Monte Carlo Parameter Estimation From Data

Chris Rackauckas

February 26, 2019

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.

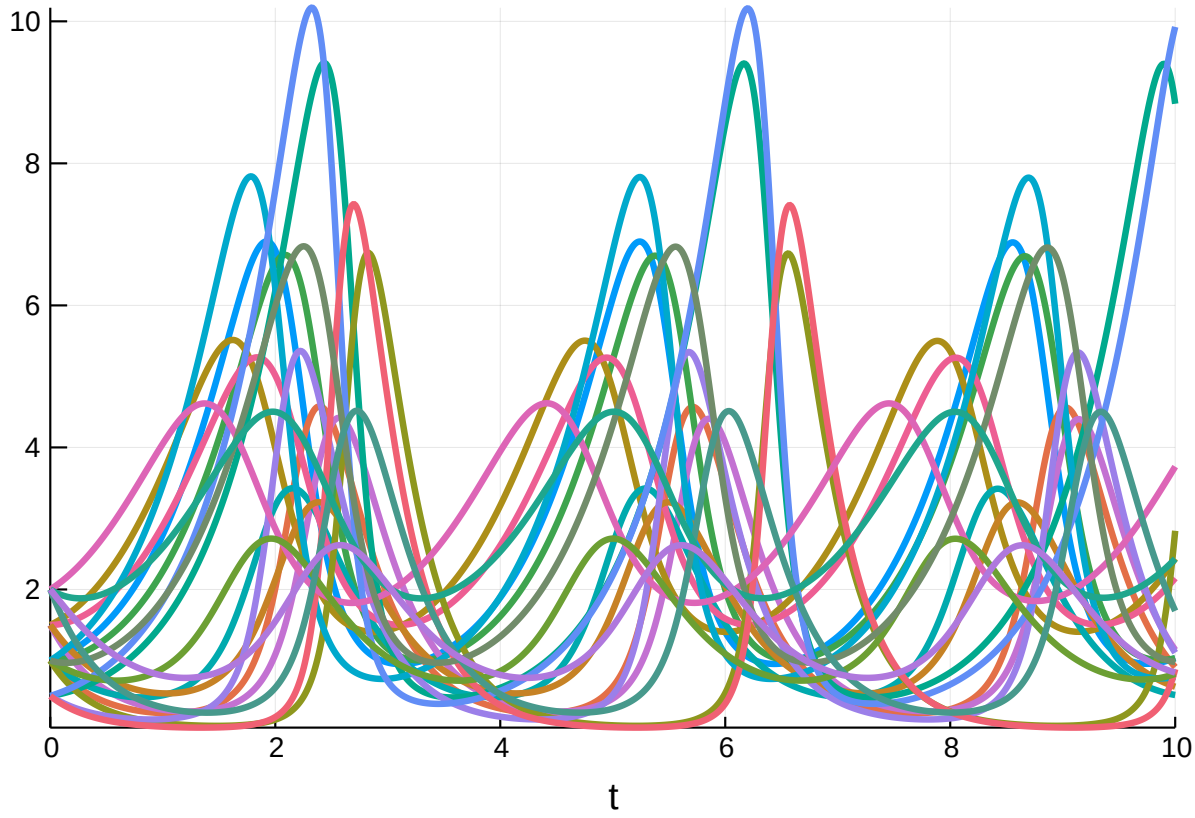
```
using DifferentialEquations, DiffEqParamEstim, Plots, Optim

# 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)
```

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)
```

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)
```



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)
```

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]
```

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)
```

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)
```

```
10108.695792044306
```

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)
```

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()))
```

```
result
```

```
Results of Optimization Algorithm
 * Algorithm: Fminbox with BFGS
 * Starting Point: [1.3,0.9]
 * Minimizer: [1.5000000000573428,1.0000000001610496]
 * Minimum: 7.028970e-16
 * Iterations: 4
 * Convergence: true
   * |x - x'| ≤ 0.0e+00: true
     |x - x'| = 0.00e+00
   * |f(x) - f(x')| ≤ 0.0e+00 |f(x)|: true
     |f(x) - f(x')| = 0.00e+00 |f(x)|
   * |g(x)| ≤ 1.0e-08: false
     |g(x)| = 1.06e-06
   * Stopped by an increasing objective: true
   * Reached Maximum Number of Iterations: false
```

```
 * Objective Calls: 195
 * Gradient Calls: 195
```

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

```
result
```

```
Results of Optimization Algorithm
 * Algorithm: Fminbox with BFGS
 * Starting Point: [1.3,0.9]
 * Minimizer: [1.5007434761923657,1.001238477498098]
 * Minimum: 4.163900e-02
 * Iterations: 5
 * Convergence: true
   * |x - x'| ≤ 0.0e+00: true
     |x - x'| = 0.00e+00
   * |f(x) - f(x')| ≤ 0.0e+00 |f(x)|: true
     |f(x) - f(x')| = 0.00e+00 |f(x)|
   * |g(x)| ≤ 1.0e-08: false
     |g(x)| = 1.66e-06
   * Stopped by an increasing objective: true
   * Reached Maximum Number of Iterations: false
 * Objective Calls: 227
 * Gradient Calls: 227
```

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

```
using DiffEqTutorials
DiffEqTutorials.tutorial_footer(WEAVE_ARGS[:folder],WEAVE_ARGS[:file])
```

```
These benchmarks are part of the DiffEqTutorials.jl repository, found at:

https://github.com/JuliaDiffEq/DiffEqTutorials.jl

To locally run this tutorial, do the following commands:

using DiffEqTutorials
```

```
DiffEqTutorials.weave_file("ode_extras","monte_carlo_parameter_estim.jmd")

Computer Information:

Julia Version 1.1.0
Commit 80516ca202 (2019-01-21 21:24 UTC)
Platform Info:
  OS: Windows (x86_64-w64-mingw32)
  CPU: Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-6.0.1 (ORCJIT, skylake)
Environment:
  JULIA_EDITOR = "C:\Users\accou\AppData\Local\atom\app-1.34.0\atom.exe" -a
  JULIA_NUM_THREADS = 6

Package Information:

    Status `C:\Users\accou\.julia\environments\v1.1\Project.toml`
  [7e558dbc] ArbNumerics v0.3.6
  [c52e3926] Atom v0.7.14
  [6e4b80f9] BenchmarkTools v0.4.2
  [336ed68f] CSV v0.4.3
  [3895d2a7] CUDAapi v0.5.4
  [be33ccc6] CUDAnative v1.0.1
  [3a865a2d] CuArrays v0.9.1
  [a93c6f00] DataFrames v0.17.1
  [55939f99] DecFP v0.4.8
  [abce61dc] Decimals v0.4.0
  [39dd38d3] Dierckx v0.4.1
  [bb2cbb15] DiffEqBenchmarks v0.0.0 [`C:\Users\accou\.julia\external\DiffE
qBenchmarks.jl`]
  [459566f4] DiffEqCallbacks v2.5.2
  [f3b72e0c] DiffEqDevTools v2.6.1
  [aae7a2af] DiffEqFlux v0.2.0
  [c894b116] DiffEqJump v6.1.0+ [`C:\Users\accou\.julia\dev\DiffEqJump`]
  [1130ab10] DiffEqParamEstim v1.6.0+ [`C:\Users\accou\.julia\dev\DiffEqPar
amEstim`]
  [055956cb] DiffEqPhysics v3.1.0
  [a077e3f3] DiffEqProblemLibrary v4.1.0
  [225cb15b] DiffEqTutorials v0.0.0 [`C:\Users\accou\.julia\external\DiffEq
Tutorials.jl`]
  [0c46a032] DifferentialEquations v6.3.0
  [497a8b3b] DoubleFloats v0.7.5
  [587475ba] Flux v0.7.3
  [f6369f11] ForwardDiff v0.10.3+ [`C:\Users\accou\.julia\dev\ForwardDiff`]
  [28b8d3ca] GR v0.38.1
  [7073ff75] IJulia v1.17.0
  [c601a237] Interact v0.9.1
  [b6b21f68] Ipopt v0.5.4
  [4076af6c] JuMP v0.19.0
  [e5e0dc1b] Juno v0.5.4
  [7f56f5a3] LSODA v0.4.0
  [eff96d63] Measurements v2.0.0
  [76087f3c] NLopt v0.5.1
  [c030b06c] ODE v2.4.0
  [54ca160b] ODEInterface v0.4.5+ [`C:\Users\accou\.julia\dev\ODEInterface`
]
  [09606e27] ODEInterfaceDiffEq v3.0.0
```

```
  [429524aa] Optim v0.17.2
  [1dea7af3] OrdinaryDiffEq v5.2.1+ [`C:\Users\accou\.julia\dev\OrdinaryDif
fEq`]
  [65888b18] ParameterizedFunctions v4.1.1
  [91a5bcdd] Plots v0.23.0
  [71ad9d73] PuMaS v0.0.0 [`C:\Users\accou\.julia\dev\PuMaS`]
  [d330b81b] PyPlot v2.7.0
  [731186ca] RecursiveArrayTools v0.20.0
  [90137ffa] StaticArrays v0.10.2
  [789caeaf] StochasticDiffEq v6.1.1+ [`C:\Users\accou\.julia\dev\Stochasti
cDiffEq`]
  [c3572dad] Sundials v3.0.0
  [1986cc42] Unitful v0.14.0
  [2a06ce6d] UnitfulPlots v0.0.0 #master (https://github.com/ajkeller34/Uni
tfulPlots.jl)
  [44d3d7a6] Weave v0.7.1 [`C:\Users\accou\.julia\dev\Weave`]
```