Bayesian Inference on a Pendulum using Turing.jl

Vaibhav Dixit

June 21, 2019

0.0.1 Set up simple pendulum problem

```
using DiffEqBayes, OrdinaryDiffEq, RecursiveArrayTools, Distributions, Plots, StatsPlots
```

Let's define our simple pendulum problem. Here our pendulum has a drag term ω and a length `L`. We get first order equations by defining the first term as the velocity and the second term as the position, getting:

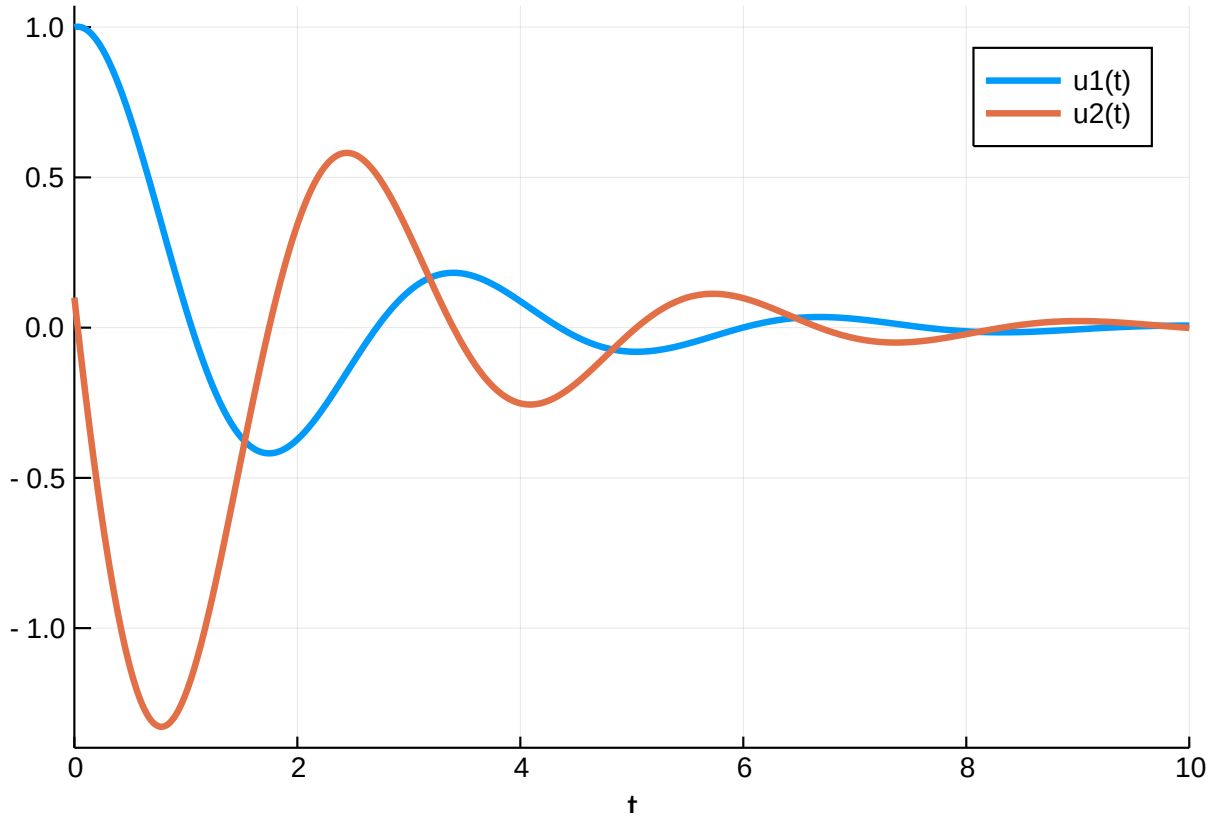
```
function pendulum(du,u,p,t)
    ω,L = p
    x,y = u
    du[1] = y
    du[2] = - ω*y -(9.8/L)*sin(x)
end

u0 = [1.0,0.1]
tspan = (0.0,10.0)
prob1 = ODEProblem(pendulum,u0,tspan,[1.0,2.5])
```

```
ODEProblem with uType Array{Float64,1} and tType Float64. In-place: true
timespan: (0.0, 10.0)
u0: [1.0, 0.1]
```

0.0.2 Solve the model and plot

To understand the model and generate data, let's solve and visualize the solution with the known parameters:

```
sol = solve(prob1,Tsit5())
plot(sol)
```



It's the pendulum, so you know what it looks like. It's periodic, but since we have not made a small angle assumption it's not exactly `sin` or `cos`. Because the true dampening parameter ω is 1, the solution does not decay over time, nor does it increase. The length `L` determines the period.

0.0.3 Create some dummy data to use for estimation

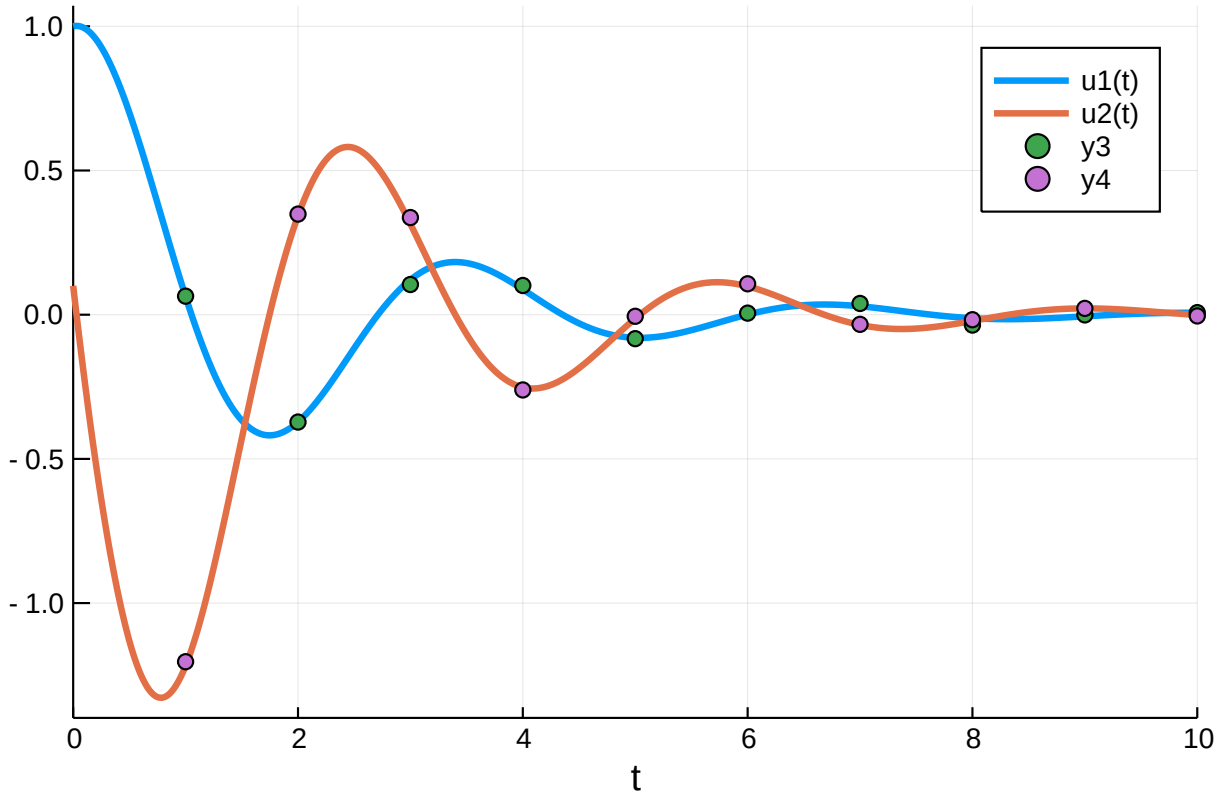
We now generate some dummy data to use for estimation

```
t = collect(range(1,stop=10,length=10))
randomized = VectorOfArray([(sol(t[i]) + .01randn(2)) for i in 1:length(t)])
data = convert(Array,randomized)
```

```
2×10 Array{Float64,2}:
  0.0642482  -0.372244  0.104671   0.101319  …  -0.000707668   0.00716293
 -1.20335     0.348918  0.336912  -0.260934      0.021927     -0.00429795
```

Let's see what our data looks like on top of the real solution

```
scatter!(data')
```



This data captures the non-dampening effect and the true period, making it perfect to attempting a Bayesian inference.

0.0.4 Perform Bayesian Estimation

Now let's fit the pendulum to the data. Since we know our model is correct, this should give us back the parameters that we used to generate the data! Define priors on our parameters. In this case, let's assume we don't have much information, but have a prior belief that ω is between 0.1 and 3.0, while the length of the pendulum L is probably around 3.0:

```
priors = [Uniform(0.1,3.0), Normal(3.0,1.0)]
```

```
2-element Array{Distribution{Univariate,Continuous},1}:
 Uniform{Float64}(a=0.1, b=3.0)
 Normal{Float64}(μ=3.0, σ=1.0)
```

Finally let's run the estimation routine from DiffEqBayes.jl using the Turing.jl backend

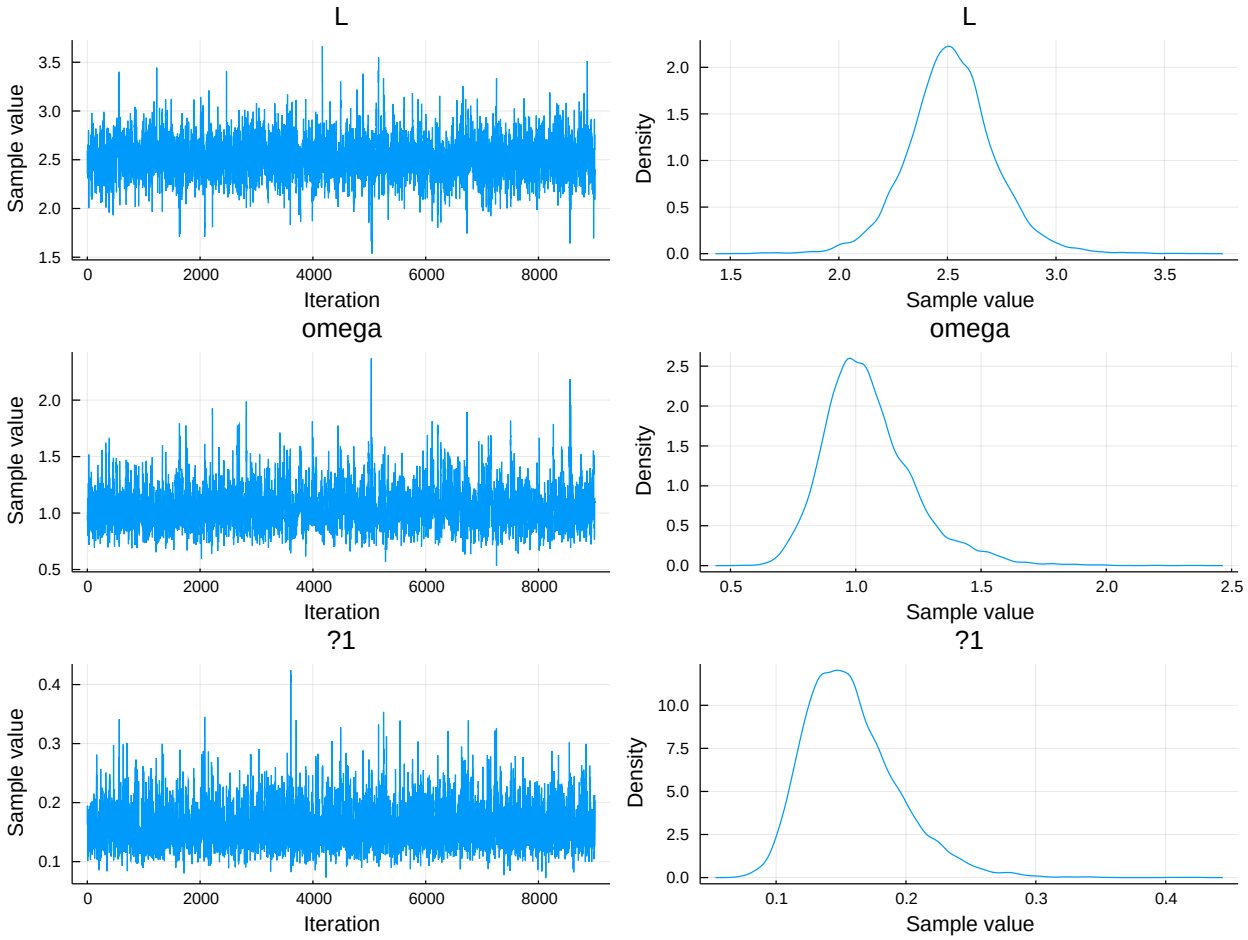
```
bayesian_result = turing_inference(prob1,Tsit5(),t,data,priors;num_samples=10_000,
                                   syms = [:omega,:L])
```

```
Object of type Chains, with data of type 9000×6×1 Array{Union{Missing, Floa
t64},3}
```

```
Log evidence      = 0.0
Iterations        = 1:9000
Thinning interval = 1
Chains            = 1
Samples per chain = 9000
internals         = eval_num, lf_eps, lp
parameters        = omega, σ1, L

2-element Array{MCMCChains.ChainDataFrame,1}

Summary Statistics
. Omitted printing of 1 columns
 Row  parameters  mean      std        naive_se     mcse        ess

      Symbol      Float64   Float64    Float64      Float64     Any



 1    L           2.51792   0.202716   0.00213681   0.00508037  1542
.58
 2    omega       1.05421   0.183051   0.00192953   0.00563315  1221
.26
 3    σ1          0.158244  0.0362432  0.000382037  0.00075894  2170
.85

Quantiles

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

 1    L           2.12357   2.39427   2.51561   2.63599   2.93412
 2    omega       0.765063  0.930827  1.02915   1.14897   1.50302
 3    σ1          0.102598  0.132802  0.153324  0.178261  0.242048
```

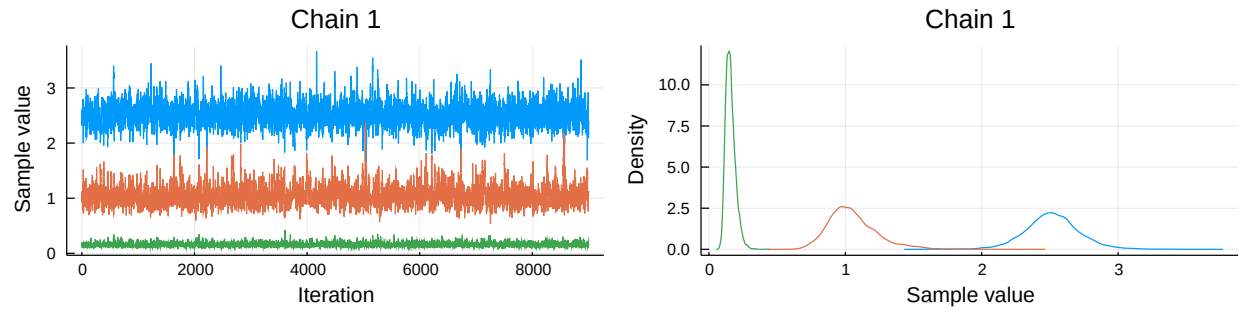
Notice that while our guesses had the wrong means, the learned parameters converged to the correct means, meaning that it learned good posterior distributions for the parameters. To look at these posterior distributions on the parameters, we can examine the chains:

```
plot(bayesian_result)
```



As a diagnostic, we will also check the parameter chains. The chain is the MCMC sampling process. The chain should explore parameter space and converge reasonably well, and we should be taking a lot of samples after it converges (it is these samples that form the posterior distribution!)

```
plot(bayesian_result, colordim = :parameter)
```



Notice that after awhile these chains converge to a "fuzzy line", meaning it found the area with the most likelihood and then starts to sample around there, which builds a posterior distribution around the true mean.