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

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 (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 (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 (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

In [1]:

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

```
    Updating registry at `C:\Users\numbe\.julia\registries\General.toml`
   Resolving package versions...
  No Changes to `C:\Users\numbe\.julia\environments\v1.10\Project.toml`
  No Changes to `C:\Users\numbe\.julia\environments\v1.10\Manifest.toml`
```

In [53]:

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

In [3]:

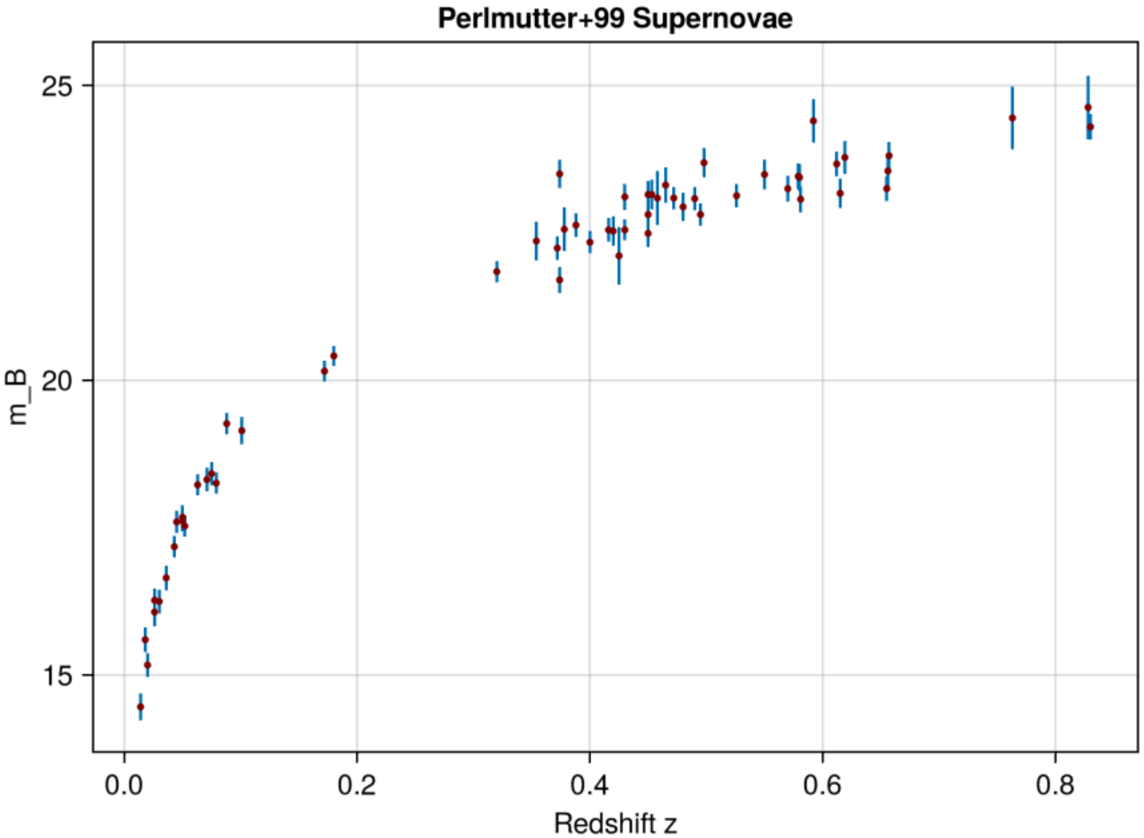
```
# There is a data file in this directory, taken basically straight out of the Per
data = CSV.read("p99-data.txt", DataFrame, delim=" ", ignorerepeated=true);
```

In [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
data.mag = data.m_b_eff
data.sigma_mag = data.sigma_m_b_eff;
```

In [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)
f
```



In [6]:

```
# Here is how we will use the "cosmology" package.  This will create a cosmology
# It does not take an Omega_Lambda parameter; instead, it takes Omega_Matter, and
# Omega_K = 1. - Omatter - Olambda.  We will also pass in "Tcmb=0", which tells i

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
```

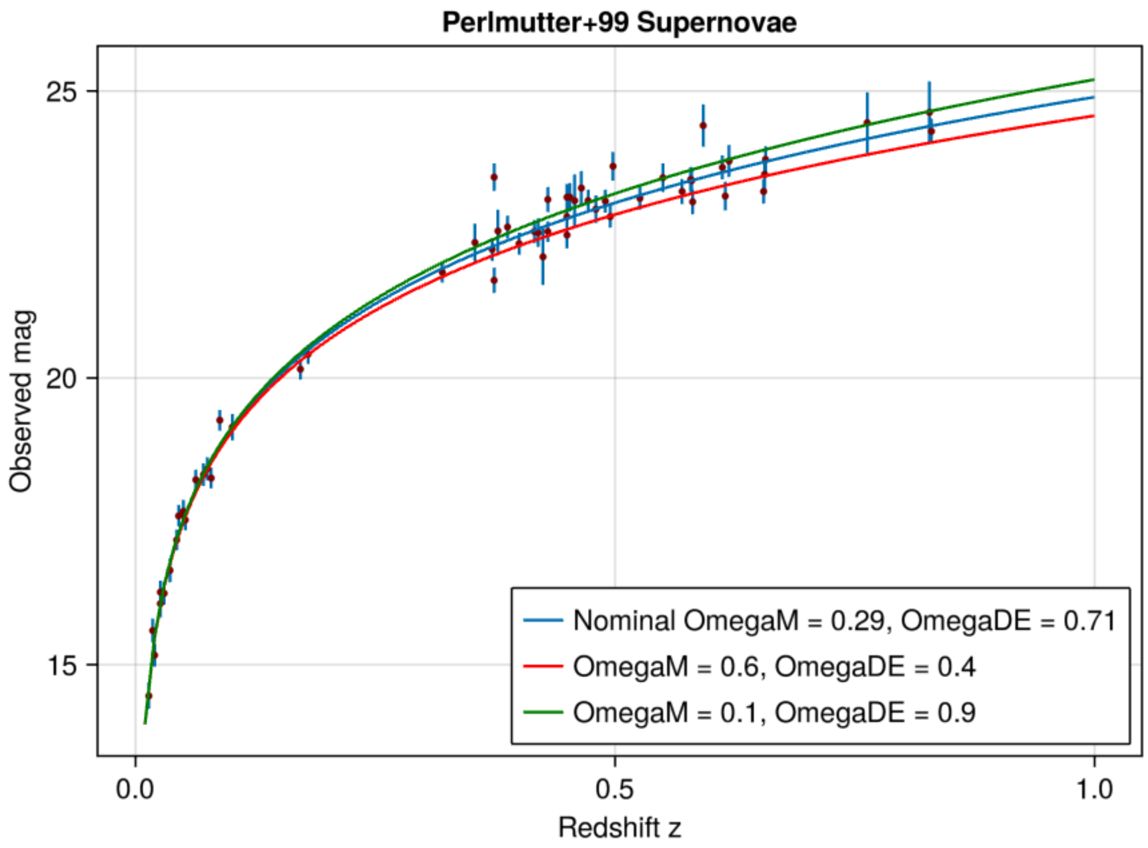
In [7]:

```
# We can then pass that "universe" object to other functions to compute things ab
# need is this `distance_modulus`, which tell you, in _magnitudes_, how much fain
# 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.

In [8]:
```
# We'll cheat a bit and use a "nominal" cosmology with currently-accepted values
nominal = cosmology(Tcmb=0)

f = Figure()
ax = Axis(f[1,1], title="Perlmutter+99 Supernovae", xlabel="Redshift z", ylabel="
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 esti
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)
f
```



In [9]:

```
# Here's our scalar estimate of the absolute mag.
abs_mag
```

```
-19.228824925301424
```

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.

In [10]:

```
function supernova_log_likelihood(z, 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

    #Universe objects defined with given parameters
    universe = cosmology(OmegaK = 1 - Ode - Omatter, OmegaM = Omatter, Tcmb=0)

    #Array of Distance moduli and redshifts
    DLx = map(x->distance_modulus(universe, x), z)

    #Adding absolute magnitude to distance modulus
    fₓ = DLx .+ M

    #Comparing with measured magnitude
    X = (fₓ .- mag) ./ mag_error

    #Computing log-likelihood
    LogLike = -0.5 .* X .^2


    # You must return a scalar value
    return sum(LogLike)
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?

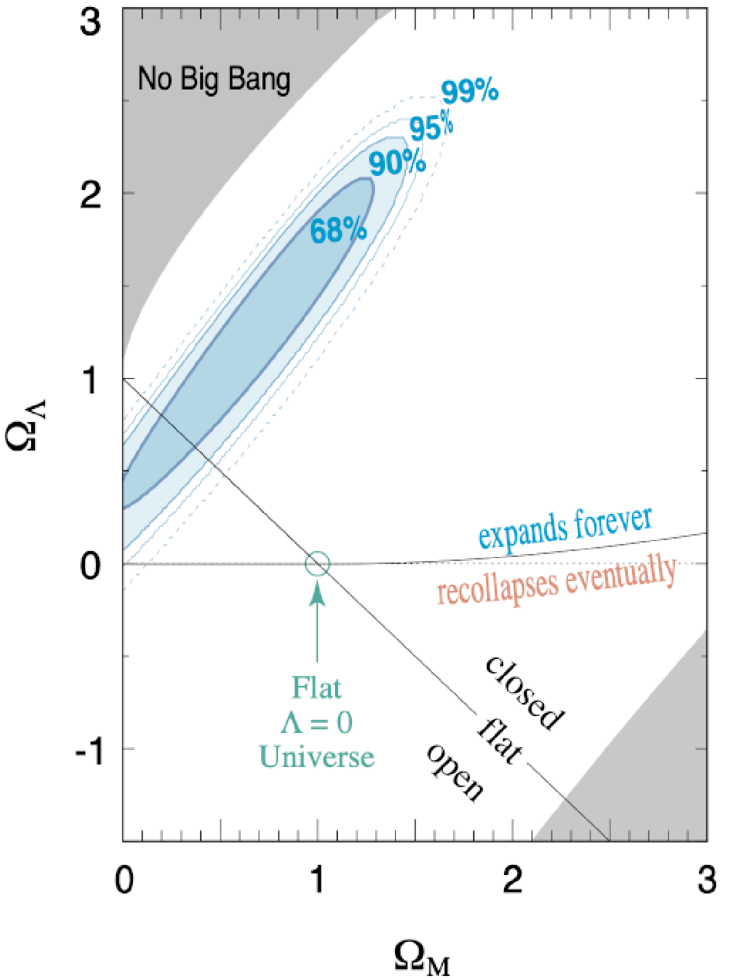
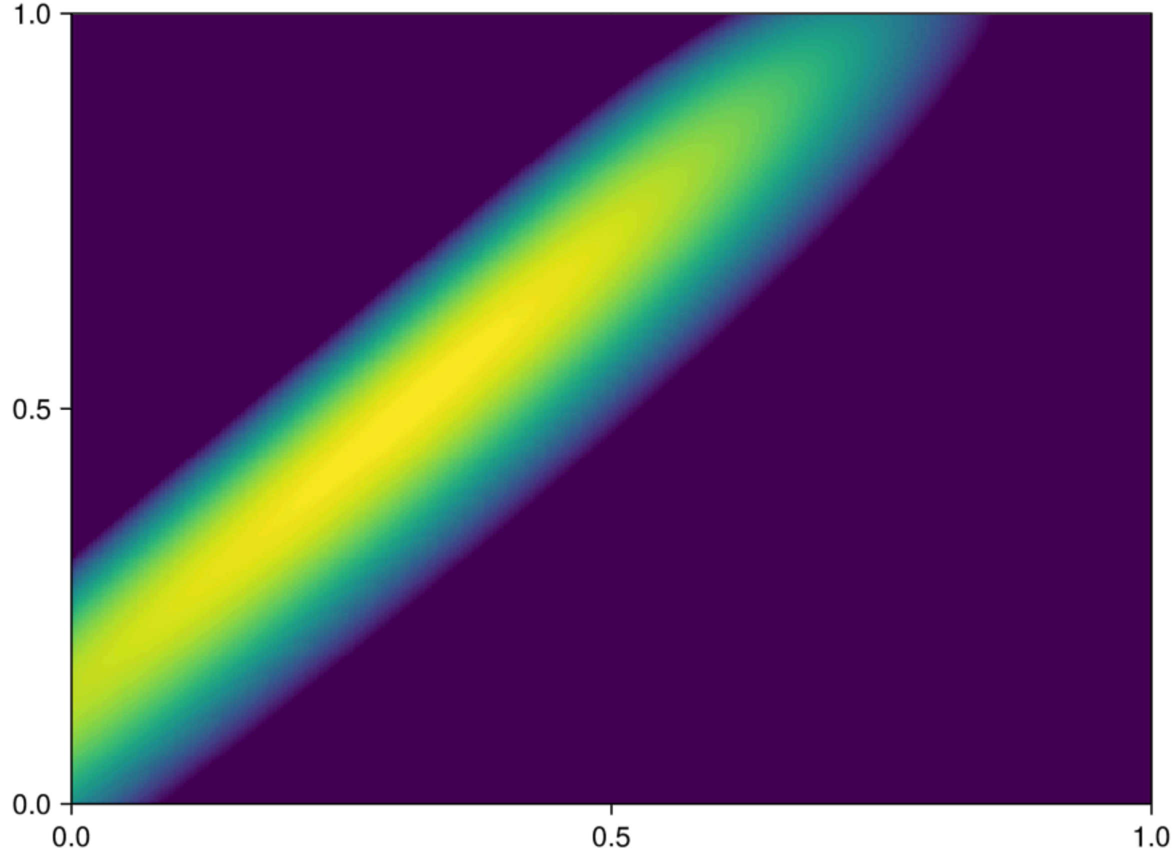


FIG. 7.— Best-fit confidence regions in the Ω_M–Ω_Λ plane for our primary analysis, Fit C.

In [11]:
```
#Setting the resolution
res = 100

LogLike_ = zeros(res, res)
for i in 1:res
    for j in 1:res
        LogLike_[i,j] = supernova_log_likelihood(data.z, data.mag, data.sigma_mag
    end
end
```

In [20]:

```
heatmap(LinRange(0, 1, res), LinRange(0, 1, res), LogLike_, colorrange=[maximum(L
```



[Asif] The general shape here does appear to resemble figure 7 in Perlmutter 1999 paper.

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:

In [13]:

```
#Setting the resolution
res = 100
#Grid expansion factor
grid_factor = 3

LogLike_ = zeros(res*grid_factor, res*grid_factor)
for i in 1:res*grid_factor
    for j in 1:res*grid_factor
        LogLike_[i,j] = supernova_log_likelihood(data.z, data.mag, data.sigma_mag
    end
end
```

```
DomainError with -1.2067371795709825e-5:
sqrt was called with a negative real argument but will only return a complex re
sult if called with a complex argument. Try sqrt(Complex(x)).

Stacktrace:
  [1] throw_complex_domainerror(f::Symbol, x::Float64)
    @ Base.Math .\math.jl:33
  [2] sqrt
    @ .\math.jl:686 [inlined]
  [3] a2E
    @ C:\Users\numbe\.julia\packages\Cosmology\eZT7X\src\Cosmology.jl:65 [inlin
ed]
  [4] #3
    @ C:\Users\numbe\.julia\packages\Cosmology\eZT7X\src\Cosmology.jl:181 [inli
ned]
  [5] evalrule(f::Cosmology.var"#3#4"{Cosmology.ClosedLCDM{Float64}}, a::Float6
4, b::Float64, x::Vector{Float64}, w::Vector{Float64}, gw::Vector{Float64}, nr
m::typeof(LinearAlgebra.norm))
    @ QuadGK C:\Users\numbe\.julia\packages\QuadGK\OtnWt\src\evalrule.jl:30
  [6] refine(f::Cosmology.var"#3#4"{Cosmology.ClosedLCDM{Float64}}, segs::Vecto
r{QuadGK.Segment{Float64, Float64, Float64}}, I::Float64, E::Float64, numeval
s::Int64, x::Vector{Float64}, w::Vector{Float64}, gw::Vector{Float64}, n::Int6
4, atol::Float64, rtol::Float64, maxevals::Int64, nrm::typeof(LinearAlgebra.nor
m))
    @ QuadGK C:\Users\numbe\.julia\packages\QuadGK\OtnWt\src\adapt.jl:70
  [7] adapt
    @ C:\Users\numbe\.julia\packages\QuadGK\OtnWt\src\adapt.jl:52 [inlined]
  [8] do_quadgk(f::Cosmology.var"#3#4"{Cosmology.ClosedLCDM{Float64}}, s::Tuple
{Float64, Float64}, n::Int64, atol::Nothing, rtol::Nothing, maxevals::Int64, nr
m::typeof(LinearAlgebra.norm), segbuf::Nothing)
```

```
    @ QuadGK C:\Users\numbe\.julia\packages\QuadGK\OtnWt\src\adapt.jl:44
  [9] #50
    @ C:\Users\numbe\.julia\packages\QuadGK\OtnWt\src\adapt.jl:253 [inlined]
 [10] handle_infinities(workfunc::QuadGK.var"#50#51"{Nothing, Nothing, Int64, I
nt64, typeof(LinearAlgebra.norm), Nothing}, f::Cosmology.var"#3#4"{Cosmology.Cl
osedLCDM{Float64}}, s::Tuple{Float64, Float64})
    @ QuadGK C:\Users\numbe\.julia\packages\QuadGK\OtnWt\src\adapt.jl:145
 [11] #quadgk#49
    @ C:\Users\numbe\.julia\packages\QuadGK\OtnWt\src\adapt.jl:252 [inlined]
 [12] quadgk
    @ C:\Users\numbe\.julia\packages\QuadGK\OtnWt\src\adapt.jl:250 [inlined]
 [13] quadgk
    @ C:\Users\numbe\.julia\packages\QuadGK\OtnWt\src\adapt.jl:247 [inlined]
 [14] Z
    @ C:\Users\numbe\.julia\packages\Cosmology\eZT7X\src\Cosmology.jl:180 [inli
ned]
 [15] comoving_transverse_dist(c::Cosmology.ClosedLCDM{Float64}, z₁::Float64, z
₂::Nothing; kws::@Kwargs{})
    @ Cosmology C:\Users\numbe\.julia\packages\Cosmology\eZT7X\src\Cosmology.j
l:203
 [16] comoving_transverse_dist (repeats 2 times)
    @ C:\Users\numbe\.julia\packages\Cosmology\eZT7X\src\Cosmology.jl:201 [inli
ned]
 [17] luminosity_dist
    @ C:\Users\numbe\.julia\packages\Cosmology\eZT7X\src\Cosmology.jl:218 [inli
ned]
 [18] distance_modulus(universe::Cosmology.FlatLCDM{Float64}, z::Float64)
    @ Main .\In[7]:6 [inlined]
 [19] #9
    @ .\In[10]:13 [inlined]
 [20] iterate(g::Base.Generator, s::Vararg{Any})
```

```
    @ Base .\generator.jl:47 [inlined]
 [21] collect_to!(dest::Vector{Float64}, itr::Base.Generator{Vector{Float64}, v
ar"#9#10"{Cosmology.ClosedLCDM{Float64}}}, offs::Int64, st::Int64)
    @ Base .\array.jl:892
 [22] collect_to_with_first!(dest::AbstractArray, v1::Any, itr::Any, st::Any)
    @ Base .\array.jl:870 [inlined]
 [23] _collect(c::Vector{Float64}, itr::Base.Generator{Vector{Float64}, var"#9#
10"{Cosmology.ClosedLCDM{Float64}}}, ::Base.EltypeUnknown, isz::Base.HasShape
{1})
    @ Base .\array.jl:864
 [24] collect_similar(cont::Vector{Float64}, itr::Base.Generator{Vector{Float6
4}, var"#9#10"{Cosmology.ClosedLCDM{Float64}}})
    @ Base .\array.jl:763
 [25] map(f::Function, A::Vector{Float64})
    @ Base .\abstractarray.jl:3282
 [26] supernova_log_likelihood(z::Vector{Float64}, mag::Vector{Float64}, mag_er
ror::Vector{Float64}, M::Float64, Omatter::Float64, Ode::Float64)
    @ Main .\In[10]:13
 [27] top-level scope
    @ .\In[13]:9
```

In [14]:

```
# Example of Julia's try-catch syntax:
ll = 0.
try:
    ll = supernova_log_likelihood(data.z, data.mag, data.sigma_mag, abs_mag, 2.0,
catch err
    ll = -Inf
end
```

```
ParseError:
# Error @ ]8;;file://C:/Users/numbe/Downloads/Numerical_Methods/Homeworks/PSI-Numerical-Methods-2024-MCMC-Homework/In[14]#3:5\In[14]:3:5]8;;\
ll = 0.
#    ┌
try:
    ll = supernova_log_likelihood(data.z, data.mag, data.sigma_mag, abs_mag, 2.0, 2.0)
#└─┘ ── whitespace not allowed after `:` used for quoting

Stacktrace:
 [1] top-level scope
   @ In[14]:3
```

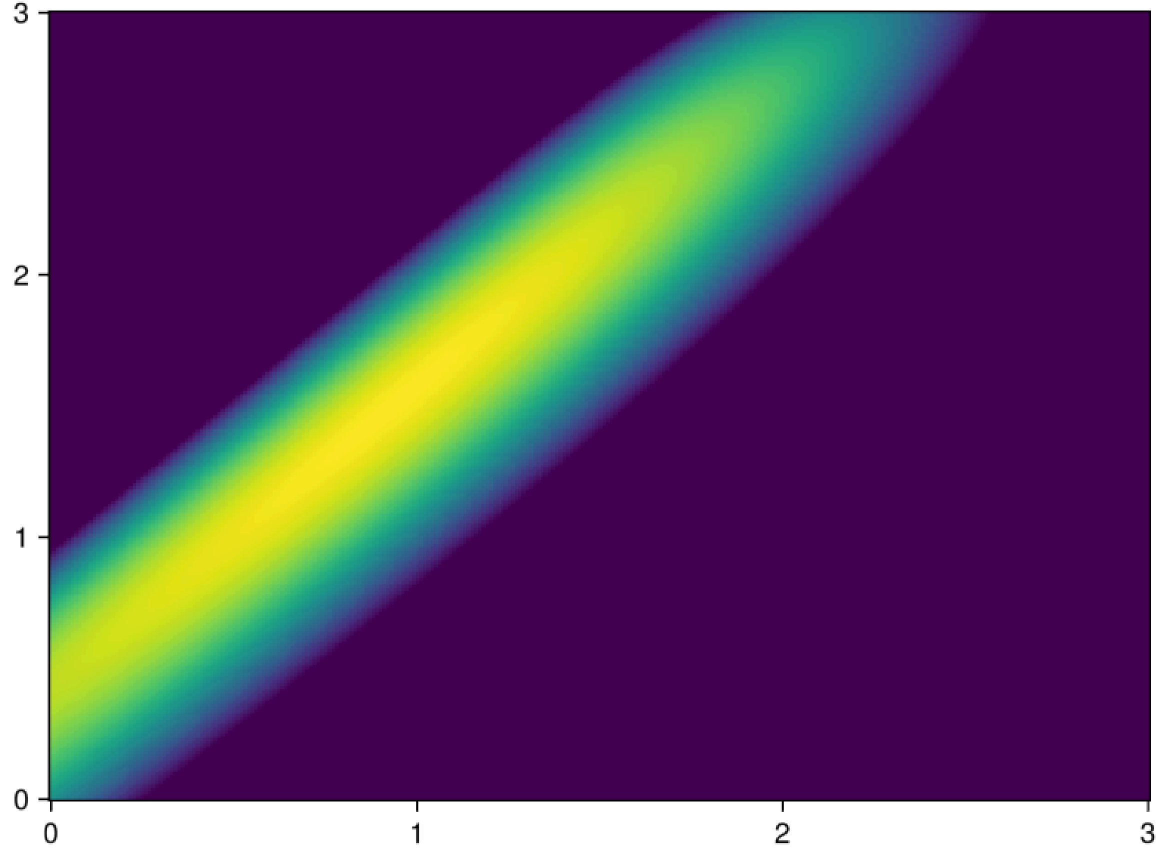
This will "try" to run the `supernova_log_likelihood` function, and if it fails, it will go into the "catch" branch.

In [15]:

```
#Setting the resolution
res = 100
#Grid expansion factor
grid_factor = 3

LogLike_ = zeros(res*grid_factor, res*grid_factor)
for i in 1:res*grid_factor
    for j in 1:res*grid_factor
        try
            LogLike_[i,j] = supernova_log_likelihood(data.z, data.mag, data.sigma
        catch err
            LogLike_[i,j] = -Inf
        end
    end
end
```

In [46]:

```
heatmap(LinRange(0, grid_factor, grid_factor*res), LinRange(0, grid_factor, grid_
```



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

In [17]:

```
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;
```

In [34]:

```
#Updating the log-likelihood function definition to include Try-Catch statement
function supernova_log_likelihood_(M, Omatter, Ode)

    universe_ = cosmology(OmegaK = 1 - Ode - Omatter, OmegaM = Omatter, Tcmb=0)

    # Array for specific given z values
    DLx = 0.
    try
        DLx = map(x -> distance_modulus(universe_, x), data.z)
    catch err
        DLx = -Inf
    end

    #Adding absolute magnitude to distance modulus
    fₓ_ = DLx .+ M

    #Comparing with measured magnitude
    X_ = (fₓ_ .- data.mag) ./ data.sigma_mag

    #Computing log-likelihood
    LogLike = -0.5 .* X_ .^2

    return sum(LogLike)

end;
```

In [35]:

```
# Defining function to generate jump sites
function propose(M, Omatter, Ode, jump_size)
    return [M, Omatter, Ode] .+ randn(length([M, Omatter, Ode])) .* jump_size
end;
```

In [36]:

```
# Defining the MCMC function
function mcmc_cyclic(logprob_func, propose_func, M, Omatter, Ode, n_steps, jump_s
    p = [M, Omatter, Ode]
    logprob = logprob_func(p[1], p[2], p[3])
    chain = zeros(n_steps, length(p))
    n_accept = zeros(length(p))
    for i in 1:n_steps
        # Updating the index one at a time
        update_index = 1 + ((i-1) % length(p))

        # Generating new values for all parameters through the propose function
        p_prop = propose_func(p[1], p[2], p[3], jump_size)

        # Keeping one of the new parameter values
        p_new = copy(p)
        p_new[update_index] = p_prop[update_index]
        logprob_new = logprob_func(p_new[1], p_new[2], p_new[3])
        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
            u = rand()
            if u < ratio
                # Move to the new parameter
                p = p_new
                logprob = logprob_new
                n_accept[update_index] += 1
            else
                # Don't move to proposed parameter
            end
        end
        chain[i, 1:end] = p_new
    end
    # The number of times we step each parameter is roughly n_steps/n_parameters
    return chain, n_accept ./ (n_steps ./ length(p))
```

```
end;
```

In [37]:

```
# Number of MCMC steps
Nsteps = 100000

values, acceptance_rate = mcmc_cyclic(supernova_log_likelihood_, propose, -15., 0
```

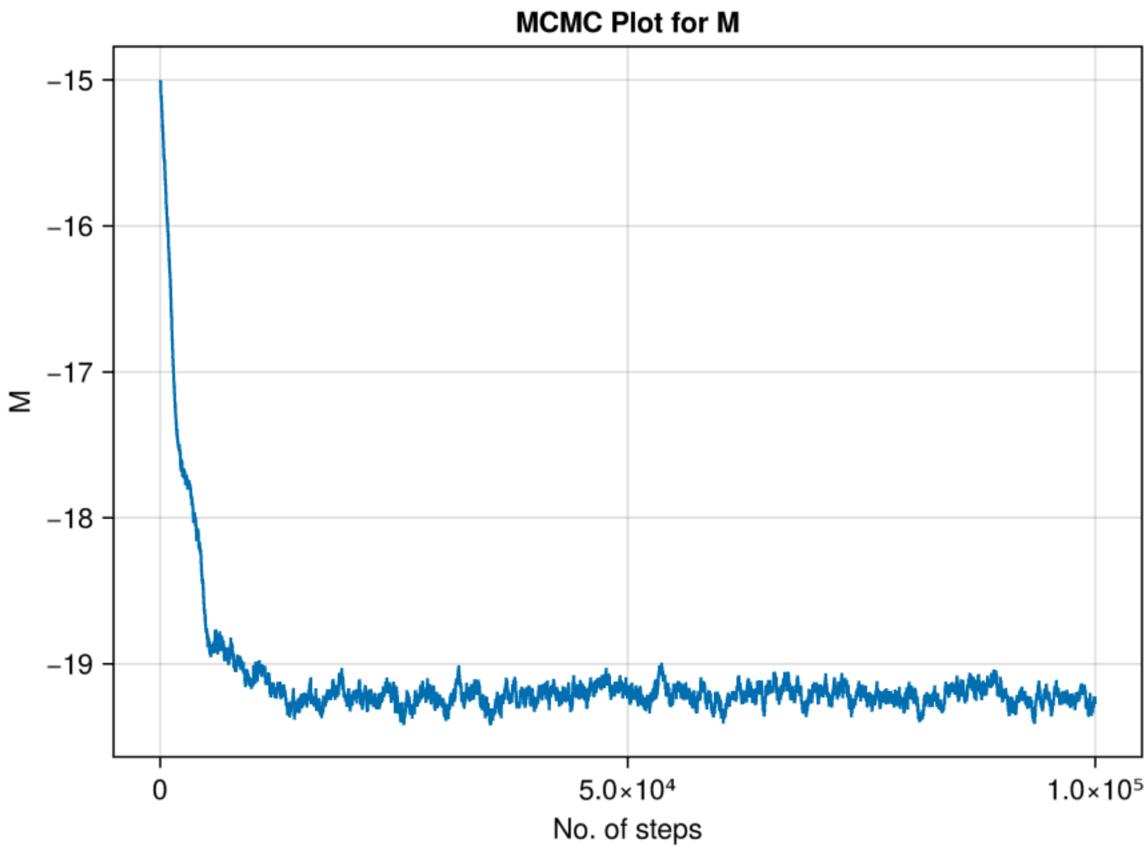
```
([-14.993384313831216 0.2 0.5; -15.0 0.18148740822998144 0.5; … ; -19.268859417
348235 1.0979079101077487 1.9394470514427906; -19.265828652166242 1.09790791010
77487 1.9394470514427906], [0.87771, 0.73842, 0.77367])
```

MCMC steps needed for convergence

Having tried various number of steps for the MCMC code above, we notice that the level of convergence starts to saturate once we go past 30000 steps or so (as evident in the figures below). So, to ensure good convergence, we choose 100000 steps.

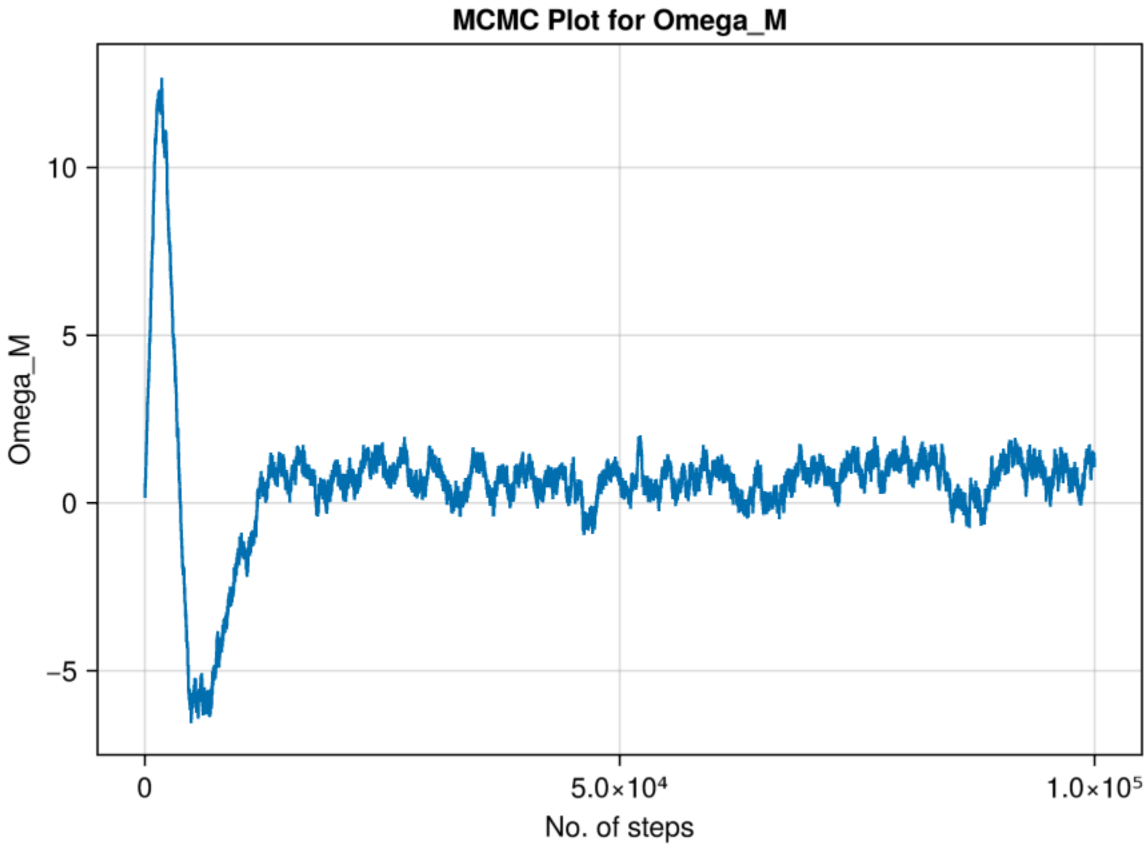
In [42]:

```
# Figure for M
f = Figure()
ax = Axis(f[1,1], title="MCMC Plot for M", ylabel="M", xlabel="No. of steps")
lines!(LinRange(1, Nsteps, Nsteps), values[:, 1])
f
```



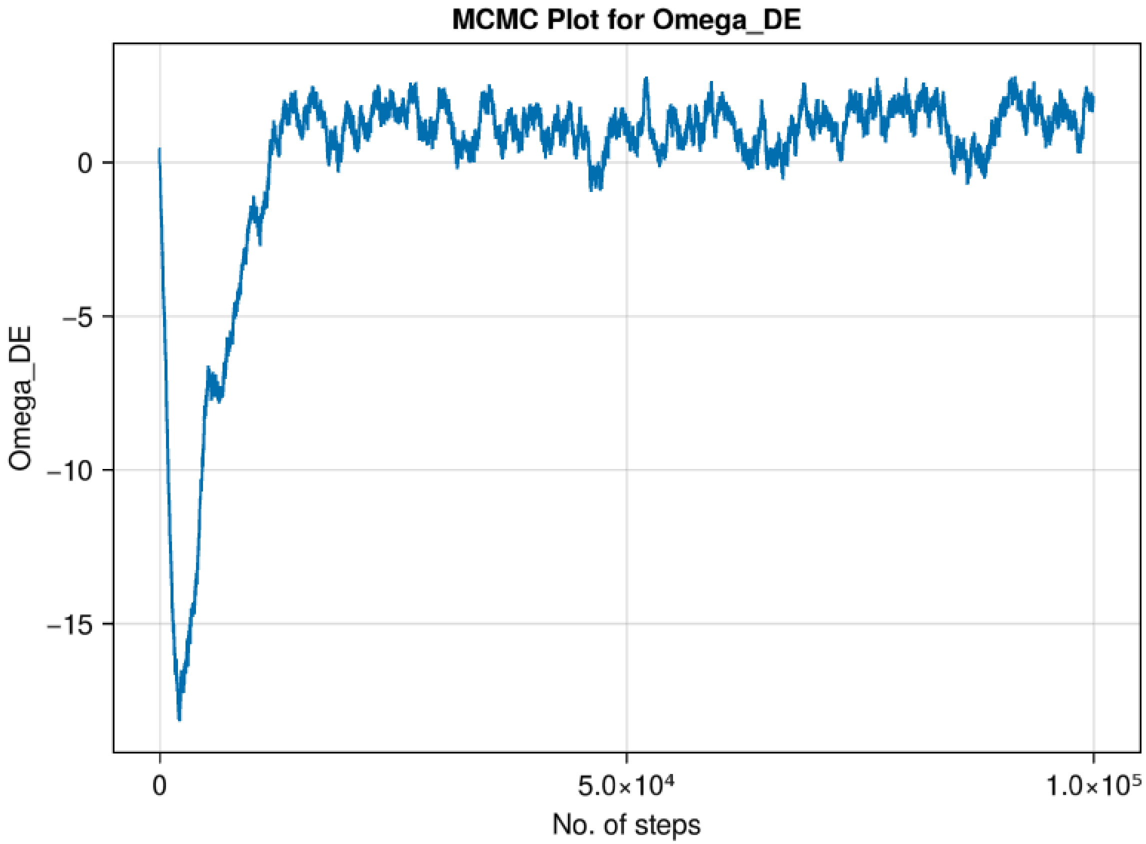
In [41]:

```
# Figure for Omega_M
f = Figure()
ax = Axis(f[1,1], title="MCMC Plot for Omega_M", ylabel="Omega_M", xlabel="No. of
lines!(LinRange(1, Nsteps, Nsteps), values[:, 2])
f
```



In [43]:

```
# Figure for Omega_DE
f = Figure()
ax = Axis(f[1,1], title="MCMC Plot for Omega_DE", ylabel="Omega_DE", xlabel="No.
lines!(LinRange(1, Nsteps, Nsteps), values[:, 3])
f
```



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 (https://juliapackages.com/p/cornerplot)](https://juliapackages.com/p/cornerplot)) but I have not had luck getting it to work for me.

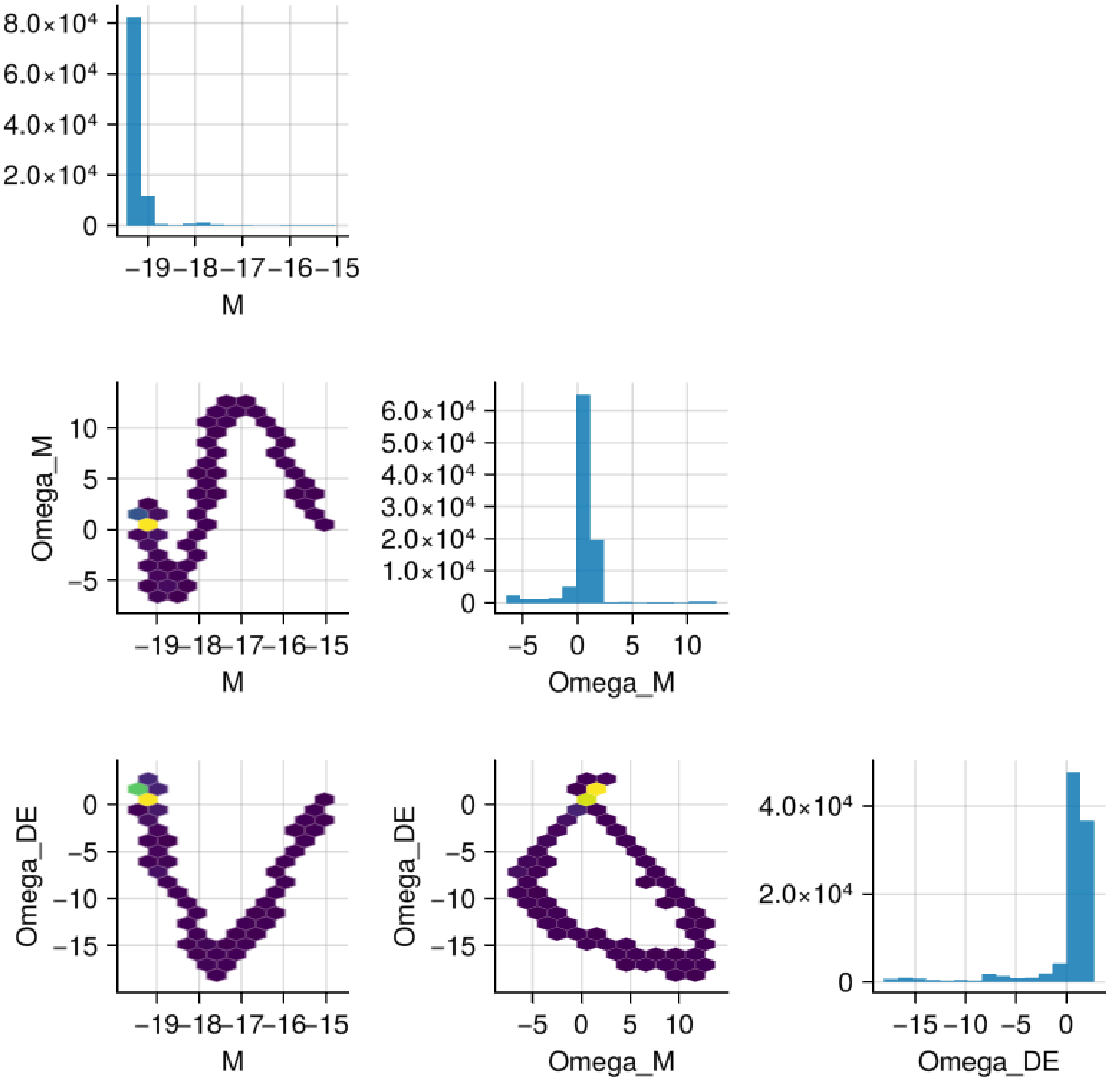
Once you have made you corner plots, please write a few sentences interpreting what you see. Is the

In [44]:

```
# Function to make cornerplots
function cornerplot(x, names; figsize=(600,600))
    # Number of columns of data
    dim = size(x, 2)
    # Number of rows of the 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
            hexbin!(x[idxs,j], x[idxs,i])
            ax.xlabel = names[j]
            ax.ylabel = names[i]
        end
    end
    f
end;
```

In [45]:

```
cornerplot(values, ["M", "Omega_M", "Omega_DE"])
```



Interpreting the corner plot

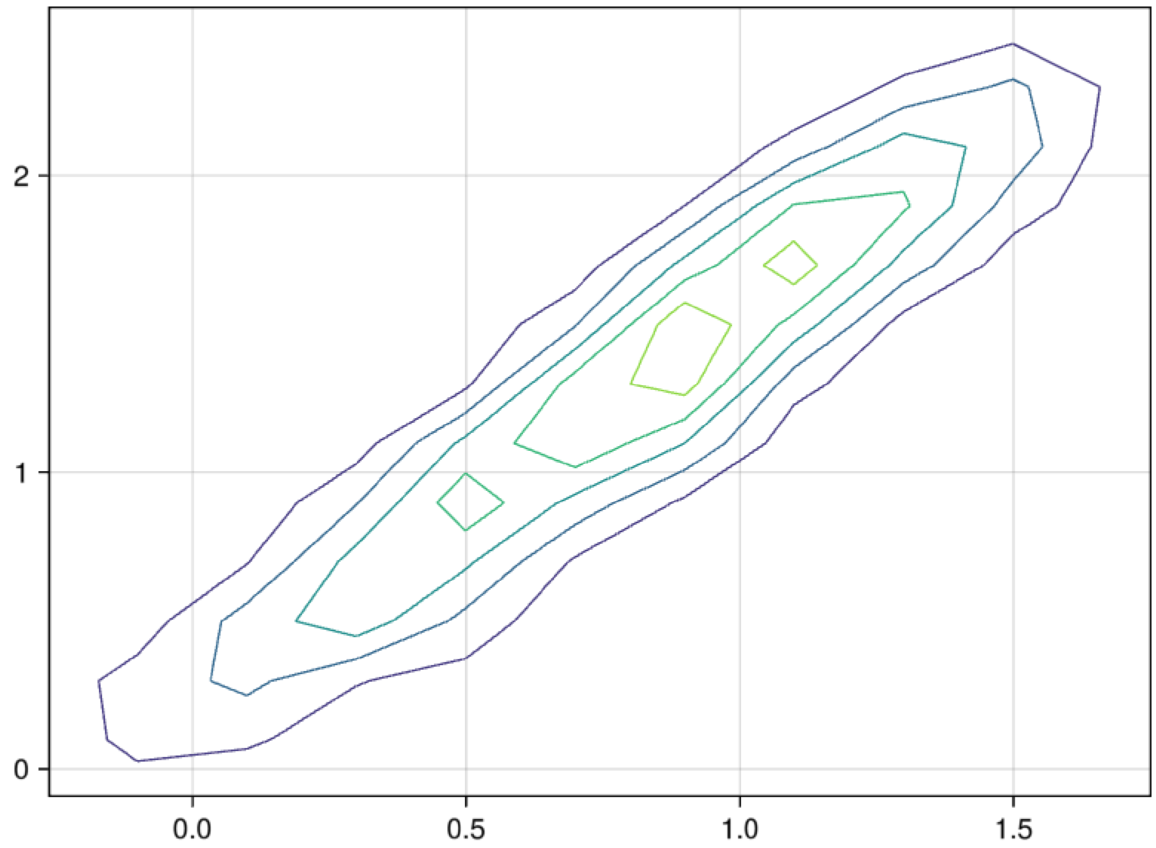
- The nuisance parameter M is not correlated with the Omegas, as we see an irregular dependence in the form of a V in Omega_DE's case, and an inverted V in Omega_M's case.
- The omegas seem to be anti-correlated given the downward trend seen in their plot. However, this is only true when we look at the overall trend. I don't have a good interpretation for the hollow section present in the plot.

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.

In [55]:

```
Pkg.add("FHist")
using FHist
h = Hist2D((values[:, 2], values[:, 3]); nbins=(100,100))

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

contour(xc, yc, counts)
```

```
   Resolving package versions...
  No Changes to `C:\Users\numbe\.julia\environments\v1.10\Project.toml`
  No Changes to `C:\Users\numbe\.julia\environments\v1.10\Manifest.toml`
```



Acknowledgements

- I would like to thank Marko for helping me with the plots and discussing the MCMC algorithm.
- I've used suggestions from Bing AI/GPT-4 for getting the Julia syntax in several places.