MCMC-homework-assignment

February 10, 2024

1 PSI Numerical Methods 2024 - Homework Assignment on Model Fitting & MCMC

1.1 Code available here

We're going to put together everything we have learned so far to re-do the data analysis for the Perlmutter et al. 1999 paper on the discovery of dark energy! (https://ui.adsabs.harvard.edu/abs/1999ApJ...517..565P/abstract)

Start by `Forking` this repository on Github: https://github.com/dstndstn/PSI-Numerical-Methods-2024-MCMC-Homework And then clone the repository to your laptop or to Symmetry. You can modify this notebook, and when you are done, save it, and then `git commit -a` the results, and `git push` them back to your fork of the repository. You will "hand in" your homework by giving a link to your Github repository, where the marker will be able to read your notebook.

First, a little bit of background on the cosmology and astrophysics. The paper reports measurements of a group of supernova explosions of a specific type, "Type 1a". These are thought to be caused by a white dwarf star that has a companion star that "donates" gas to the white dwarf. It gradually gains mass until it exceeds the Chandresekhar mass, and explodes. Since they all explode through the same mechanism, and with the same mass, they should all have the same intrinsic brightess. It turns out to be a *little* more complicated than that, but in the end, these Type-1a supernovae can be turned into "standard candles", objects that are all the same brightness. If you can also measure the redshift of each galaxy containing the supernova, then you can map out this brightness–redshift relation, and the shape of that relation depends on how the universe grows over cosmic time. In turn, the growth rate of the universe depends on the contents of the universe!

In this way, these Type-1a supernova allow us to constrain the parameters of a model of the universe. Specifically, the model is called "Lambda-CDM", a universe containing dark energy and matter (cold dark matter, plus regular matter). We will consider a two-parameter version of this model: Ω_M, the amount of matter, and Ω_Λ, the amount of dark energy. These are in cosmology units of "energy density now relative to the critical density", where the critical density is the energy density you need for the universe to be spatially flat (angles of a large triangle sum to 180 degrees). So $\Omega_M = 1, \Omega_\Lambda = 0$ would be a flat universe containing all matter, while $\Omega_M = 0.25, \Omega_\Lambda = 0.5$ would be a spatially closed universe with dark energy and matter. Varying these ingredients changes the growth history of the universe, which changes how much the light from a supernova is redshifted, and how its brightness drops off with distance.

(In the code below, we will call these `Omega_M` $= \Omega_M$ and `Omega_DE` $= \Omega_\Lambda$.)

Distance measurements in cosmology are complicated – see https://arxiv.org/abs/astro-ph/9905116 for details! For this assignment, we will use a cosmology package that will handle all this for us.

All we need to use is the "luminosity distance", which is the one that tells you how objects get fainter given a redshift.

[1]:
```
# Let's start by installing the Cosmology package!
using Pkg
Pkg.add("Cosmology")
```

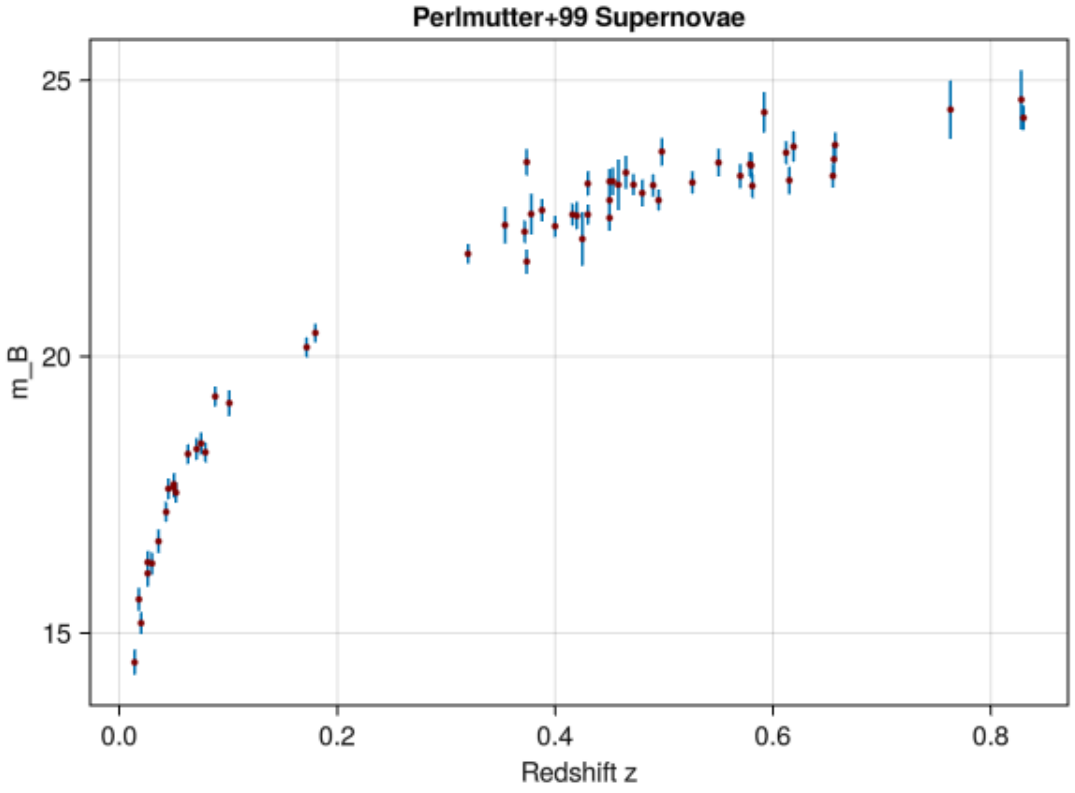
```
  Resolving package versions…
 No Changes to `~/.julia/environments/v1.10/Project.toml`
 No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

[2]:
```
# We'll also end up using all our old friends:
using WGLMakie
using CSV
using DataFrames
using Cosmology
using Statistics
```

[3]:
```
# There is a data file in this directory, taken basically straight out of the␣
 ↪Perlmutter+1999 paper.  We can read it with the CSV package.
data = CSV.read("p99-data.txt", DataFrame, delim=" ", ignorerepeated=true);
```

[4]:
```
# Make a copy of the data columns that we want to treat as the "y" measurements.
# These are the measured brightnesses, and their Gaussian uncertainties␣
 ↪(standard deviations).
data.mag = data.m_b_eff
data.sigma_mag = data.sigma_m_b_eff;
```

[5]:
```
f = Figure()
Axis(f[1,1], title="Perlmutter+99 Supernovae", xlabel="Redshift z",␣
 ↪ylabel="m_B")
errorbars!(data.z, data.mag, data.sigma_mag)
scatter!(data.z, data.mag, markersize=5, color=:maroon)
save("perlmutterPlot.png", f)
display(f)
```




```
Bonito.App(Bonito.var"#8#14"{WGLMakie.var"#20#21"{WGLMakie.Screen,␣
 ↪Scene}}(WGLMakie.var"#20#21"{WGLMakie.Screen, Scene}(WGLMakie.Screen(
    framerate = 30.0,
    resize_to = nothing,
    px_per_unit = automatic,
    scalefactor = automatic,
    session = nothing,
    three = Channel{Bool}(1) (empty),
    scene = Scene (600px, 450px):
  0 Plots
  1 Child Scene:
      Scene (600px, 450px),
), Scene (600px, 450px):
  0 Plots
  1 Child Scene:
      Scene (600px, 450px))), Base.RefValue{Union{Nothing, Bonito.
 ↪Session}}(nothing), "Bonito App", false)
```

[5]:
```
WGLMakie.Screen(
    framerate = 30.0,
    resize_to = nothing,
    px_per_unit = automatic,
    scalefactor = automatic,
```

```
    session = true,
    three = Channel{Bool}(1) (1 item available),
    scene = Scene (600px, 450px):
  0 Plots
  1 Child Scene:
    Scene (600px, 450px),
)
```

[6]:
```
# Here is how we will use the "cosmology" package.  This will create a␣
 ↪cosmology "object" with the parameters we pass in.
# It does not take an Omega_Lambda parameter; instead, it takes Omega_Matter,␣
 ↪and Omega_K (for "curvature"), where
# Omega_K = 1. - Omatter - Olambda.  We will also pass in "Tcmb=0", which tells␣
 ↪it to ignore the effects of radiation.

universe = cosmology(OmegaK=0.1, OmegaM=0.4, Tcmb=0)
@show universe
@show universe.Ω_Λ;
```

```
universe = Cosmology.OpenLCDM{Float64}(0.69, 0.1, 0.5, 0.4, 0.0)
universe.Ω_Λ = 0.5
```

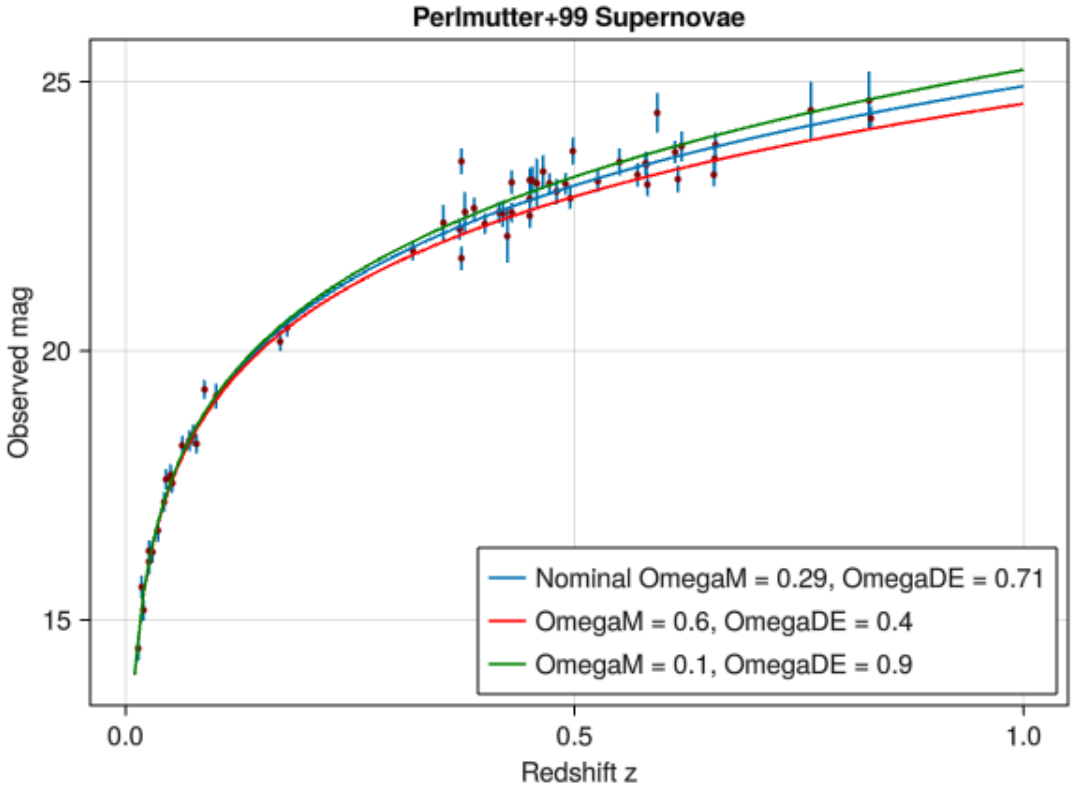
[7]:
```
# We can then pass that "universe" object to other functions to compute things␣
 ↪about it.  Basically the only one you'll
# need is this `distance_modulus`, which tell you, in _magnitudes_, how much␣
 ↪fainter an object is at the given redshift,
# versus how faint it would be if it were 10 parsecs away.

function distance_modulus(universe, z)
    DL = luminosity_dist(universe, z)
    # DL is in Megaparsecs; the distance for absolute to observed mag is 10 pc.
    5. * log10.(DL.val * 1e6 / 10.)
end;
```

There is one more parameter to the model we will be fitting: M, the *absolute magnitude* of the supernovae. This is a "nuisance parameter" - a parameter that we have to fit for, but that we don't really care about; it's basically a calibration of what the intrinsic brightness of a supernova is. To start out, we will fix this value to a constant, but later we will fit for it along with our Omegas.

The *observed* brightness of a supernova will be its *absolute mag* plus its *distance modulus*. The *distance modulus* depends on the redshift z and our parameters Omega_M and Omega_DE.

[8]:
```
# We'll cheat a bit and use a "nominal" cosmology with currently-accepted␣
 ↪values of Omega_M = 0.29, Omega_DE = 0.71.
nominal = cosmology(Tcmb=0)

f = Figure()
```

```
ax = Axis(f[1,1], title="Perlmutter+99 Supernovae", xlabel="Redshift z",␣
 ↪ylabel="Observed mag")
errorbars!(data.z, data.mag, data.sigma_mag)
scatter!(data.z, data.mag, markersize=5, color=:maroon)

# Compute the average absolute magnitude M given nominal cosmology -- ie, an␣
 ↪estimate of the absolute mag of the supernovae
DLx = map(z->distance_modulus(nominal, z), data.z)
abs_mag = median(data.mag - DLx)

# Here's another way to plot a function evaluated on a grid of values.
zgrid = 0.01:0.01:1.
DL = map(z->distance_modulus(nominal, z), zgrid)
lines!(zgrid, DL .+ abs_mag, label="Nominal OmegaM = 0.29, OmegaDE = 0.71")

universe = cosmology(OmegaK=0.0, OmegaM=0.6, Tcmb=0)
DL = map(z->distance_modulus(universe, z), zgrid)
lines!(zgrid, DL .+ abs_mag, color=:red, label="OmegaM = 0.6, OmegaDE = 0.4")

universe = cosmology(OmegaK=0.0, OmegaM=0.1, Tcmb=0)
DL = map(z->distance_modulus(universe, z), zgrid)
lines!(zgrid, DL .+ abs_mag, color=:green, label="OmegaM = 0.1, OmegaDE = 0.9")

#f[2,1] = Legend(f, ax, "Cosmologies", framevisible = false)
# Create a legend for our plot
axislegend(ax, position = :rb)
save("PerlmutterLines.png")
f
```



[9]:
```
# Here's our scalar estimate of the absolute mag.
abs_mag
```

[9]:
```
-19.228824925301424
```

1.2 Part 1 - The Log-likelihood terrain

First, you have to write out the likelihood function for the observed supernova data, given cosmological model parameters.

That is, please complete the following function. It will be passed vectors of `z`, `mag`, and `mag_error` measurements, plus scalar parameters `M`, `Omega_M` and `Omega_DE`. You will need to create a "cosmology" object, find the *distance modulus* for each redshift `z`, and add that to the absolute mag `M` to get the *predicted* magnitude. You will then compare that to each measured magnitude, and compute the likelihood.

[10]:
```
function supernova_log_likelihood(zs, mag, mag_error, M, Omatter, Ode)
    # z: vector of redshifts
    # mag: vector of measured magnitudes
    # mag_error: vector of uncertainties on the measured magnitudes (sigmas).
    # M: scalar, absolute magnitude of a Type-1a supernova
    # Omatter: scalar Omega_M, amount of matter in the universe
    # Ode: scalar Omega_DE, amount of dark energy in the universe
```

```
    ###   YOUR CODE HERE!!
    cosModel = cosmology(OmegaK=1. - Omatter - Ode, OmegaM=Omatter, Tcmb=0)␣
 ↪#make the cosmology "object"

    mag_pred = (z -> distance_modulus(cosModel, z) + M).(zs) #vectorize the␣
 ↪operations on zs

    chi = (mag_pred - mag) ./ mag_error
    sum(-0.5 .* chi.^2)

    # You must return a scalar value
end;
```

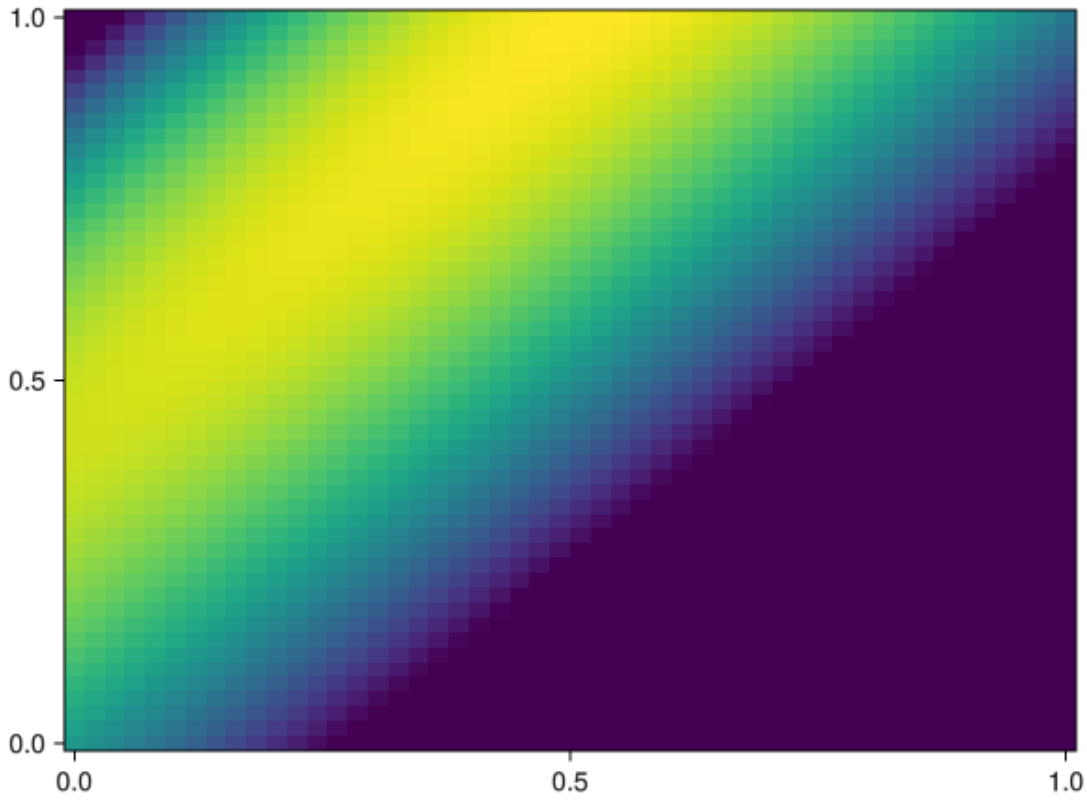
Next, please keep `M` fixed to the `abs_mag` value we computed above, and call your `supernova_log_likelihood` on a grid of `Omega_M` and `Omega_DE` values. (You will pass in `data.z`, `data.mag`, and `data.sigma_mag` for the `z`, `mag`, and `mag_error` values.)

Try a grid from 0 to 1 for both Omega_M and Omega_DE, and show the `supernova_log_likelihood` values using the `heatmap` function. You may find it helpful to limit the range using something like `heatmap(om_grid, ode_grid, sn_ll, colorrange=[maximum(sn_ll)-20, maximum(sn_ll)])`.

Another thing you can do is, instead of showing the *log*-likelihood, show the likelihood by taking the `exp` of your `sn_ll` grid, like this, `heatmap(om_grid, ode_grid, exp.(sn_ll))`.

Please compare your plot to Figure 7 in the Perlmutter et al. 1999 paper, shown below. Does your likelihood contour look consistent with the blue ellipses?

[28]:
```
n_om, n_ode = 50,50
om_grid = LinRange(0, 1, n_om)
ode_grid = LinRange(0, 1, n_ode)
sn_ll = zeros(n_om, n_ode)
for i in 1:n_om
    for j in 1:n_ode
        sn_ll[i, j] = supernova_log_likelihood(data.z, data.mag, data.
 ↪sigma_mag, abs_mag, om_grid[i], ode_grid[j])
    end
end
f = Figure()
ax = Axis(f[1, 1])

heatmap!(om_grid, ode_grid, sn_ll, colorrange=[maximum(sn_ll)-20,␣
 ↪maximum(sn_ll)])

save("FirstHeatmapLimitedRange.png", f)
f
```

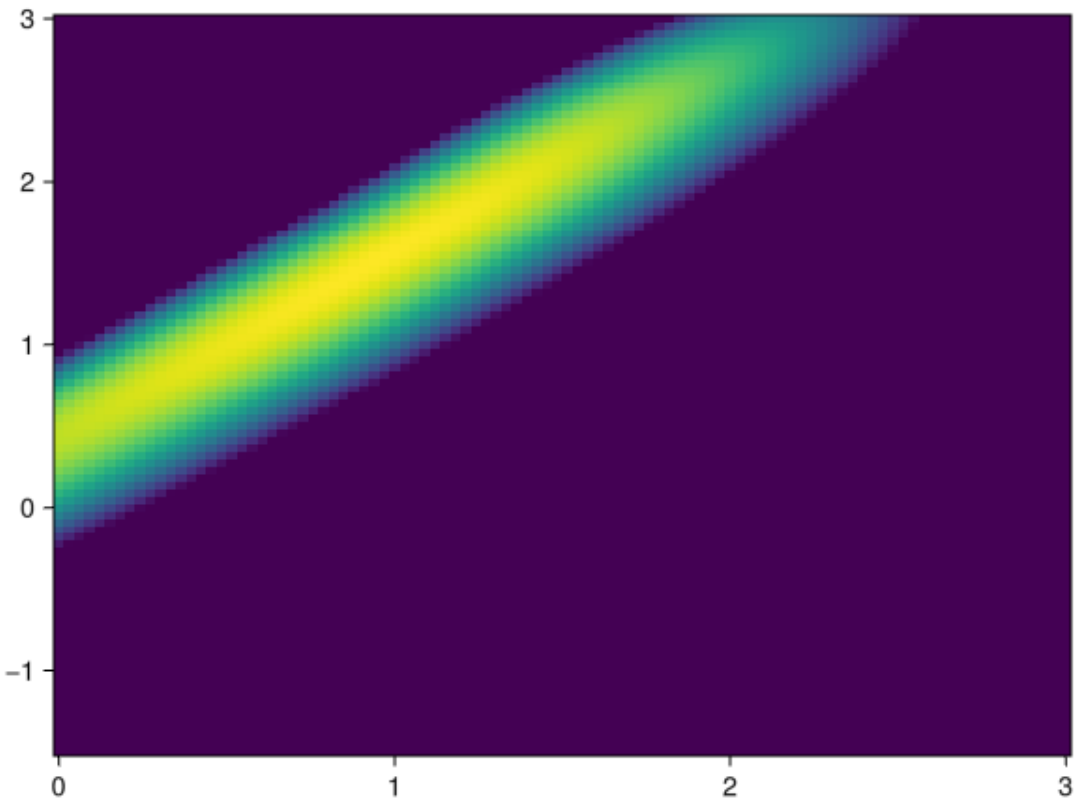



```
Bonito.App(Bonito.var"#8#14"{WGLMakie.var"#20#21"{WGLMakie.Screen,␣
 ↪Scene}}(WGLMakie.var"#20#21"{WGLMakie.Screen, Scene}(WGLMakie.Screen(
    framerate = 30.0,
    resize_to = nothing,
    px_per_unit = automatic,
    scalefactor = automatic,
    session = nothing,
    three = Channel{Bool}(1) (empty),
    scene = Scene (600px, 450px):
  0 Plots
  1 Child Scene:
      Scene (600px, 450px),
), Scene (600px, 450px):
  0 Plots
  1 Child Scene:
      Scene (600px, 450px))), Base.RefValue{Union{Nothing, Bonito.
 ↪Session}}(nothing), "Bonito App", false)
```

The resulting heat map looks pretty similar to the contour plot in Perlmutter, same shape and direction in the restricted region.

Next, try expanding the grid ranges for Omega_M and Omega_DE up to, say, 0 to 2 or 0 to 3. You should encounter a problem – the cosmology package will fail to compute the `distance_modulus` for some combinations! You can work around this by using Julia's `try...catch` syntax, like this:

[12]:
```
n_om2, n_ode2 = 100,100
om_grid2 = LinRange(0, 3, n_om2)
ode_grid2 = LinRange(-1.5, 3, n_ode2)
sn_ll2 = zeros(n_om2, n_ode2)
for i in 1:n_om2
    for j in 1:n_ode2
        try #very bad way of doing this, should rather make␣
 ↪supernova_log_likelihood to check for parameters.
            sn_ll2[i, j] = supernova_log_likelihood(data.z, data.mag, data.
 ↪sigma_mag, abs_mag, om_grid2[i], ode_grid2[j])
        catch err
            sn_ll2[i, j] = -Inf
        end
    end
end

f = Figure()
ax = Axis(f[1, 1])

heatmap!(om_grid2, ode_grid2, sn_ll2, colorrange=[maximum(sn_ll2)-20,␣
 ↪maximum(sn_ll2)])

save("HeatmapExtendedRange.png", f)
f
```




```
Bonito.App(Bonito.var"#8#14"{WGLMakie.var"#20#21"{WGLMakie.Screen,␣
 ↪Scene}}(WGLMakie.var"#20#21"{WGLMakie.Screen, Scene}(WGLMakie.Screen(
    framerate = 30.0,
    resize_to = nothing,
    px_per_unit = automatic,
    scalefactor = automatic,
    session = nothing,
    three = Channel{Bool}(1) (empty),
    scene = Scene (600px, 450px):
  0 Plots
  1 Child Scene:
      Scene (600px, 450px),
), Scene (600px, 450px):
  0 Plots
  1 Child Scene:
      Scene (600px, 450px))), Base.RefValue{Union{Nothing, Bonito.
 ↪Session}}(nothing), "Bonito App", false)
```

This will "try" to run the `supernova_log_likelihood` function, and if it fails, it will go into the "catch" branch.

1.3 Part 2 - Using MCMC to sample from the likelihood

Next, we will use Markov Chain Monte Carlo to draw samples from the likelihood distribution.

You can start with the `mcmc` function from the lecture.

You will need to tune the MCMC proposal's step sizes (also known as "jump sizes"). To do this, you can use the variant of the `mcmc` routine that cycles through the parameters and only jumps one at a time, named `mcmc_cyclic` in the updated lecture notebook. After tuning the step sizes with `mcmc_cyclic`, you can go back to the plain `mcmc` routine if you want, or stick with `mcmc_cyclic`; it is up to you.

Please plot the samples from your MCMC chains, to demonstrate that the chain looks like it has converged. Ideally, you would like to see reasonable acceptance rates, and you would like to see the samples "exploring" the parameter space. Decide how many step you need to run the MCMC routine for, and write a sentence or two describing why you think that's a good number.

For this part, please include the `M` (absolute magnitude) as a parameter that you are fitting – so you are fitting for `M` in addition to `Omega_M` and `Omega_DE`. This is a quite standard situation where you have a "nuisance" parameter `M` that you don't really care about, in addition to the `Omega` parameters that you do care about.

It is quite common to plot the results from an MCMC sampling using a "corner plot", which shows the distribution of each of the individual parameters, and the joint distributions of pairs of parameters. This will help you determine whether some of the parameters are correlated with each other.

Below is a function you can use to generate corner plots from your chain – call it like `cornerplot(chain, ["M", "Omega_M", "Omega_DE"])`. There is also a CornerPlot package (https://juliapackages.com/p/cornerplot) but I have not had luck getting it to work for me.

Once you have made you corner plots, please write a few sentences interpreting what you see. Is the nuisance parameter `M` correlated with the Omegas? Are the Omegas correlated with each other?

[13]:
```
function cornerplot(x, names; figsize=(600,600))
    # how many columns of data
    dim = size(x, 2)
    # rows to plot
    idxs = 1:size(x,1)
    f = Figure(size=figsize)
    for i in 1:dim, j in 1:dim
        if i < j
            continue
        end
        ax = Axis(f[i, j], aspect = 1,
                  topspinevisible = false,
                  rightspinevisible = false,)
        if i == j
            hist!(x[idxs,i], direction=:y)
            ax.xlabel = names[i]
        else
            #scatter!(x[idxs,j], x[idxs,i], markersize=4)
            hexbin!(x[idxs,j], x[idxs,i])
            ax.xlabel = names[j]
```

```
            ax.ylabel = names[i]
        end
    end
    f
end;
```

Here is the cyclic Markov Chain Monte Carlo function that I got from the lecture notes.

[14]:
```
function mcmc_cyclic(logprob_func, propose_func, initial_p, n_steps)
    p = initial_p
    logprob = logprob_func(p)
    chain = zeros(n_steps, length(p))
    n_accept = zeros(length(p))

    for i in 1:n_steps

        # We're going to update one index at a time... 1, 2, 1, 2, ....
        update_index = 1 + ((i-1) % length(p))

        # Call the proposal function to generate new values for all parameters..
 ↪.
        p_prop = propose_func(p)
        # ... but then only keep one of the new parameter values!
        p_new = copy(p)
        p_new[update_index] = p_prop[update_index]

        logprob_new = logprob_func(p_new)

        ratio = exp(logprob_new - logprob)
        if ratio > 1
            # Jump to the new place
            p = p_new
            logprob = logprob_new
            n_accept[update_index] += 1
        else
            # Jump to the new place with probability "ratio"
            u = rand()
            if u < ratio
                # Jump to the new place
                p = p_new
                logprob = logprob_new
                n_accept[update_index] += 1
            else
                # Stay where we are
            end
        end
        chain[i, 1:end] = p
```

```
    end
    # The number of times we step each parameter is roughly n_steps /␣
 ↪n_parameters
    chain, n_accept ./ (n_steps ./ length(p))
end;
```

Here I run the cyclic Monte Carlo Markov chain so that I can tune the parameters. I went for more samples and smaller jumps as this function does not take too long to execute. I took a look at acceptance rates too to make sure that most jumps were good jumps, I chose the smaller jumps size to increase the acceptance rate of sampling with all paramters having similar acceptance rates.

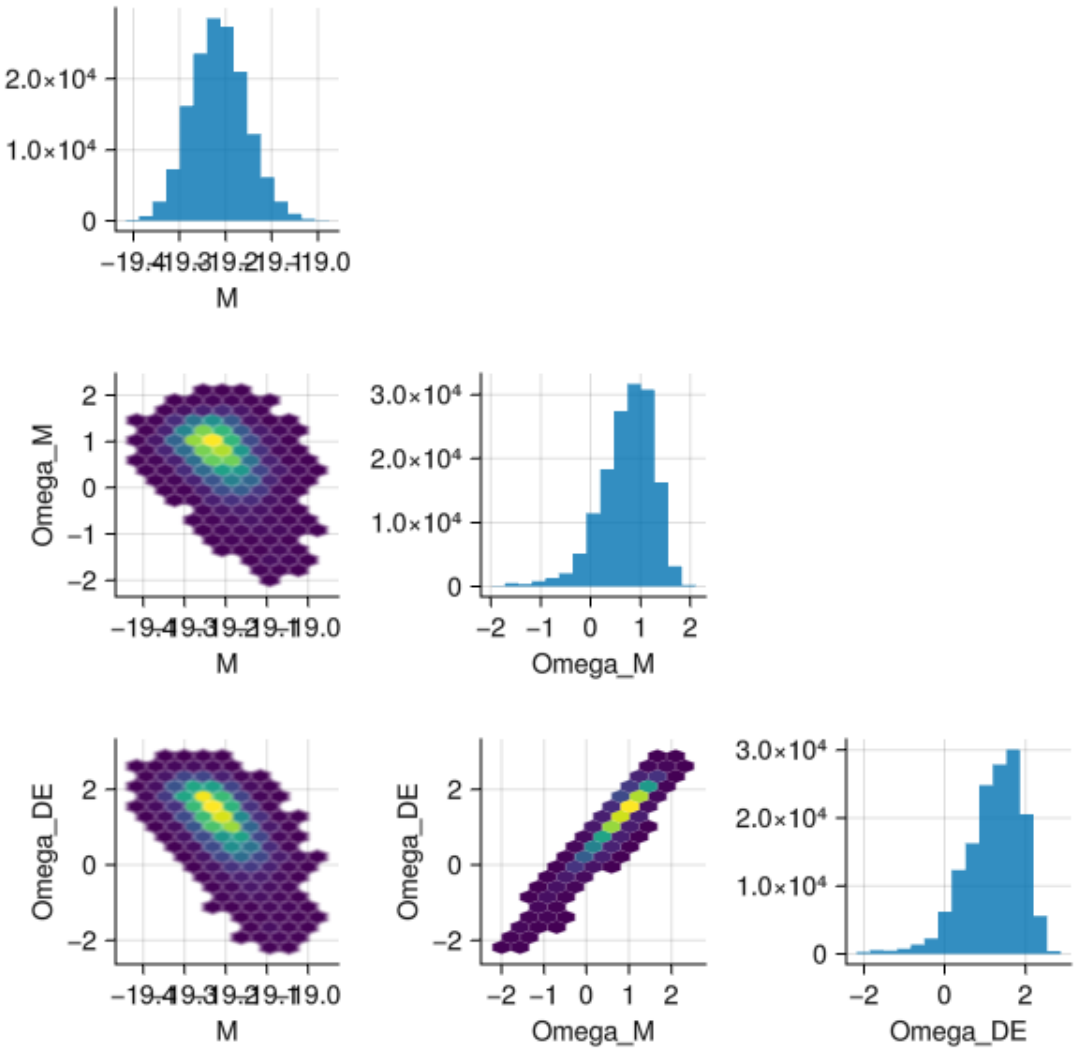
[15]:
```
function propose(p, jump_sizes)
    p .+ randn(length(p)) .* jump_sizes
end;

initial_guess = [abs_mag, .5, .5]
jump_sizes = [0.02, 0.09, 0.1]

chain, accept_rate = mcmc_cyclic(p ->
        supernova_log_likelihood(data.z, data.mag, data.sigma_mag, p[1], p[2],␣
 ↪p[3]),
p -> propose(p, jump_sizes),
initial_guess, 150000)
println("Acceptance Rates")
accept_rate
```

```
Acceptance Rates
```

[15]:
```
3-element Vector{Float64}:
 0.78388
 0.69756
 0.71164
```

I only plot the samples as the corner plot as it makes the sampling most clear.

[29]:
```
cplot1 = cornerplot(chain, ["M", "Omega_M", "Omega_DE"])
save("cornerplotFirstMCMC.png", cplot1)
```

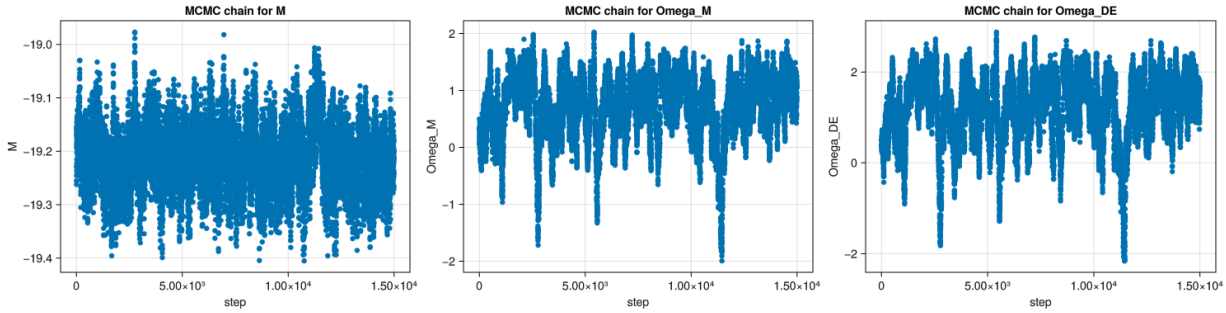



```
Bonito.App(Bonito.var"#8#14"{WGLMakie.var"#20#21"{WGLMakie.Screen,␣
 ↪Scene}}(WGLMakie.var"#20#21"{WGLMakie.Screen, Scene}(WGLMakie.Screen(
    framerate = 30.0,
    resize_to = nothing,
    px_per_unit = automatic,
    scalefactor = automatic,
    session = nothing,
    three = Channel{Bool}(1) (empty),
    scene = Scene (600px, 600px):
  0 Plots
  6 Child Scenes:
     Scene (600px, 600px)
     Scene (600px, 600px)
     Scene (600px, 600px)
     Scene (600px, 600px)
     Scene (600px, 600px)
```

```
    Scene (600px, 600px),
), Scene (600px, 600px):
  0 Plots
  6 Child Scenes:
    Scene (600px, 600px)
    Scene (600px, 600px)
    Scene (600px, 600px)
    Scene (600px, 600px)
    Scene (600px, 600px)
    Scene (600px, 600px))), Base.RefValue{Union{Nothing, Bonito.
 ↪Session}}(nothing), "Bonito App", false)
```

From the above corner plot, Ω_M and M seem slightly negatively correlated, Ω_{DE} and M also seem slightly negatively correlated. Finally, Ω_M and Ω_{DE} are strongly positively correlated.

[17]:
```
f = Figure(size = (1500, 400))
ax = Axis(f[1,1], title="MCMC chain for M", xlabel="step", ylabel="M")

scatter!(chain[1:10:end, 1])
ax = Axis(f[1,2], title="MCMC chain for Omega_M", xlabel="step",␣
 ↪ylabel="Omega_M")
scatter!(chain[1:10:end, 2])
ax = Axis(f[1,3], title="MCMC chain for Omega_DE", xlabel="step",␣
 ↪ylabel="Omega_DE")
scatter!(chain[1:10:end, 3])

save("MCMCChain.png", f)
display(f)
```




```
Bonito.App(Bonito.var"#8#14"{WGLMakie.var"#20#21"{WGLMakie.Screen,␣
 ↪Scene}}(WGLMakie.var"#20#21"{WGLMakie.Screen, Scene}(WGLMakie.Screen(
    framerate = 30.0,
    resize_to = nothing,
    px_per_unit = automatic,
    scalefactor = automatic,
    session = nothing,
    three = Channel{Bool}(1) (empty),
    scene = Scene (1500px, 400px):
```

```
  0 Plots
  3 Child Scenes:
     Scene (1500px, 400px)
     Scene (1500px, 400px)
     Scene (1500px, 400px),
), Scene (1500px, 400px):
  0 Plots
  3 Child Scenes:
     Scene (1500px, 400px)
     Scene (1500px, 400px)
     Scene (1500px, 400px))), Base.RefValue{Union{Nothing, Bonito.
↪Session}}(nothing), "Bonito App", false)
```

[17]:
```
WGLMakie.Screen(
    framerate = 30.0,
    resize_to = nothing,
    px_per_unit = automatic,
    scalefactor = automatic,
    session = true,
    three = Channel{Bool}(1) (1 item available),
    scene = Scene (1500px, 400px):
  0 Plots
  3 Child Scenes:
     Scene (1500px, 400px)
     Scene (1500px, 400px)
     Scene (1500px, 400px),
)
```

Finally, please try to make a contour plot similar to Perlmutter et al.'s Figure 7. From your MCMC chain, you can pull out the `Omega_M` and `Omega_DE` arrays, and then create a 2-d histogram. Once you have a 2-d histogram, you can use the `contour` function to find and plot the contours in that histogram.

[18]:
```
OmegaMMCMC = chain[begin:end, 2]
OmegaDEMCMC = chain[begin:end, 3];
```

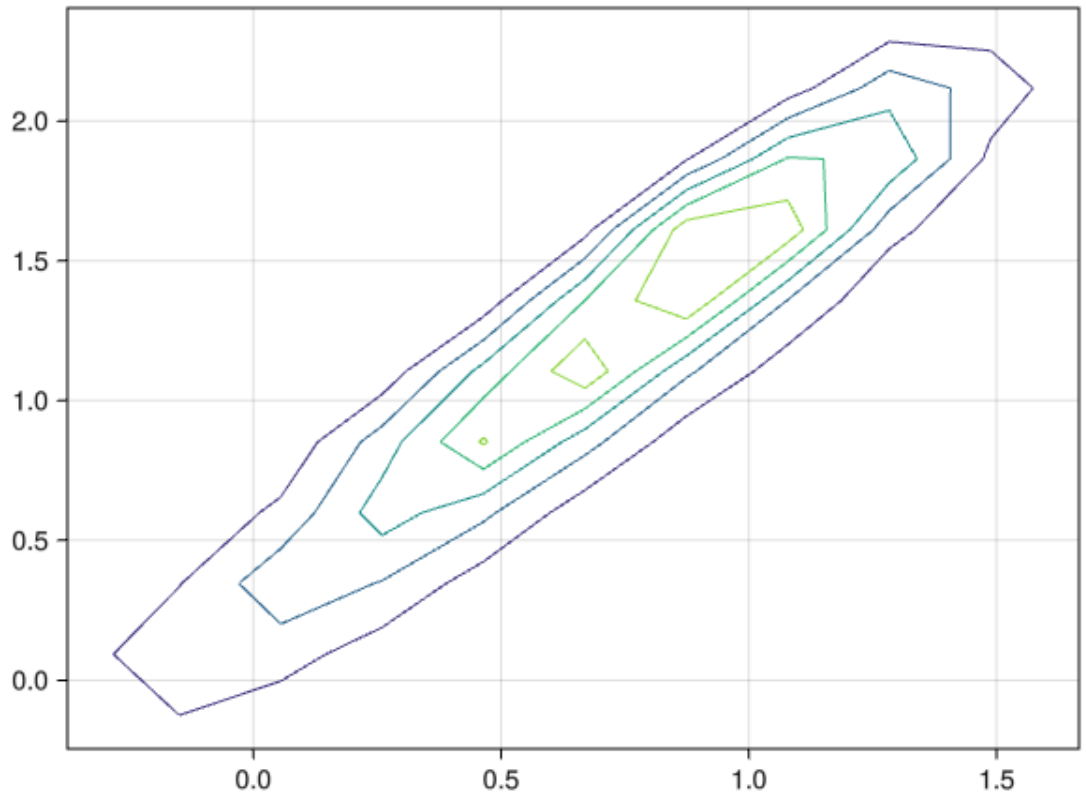
Here I bin the data in a 2D histogram. I did it in a for loop placing the histogram values in appropriate bins.

[19]:
```
n_ombins, n_odebins = 20,20

om_gridMCMC = LinRange(minimum(OmegaMMCMC), maximum(OmegaMMCMC), n_ombins + 1)
ode_gridMCMC = LinRange(minimum(OmegaDEMCMC), maximum(OmegaDEMCMC), n_odebins +␣
↪1)

minOmegaMMCMC = minimum(OmegaMMCMC)
minOmegaDEMCMC = minimum(OmegaDEMCMC)
maxOmegaMMCMC = maximum(OmegaMMCMC)
```

```
maxOmegaDEMCMC = maximum(OmegaDEMCMC)
rangeOmegaMMCMC = maxOmegaMMCMC - minOmegaMMCMC
rangeOmegaDEMCMC = maxOmegaDEMCMC - minOmegaDEMCMC

histogramValues = zeros(n_ombins + 1, n_odebins + 1)
for i in 1:size(OmegaDEMCMC)[begin]
    histogramValues[trunc(Int, (OmegaMMCMC[i] - minOmegaMMCMC) * n_ombins /␣
 ↪(rangeOmegaMMCMC)) + 1, trunc(Int, (OmegaDEMCMC[i] - minOmegaDEMCMC) *␣
 ↪n_odebins / rangeOmegaDEMCMC) + 1] += 1
## I was doing to more manually, which is probably correct, but very slow.
#    for j in 1:(n_ombins - 1)
#        for k in 1:(n_odebins - 1)
#            histogramValues[j, k] += (om_gridMCMC[j] <= OmegaMMCMC[i] &&␣
 ↪OmegaMMCMC[i] <= om_gridMCMC[j + 1] && ode_gridMCMC[k] <= OmegaDEMCMC[i] &&␣
 ↪OmegaDEMCMC[i] <= ode_gridMCMC[k + 1])
#        end
#    end
end
histogramValues;
```

Next I just call the contour plot function on the histogram matrix.

[20]:
```
f = Figure()
Axis(f[1, 1])

contour!(om_gridMCMC, ode_gridMCMC, histogramValues)

save("MyHistogram.png", f)

display(f)
```




```
Bonito.App(Bonito.var"#8#14"{WGLMakie.var"#20#21"{WGLMakie.Screen,␣
 ↪Scene}}(WGLMakie.var"#20#21"{WGLMakie.Screen, Scene}(WGLMakie.Screen(
    framerate = 30.0,
    resize_to = nothing,
    px_per_unit = automatic,
    scalefactor = automatic,
    session = nothing,
    three = Channel{Bool}(1) (empty),
    scene = Scene (600px, 450px):
  0 Plots
  1 Child Scene:
      Scene (600px, 450px),
), Scene (600px, 450px):
  0 Plots
  1 Child Scene:
      Scene (600px, 450px))), Base.RefValue{Union{Nothing, Bonito.
 ↪Session}}(nothing), "Bonito App", false)
```

[20]:
```
WGLMakie.Screen(
    framerate = 30.0,
    resize_to = nothing,
    px_per_unit = automatic,
    scalefactor = automatic,
```

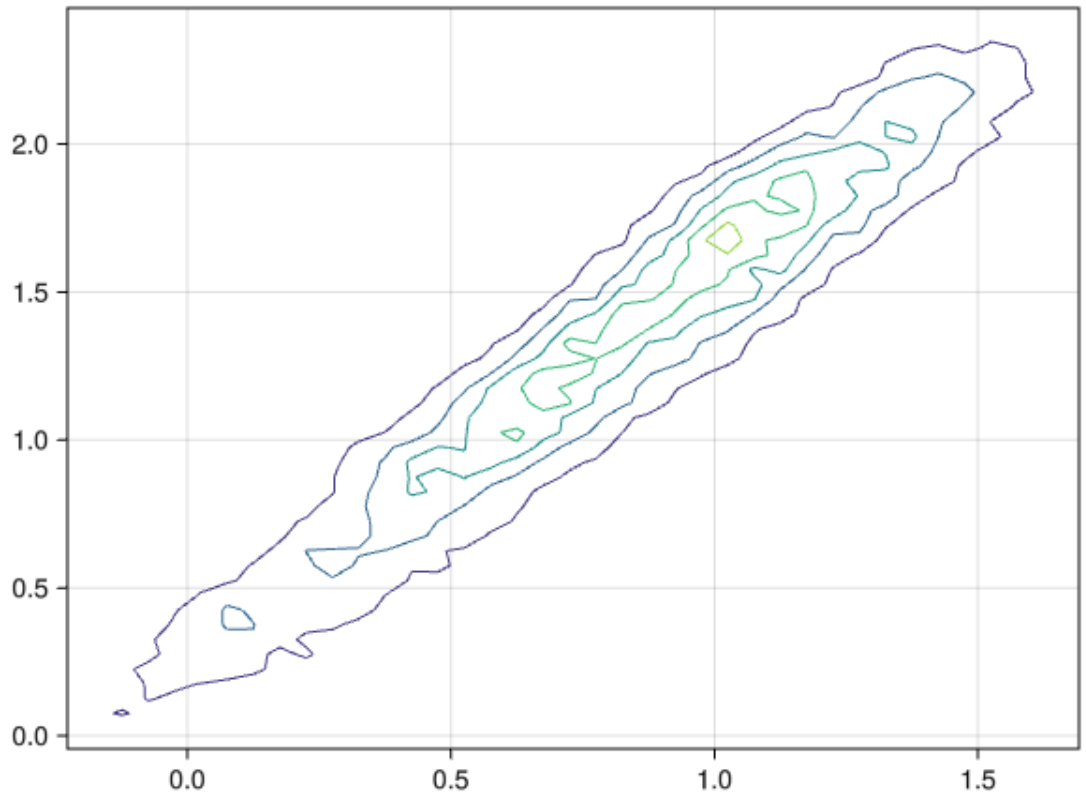
```
    session = true,
    three = Channel{Bool}(1) (1 item available),
    scene = Scene (600px, 450px):
  0 Plots
  1 Child Scene:
    Scene (600px, 450px),
)
```

[21]:
```
] add FHist AffineInvariantMCMC
```

```
  Resolving package versions…
 No Changes to `~/.julia/environments/v1.10/Project.toml`
 No Changes to `~/.julia/environments/v1.10/Manifest.toml`
```

Here I tested the FHist package that was mentioned in the lectures.

[22]:
```
using FHist

h = FHist.Hist2D((chain[:,2], chain[:,3]); nbins=(100,100))

counts = bincounts(h);
xc,yc = bincenters(h);

f = Figure()
Axis(f[1, 1])

contour!(xc, yc, counts) #, levels=[10,50,100])
#really small/many bins make this contour plot really jaggered

save("MyFirstJaggeredContour.png", f)
f
```

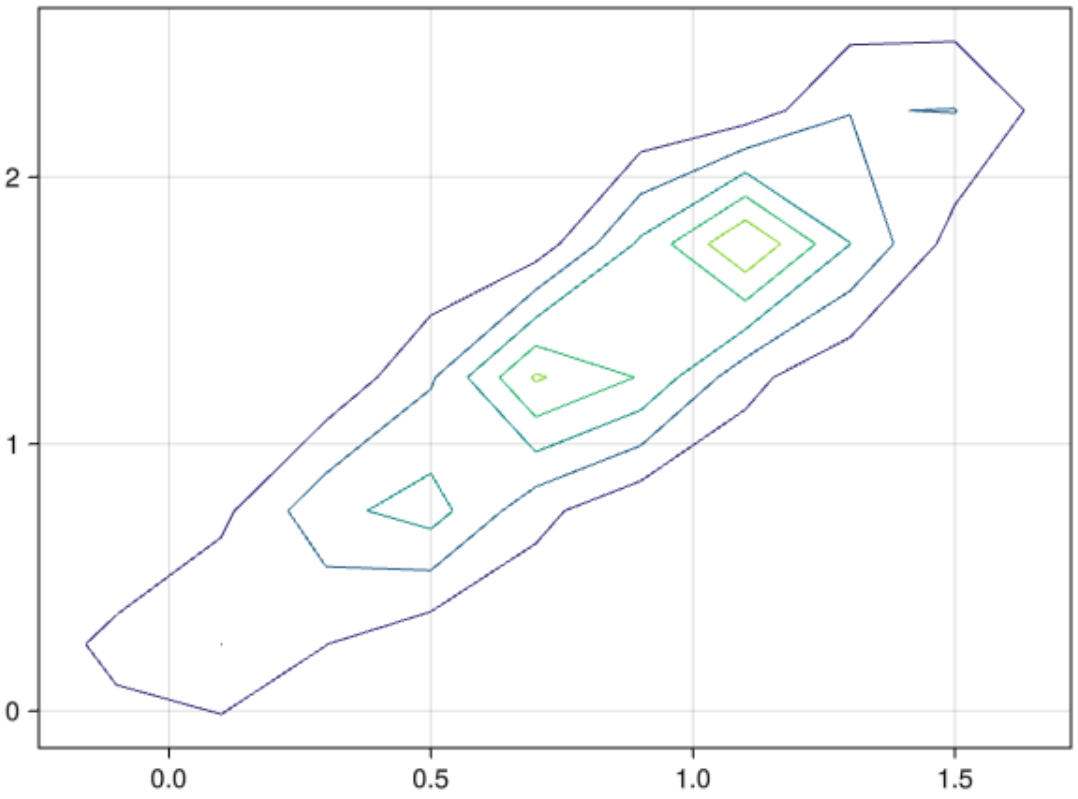



```
Bonito.App(Bonito.var"#8#14"{WGLMakie.var"#20#21"{WGLMakie.Screen,␣
 ↪Scene}}(WGLMakie.var"#20#21"{WGLMakie.Screen, Scene}(WGLMakie.Screen(
    framerate = 30.0,
    resize_to = nothing,
    px_per_unit = automatic,
    scalefactor = automatic,
    session = nothing,
    three = Channel{Bool}(1) (empty),
    scene = Scene (600px, 450px):
  0 Plots
  1 Child Scene:
      Scene (600px, 450px),
), Scene (600px, 450px):
  0 Plots
  1 Child Scene:
      Scene (600px, 450px))), Base.RefValue{Union{Nothing, Bonito.
 ↪Session}}(nothing), "Bonito App", false)
```

Doing this with fewer bigger bins makes the contours smoother, similar to my manual way. FHist seems to be binning data slightly differently to me…

[23]:
```
h = FHist.Hist2D((chain[:,2], chain[:,3]); nbins=(20,20))

counts = bincounts(h);
```

```
xc,yc = bincenters(h);

f = Figure()
Axis(f[1, 1])

contour!(xc, yc, counts)#, levels=[10,50,100])

save("SmallerBinsCountour.png", f)
f
```




```
Bonito.App(Bonito.var"#8#14"{WGLMakie.var"#20#21"{WGLMakie.Screen,␣
  ↪Scene}}(WGLMakie.var"#20#21"{WGLMakie.Screen, Scene}(WGLMakie.Screen(
    framerate = 30.0,
    resize_to = nothing,
    px_per_unit = automatic,
    scalefactor = automatic,
    session = nothing,
    three = Channel{Bool}(1) (empty),
    scene = Scene (600px, 450px):
  0 Plots
  1 Child Scene:
      Scene (600px, 450px),
), Scene (600px, 450px):
```

```
  0 Plots
  1 Child Scene:
     Scene (600px, 450px))), Base.RefValue{Union{Nothing, Bonito.
↪Session}}(nothing), "Bonito App", false)
```

Doing samples with the Affine MCMC method mentioned in the lectures. "Self-tunes parameters".

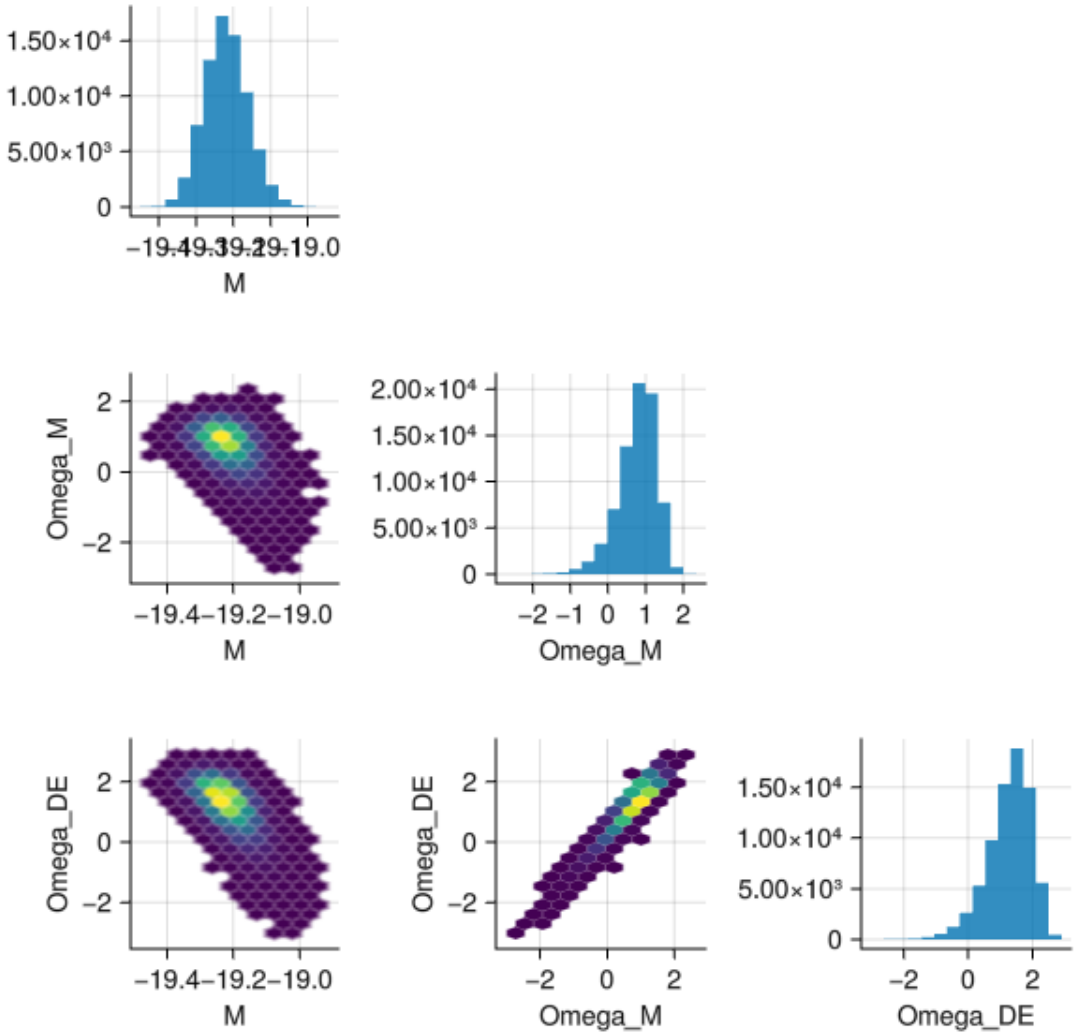
[24]:
```
using AffineInvariantMCMC

numdims = 3
numwalkers = 50
thinning = 10
numsamples_perwalker = 15000
burnin = 5000

# Start out by doing a "burn-in" short run...
initial = [abs_mag, 0.5, 0.5] .+ randn(numdims, numwalkers)*0.01
chain, ll = AffineInvariantMCMC.sample( p ->
        try
            supernova_log_likelihood(data.z, data.mag, data.sigma_mag, p[1],␣
 ↪p[2], p[3])
        catch err
            -Inf
        end, numwalkers, initial, burnin, 1)

# Then start the "main" run where the burn-in finished:
initial = chain[:,:,end]
chain, ll = AffineInvariantMCMC.sample(
   p ->
        try
            supernova_log_likelihood(data.z, data.mag, data.sigma_mag, p[1],␣
 ↪p[2], p[3])
        catch err
            -Inf
        end
    , numwalkers, initial, numsamples_perwalker, thinning)
flatchain, flatllhoodvals = AffineInvariantMCMC. flattenmcmcarray(chain, ll);

# This "flatchain" is transposed from the way we produced it in our code, so␣
 ↪you can do

chain = flatchain';
```

```
Progress: 100%|                          | Time:
0:00:12
```

The affine MCMC doesn't seem to give perfect sampling, some samples seem to be out of the main distribution, even starting on the main distribution, but does an amazing job considering the lack

of needing to tune jump size parameters.

[25]:
```
cplot2 = cornerplot(chain, ["M", "Omega_M", "Omega_DE"])

save("AffineMCMCCPlot.png", cplot2)
cplot2
```

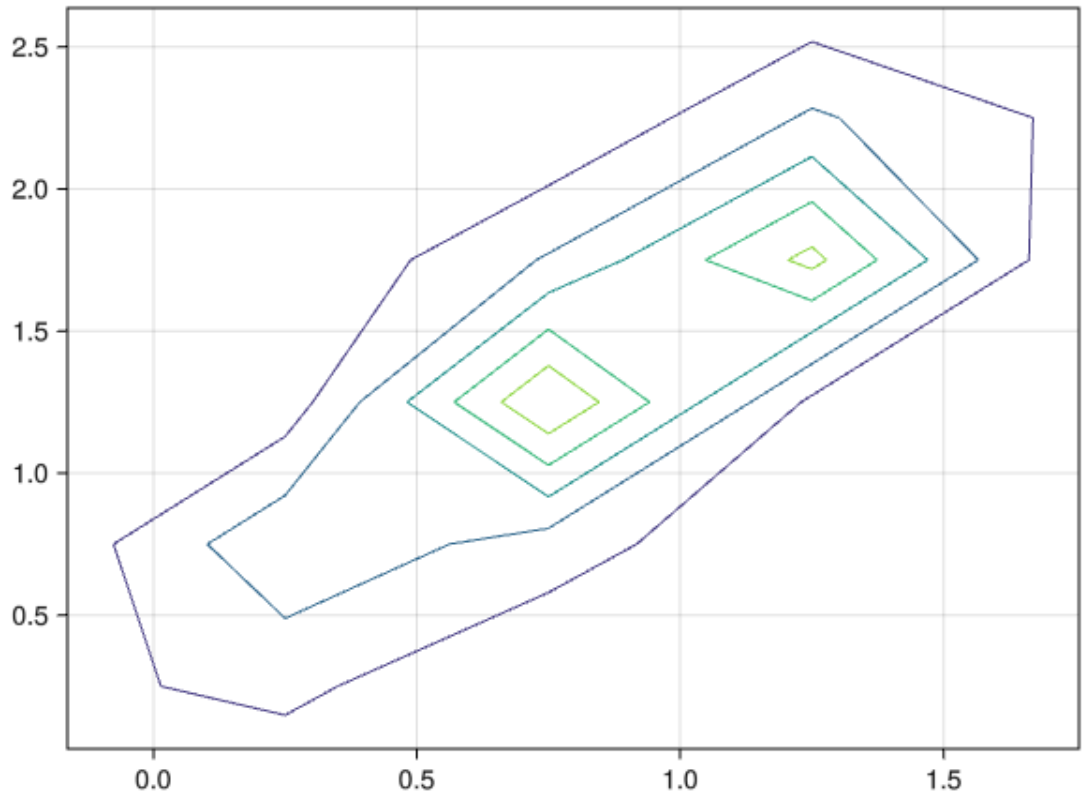



```
Bonito.App(Bonito.var"#8#14"{WGLMakie.var"#20#21"{WGLMakie.Screen,␣
 ↪Scene}}(WGLMakie.var"#20#21"{WGLMakie.Screen, Scene}(WGLMakie.Screen(
    framerate = 30.0,
    resize_to = nothing,
    px_per_unit = automatic,
    scalefactor = automatic,
    session = nothing,
    three = Channel{Bool}(1) (empty),
    scene = Scene (600px, 600px):
```

```
  0 Plots
  6 Child Scenes:
     Scene (600px, 600px)
     Scene (600px, 600px)
     Scene (600px, 600px)
     Scene (600px, 600px)
     Scene (600px, 600px)
     Scene (600px, 600px),
), Scene (600px, 600px):
  0 Plots
  6 Child Scenes:
     Scene (600px, 600px)
     Scene (600px, 600px)
     Scene (600px, 600px)
     Scene (600px, 600px)
     Scene (600px, 600px)
     Scene (600px, 600px))), Base.RefValue{Union{Nothing, Bonito.
↪Session}}(nothing), "Bonito App", false)
```

Now I plot these contours, pretty smooth :)

[26]:
```
h = FHist.Hist2D((chain[:,2], chain[:,3]); nbins=(20,20))

counts = bincounts(h);
xc,yc = bincenters(h);

f = Figure()
Axis(f[1, 1])

contour!(xc, yc, counts)#, levels=[10,50,100])

save("AffineMCMCContour.png", f)
f
```

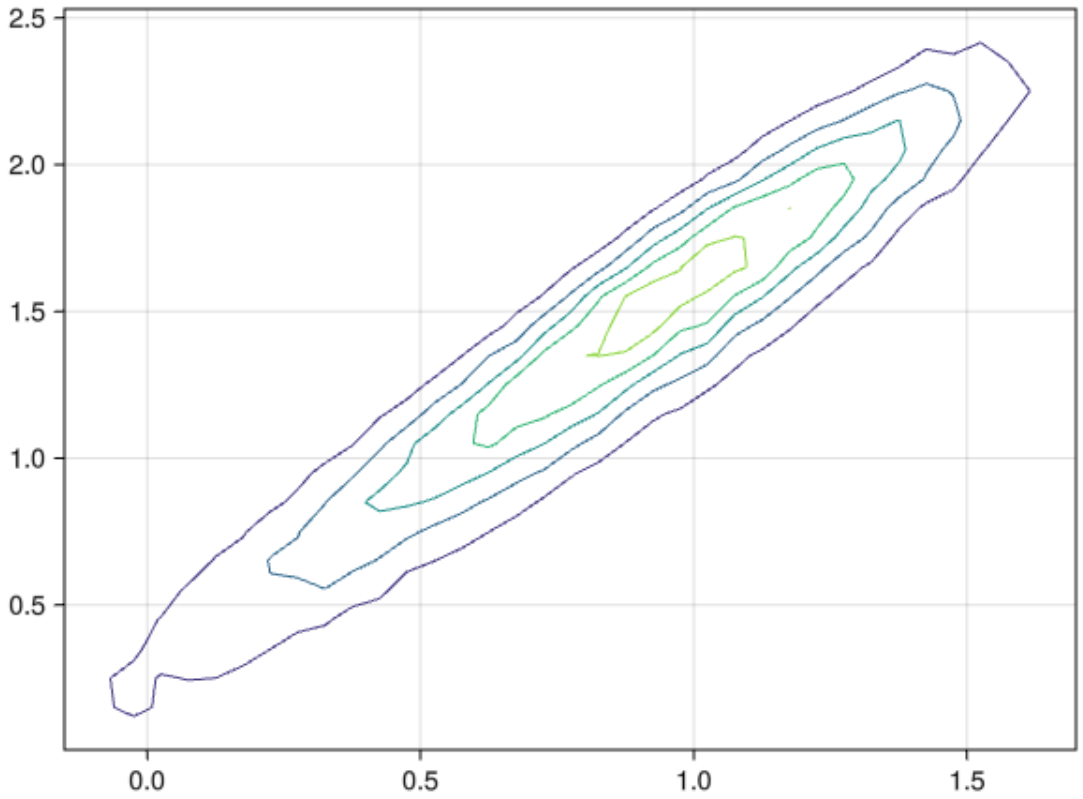



```
Bonito.App(Bonito.var"#8#14"{WGLMakie.var"#20#21"{WGLMakie.Screen,␣
 ↪Scene}}(WGLMakie.var"#20#21"{WGLMakie.Screen, Scene}(WGLMakie.Screen(
    framerate = 30.0,
    resize_to = nothing,
    px_per_unit = automatic,
    scalefactor = automatic,
    session = nothing,
    three = Channel{Bool}(1) (empty),
    scene = Scene (600px, 450px):
  0 Plots
  1 Child Scene:
      Scene (600px, 450px),
), Scene (600px, 450px):
  0 Plots
  1 Child Scene:
      Scene (600px, 450px))), Base.RefValue{Union{Nothing, Bonito.
 ↪Session}}(nothing), "Bonito App", false)
```

This contour looks much smoother now

[27]:
```
h = FHist.Hist2D((chain[:,2], chain[:,3]); nbins=(100,100))

counts = bincounts(h);
xc,yc = bincenters(h);
```

```
f = Figure()
Axis(f[1, 1])

contour!(xc, yc, counts) #, levels=[10,50,100])

save("finalsmoothcountour.png", f)
f
```




```
Bonito.App(Bonito.var"#8#14"{WGLMakie.var"#20#21"{WGLMakie.Screen,␣
 ↪Scene}}(WGLMakie.var"#20#21"{WGLMakie.Screen, Scene}(WGLMakie.Screen(
    framerate = 30.0,
    resize_to = nothing,
    px_per_unit = automatic,
    scalefactor = automatic,
    session = nothing,
    three = Channel{Bool}(1) (empty),
    scene = Scene (600px, 450px):
  0 Plots
  1 Child Scene:
      Scene (600px, 450px),
), Scene (600px, 450px):
  0 Plots
```

```
 1 Child Scene:
    Scene (600px, 450px))), Base.RefValue{Union{Nothing, Bonito.
↪Session}}(nothing), "Bonito App", false)
```