Monte Carlo Parameter Estimation From Data

Chris Rackauckas

July 4, 2020

First you want to create a problem which solves multiple problems at the same time. This is the Monte Carlo Problem. When the parameter estimation tools say it will take any DEProblem, it really means ANY DEProblem!

So, let's get a Monte Carlo problem setup that solves with 10 different initial conditions.

```
using DifferentialEquations, DiffEqParamEstim, Plots, Optim
```

```
Error: Failed to precompile DiffEqParamEstim [1130ab10-4a5a-5621-a13d-e4788
d82bd4c] to /builds/JuliaGPU/DiffEqTutorials.jl/.julia/compiled/v1.4/DiffEq
ParamEstim/nWq0E_XzcJh.ji.
```

```
# Monte Carlo Problem Set Up for solving set of ODEs with different initial conditions

# Set up Lotka-Volterra system
function pf_func(du,u,p,t)
  du[1] = p[1] * u[1] - p[2] * u[1]*u[2]
  du[2] = -3 * u[2] + u[1]*u[2]
end
p = [1.5,1.0]
prob = ODEProblem(pf_func,[1.0,1.0],(0.0,10.0),p)
```

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

Now for a MonteCarloProblem we have to take this problem and tell it what to do N times via the prob_func. So let's generate N=10 different initial conditions, and tell it to run the same problem but with these 10 different initial conditions each time:

```
# Setting up to solve the problem N times (for the N different initial conditions)
N = 10;
initial_conditions = [[1.0,1.0], [1.0,1.5], [1.5,1.0], [1.5,1.5], [0.5,1.0], [1.0,0.5],
[0.5,0.5], [2.0,1.0], [1.0,2.0], [2.0,2.0]]
function prob_func(prob,i,repeat)
  ODEProblem(prob.f,initial_conditions[i],prob.tspan,prob.p)
end
monte_prob = MonteCarloProblem(prob,prob_func=prob_func)
```

```
EnsembleProblem with problem ODEProblem
```

We can check this does what we want by solving it:

```
# Check above does what we want
sim = solve(monte_prob,Tsit5(),num_monte=N)
plot(sim)
```

```
Error: UndefVarError: plot not defined
```

num*monte=N means "run N times", and each time it runs the problem returned by the prob*func, which is always the same problem but with the ith initial condition.

Now let's generate a dataset from that. Let's get data points at every t=0.1 using saveat, and then convert the solution into an array.

```
# Generate a dataset from these runs
data_times = 0.0:0.1:10.0
sim = solve(monte_prob,Tsit5(),num_monte=N,saveat=data_times)
data = Array(sim)
```

```
2×101×10 Array{Float64,3}:
[:, :, 1] =
 1.0  1.06108   1.14403   1.24917   1.37764   …  0.956979  0.983561  1.0337
6
 1.0  0.821084  0.679053  0.566893  0.478813     1.35559   1.10629   0.9063
7

[:, :, 2] =
 1.0  1.01413  1.05394  1.11711   …  1.05324  1.01309  1.00811  1.03162
 1.5  1.22868  1.00919  0.833191     2.08023  1.70818  1.39972  1.14802

[:, :, 3] =
 1.5  1.58801   1.70188   1.84193   2.00901   …  2.0153    2.21084   2.4358
9
 1.0  0.864317  0.754624  0.667265  0.599149     0.600943  0.549793  0.5136
8

[:, :, 4] =
 1.5  1.51612  1.5621   1.63555   1.73531   …  1.83822   1.98545   2.15958
 1.5  1.29176  1.11592  0.969809  0.850159     0.771088  0.691421  0.630025

[:, :, 5] =
 0.5  0.531705  0.576474  0.634384  0.706139  …  9.05366   9.4006    8.8391
 1.0  0.77995   0.610654  0.480565  0.380645     0.809383  1.51708  2.82619

[:, :, 6] =
 1.0  1.11027   1.24238   1.39866   1.58195   …  0.753107  0.748814  0.7682
84
 0.5  0.411557  0.342883  0.289812  0.249142     1.73879   1.38829   1.1093
2

[:, :, 7] =
 0.5  0.555757  0.623692  0.705084  0.80158   …  8.11213   9.10669   9.9216
9
 0.5  0.390449  0.30679   0.24286   0.193966     0.261294  0.455928  0.8787
92

[:, :, 8] =
 2.0  2.11239   2.24921   2.41003   2.59433   …  3.22292   3.47356   3.7301
1
 1.0  0.909749  0.838025  0.783532  0.745339     0.739406  0.765524  0.8130
04

[:, :, 9] =
 1.0  0.969326  0.971358  1.00017  …  1.25065  1.1012   1.01733  0.979304
 2.0  1.63445   1.33389   1.09031     3.02672  2.52063  2.07503  1.69808
```

```
[:, :, 10] =
 2.0  1.92148  1.88215  1.87711  1.90264  ...  2.15079  2.27937   2.43105
 2.0  1.80195  1.61405  1.4426   1.2907      0.95722  0.884825  0.829478
```

Here, data[i,j,k] is the same as sim[i,j,k] which is the same as sim[k]i,j. So data[i,j,k] is the jth timepoint of the ith variable in the kth trajectory.

Now let's build a loss function. A loss function is some loss(sol) that spits out a scalar for how far from optimal we are. In the documentation I show that we normally do loss = L2Loss(t,data), but we can bootstrap off of this. Instead lets build an array of N loss functions, each one with the correct piece of data.

```
# Building a loss function
losses = [L2Loss(data_times,data[:,:,i]) for i in 1:N]
```

```
Error: UndefVarError: L2Loss not defined
```

So losses[i] is a function which computes the loss of a solution against the data of the ith trajectory. So to build our true loss function, we sum the losses:

```
loss(sim) = sum(losses[i](sim[i]) for i in 1:N)
```

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

As a double check, make sure that loss(sim) outputs zero (since we generated the data from sim). Now we generate data with other parameters:

```
prob = ODEProblem(pf_func,[1.0,1.0],(0.0,10.0),[1.2,0.8])
function prob_func(prob,i,repeat)
  ODEProblem(prob.f,initial_conditions[i],prob.tspan,prob.p)
end
monte_prob = MonteCarloProblem(prob,prob_func=prob_func)
sim = solve(monte_prob,Tsit5(),num_monte=N,saveat=data_times)
loss(sim)
```

```
Error: UndefVarError: losses not defined
```

and get a non-zero loss. So we now have our problem, our data, and our loss function... we have what we need.

Put this into build*loss*objective.

```
obj = build_loss_objective(monte_prob,Tsit5(),loss,num_monte=N,
                           saveat=data_times)
```

```
Error: UndefVarError: build_loss_objective not defined
```

Notice that I added the kwargs for solve into this. They get passed to an internal solve command, so then the loss is computed on N trajectories at data_times.

Thus we take this objective function over to any optimization package. I like to do quick things in Optim.jl. Here, since the Lotka-Volterra equation requires positive parameters, I use Fminbox to make sure the parameters stay positive. I start the optimization with [1.3,0.9], and Optim spits out that the true parameters are:

```
lower = zeros(2)
upper = fill(2.0,2)
result = optimize(obj, lower, upper, [1.3,0.9], Fminbox(BFGS()))
```

```
Error: UndefVarError: BFGS not defined
```

```
result
```

```
Error: UndefVarError: result not defined
```

Optim finds one but not the other parameter.

I would run a test on synthetic data for your problem before using it on real data. Maybe play around with different optimization packages, or add regularization. You may also want to decrease the tolerance of the ODE solvers via

```
obj = build_loss_objective(monte_prob,Tsit5(),loss,num_monte=N,
                           abstol=1e-8,reltol=1e-8,
                           saveat=data_times)
```

```
Error: UndefVarError: build_loss_objective not defined
```

```
result = optimize(obj, lower, upper, [1.3,0.9], Fminbox(BFGS()))
```

```
Error: UndefVarError: BFGS not defined
```

```
result
```

```
Error: UndefVarError: result not defined
```

if you suspect error is the problem. However, if you're having problems it's most likely not the ODE solver tolerance and mostly because parameter inference is a very hard optimization problem.

0.1 Appendix

This tutorial is part of the DiffEqTutorials.jl repository, found at: https://github.com/JuliaDiffEq/DiffEq

To locally run this tutorial, do the following commands:

```
using DiffEqTutorials
DiffEqTutorials.weave_file("model_inference","02-monte_carlo_parameter_estim.jmd")
```

Computer Information:

```
Julia Version 1.4.2
Commit 44fa15b150* (2020-05-23 18:35 UTC)
Platform Info:
  OS: Linux (x86_64-pc-linux-gnu)
  CPU: Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-8.0.1 (ORCJIT, skylake)
Environment:
  JULIA_DEPOT_PATH = /builds/JuliaGPU/DiffEqTutorials.jl/.julia
  JULIA_CUDA_MEMORY_LIMIT = 2147483648
  JULIA_PROJECT = @.
  JULIA_NUM_THREADS = 4
```

Package Information:

```
Status `/builds/JuliaGPU/DiffEqTutorials.jl/tutorials/model_inference/Project.toml`
[6e4b80f9-dd63-53aa-95a3-0cdb28fa8baf] BenchmarkTools 0.5.0
[593b3428-ca2f-500c-ae53-031589ec8ddd] CmdStan 6.0.6
[ebbdde9d-f333-5424-9be2-dbf1e9acfb5e] DiffEqBayes 2.16.0
[1130ab10-4a5a-5621-a13d-e4788d82bd4c] DiffEqParamEstim 1.15.0
[0c46a032-eb83-5123-abaf-570d42b7fbaa] DifferentialEquations 6.15.0
[31c24e10-a181-5473-b8eb-7969acd0382f] Distributions 0.23.4
[bbc10e6e-7c05-544b-b16e-64fede858acb] DynamicHMC 2.1.0
[429524aa-4258-5aef-a3af-852621145aeb] Optim 0.22.0
[1dea7af3-3e70-54e6-95c3-0bf5283fa5ed] OrdinaryDiffEq 5.41.0
[91a5bcdd-55d7-5caf-9e0b-520d859cae80] Plots 1.5.2
[731186ca-8d62-57ce-b412-fbd966d074cd] RecursiveArrayTools 2.5.0
[f3b207a7-027a-5e70-b257-86293d7955fd] StatsPlots 0.14.6
[84d833dd-6860-57f9-a1a7-6da5db126cff] TransformVariables 0.3.9
```