

# Monte Carlo Parameter Estimation From Data

Chris Rackauckas

June 27, 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)
```

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

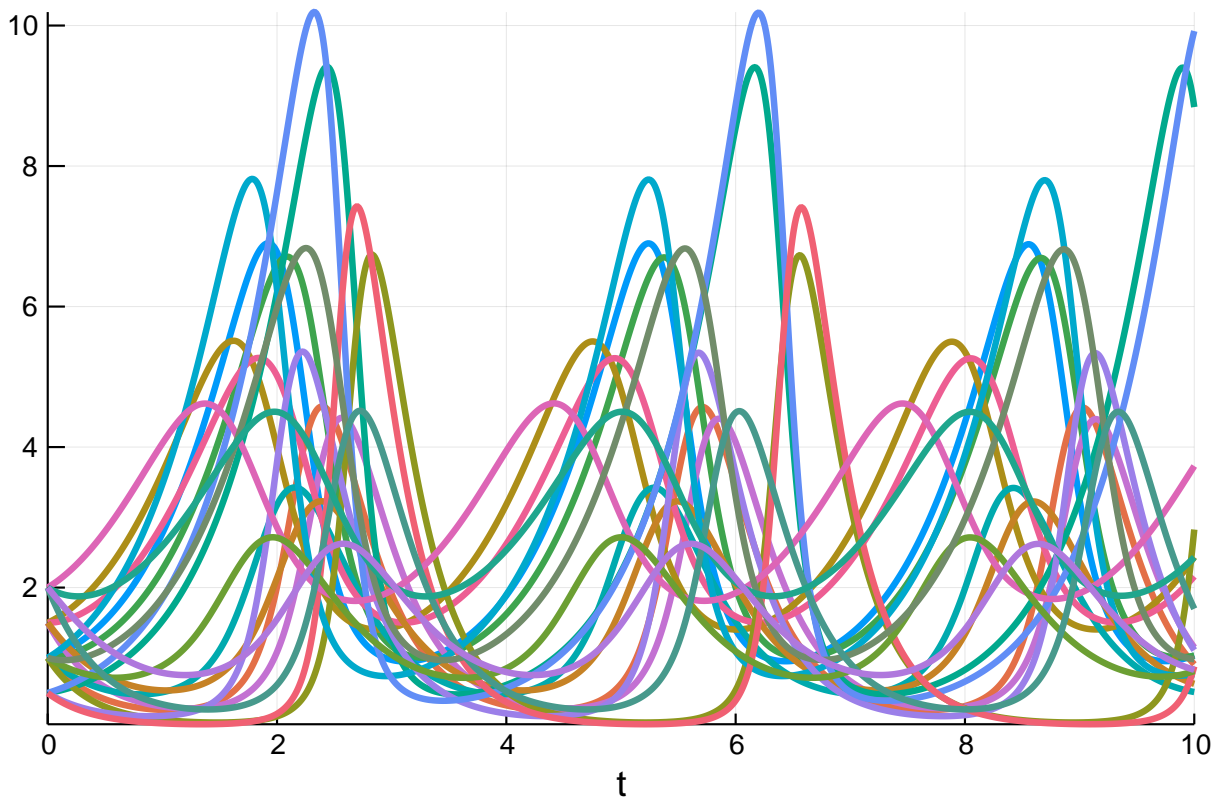
Now for a MonteCarloProblem we have to take this problem and tell it what to do N times via the prob\_func. So let's generate N=10 different initial conditions, and tell it to run the same problem but with these 10 different initial conditions each time:

```
# 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)
```

```
MonteCarloProblem 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)
```



`num_monte=N` means "run  $N$  times", and each time it runs the problem returned by the `probfnc`, which is always the same problem but with the  $i$ th 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
71

[:, :, 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  1.39973  1.14803

[:, :, 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 0.600942 0.549793 0.5136
79

[:, :, 4] =
1.5 1.51612 1.5621 1.63555 1.73531 ... 1.83823 1.98545 2.15958
1.5 1.29176 1.11592 0.969809 0.850159 0.771089 0.691421 0.630025

[:, :, 5] =
0.5 0.531705 0.576474 0.634384 0.706139 ... 9.05366 9.4006 8.83911
1.0 0.77995 0.610654 0.480565 0.380645 0.809382 1.51708 2.82619

[:, :, 6] =
1.0 1.11027 1.24238 1.39866 1.58195 ... 0.753108 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 ... 8.11216 9.10671 9.9217

0.5 0.390449 0.30679 0.24286 0.193966 0.261298 0.455937 0.8788
1

[:, :, 8] =
2.0 2.11239 2.24921 2.41003 2.59433 ... 3.223 3.47362 3.7301
4
1.0 0.909749 0.838025 0.783532 0.745339 0.739471 0.765597 0.8130
86

[:, :, 9] =
1.0 0.969326 0.971358 1.00017 ... 1.25065 1.1012 1.01733 0.979306
2.0 1.63445 1.33389 1.09031 3.02671 2.52063 2.07502 1.69807

[:, :, 10] =
2.0 1.92148 1.88215 1.87711 1.90264 ... 2.15079 2.27938 2.43105
2.0 1.80195 1.61405 1.4426 1.2907 0.957221 0.884827 0.82948

```

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 `j`th timepoint of the `i`th variable in the `k`th 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 let's 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]

10-element Array{DiffEqParamEstim.L2Loss{StepRangeLen{Float64,Base.TwicePrecision{Float64}},Base.TwicePrecision{Float64}},Array{Float64,2},Nothing,Nothing,Nothing},1}:
 DiffEqParamEstim.L2Loss{StepRangeLen{Float64,Base.TwicePrecision{Float64}},Base.TwicePrecision{Float64}},Array{Float64,2},Nothing,Nothing,Nothing}(0.0 : 0.1 : 10.0, [1.0 1.06108 ... 0.983561 1.03376; 1.0 0.821084 ... 1.10629 0.906371], nothing, nothing, nothing, nothing)
 DiffEqParamEstim.L2Loss{StepRangeLen{Float64,Base.TwicePrecision{Float64}},

```

```

Base.TwicePrecision{Float64}},Array{Float64,2},Nothing,Nothing,Nothing}(0.0
:0.1:10.0, [1.0 1.01413 ... 1.00811 1.03162; 1.5 1.22868 ... 1.39973 1.14803],
nothing, nothing, nothing, nothing)
DiffEqParamEstim.L2Loss{StepRangeLen{Float64,Base.TwicePrecision{Float64},
Base.TwicePrecision{Float64}},Array{Float64,2},Nothing,Nothing,Nothing}(0.0
:0.1:10.0, [1.5 1.58801 ... 2.21084 2.43589; 1.0 0.864317 ... 0.549793 0.513679
], nothing, nothing, nothing, nothing)
DiffEqParamEstim.L2Loss{StepRangeLen{Float64,Base.TwicePrecision{Float64},
Base.TwicePrecision{Float64}},Array{Float64,2},Nothing,Nothing,Nothing}(0.0
:0.1:10.0, [1.5 1.51612 ... 1.98545 2.15958; 1.5 1.29176 ... 0.691421 0.630025]
, nothing, nothing, nothing, nothing)
DiffEqParamEstim.L2Loss{StepRangeLen{Float64,Base.TwicePrecision{Float64},
Base.TwicePrecision{Float64}},Array{Float64,2},Nothing,Nothing,Nothing}(0.0
:0.1:10.0, [0.5 0.531705 ... 9.4006 8.83911; 1.0 0.77995 ... 1.51708 2.82619],
nothing, nothing, nothing, nothing)
DiffEqParamEstim.L2Loss{StepRangeLen{Float64,Base.TwicePrecision{Float64},
Base.TwicePrecision{Float64}},Array{Float64,2},Nothing,Nothing,Nothing}(0.0
:0.1:10.0, [1.0 1.11027 ... 0.748814 0.768284; 0.5 0.411557 ... 1.38829 1.10932
], nothing, nothing, nothing, nothing)
DiffEqParamEstim.L2Loss{StepRangeLen{Float64,Base.TwicePrecision{Float64},
Base.TwicePrecision{Float64}},Array{Float64,2},Nothing,Nothing,Nothing}(0.0
:0.1:10.0, [0.5 0.555757 ... 9.10671 9.9217; 0.5 0.390449 ... 0.455937 0.87881]
, nothing, nothing, nothing, nothing)
DiffEqParamEstim.L2Loss{StepRangeLen{Float64,Base.TwicePrecision{Float64},
Base.TwicePrecision{Float64}},Array{Float64,2},Nothing,Nothing,Nothing}(0.0
:0.1:10.0, [2.0 2.11239 ... 3.47362 3.73014; 1.0 0.909749 ... 0.765597 0.813086
], nothing, nothing, nothing, nothing)
DiffEqParamEstim.L2Loss{StepRangeLen{Float64,Base.TwicePrecision{Float64},
Base.TwicePrecision{Float64}},Array{Float64,2},Nothing,Nothing,Nothing}(0.0
:0.1:10.0, [1.0 0.969326 ... 1.01733 0.979306; 2.0 1.63445 ... 2.07502 1.69807]
, nothing, nothing, nothing, nothing)
DiffEqParamEstim.L2Loss{StepRangeLen{Float64,Base.TwicePrecision{Float64},
Base.TwicePrecision{Float64}},Array{Float64,2},Nothing,Nothing,Nothing}(0.0
:0.1:10.0, [2.0 1.92148 ... 2.27938 2.43105; 2.0 1.80195 ... 0.884827 0.82948],
nothing, nothing, nothing, nothing)

```

So `losses[i]` is a function which computes the loss of a solution against the data of the *i*th 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)

```

10108.695792027418

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_lossobjective`.

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

```
(::DiffEqParamEstim.DiffEqObjective{getfield(DiffEqParamEstim, Symbol("##29
#34")){Nothing, Bool, Int64, typeof(DiffEqParamEstim.STANDARD_PROB_GENERATOR),
Base.Iterators.Pairs{Symbol, Any, Tuple{Symbol, Symbol}, NamedTuple{(:num_monte
, :saveat), Tuple{Int64, StepRangeLen{Float64, Base.TwicePrecision{Float64}, Ba
se.TwicePrecision{Float64}}}}}, DiffEqBase.MonteCarloProblem{DiffEqBase.ODEP
roblem{Array{Float64, 1}, Tuple{Float64, Float64}, true, Array{Float64, 1}, DiffEq
Base.ODEFunction{true, typeof(Main.WeaveSandBox27.pf_func), LinearAlgebra.Uni
formScaling{Bool}, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing, N
othing, Nothing}, Nothing, DiffEqBase.StandardODEProblem}, typeof(Main.WeaveSan
dBox27.prob_func), getfield(DiffEqBase, Symbol("##282#288")), getfield(DiffEq
Base, Symbol("##284#290")), Array{Any, 1}}, OrdinaryDiffEq.Tsit5, typeof(Main.W
eaveSandBox27.loss), Nothing}, getfield(DiffEqParamEstim, Symbol("##33#39"))}
) (generic function with 2 methods)
```

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()))
```

```
Results of Optimization Algorithm
* Algorithm: Fminbox with BFGS
* Starting Point: [1.3, 0.9]
* Minimizer: [1.5000000000581937, 1.0000000001633538]
* Minimum: 7.119929e-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)| = 1.07e-06
* Stopped by an increasing objective: true
* Reached Maximum Number of Iterations: false
* Objective Calls: 193
* Gradient Calls: 193
```

```
result
```

```
Results of Optimization Algorithm
* Algorithm: Fminbox with BFGS
* Starting Point: [1.3,0.9]
* Minimizer: [1.5000000000581937,1.0000000001633538]
* Minimum: 7.119929e-16
* Iterations: 4
* Convergence: true
  *  $|x - x'| \leq 0.0e+00$ : true
     $|x - x'| = 0.00e+00$ 
  *  $|f(x) - f(x')| \leq 0.0e+00$   $|f(x)|$ : true
     $|f(x) - f(x')| = 0.00e+00$   $|f(x)|$ 
  *  $|g(x)| \leq 1.0e-08$ : false
     $|g(x)| = 1.07e-06$ 
  * Stopped by an increasing objective: true
  * Reached Maximum Number of Iterations: false
* Objective Calls: 193
* Gradient Calls: 193
```

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()))
```

```
Results of Optimization Algorithm
* Algorithm: Fminbox with BFGS
* Starting Point: [1.3,0.9]
* Minimizer: [1.500743476201907,1.001238477622136]
* Minimum: 4.163900e-02
* Iterations: 5
* Convergence: true
  *  $|x - x'| \leq 0.0e+00$ : true
     $|x - x'| = 0.00e+00$ 
  *  $|f(x) - f(x')| \leq 0.0e+00$   $|f(x)|$ : true
     $|f(x) - f(x')| = 0.00e+00$   $|f(x)|$ 
  *  $|g(x)| \leq 1.0e-08$ : false
     $|g(x)| = 1.07e-06$ 
  * Stopped by an increasing objective: false
  * Reached Maximum Number of Iterations: false
* Objective Calls: 224
* Gradient Calls: 224
```

```
result
```

```

Results of Optimization Algorithm
* Algorithm: Fminbox with BFGS
* Starting Point: [1.3,0.9]
* Minimizer: [1.500743476201907,1.001238477622136]
* Minimum: 4.163900e-02
* Iterations: 5
* Convergence: true
  *  $|x - x'| \leq 0.0e+00$ : true
     $|x - x'| = 0.00e+00$ 
  *  $|f(x) - f(x')| \leq 0.0e+00$   $|f(x)|$ : true
     $|f(x) - f(x')| = 0.00e+00$   $|f(x)|$ 
  *  $|g(x)| \leq 1.0e-08$ : false
     $|g(x)| = 1.07e-06$ 
  * Stopped by an increasing objective: false
  * Reached Maximum Number of Iterations: false
* Objective Calls: 224
* Gradient Calls: 224

```

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/DiffEqTutorials.jl>

To locally run this tutorial, do the following commands:

```

using DiffEqTutorials
DiffEqTutorials.weave_file("ode_extras","04-monte_carlo_parameter_estim.jmd")

```

Computer Information:

```

Julia Version 1.1.1
Commit 55e36cc308 (2019-05-16 04:10 UTC)
Platform Info:
  OS: Linux (x86_64-pc-linux-gnu)
  CPU: Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-6.0.1 (ORCJIT, ivybridge)

```

Package Information:

```

Status `~/julia/environments/v1.1/Project.toml`
[7e558dbc-694d-5a72-987c-6f4ebed21442] ArbNumerics 0.5.4
[6e4b80f9-dd63-53aa-95a3-0cdb28fa8baf] BenchmarkTools 0.4.2
[be33ccc6-a3ff-5ff2-a52e-74243cff1e17] CUDAnative 2.2.0
[3a865a2d-5b23-5a0f-bc46-62713ec82fae] CuArrays 1.0.2
[55939f99-70c6-5e9b-8bb0-5071ed7d61fd] DecFP 0.4.8

```

[abce61dc-4473-55a0-ba07-351d65e31d42] Decimals 0.4.0  
 [ebbdde9d-f333-5424-9be2-dbf1e9acfb5e] DiffEqBayes 1.1.0  
 [eb300fae-53e8-50a0-950c-e21f52c2b7e0] DiffEqBiological 3.8.2  
 [459566f4-90b8-5000-8ac3-15dfb0a30def] DiffEqCallbacks 2.5.2  
 [f3b72e0c-5b89-59e1-b016-84e28bfd966d] DiffEqDevTools 2.9.0  
 [1130ab10-4a5a-5621-a13d-e4788d82bd4c] DiffEqParamEstim 1.6.0  
 [055956cb-9e8b-5191-98cc-73ae4a59e68a] DiffEqPhysics 3.1.0  
 [6d1b261a-3be8-11e9-3f2f-0b112a9a8436] DiffEqTutorials 0.1.0  
 [0c46a032-eb83-5123-abaf-570d42b7fbba] DifferentialEquations 6.4.0  
 [31c24e10-a181-5473-b8eb-7969acd0382f] Distributions 0.20.0  
 [497a8b3b-efae-58df-a0af-a86822472b78] DoubleFloats 0.9.1  
 [f6369f11-7733-5829-9624-2563aa707210] ForwardDiff 0.10.3  
 [c91e804a-d5a3-530f-b6f0-dfbca275c004] Gadfly 1.0.1  
 [7073ff75-c697-5162-941a-fcdaad2a7d2a] IJulia 1.18.1  
 [4138dd39-2aa7-5051-a626-17a0bb65d9c8] JLD 0.9.1  
 [23fbe1c1-3f47-55db-b15f-69d7ec21a316] Latexify 0.8.2  
 [eff96d63-e80a-5855-80a2-b1b0885c5ab7] Measurements 2.0.0  
 [961ee093-0014-501f-94e3-6117800e7a78] ModelingToolkit 0.2.0  
 [76087f3c-5699-56af-9a33-bf431cd00edd] NLOpt 0.5.1  
 [2774e3e8-f4cf-5e23-947b-6d7e65073b56] NLSolve 4.0.0  
 [429524aa-4258-5aef-a3af-852621145aeb] Optim 0.18.1  
 [1dea7af3-3e70-54e6-95c3-0bf5283fa5ed] OrdinaryDiffEq 5.8.1  
 [65888b18-ceab-5e60-b2b9-181511a3b968] ParameterizedFunctions 4.1.1  
 [91a5bcd7-55d7-5caf-9e0b-520d859cae80] Plots 0.25.1  
 [d330b81b-6aea-500a-939a-2ce795aea3ee] PyPlot 2.8.1  
 [731186ca-8d62-57ce-b412-fbd966d074cd] RecursiveArrayTools 0.20.0  
 [90137ffa-7385-5640-81b9-e52037218182] StaticArrays 0.11.0  
 [f3b207a7-027a-5e70-b257-86293d7955fd] StatsPlots 0.11.0  
 [c3572dad-4567-51f8-b174-8c6c989267f4] Sundials 3.6.1  
 [1986cc42-f94f-5a68-af5c-568840ba703d] Unitful 0.15.0  
 [44d3d7a6-8a23-5bf8-98c5-b353f8df5ec9] Weave 0.9.0  
 [b77e0a4c-d291-57a0-90e8-8db25a27a240] InteractiveUtils  
 [37e2e46d-f89d-539d-b4ee-838fcccc9c8e] LinearAlgebra  
 [44cfe95a-1eb2-52ea-b672-e2afdf69b78f] Pkg