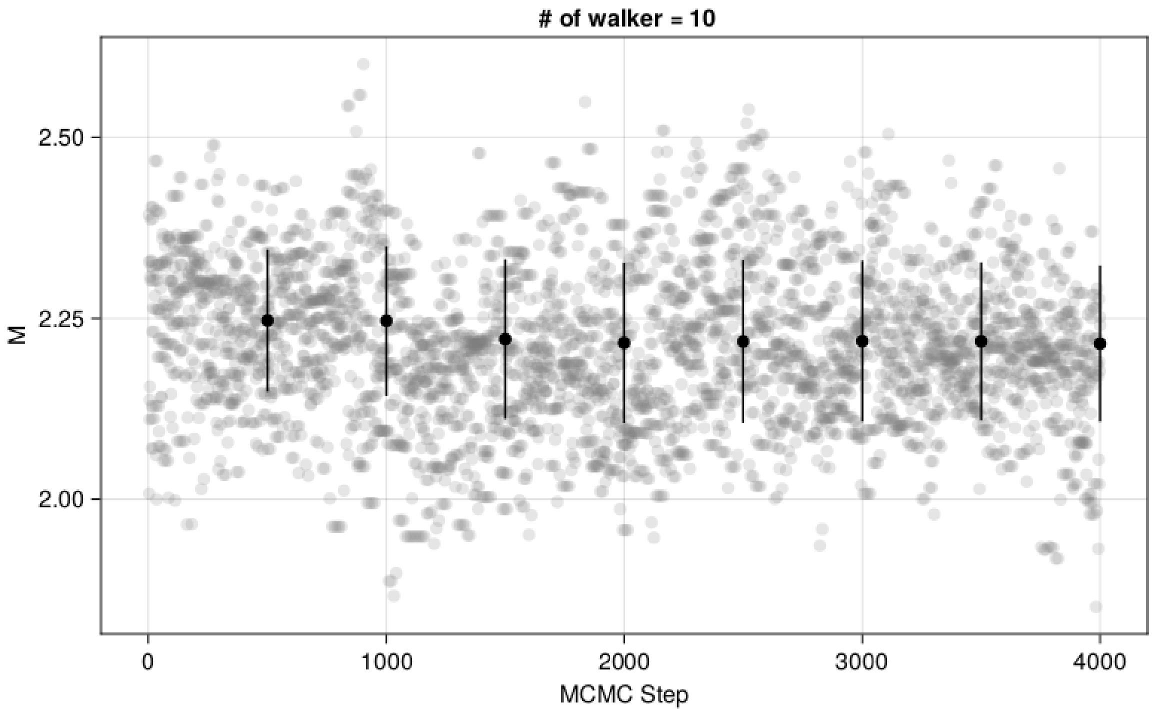
```
function walker_affine_plot(numwalkers)
    nburn_aff = nburn ÷ numwalkers
    nsteps_aff = (nsteps - nburn) ÷ numwalkers
    x0 = rand(numdims, numwalkers)
    x0[1,:] .+= 0.0;
    x0[2,:] .+= 1.0;
    chain2, _ = AffineInvariantMCMC.sample(ll_func, numwalkers, x0, nburn_aff, 1)
    chain, llvals = AffineInvariantMCMC.sample(ll_func, numwalkers, chain2[:, :, e
    flatchain2, flat_llvals = AffineInvariantMCMC.flattenmcmcarray(chain2, llvals)
    chain2 = flatchain2'
    flatchain, flat_llvals = AffineInvariantMCMC.flattenmcmcarray(chain, llvals);
    chain = flatchain'
    f = Figure(resolution=(800,1000))
    ax = Axis(f[1, 1], xlabel="MCMC Step", ylabel="M", title=" # of walker = "*str
    scatter!(chain[:,2], color=(:grey, 0.2), label="Samples")
    runsteps = 500
    ss,mn,st = running_mean_std(chain[:,2], runsteps)

    scatter!(ss, mn, color=:black, label="Mean + Std up to this sample")
    errorbars!(ss, mn, st)
    Axis(f[2,1], xlabel="B", ylabel="M", title="MCMC samples (mean location)"*" # 
    cc = vcat(chain2, chain)
    mean_location_M = Float64[]
    mean_location_B = Float64[]
    for i in 1:numwalkers:(size(cc,1))
        push!(mean_location_M, mean(cc[i:(i+numwalkers-1),1]))
        push!(mean_location_B, mean(cc[i:(i+numwalkers-1),2]))
        # scatter!(x0[1,:], x0[2,:], color=:red)
    end
    lines!(mean_location_M,mean_location_B, color=:blue)
    scatter!(mean_location_M,mean_location_B, color=:red)
    [@printf("The mean and variance after step %.2f is %.2f +/- %.2f \n", ss[i] ,m

    return f
end
```

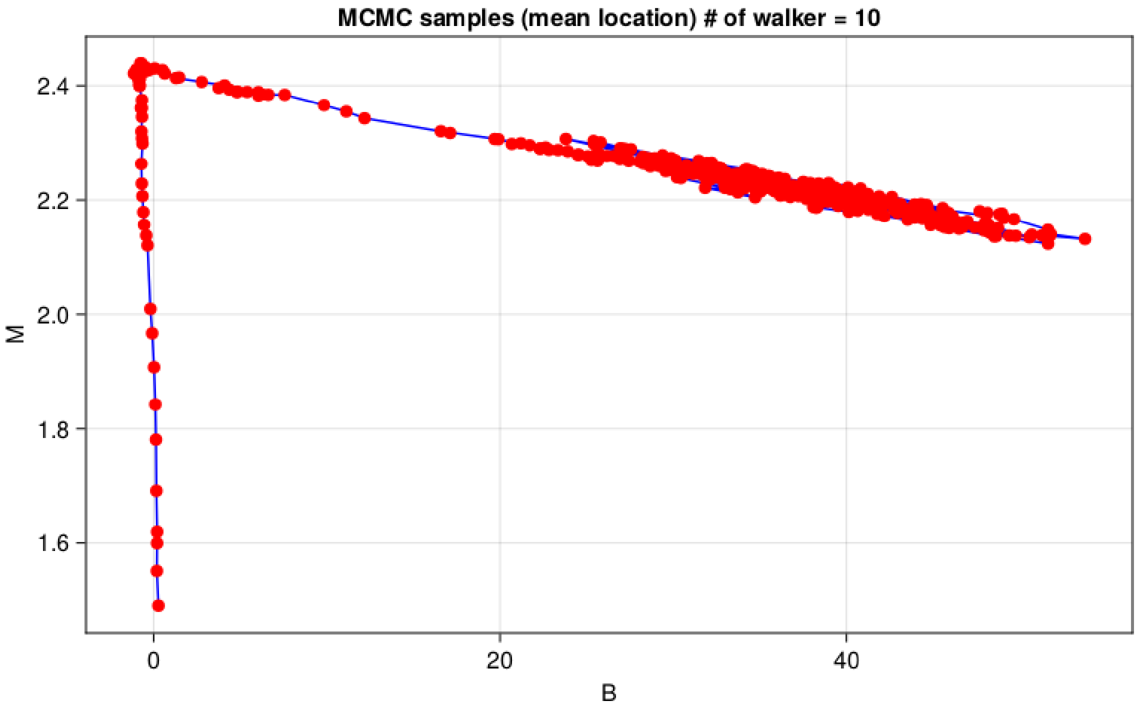
Out[177]:
```
walker_affine_plot (generic function with 1 method)
```

In [178…
```
f = walker_affine_plot(10)
```

```
The mean and variance after step 500.00 is 2.25 +/- 0.10
The mean and variance after step 1000.00 is 2.25 +/- 0.10
The mean and variance after step 1500.00 is 2.22 +/- 0.11
The mean and variance after step 2000.00 is 2.22 +/- 0.11
The mean and variance after step 2500.00 is 2.22 +/- 0.11
The mean and variance after step 3000.00 is 2.22 +/- 0.11
The mean and variance after step 3500.00 is 2.22 +/- 0.11
The mean and variance after step 4000.00 is 2.22 +/- 0.11
```

Out[178]:



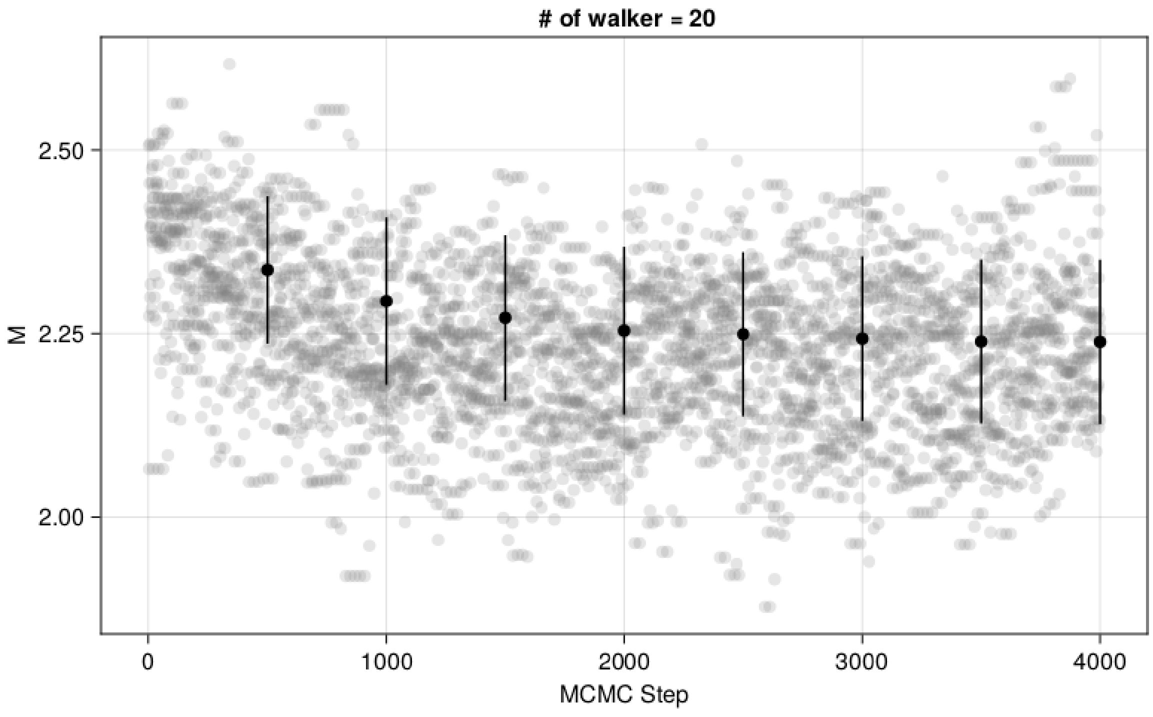


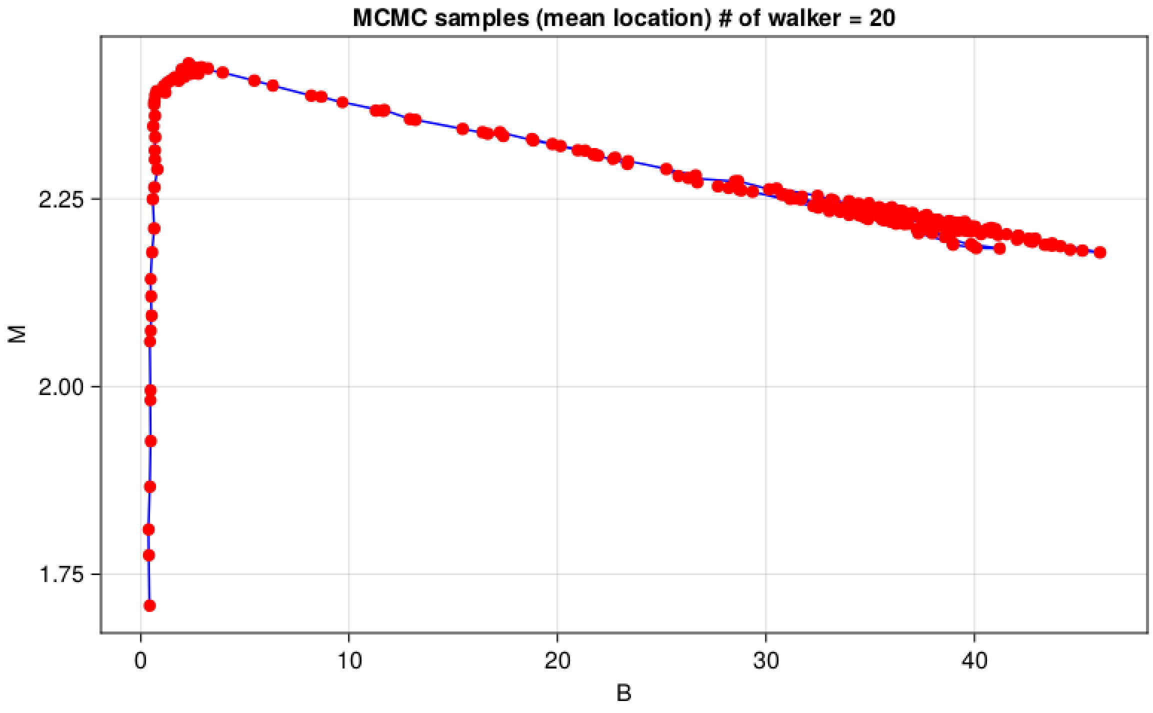
In [179…

```
f = walker_affine_plot(20)
```

```
The mean and variance after step 500.00 is 2.34 +/- 0.10
The mean and variance after step 1000.00 is 2.29 +/- 0.11
The mean and variance after step 1500.00 is 2.27 +/- 0.11
The mean and variance after step 2000.00 is 2.25 +/- 0.11
The mean and variance after step 2500.00 is 2.25 +/- 0.11
The mean and variance after step 3000.00 is 2.24 +/- 0.11
The mean and variance after step 3500.00 is 2.24 +/- 0.11
The mean and variance after step 4000.00 is 2.24 +/- 0.11
```

Out[179]:



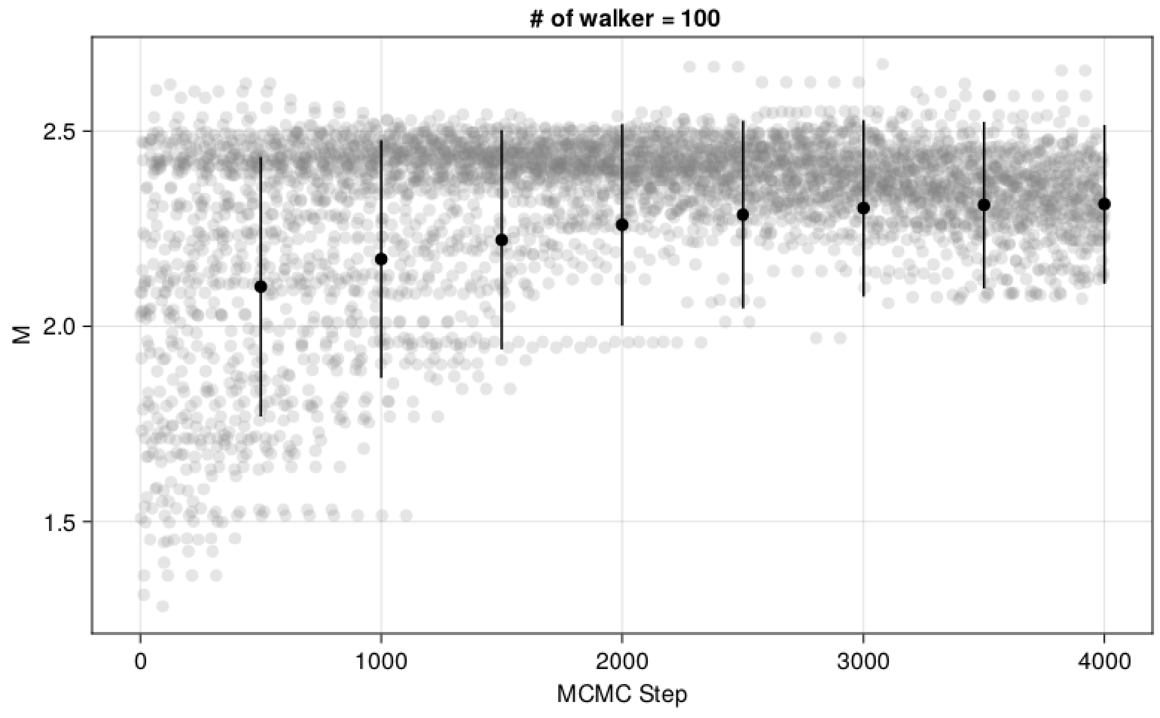


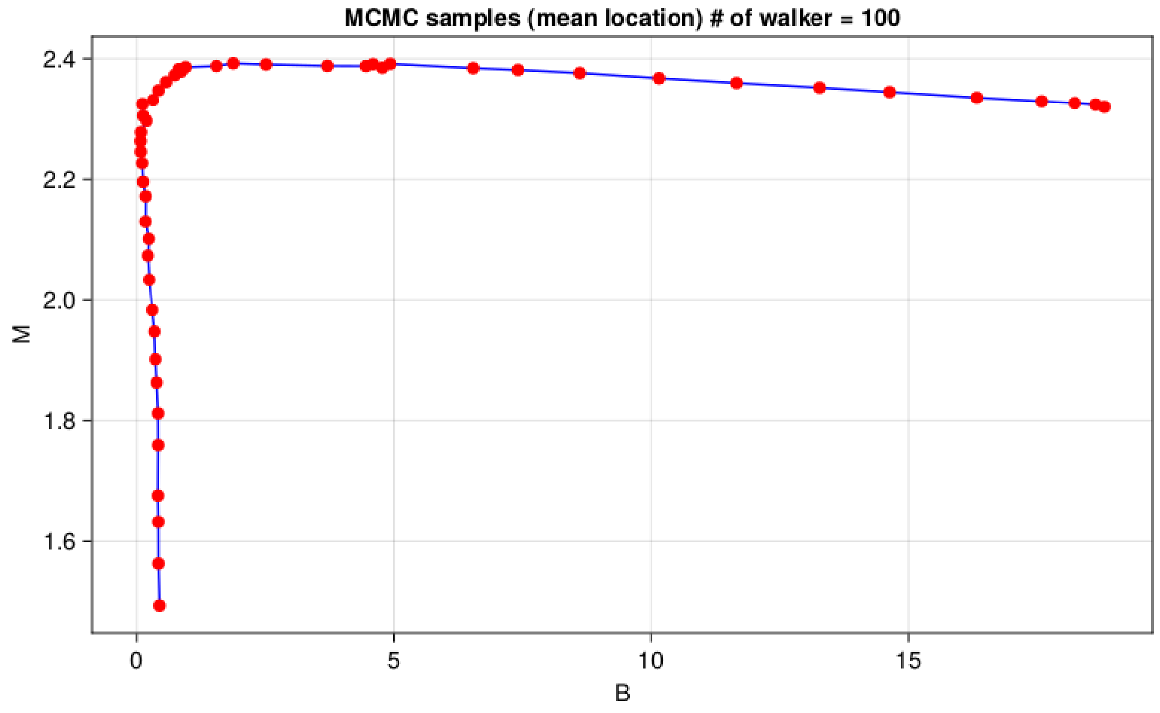
In [180…

```
f = walker_affine_plot(100)
```

```
The mean and variance after step 500.00 is 2.10 +/- 0.33
The mean and variance after step 1000.00 is 2.17 +/- 0.30
The mean and variance after step 1500.00 is 2.22 +/- 0.28
The mean and variance after step 2000.00 is 2.26 +/- 0.26
The mean and variance after step 2500.00 is 2.29 +/- 0.24
The mean and variance after step 3000.00 is 2.30 +/- 0.23
The mean and variance after step 3500.00 is 2.31 +/- 0.21
The mean and variance after step 4000.00 is 2.31 +/- 0.20
```

Out[180]:





It seems convergence with 10 is better and also walker burn-in is quick. May be the optimum choice might be close to 10 for this data.. .

The End! Good work!

Acknowledgement

I didn't collaborate with anyone for this homework.