Project 1

Transiting Exoplanet

Your team detected a planet candidate in the data from the Transiting Exoplanet Survey Satellite (TESS). The data is very noisy, but you are relatively confident the planet is there. Based on the TESS data, you can predict that the center of the next transit will be between the time t = 1.212 days and t = 1.362 days [Note: t is measured relative to an arbitrary reference time]. The TESS data also tell you that the radius of the planet is between 1% and 10% the radius of the star. Unfortunately, you don't know where the planet crosses the star, i.e. the impact parameter is unconstrained between 0 and 1. Assume that the orbital period is exactly known to be 3.0 days. You wrote a proposal and successfully convinced the time allocation committee of the James Webb Space Telescope (JWST) to obtain new very precise data for you. YAY!!! You receive the data in Project1_JWST_data.csv. The errors are 10-4=0.01% on each data point in the light curve. From these data, you want to determine new estimates with uncertainties for the transit time, the planet-to-star radius ratio, and the impact parameter.

Batman: https://lkreidberg.github.io/batman/docs/html/index.html joshspeagle/dynesty: Dynamic Nested Sampling package for computing Bayesian posteriors and evidences

Julia packages:

madsjulia/AffineInvariantMCMC.jl: Affine Invariant Markov Chain Monte Carlo (MCMC) Ensemble sampler chalk-lab/NestedSamplers.jl: Implementations of single and multi-ellipsoid nested sampling bat/BAT.jl: A Bayesian Analysis Toolkit in Julia

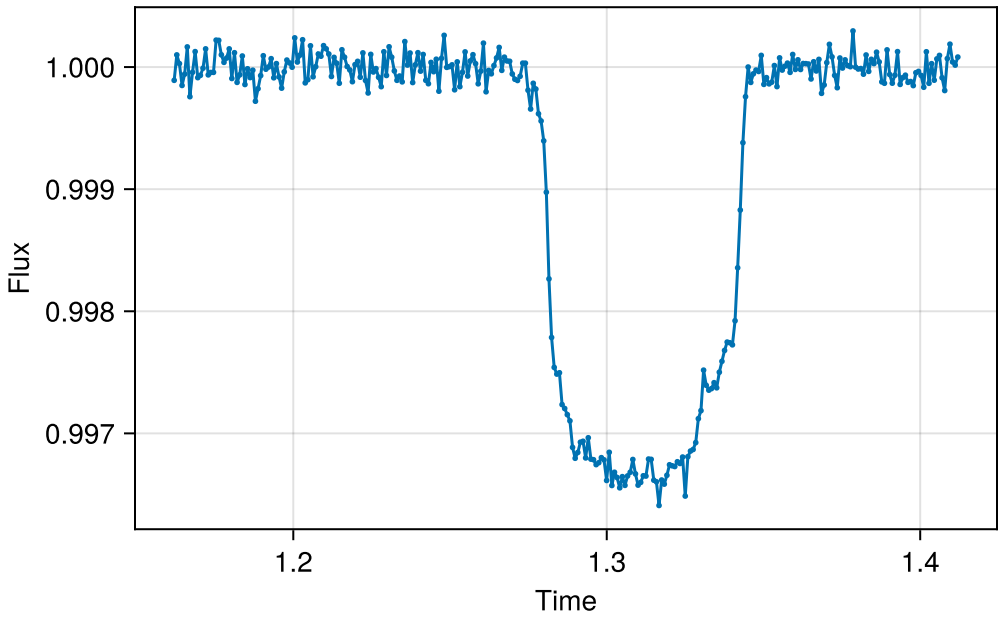
> **Note**
>
> In the updated version of the project, we use `Transits.jl` instead of `batman`, which is more performant and provides features like automatic differentiation.

Introduction

```
dir = "docs/courses/epss298_DataAnalysis"
if isdir(dir)
    cd(dir)
    Pkg.activate(".")
    Pkg.instantiate()
```

```
    cd("projects")
end
```

```
using CSV, DataFrames
using CairoMakie
path = "Project1_JWST_data.csv"
data = CSV.read(path, DataFrame)

f = Figure()
markersize = 4
ax = Axis(f[1, 1], xlabel="Time", ylabel="Flux")
scatterlines!(ax, data.time, data.flux; markersize)
f
```



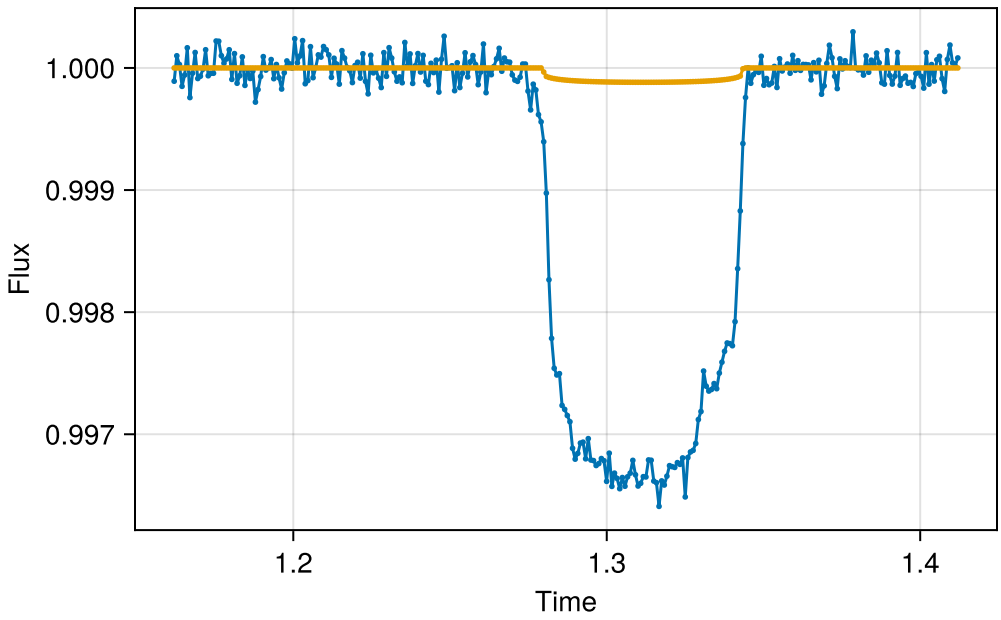
Part A

Step 1: Define the physical model

Write a function for your physical "forward" model that takes in a vector with these three parameters:

```
using Orbits
using Transits
```

```
t0_i = 1.311544
b_i = 0.1
rp_i = 0.01

create_orbit(t0, b; a=15, per=3, kw...) = KeplerianOrbit(; period=per, t0, a,
b, kw...)
create_orbit(t::Tuple; kw...) = create_orbit(t[1], t[2]; kw...)

function light_curve(time, t0, b, rp; u=[0.3, 0.3], kwargs...)
    orbit = create_orbit(t0, b; kwargs...)
    ld = QuadLimbDark(u)
    return @. ld(orbit, time, rp)
end

flux = light_curve(data.time, t0_i, b_i, rp_i)
scatterlines!(f[1, 1], data.time, flux; markersize)
f
```




```
function light_curve_n(time, params::NTuple{N}; ld=QuadLimbDark([0.3, 0.3]),
kwargs...) where {N}
    rps = getindex.(params, 3)
    orbits = create_orbit.(params; kwargs...)
    map(time) do t
        s = reduce(+, ntuple(i -> ld(orbits[i], t, rps[i]), N))
        s - (N - 1) * one(s)
```

```
    end
end
```

```
light_curve_n (generic function with 1 method)
```

Step 2: Define the Bayesian analysis

1. Write the given prior information as a log-prior function
2. Write a log-likelihood function
3. Write a log-probability function

Vanilla log-prior, log-likelihood, and log-probability functions could be written as:

```
log_prior(x, (lb, ub)) = lb <= x <= ub ? 0.0 : -Inf

# Transit time between 1.212 and 1.362 days
# Planet radius between 1% and 10% of star radius
# Impact parameter between 0 and 1
log_prior(t0, rp, b) = log_prior(t0, (1.212, 1.362)) +
    log_prior(rp, (0.01, 0.10)) +
    log_prior(b, (0.0, 1.0))

function log_likelihood(params, time, obs, error;
    per=3.0, # Period in days
    a=15.0, # Semi-major axis in stellar radii
    u=[0.3, 0.3] # Limb darkening coefficients
)
    t0, rp, b = params
    inc = acosd(b / a)  # Inclination in degrees
    model_flux = light_curve(time; t0, per, rp, a, inc, u)

    # Calculate log-likelihood (assuming Gaussian errors)
    residuals = obs .- model_flux
    chi_squared = sum((residuals ./ error) .^ 2)
    -0.5 * chi_squared
end

log_probability(params, time, obs, error) = log_prior(params) +
log_likelihood(params, time, obs, error)
```

Using `Turing.jl`, we can write the same model more compactly:

```
using Turing

@model function transit_model(time, flux; error=1e-4)
```

```
    t0 ~ Uniform(1.212, 1.362)
    rp ~ Uniform(0.01, 0.10)
    b ~ Uniform(0.0, 0.8)
    model_flux = light_curve(time, t0, b, rp)
    return flux ~ MvNormal(model_flux, error)
end

model = transit_model(data.time, data.flux)
```

```
DynamicPPL.Model{typeof(transit_model), (:time, :flux), (:error,), (),
Tuple{Vector{Float64}, Vector{Float64}}, Tuple{Float64},
DynamicPPL.DefaultContext}(transit_model, (time = [1.1620000000000001,
1.162836120401338, 1.1636722408026758, 1.1645083612040135, 1.1653444816053513,
1.1661806020066892, 1.167016722408027, 1.1678528428093646, 1.1686889632107025,
1.1695250836120403  …  1.40447491638796, 1.4053110367892978,
1.4061471571906357, 1.4069832775919733, 1.4078193979933111, 1.408655518394649,
1.4094916387959868, 1.4103277591973247, 1.4111638795986623,
1.4120000000000001], flux = [0.99989143693967, 1.0000997345446583,
1.0000282978498052, 0.9998493705286082, 0.9999421399748032,
1.0001651436537098, 0.9997573320756606, 0.9999571087371144,
1.0001265936258705, 0.9999133259597734  …  0.9998925233406881,
1.000066831686674, 1.0000955832355167, 0.9999122386413373, 0.9998076284270303,
1.000069578731914, 1.0001875800546713, 1.0000415694539893, 1.000016054442148,
1.000081976060961]), (error = 0.0001,), DynamicPPL.DefaultContext())
```

Step 3: Demonstrate solution on a grid

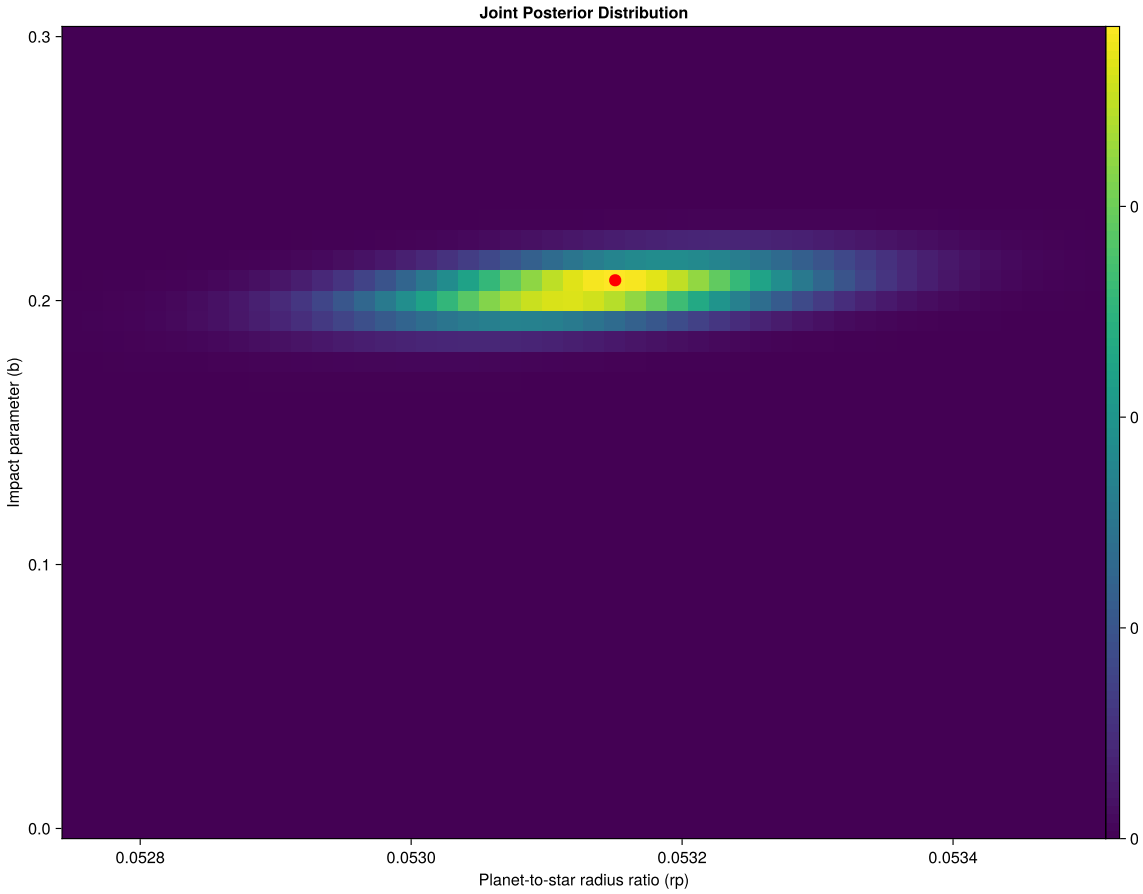
Determine new estimates with uncertainties of the planet-to-star radius ratio and the impact parameter. Fix the transit mid-time to t0 = 1.311544 days. Make a 50x50 grid that explores planet-to-star radius ratios between 0.05275 and 0.53505 as well as impact parameters between 0 and 0.3.

1. Plot the joint posterior distribution with and mark the best fit with a marker

```
using LinearAlgebra  # For eigen decomposition and other matrix operations
using Statistics     # For statistical functions
using StatsBase

fixed_t0 = 1.311544

n_grid = 50
rp_grid = range(0.05275, 0.053505, length=n_grid)
b_grid = range(0.0, 0.3, length=n_grid - 10)

log_posterior_grid = [
    logjoint(model, (; rp, b, t0=fixed_t0))
```

```
    for rp in rp_grid, b in b_grid
]

posterior_grid = exp.(log_posterior_grid .- maximum(log_posterior_grid)) #
(subtract max to avoid numerical issues)
posterior_grid = posterior_grid ./ sum(posterior_grid) # Normalize the
posterior

# Find the maximum posterior point
max_idx = argmax(posterior_grid)
best_rp = rp_grid[max_idx[1]]
best_b = b_grid[max_idx[2]]
@info "Best fit values" best_rp best_b

b_label = "Impact parameter (b)"
rp_label = "Planet-to-star radius ratio (rp)"
fig = Figure(size=(1000, 800))
ax1 = Axis(fig[1, 1], xlabel=rp_label, ylabel=b_label, title="Joint Posterior
Distribution")

hm = heatmap!(ax1, rp_grid, b_grid, posterior_grid)
scatter!(ax1, [best_rp], [best_b], color=:red, markersize=15)
Colorbar(fig[1, 1, Right()], hm, label="Posterior Probability")
fig
```

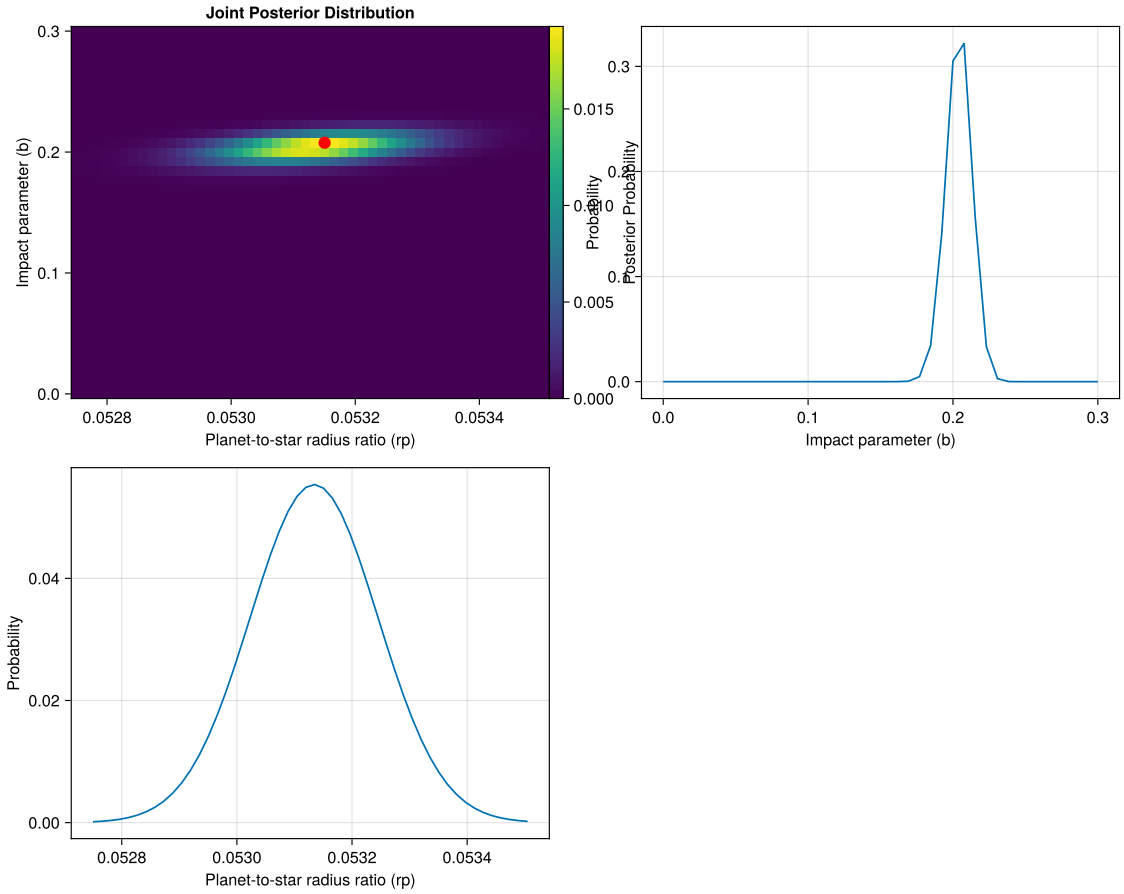
```
┌ Info: Best fit values
│   best_rp = 0.053150612244897956
└   best_b = 0.2076923076923077
```



2. Marginalized posterior distributions

Plot the marginalized posterior distributions for the planet-to-star radius ratio and the impact parameters.

```
rp_posterior = vec(sum(posterior_grid, dims=2))
rp_posterior = weights(rp_posterior ./ sum(rp_posterior))  # Normalize

b_posterior = vec(sum(posterior_grid, dims=1))
b_posterior = weights(b_posterior ./ sum(b_posterior))  # Normalize

ax2 = Axis(fig[2, 1], xlabel=rp_label, ylabel="Probability")
lines!(ax2, rp_grid, rp_posterior)
ax3 = Axis(fig[1, 2], xlabel=b_label, ylabel="Probability")
lines!(ax3, b_grid, b_posterior)
fig
```



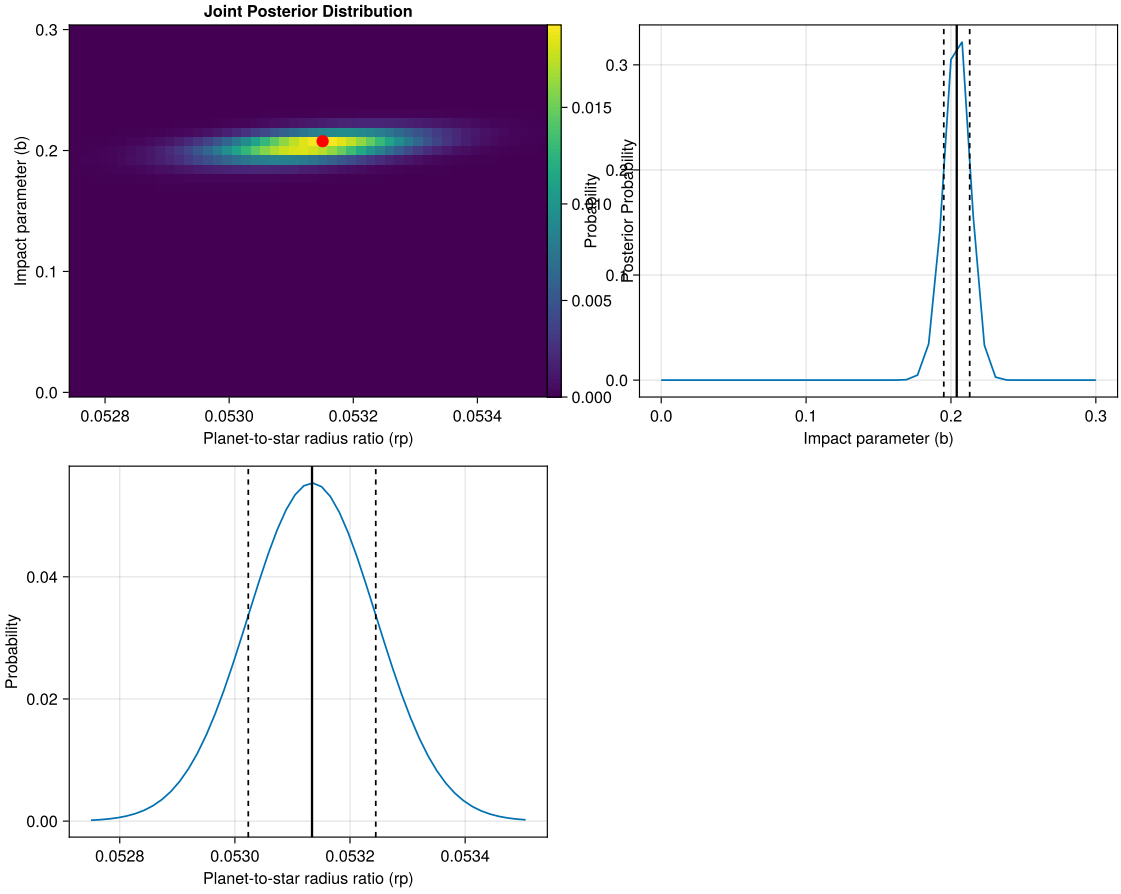
3. Best estimates and uncertainties

Calculate the mean and +/-1 sigma uncertainties for both parameters. Mark the mean value with a solid vertical line in the figure created in Step 3.2. Make the +/-1 sigma uncertainties with dashed vertical lines. Report the best estimates (mean and uncertainty) for both parameters quantitatively.

```
using Measurements
# For rp
rp_mean = mean(rp_grid, rp_posterior)
rp_std = std(rp_grid, rp_posterior)
rp = measurement(rp_mean, rp_std)

# For b
b_mean = mean(b_grid, b_posterior)
b_std = std(b_grid, b_posterior)
b = measurement(b_mean, b_std)
```

```
# Add mean and +/- 1 sigma to the marginalized plots
vlines!(ax2, [rp_mean], color=:black, linewidth=2)
vlines!(ax2, [rp_mean - rp_std, rp_mean + rp_std], color=:black,
linestyle=:dash)

vlines!(ax3, [b_mean], color=:black, linewidth=2)
vlines!(ax3, [b_mean - b_std, b_mean + b_std], color=:black, linestyle=:dash)

@info "Planet-to-star radius ratio" rp
@info "Impact parameter" b
fig
```

```
┌ Info: Planet-to-star radius ratio
└   rp = 0.05313 ± 0.00011
┌ Info: Impact parameter
└   b = 0.204 ± 0.0089
```



4. Covariance matrix and confidence regions

Calculate the covariance matrix that best represents the joint posterior distribution. What is the correlation coefficient between the planet-to-star radius ratios and the impact parameter? Plot the 68% and 95% confidence regions onto the figure created in Step 3.1.

```
"""Calculate covariance matrix elements"""
function covariance_matrix(p, x, y)
    x_2d = repeat(x, 1, length(y))
    y_2d = repeat(y', length(x), 1)
    wp = weights(p)
    x_mean = mean(x_2d, wp)
    y_mean = mean(y_2d, wp)

    # Calculate covariance matrix elements
    cov_x_x = mean((x_2d .- x_mean) .^ 2, wp)
    cov_y_y = mean((y_2d .- y_mean) .^ 2, wp)
    cov_x_y = mean((x_2d .- x_mean) .* (y_2d .- y_mean), wp)
    return [cov_x_x cov_x_y; cov_x_y cov_y_y]
end

cov_matrix = covariance_matrix(posterior_grid, rp_grid, b_grid)
@info "Covariance matrix:" cov_matrix

# Calculate correlation coefficient
corr_coef = cov_matrix[1, 2] / (sqrt(cov_matrix[1, 1]) * sqrt(cov_matrix[2,
2]))
@info "Correlation coefficient:" corr_coef
```

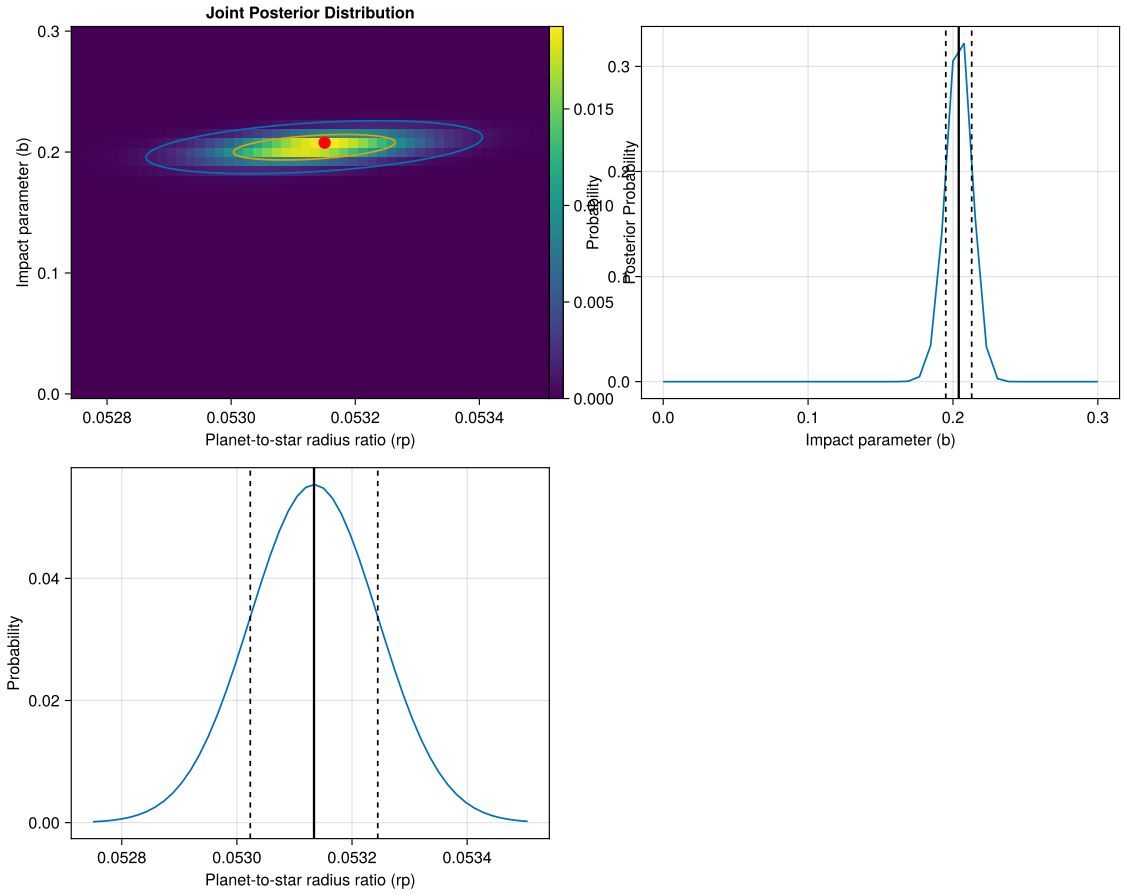
```
┌ Info: Covariance matrix:
│   cov_matrix =
│    2×2 Matrix{Float64}:
│     1.22606e-8  3.77043e-7
└     3.77043e-7  7.99982e-5
┌ Info: Correlation coefficient:
└   corr_coef = 0.3807109735554177
```

```
using Distributions

function getellipsepoints(cx, cy, rx, ry, θ; length=100)
    t = range(0, 2π; length)
    ellipse_x_r = @. rx * cos(t)
    ellipse_y_r = @. ry * sin(t)
    R = [cos(θ) sin(θ); -sin(θ) cos(θ)]
    r_ellipse = [ellipse_x_r ellipse_y_r] * R
```

```
    x = @. cx + r_ellipse[:, 1]
    y = @. cy + r_ellipse[:, 2]
    (x, y)
end

function getellipsepoints(μ, Σ, confidence=0.95)
    quant = quantile(Chisq(2), confidence) |> sqrt

    egvs = eigvals(Σ)
    if egvs[1] > egvs[2]
        idxmax = 1
        largestegv = egvs[1]
        smallesttegv = egvs[2]
    else
        idxmax = 2
        largestegv = egvs[2]
        smallesttegv = egvs[1]
    end

    rx = quant * sqrt(largestegv)
    ry = quant * sqrt(smallesttegv)

    eigvecmax = eigvecs(Σ)[:, idxmax]
    θ = atan(eigvecmax[2] / eigvecmax[1])
    if θ < 0
        θ += 2π
    end
    getellipsepoints(μ..., rx, ry, θ)
end
```

```
getellipsepoints (generic function with 3 methods)
```

```
μ = [rp_mean, b_mean]
Σ = cov_matrix

lines!(ax1, getellipsepoints(μ, Σ)..., label="95% confidence interval")
lines!(ax1, getellipsepoints(μ, Σ, 0.5)..., label="50% confidence interval")
fig
```



Part B

Step 4: Solve the full problem using MCMC and Nested Sampling

Determine new estimates with uncertainties of the transit time, the planet-to-star radius ratio, and the impact parameter.

1. Using Emcee. Make all possible plots and quantitative assessments to convince the reviewer that your results are converged
2. Using Dynesty. Make all possible plots and quantitative assessments to convince the reviewer that your results are converged

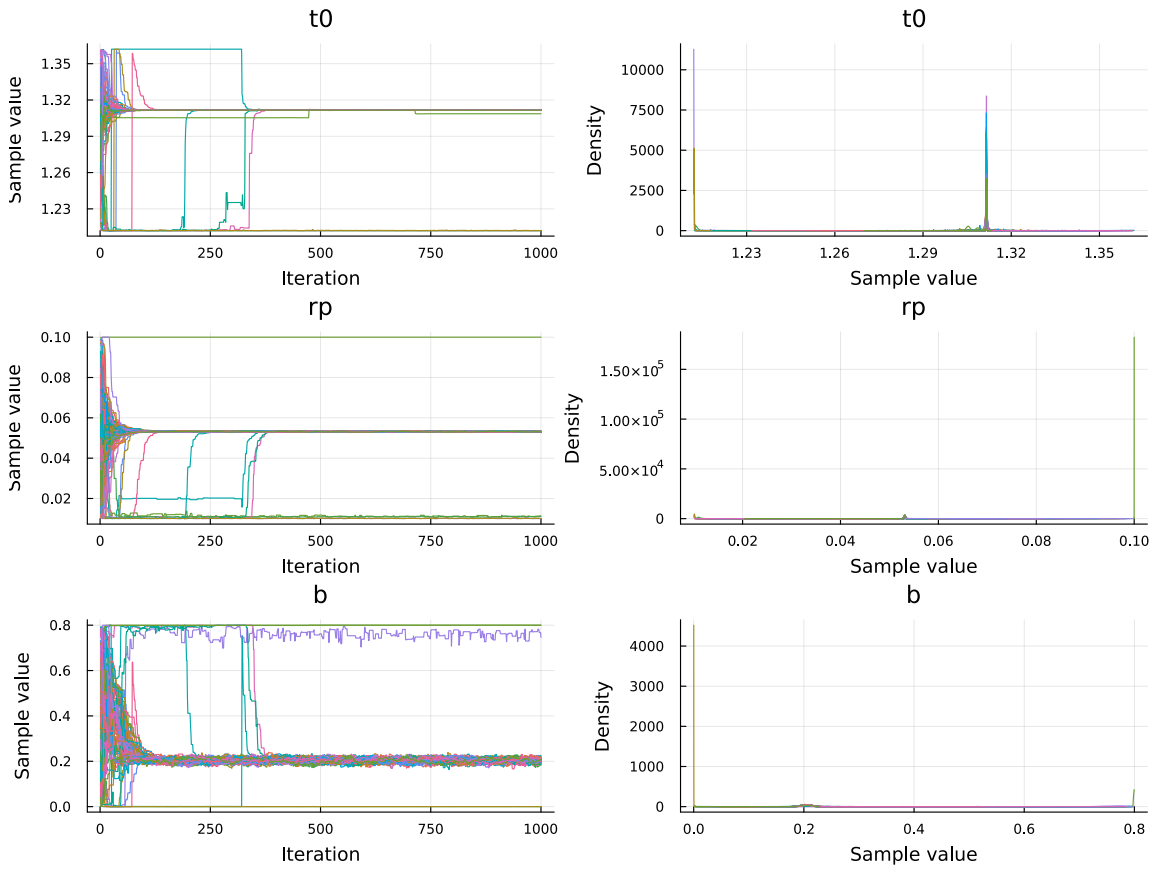
Affine Invariant MCMC (Emcee-like)

```
chain = sample(model, Emcee(20), MCMCThreads(), 1000, 4; progress=false)
```

```
Chains MCMC chain (1000×4×80 Array{Float64, 3}):
```

```
Iterations        = 1:1:1000
Number of chains  = 80
Samples per chain = 1000
parameters        = t0, rp, b
internals         = lp

Summary Statistics
  parameters      mean       std      mcse   ess_bulk   ess_tail      rhat   e
…
      Symbol   Float64   Float64   Float64    Float64    Float64   Float64
…

          t0    1.3055    0.0244    0.0018   454.9317   193.9590    1.2169
…
          rp    0.0509    0.0121    0.0009   390.4781   204.4670    1.2722
…
           b    0.2354    0.1386    0.0097   383.4920   190.2810    1.2797
…
                                                                1 column
omitted

Quantiles
  parameters      2.5%     25.0%     50.0%     75.0%     97.5%
      Symbol   Float64   Float64   Float64   Float64   Float64

          t0    1.2120    1.3115    1.3115    1.3116    1.3121
          rp    0.0102    0.0530    0.0531    0.0532    0.0561
           b    0.0002    0.1985    0.2057    0.2136    0.7988
```

```
using StatsPlots
StatsPlots.plot(chain)
```



NestedSamplers (Dynesty-like)

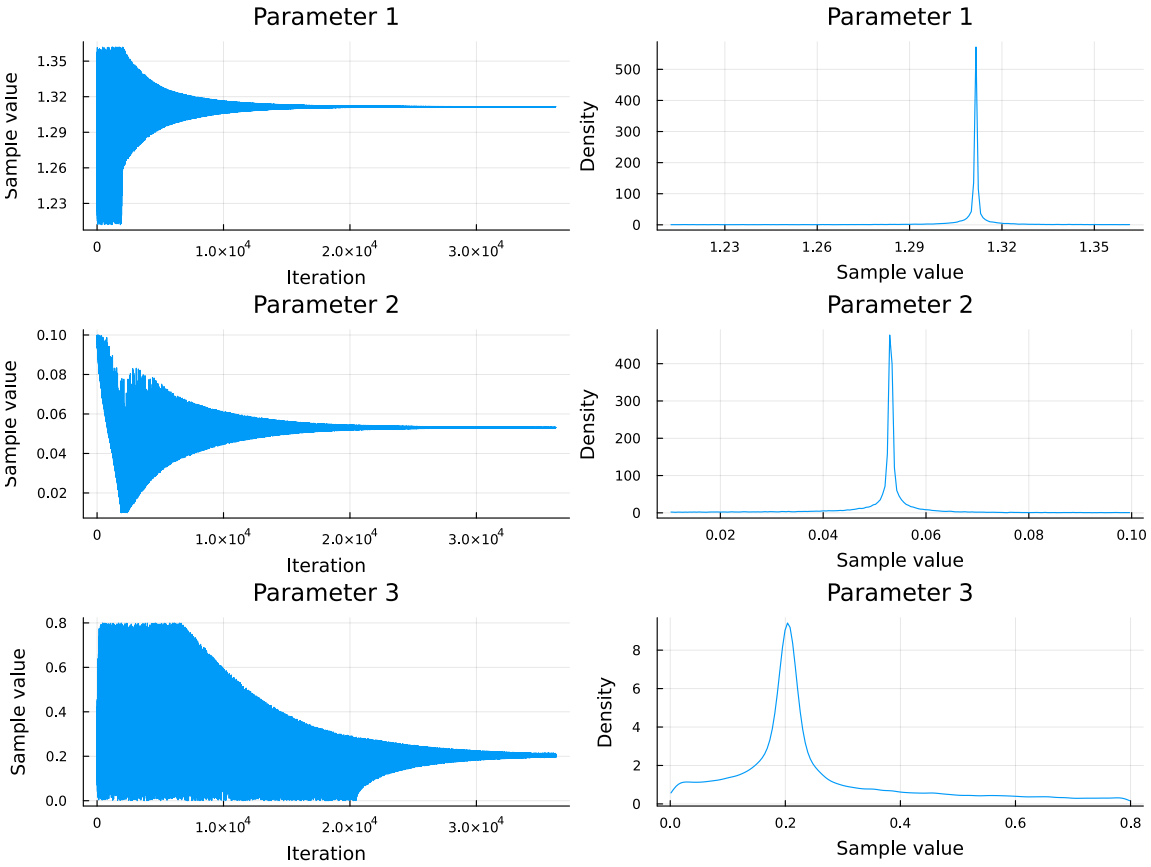
chalk-lab/NestedSamplers.jl: Implementations of single and multi-ellipsoid nested sampling

```
using NestedSamplers

loglike = let m = model
    (X) -> loglikelihood(m, (t0=X[1], rp=X[2], b=X[3]))
end

priors = [Uniform(1.212, 1.362), Uniform(0.01, 0.10), Uniform(0.0, 0.8)]
nmodel = NestedModel(loglike, priors)

sampler = Nested(3, 2000)
chain, state = sample(nmodel, sampler, progress=false)
```

```
(MCMC chain (36277×4×1 Array{Float64, 3}), (it = 36277, us =
[0.6633419982972202 0.6634610489768484 … 0.6636630535044208
0.6635219756683381; 0.4794670093953377 0.48008582122653015 …
```

```
0.47904932981186515 0.4782377822234889; 0.2638313976733687 0.2648857712374485
… 0.26095956561724315 0.2441352507852346], vs = [1.3115012997445832
1.3115191573465272 … 1.3115494580256633 1.3115282963502508; 0.0531520308455804
0.05320772391038772 … 0.05311443968306787 0.05304140040011401;
0.211065118138695 0.2119086169899588 … 0.20876765249379453
0.19530820062818768], logl = 2232.7468670304697, logz = 2217.0932433724383,
logzerr = 0.08609346808799914, logvol = -24.739402459542085, since_update =
924, has_bounds = true, active_bound = MultiEllipsoid{Float64}(ndims=3)))
```

```
StatsPlots.plot(chain)
```

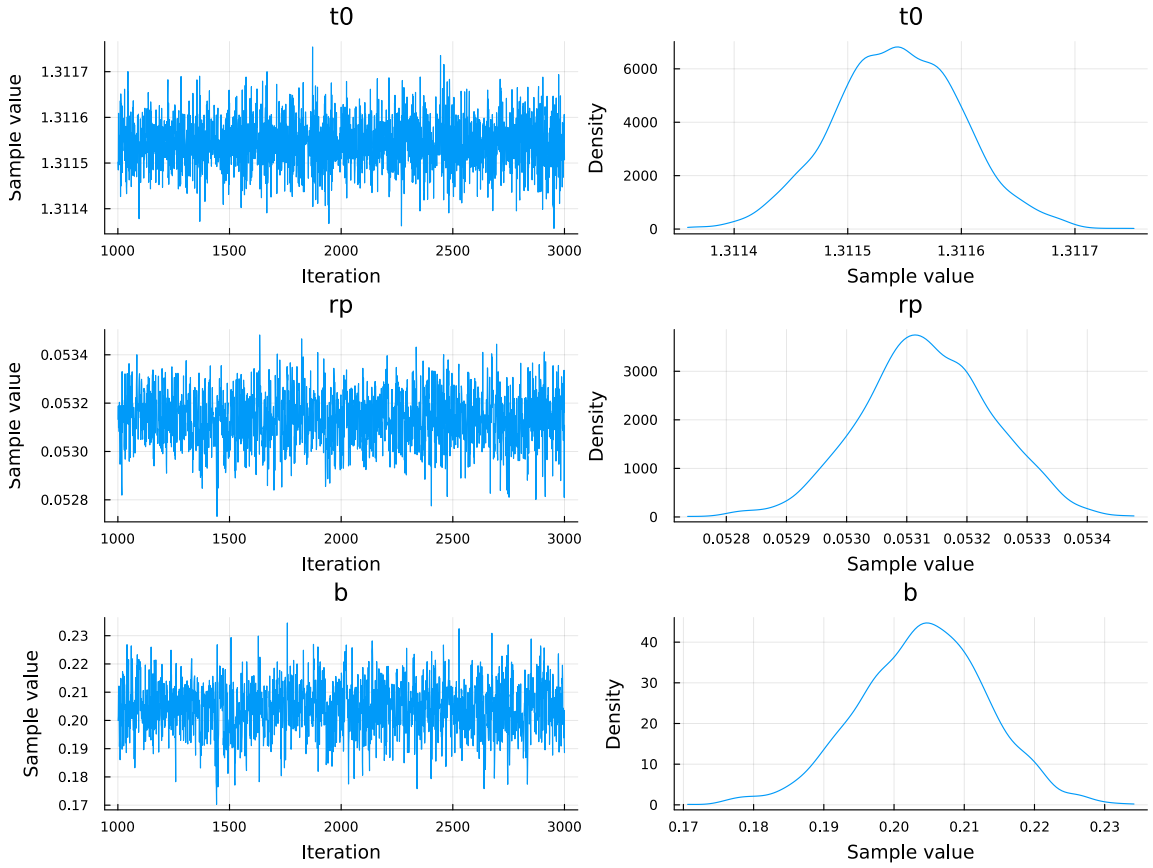


Hamiltonian Monte Carlo (HMC)

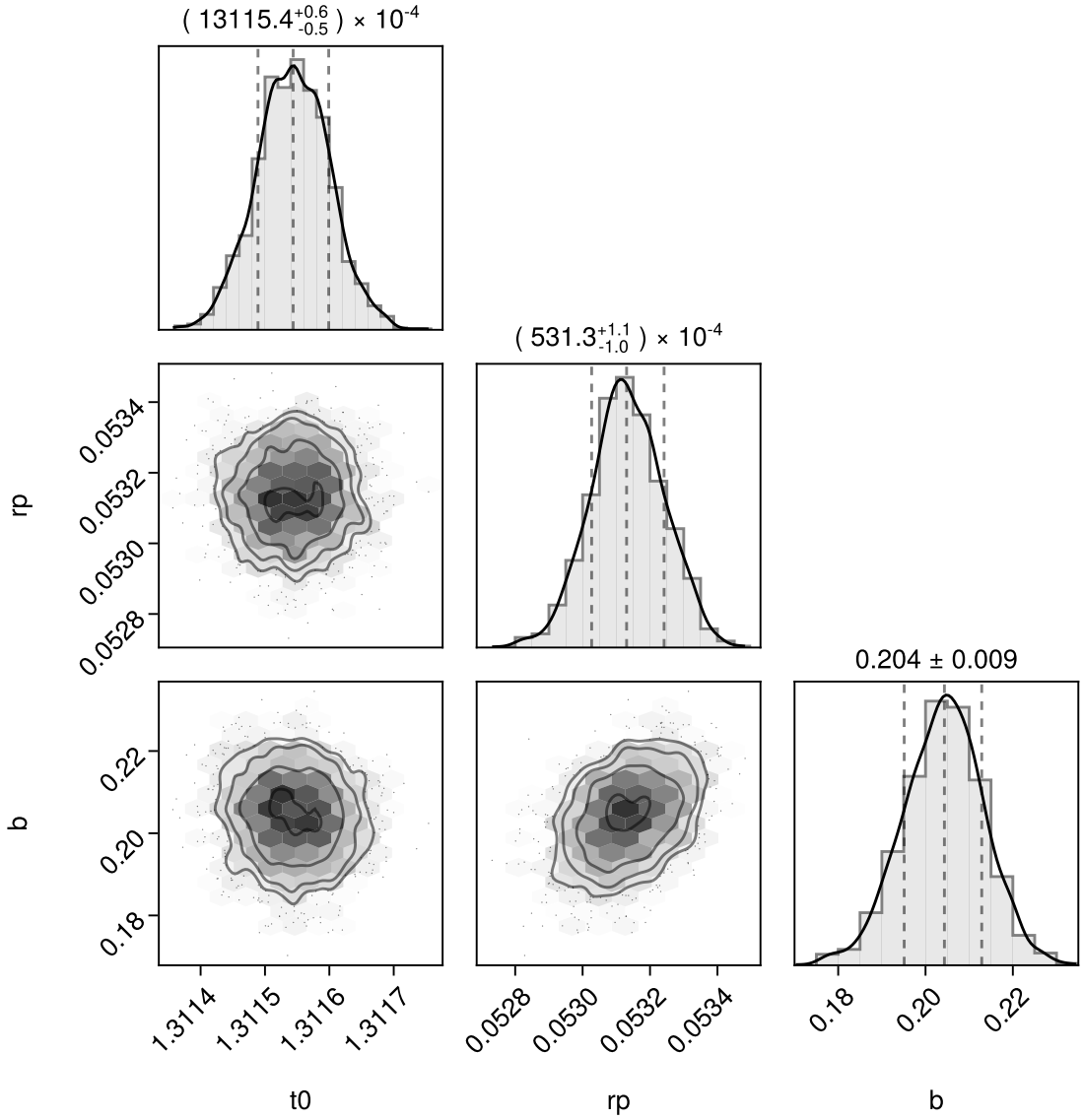
Since now the light curve is auto-differentiated, we can use algorithms like `No U-Turn Sampler (NUTS)` which uses Hamiltonian dynamics (very efficient).

```
chain = sample(model, NUTS(), 2000, progress=false)
StatsPlots.plot(chain)
```

```
┌ Info: Found initial step size
└   ϵ = 0.0015625
```




```
using PairPlots: pairplot
pairplot(chain)
```



Performance comparison

```
using Chairmarks
display(@b sample(model, NUTS(), 2000; progress=false))
display(@b sample(model, Emcee(16), 2000; progress=false))
display(@b sample(nmodel, sampler, progress=false))
```

```
┌ Info: Found initial step size
└   ϵ = 0.00625
```

```
1.387 s (11860669 allocs: 1.100 GiB, 7.82% gc time, without a warmup)
```

```
2.716 s (15638912 allocs: 1.159 GiB, 4.87% gc time, 17.98% compile time,
without a warmup)
```

```
34.764 s (49698346 allocs: 11.488 GiB, 6.78% gc time, without a warmup)
```

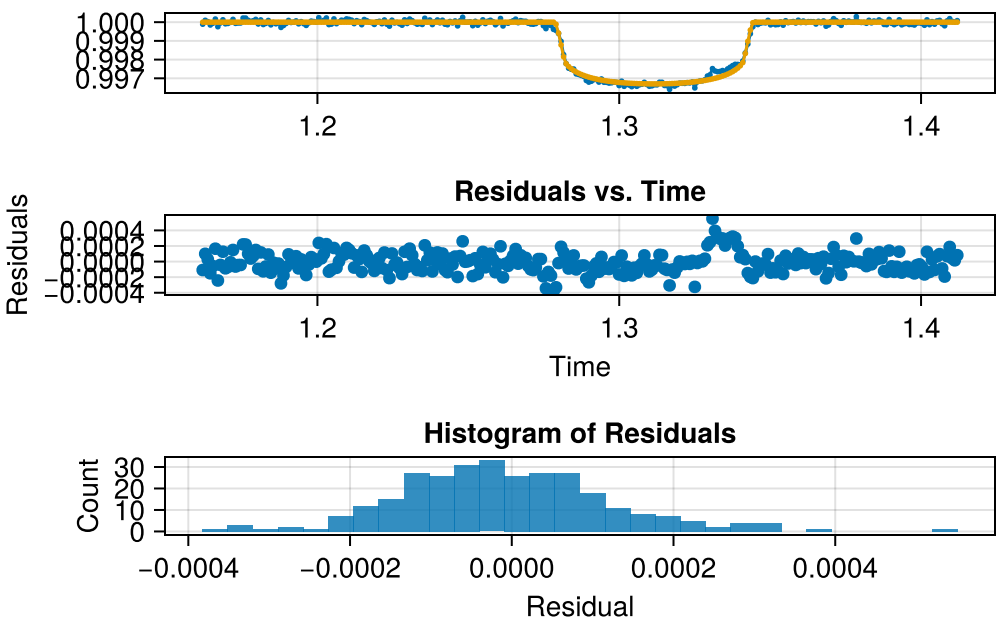
We can see that NUTS is the fastest, followed by Emcee, and then NestedSamplers. Also NUTS seems to converge faster than Emcee and NestedSamplers.

Step 5: Investigate whether you model is a good fit to the data

> Investigate the residuals. What do you think? Quantitatively and graphically assess whether your model is a good fit to the data.

```
using Statistics

t0_fit = median(chain[:t0])
rp_fit = median(chain[:rp])
b_fit = median(chain[:b])
model_flux = light_curve(data.time, t0_fit, b_fit, rp_fit)
residuals = data.flux .- model_flux

f1 = Figure()
markersize = 4
scatterlines(f1[1, 1], data.time, data.flux; markersize)
scatterlines!(f1[1, 1], data.time, model_flux; markersize)

Axis(f1[2, 1], xlabel="Time", ylabel="Residuals", title="Residuals vs. Time")
scatter!(f1[2, 1], data.time, residuals)
Axis(f1[3, 1], xlabel="Residual", ylabel="Count", title="Histogram of
Residuals")
hist!(f1[3, 1], residuals, bins=30)
display(f1)

@info "Mean residual: " mean(residuals)
@info "Std of residuals: " std(residuals)
```

```
┌ Info: Mean residual:
└   mean(residuals) = -8.141298699683164e-6
┌ Info: Std of residuals:
└   std(residuals) = 0.00013008406925382225
```



Step 6: Refine your physical model

1. What could explain your findings in Part 4? Tip: When a distant observer sees a transit of the Earth in front of the sun, would it only be the Earth that is transiting at that time?

There is a increase in the residuals at the time of the 1.33. This could be due to the presence of another object transiting the star. For example, if the system contains both a planet and a moon (or another planet), the observed light curve would show additional dips or asymmetries not explained by a single-object model.

2. Code a new physical "forward" model and log-likelihood function that better explains the data.

```
@model function transit_model_two_bodies(time, flux; error=1e-4)
    # Priors for planet
    t0_1 ~ Uniform(1.212, 1.362)
    rp_1 ~ Uniform(0.01, 0.10)
    b_1 ~ Uniform(0.0, 0.8)
    # Priors for moon (or second planet)
    t0_2 ~ Uniform(1.212, 1.362)
    rp_2 ~ Uniform(0.01, 0.10)
    b_2 ~ Uniform(0.0, 0.8)

    model_flux = light_curve_n(time, ((t0_1, b_1, rp_1), (t0_2, b_2, rp_2)))
```

```
    return flux ~ MvNormal(model_flux, error)
end
```

```
transit_model_two_bodies (generic function with 2 methods)
```

3. Use Dynesty to find the constraints of both objects. Treat the moon like another body transiting shortly before or after. Use the same prior for the moon. That makes six parameters.

For performance reason, we use `NUTS` instead of `NestedSamplers`.

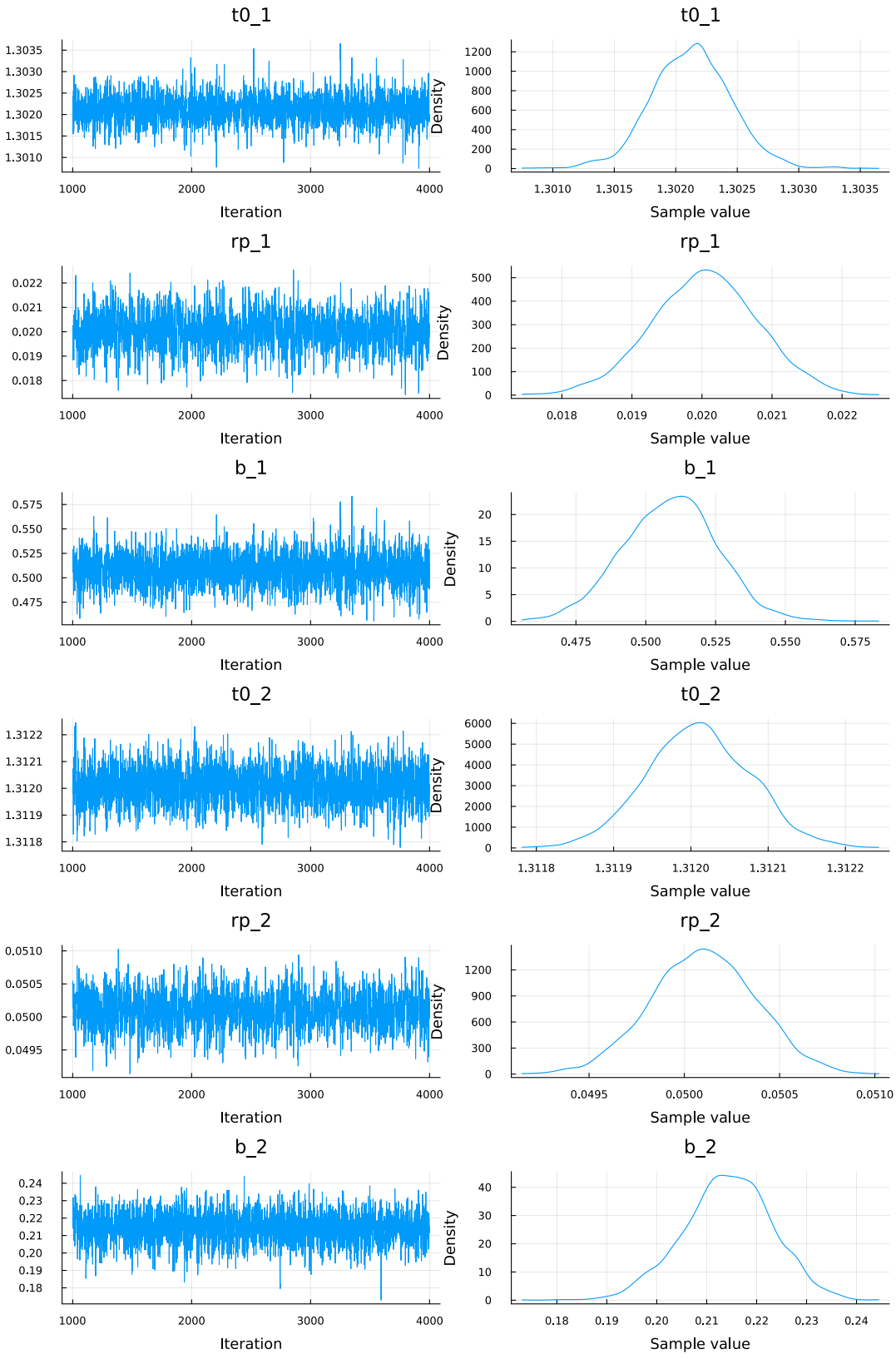
```
model2 = transit_model_two_bodies(data.time, data.flux)
chain2 = sample(model2, NUTS(), 3000, progress=false)
```

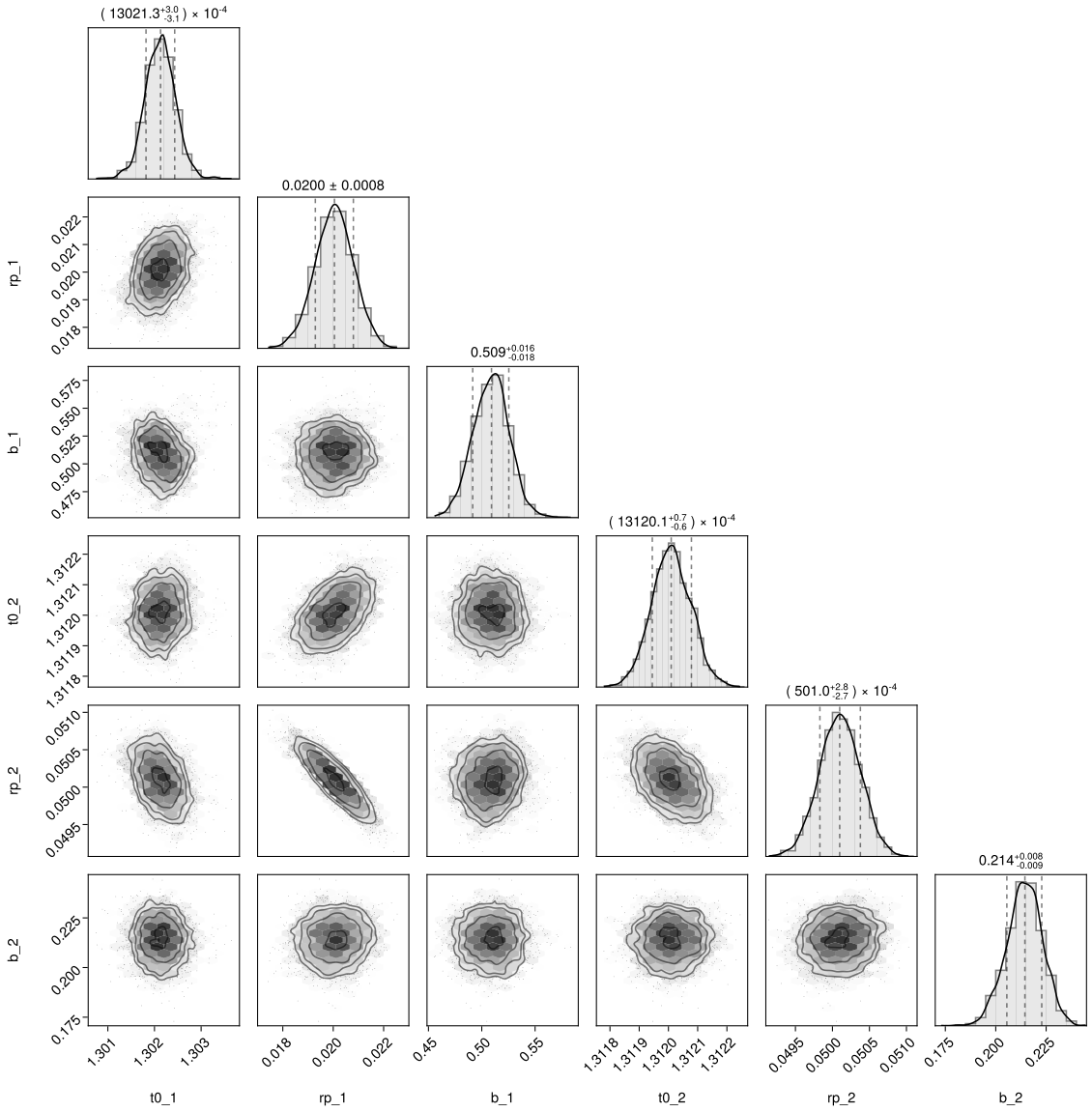
```
┌ Info: Found initial step size
└   ϵ = 0.00625
```

```
Chains MCMC chain (3000×18×1 Array{Float64, 3}):

Iterations        = 1001:1:4000
Number of chains  = 1
Samples per chain = 3000
Wall duration     = 23.63 seconds
Compute duration  = 23.63 seconds
parameters        = t0_1, rp_1, b_1, t0_2, rp_2, b_2
internals         = lp, n_steps, is_accept, acceptance_rate, log_density,
hamiltonian_energy, hamiltonian_energy_error, max_hamiltonian_energy_error,
tree_depth, numerical_error, step_size, nom_step_size

Summary Statistics
  parameters      mean       std      mcse    ess_bulk    ess_tail      rhat
⋯
      Symbol   Float64   Float64   Float64     Float64     Float64   Float64
⋯

        t0_1    1.3021    0.0003    0.0000   2143.2478   1936.2447    0.9998
⋯
        rp_1    0.0200    0.0008    0.0000   1300.3433   1725.2654    1.0005
⋯
         b_1    0.5087    0.0170    0.0003   2650.6522   1673.3757    1.0009
⋯
        t0_2    1.3120    0.0001    0.0000   2488.5882   2421.0591    1.0000
⋯
```

```
        rp_2    0.0501    0.0003    0.0000   1376.5892   1835.7400    1.0007
⋯
         b_2    0.2142    0.0089    0.0002   2677.1743   1705.1781    1.0005
⋯
                                                                  1 column
omitted

Quantiles
  parameters      2.5%     25.0%     50.0%     75.0%     97.5%
      Symbol   Float64   Float64   Float64   Float64   Float64

        t0_1    1.3015    1.3019    1.3021    1.3023    1.3028
        rp_1    0.0185    0.0195    0.0200    0.0206    0.0215
         b_1    0.4747    0.4974    0.5091    0.5200    0.5419
        t0_2    1.3119    1.3120    1.3120    1.3121    1.3121
        rp_2    0.0496    0.0499    0.0501    0.0503    0.0506
         b_2    0.1961    0.2086    0.2145    0.2201    0.2307
```

```
StatsPlots.plot(chain2) |> display
pairplot(chain2)
```


t0_1
Iteration
Density
t0_1
Sample value
rp_1
Iteration
Density
rp_1
Sample value
b_1
Iteration
Density
b_1
Sample value
t0_2
Iteration
Density
t0_2
Sample value
rp_2
Iteration
Density
rp_2
Sample value
b_2
Iteration
Density
b_2
Sample value



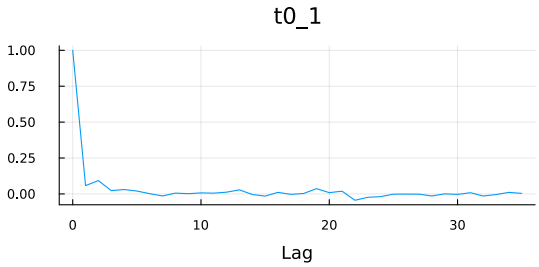
4. Make plots to convince the reviewer that your results are converged

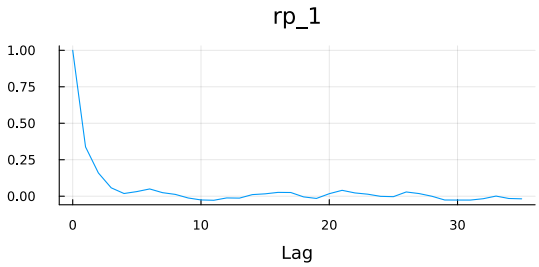
The figure below shows the trace and autocorrelation plots for all parameters. The chains appear well-mixed and stationary, indicating convergence. The Monte Carlo standard error is small further supporting convergence.

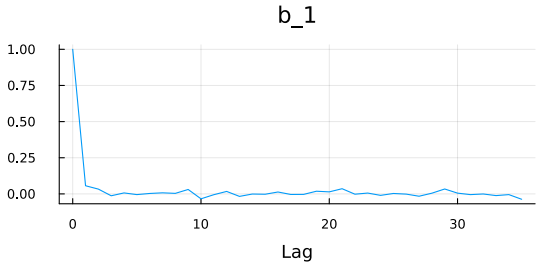
Reference: Autocorrelation analysis & convergence — emcee

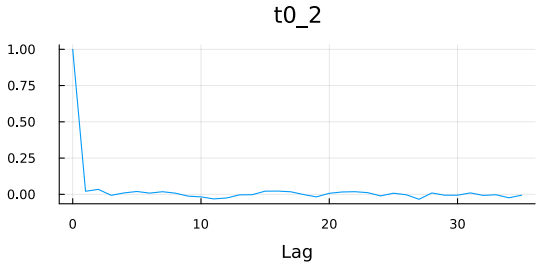
```
@info "Monte Carlo standard error" mcse(chain2)
autocorplot(chain2)
```

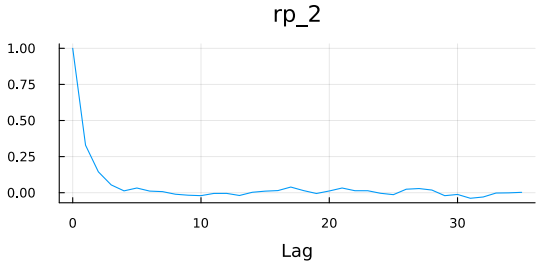
```
┌ Info: Monte Carlo standard error
│   mcse(chain2) =
│    MCSE
│      parameters      mcse
│          Symbol   Float64
│
│           t0_1    0.0000
│           rp_1    0.0000
│            b_1    0.0003
│           t0_2    0.0000
│           rp_2    0.0000
│            b_2    0.0002
└
```

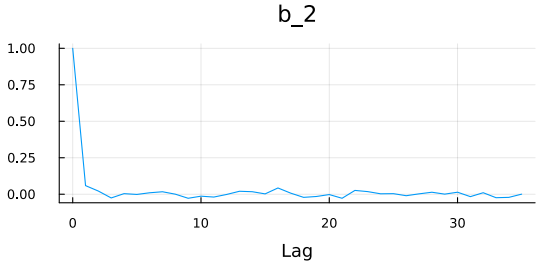

t0_1
1.00
0.75
0.50
0.25
0.00
0
10
20
30
Lag


rp_1
1.00
0.75
0.50
0.25
0.00
0
10
20
30
Lag


b_1
1.00
0.75
0.50
0.25
0.00
0
10
20
30
Lag


t0_2
1.00
0.75
0.50
0.25
0.00
0
10
20
30
Lag


rp_2
1.00
0.75
0.50
0.25
0.00
0
10
20
30
Lag

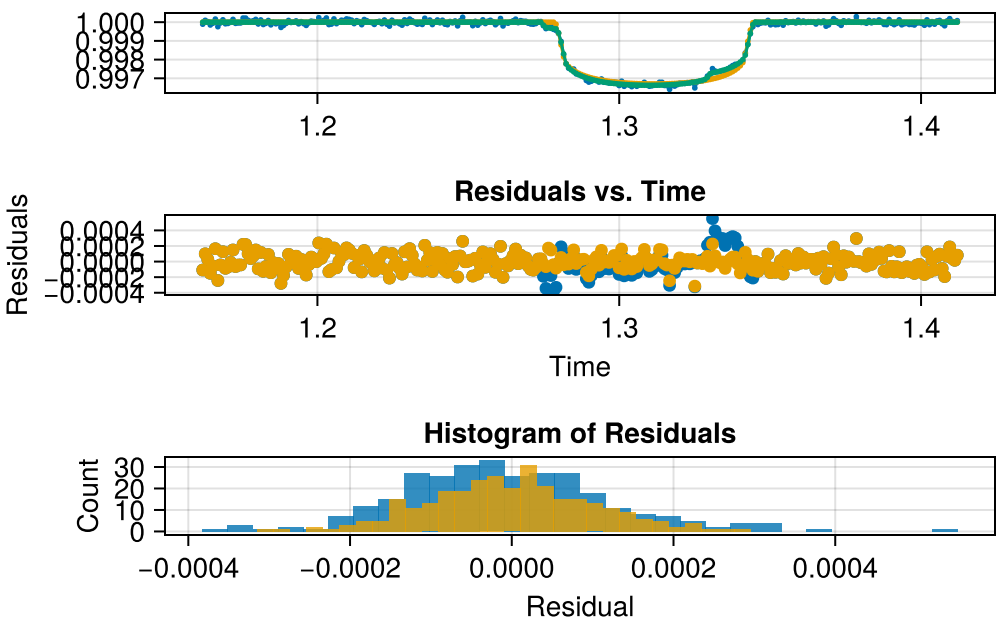

b_2
1.00
0.75
0.50
0.25
0.00
0
10
20
30
Lag

5. Determine quantitatively whether the model justifies applying this more complex model? How confident are you that you detected a moon?

Here we do a posterior predictive checks to compare how well each model reproduces the observed data

```
t0_1 = median(chain2[:t0_1])
rp_1 = median(chain2[:rp_1])
b_1 = median(chain2[:b_1])
t0_2 = median(chain2[:t0_2])
rp_2 = median(chain2[:rp_2])
b_2 = median(chain2[:b_2])
model_flux2 = light_curve_n(data.time, ((t0_1, b_1, rp_1), (t0_2, b_2, rp_2)))
residuals2 = data.flux .- model_flux2

scatterlines!(f1[1, 1], data.time, model_flux2; markersize)
scatter!(f1[2, 1], data.time, residuals2)
hist!(f1[3, 1], residuals2, bins=30)
display(f1)

@info "Mean residual: " mean(residuals2)
@info "Std of residuals: " std(residuals2)
```

```
┌ Info: Mean residual:
└   mean(residuals2) = -3.4637321423132877e-6
┌ Info: Std of residuals:
└   std(residuals2) = 0.00010219966645211133
```



The two-body yields lower residuals than the single-body model, indicating a better fit.

6. Summarize your results for the parameters for each object?

The summary statistics for the two-body model are shown below:

```
chain2
```

```
Chains MCMC chain (3000×18×1 Array{Float64, 3}):

Iterations        = 1001:1:4000
Number of chains  = 1
Samples per chain = 3000
Wall duration     = 23.63 seconds
Compute duration  = 23.63 seconds
parameters        = t0_1, rp_1, b_1, t0_2, rp_2, b_2
internals         = lp, n_steps, is_accept, acceptance_rate, log_density,
hamiltonian_energy, hamiltonian_energy_error, max_hamiltonian_energy_error,
tree_depth, numerical_error, step_size, nom_step_size

Summary Statistics
  parameters      mean       std      mcse    ess_bulk    ess_tail      rhat
⋯
      Symbol   Float64   Float64   Float64     Float64     Float64   Float64
⋯

        t0_1    1.3021    0.0003    0.0000   2143.2478   1936.2447    0.9998
```

```
…
        rp_1    0.0200    0.0008    0.0000   1300.3433   1725.2654    1.0005
…
         b_1    0.5087    0.0170    0.0003   2650.6522   1673.3757    1.0009
…
        t0_2    1.3120    0.0001    0.0000   2488.5882   2421.0591    1.0000
…
        rp_2    0.0501    0.0003    0.0000   1376.5892   1835.7400    1.0007
…
         b_2    0.2142    0.0089    0.0002   2677.1743   1705.1781    1.0005
…
                                                                1 column
omitted

Quantiles
  parameters      2.5%     25.0%     50.0%     75.0%     97.5%
      Symbol   Float64   Float64   Float64   Float64   Float64

        t0_1    1.3015    1.3019    1.3021    1.3023    1.3028
        rp_1    0.0185    0.0195    0.0200    0.0206    0.0215
         b_1    0.4747    0.4974    0.5091    0.5200    0.5419
        t0_2    1.3119    1.3120    1.3120    1.3121    1.3121
        rp_2    0.0496    0.0499    0.0501    0.0503    0.0506
         b_2    0.1961    0.2086    0.2145    0.2201    0.2307
```

7. Why do you get a multi-modal solution? What could you do to avoid this in this simple example?

It seems that using `NUTS` sampler with only one chain could avoid the multi-modal solution. Multi-modal arises when there are multiple chains and the modes switch between chains.

> This is because it's possible for either model parameter to be assigned to either of the corresponding true means, and this assignment need not be consistent between chains. That is, the posterior is fundamentally multimodal, and different chains can end up in different modes, complicating inference. One solution here is to enforce an ordering on our parameters, requiring for all. `Bijectors.jl` provides an easy transformation (ordered()) for this purpose

Another solution is to post-process the chain to exchange parameters to enforce ordering.

Step 7:

> Describe in a few sentences one example for a problem or data set that you could analogously solve in your research domain?

In plasma physics, particle velocity distributions are often non-Maxwellian and may be composed of several populations (core, halo, beam). By modeling the observed distribution as the sum of

several Maxwellian or kappa distributions, and using Bayesian inference to fit the parameters (density, temperature, drift speed for each component), you can determine how many components are justified and estimate their properties, just as you determine the number of transiting bodies in the exoplanet problem.

Old codes (using `Batman`)

Julia interface to Python and Batman

```
#| output: false
using CondaPkg
using PythonCall
import PythonCall: Py
pkg = "batman-package"
deps = CondaPkg.read_deps()["deps"]
haskey(deps, pkg) || CondaPkg.add(pkg)
# CondaPkg.add("corner")

const batman = pyimport("batman")
const corner = pyimport("corner")
const np = pyimport("numpy")
```

```
using DrWatson: @dict

function set_params!(params; kwargs...)
    for (key, value) in kwargs
        setproperty!(params, key, value)
    end
    return params
end

"""Like `batman.TransitParams` in Python, but with some keyword arguments for
convenience."""
function TransitParams(; ecc=0.0, w=0.0, limb_dark="quadratic", kwargs...)
    params = batman.TransitParams()
    set_params!(params; ecc, w, limb_dark)
    set_params!(params; kwargs...)
    return params
end

TransitModel(params, time) = batman.TransitModel(params, Py(time).to_numpy())

light_curve(model::Py, params::Py) = PyArray(model.light_curve(params))
function light_curve(params, time)
    m = TransitModel(params, time)
    return light_curve(m, params)
```

```
end
light_curve(time; kwargs...) = light_curve(TransitParams(; kwargs...), time)
```

```
inc(a, b) = acosd(b / a)
inc_i = acosd(b_i / a_i)

flux = let rp = rp_i, t0 = t0_i, b = b_i, a = a_i, u = u_i, per = per_i
    inc = acosd(b / a)
    light_curve(data.time; t0, per, rp, a, inc, u)
end
```

Emcee sampler

```
inc(p::Py, b) = inc(pyconvert(Any, p.a), b)

@model function transit_model(time, flux, params,
trans_model=TransitModel(params, data.time); error=1e-4)
    # Priors
    t0 ~ Uniform(1.212, 1.362)
    rp ~ Uniform(0.01, 0.10)
    b ~ Uniform(0.0, 0.99)
    params = set_params!(params; t0, rp, inc=inc(params, b))
    model_flux = light_curve(trans_model, params)
    return flux ~ MvNormal(model_flux, error)
end

params = TransitParams(; rp=rp_i, t0=t0_i, per=3.0, a=a_i, u=u_i, inc=inc_i)
trans_model = TransitModel(params, data.time)
model = transit_model(data.time, data.flux, params)
chain = sample(model, Emcee(100), 10000; progress=false)
```

Reference

- DACE tutorial on photometry
- Dynamic Hamiltonian Monte Carlo (HMC)
  - tpapp/DynamicHMC.jl: Implementation of robust dynamic Hamiltonian Monte Carlo methods (NUTS) in Julia

Bibliography