Defining and Simulating Populations

Vijay Ivaturi, Chris Rackauckas

July 19, 2019

```julia
using Pumas, DataFrames, LinearAlgebra, Plots
```

1 Introduction

In this tutorial, we will cover the fundamentals of generating populations to simulate with Pumas. We will demonstrate how to specify dosage regimens and covariates, and then how to piece these together to form a population to simulate.

1.1 The model

Below is a Pumas model that specifies a 1-compartment oral absorption system with between-subject variability on all the parameters. Details of the model specification are provided in the introduction tutorial.

```julia
model = @model begin
  @param begin
    θ ∈ VectorDomain(4)
    Ω ∈ PSDDomain(3)
    σ_prop ∈ RealDomain(init=0.1)
  end

  @random begin
    η ~ MvNormal(Ω)
  end

  @covariates isPM Wt

  @pre begin
    TVCL = isPM == 1 ? θ[1] : θ[4]
    CL = θ[1]*(Wt/70)^0.75*exp(η[1])
    V = θ[2]*(Wt/70)^0.75*exp(η[2])
    Ka = θ[3]*exp(η[3])
  end

  @dynamics begin
    Depot'   = -Ka*Depot
    Central' =  Ka*Depot - Central*CL/V
  end

  @vars begin
```

```
    conc = Central/V
  end

  @derived begin
    dv ~ @.Normal(conc,sqrt(conc^2*σ_prop+ eps()))
  end

end
```

```
PumasModel
  Parameters: θ, Ω, σ_prop
  Random effects: η
  Covariates: isPM, Wt
  Dynamical variables: Depot, Central
  Derived: conc, dv
  Observed: conc, dv
```

1.2 Setting up parameters

Next we provide the initial estimates of the parameters to simulate from. The fixed effects are provided in the θ vector (CL, V, Ka) and the between-subject variability parameteres are provided in the Ω vector as variances. So, 0.04 variance on Ω*11 suggests a 20% coefficient of variation. Similarly,* σprop has a 20% proportional residual error.

```
fixeffs = (
  θ = [0.4,20,1.1,2],
  Ω = diagm(0 => [0.04,0.04,0.04]),
  σ_prop = 0.04
  )
```

```
(θ = [0.4, 20.0, 1.1, 2.0], Ω = [0.04 0.0 0.0; 0.0 0.04 0.0; 0.0 0.0 0.04],
 σ_prop = 0.04)
```

1.3 Single dose example

`DosageRegimen()` is the function that lets you construct a dosing regimen. The first argument of the `DosageRegimen` is `amt` and is not a named argument. All subsequent arguments need to be named. Lets try a simple example where you provide a 100 mg dose at `time=0`.

```
ev = DosageRegimen(100, time=0)
first(ev.data)
```

	time	cmt	amt	evid	ii	addl	rate	ss
	Float64	Int64	Float64	Int8	Float64	Int64	Float64	Int8
1	0.0	1	100.0	1	0.0	0	0.0	0

As you can see above, we provided a single 100 mg dose. `DosageRegimen` provides some defaults when it creates the dataset, `time=0`, `evid=1`, `cmt=1`, `rate=0`, `ii=0` & `addl=0`. We can also provide units to the `amt` and any other variable that is derived from `amt`, e.g. `rate`, will have associated units. Handling of units will be covered in a different tutorial.

Note that `ev` is of type `DosageRegimen`. Specified like above, `DosageRegimen` is one of the four fundamental building block of a `Subject` (more on `Subject` below).

1.3.1 Building Subjects

Let's create a single subject

```
s1 = Subject(id=1,evs=ev,cvs=(isPM=0, Wt=70))
for fn in fieldnames(Subject)
           x = getproperty(s1, fn)
           if !isa(x, Nothing)
               println(fn)
               println(x)
           end
end
```

```
id
1
covariates
(isPM = 0, Wt = 70)
events
Pumas.Event[Dose event
  dose amount = 100.0
  dose time = 0.0
  compartment = 1
  instantaneous
  interdose interval = 0.0
  infusion start time = 0.0
]
```

Note that each `Subject` is an individual composed of:

- `id`: an unique identifier
- `obs`: observations, represented by `Pumas.Observation[]`
- `cvs`: covariates
- `evs`: events, represented by `Pumas.Event[]`

In the example above, we only provided the `id`, `evs`, and the `cvs`. Since `obs` were not provided, they are represented by an empty array. Lets take a closer at the events for this subject 1.

```
s1.events
```

```
1-element Array{Pumas.Event,1}:
 Dose event
  dose amount = 100.0
  dose time = 0.0
  compartment = 1
  instantaneous
  interdose interval = 0.0
  infusion start time = 0.0
```

The events are presented by basic information such as the dose of drug and associated units if specified, the time of dose administration, the compartment number for administration and whether the dose is an instantaneous input or an infusion.

Below is how the covariates are represented

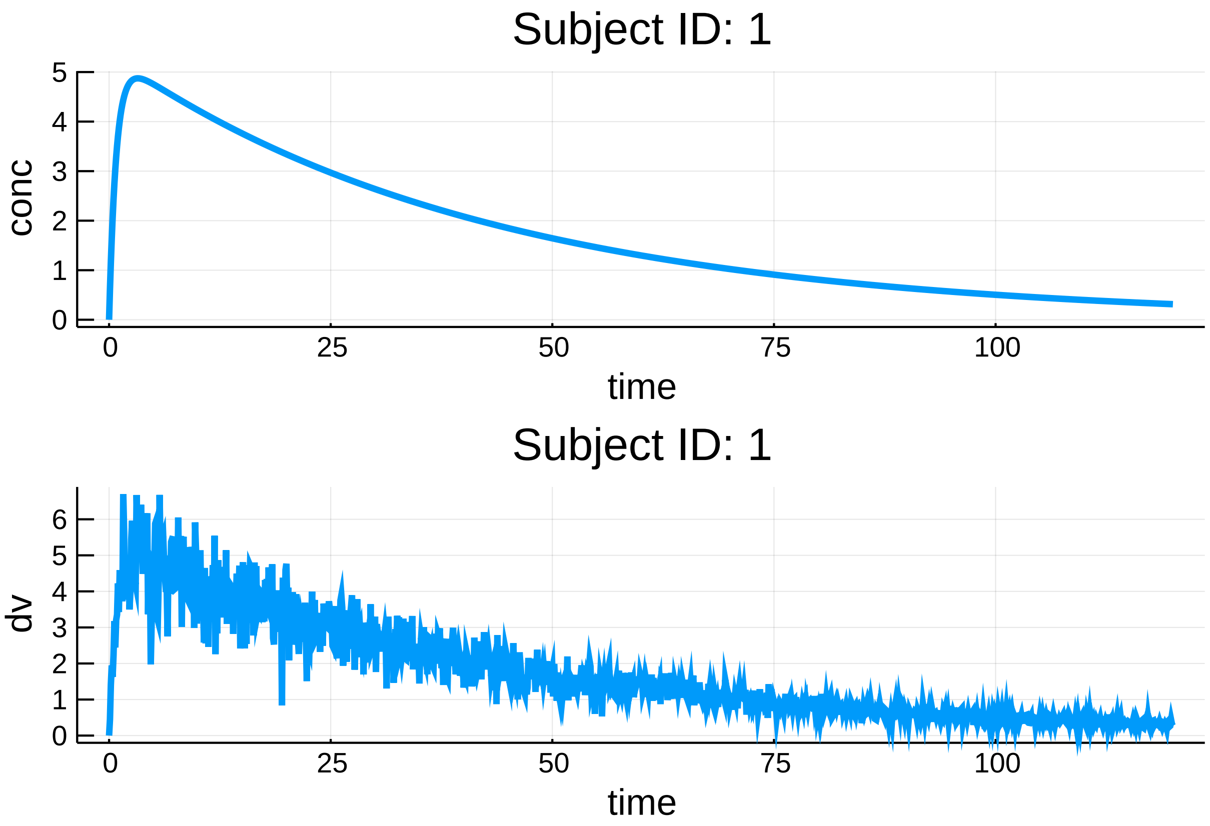
```
s1.covariates
```

```
(isPM = 0, Wt = 70)
```

(Note: defining distributions for covariates will be discussed in detail later.)

Using this one subject, `s1`, let us simulate a simple concentration time profile using the model above:

```
obs = simobs(model,s1,fixeffs,obstimes=0:0.1:120)
plot(obs)
```



1.3.2 Building Populations

Now, lets create one more subject, `s2`.

```
s2 = Subject(id=2,evs=ev,cvs=(isPM=1,Wt=70))
```

```
Subject
  ID: 2
  Events: 1
```

If we want to simulate both `s1` and `s2` together, we need to bring these subjects together to form a `Population`. A `Population` is essentially a collection of subjects.

```
twosubjs =  Population([s1,s2])
```

```
Population
  Subjects: 2
  Covariates: isPM, Wt
```

Let's see the details of the first and the second subject

```
twosubjs[1]
```

```
Subject
  ID: 1
  Events: 1
```

```
twosubjs[2]
```

```
Subject
  ID: 2
  Events: 1
```

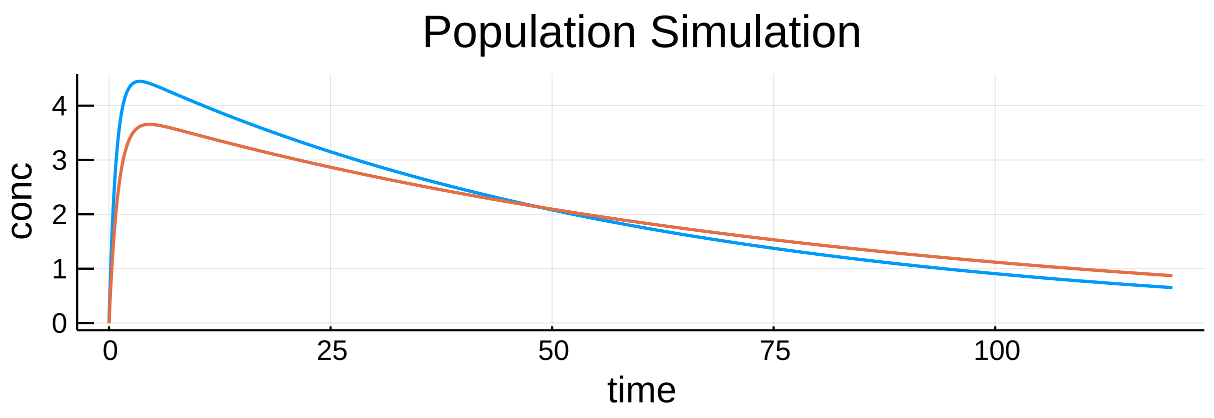
Now, we can simulate this `Population` of 2 subjects as below

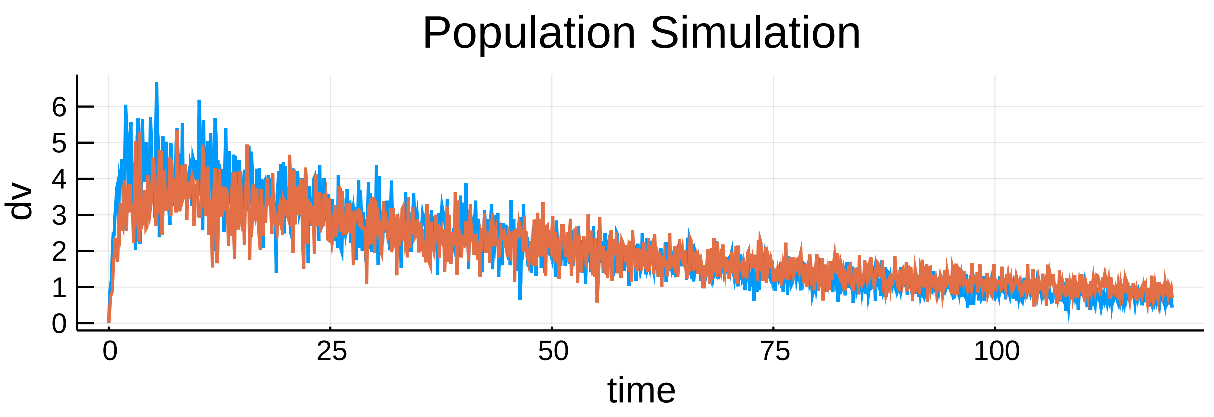
```
obs = simobs(model,twosubjs,fixeffs,obstimes=0:0.1:120)
```

```
Pumas.SimulatedPopulation{Array{Pumas.SimulatedObservations{Pumas.Subject{N
othing,NamedTuple{(:isPM, :Wt),Tuple{Int64,Int64}},Array{Pumas.Event,1},Not
hing},StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision
{Float64}},NamedTuple{(:conc, :dv),Tuple{Array{Float64,1},Array{Float64,1}}
}},1}}(Pumas.SimulatedObservations{Pumas.Subject{Nothing,NamedTuple{(:isPM,
 :Wt),Tuple{Int64,Int64}},Array{Pumas.Event,1},Nothing},StepRangeLen{Float6
4,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}},NamedTuple{(:c
onc, :dv),Tuple{Array{Float64,1},Array{Float64,1}}}}[SimulatedObservations{
Subject{Nothing,NamedTuple{(:isPM, :Wt),Tuple{Int64,Int64}},Array{Event,1},
Nothing},StepRangeLen{Float64,TwicePrecision{Float64},TwicePrecision{Float6
4}},NamedTuple{(:conc, :dv),Tuple{Array{Float64,1},Array{Float64,1}}}}(Subj
ect
  ID: 1
  Events: 1
, 0.0:0.1:120.0, (conc = [0.0, 0.563517, 1.05861, 1.49348, 1.87533, 2.21052
, 2.50464, 2.76261, 2.98875, 3.18689  …  0.661089, 0.659993, 0.658899, 0.65
7806, 0.656715, 0.655626, 0.654539, 0.653454, 0.65237, 0.651289], dv = [7.5
3737e-9, 0.848213, 1.0385, 1.21841, 1.97074, 2.47146, 2.46258, 2.99235, 3.3
4175, 3.73778  …  0.589284, 0.477367, 0.707577, 0.525525, 0.571284, 0.78563
6, 0.634079, 0.567972, 0.64268, 0.43337])), SimulatedObservations{Subject{N
othing,NamedTuple{(:isPM, :Wt),Tuple{Int64,Int64}},Array{Event,1},Nothing},
StepRangeLen{Float64,TwicePrecision{Float64},TwicePrecision{Float64}},Named
Tuple{(:conc, :dv),Tuple{Array{Float64,1},Array{Float64,1}}}}(Subject
  ID: 2
  Events: 1
, 0.0:0.1:120.0, (conc = [0.0, 0.355126, 0.677207, 0.969279, 1.23409, 1.474
16, 1.69174, 1.88891, 2.06754, 2.22933  …  0.881061, 0.879957, 0.878854, 0.
877753, 0.876653, 0.875555, 0.874458, 0.873362, 0.872268, 0.871175], dv = [
6.52876e-9, 0.34352, 0.741264, 0.807951, 0.891645, 1.54727, 1.83182, 2.3682
6, 1.82296, 2.15159  …  0.851319, 1.14245, 0.993811, 0.931137, 0.80287, 1.0
7599, 1.01436, 0.861808, 0.960052, 0.703958]))])
```

When using `simobs` on more than one subject, i.e., on a `Population`, the simulation is automatically parallelized across the subejcts.

```
plot(obs)
```





Similarly, we can build a population of any number of subjects. But before we do that, let's dive into covariate generation.

1.3.3 Covariates

As was discussed earlier, a `Subject` can also be provided details regarding covariates. In the model above, there are two covariates, `isPM` which stands for *is the subject a poor metabolizer* and takes a boolean of *yes* and *no*. The second covariate is a continuous covariate where body weight `Wt` impacts both `CL` and `V`. Let us now specify covariates to a population of 10 subjects.

```
choose_covariates() = (isPM = rand([1, 0]),
                       Wt = rand(55:80))
```

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

`choose_covariates` will randomly choose a `isPM` and an `Wt` between 55-80 kgs

We can make a list with covariates for ten subjects through a list comprehension

```
cvs = [ choose_covariates() for i in 1:10 ]
DataFrame(cvs)
```

	isPM	Wt
	Int64	Int64
1	1	63
2	1	59
3	1	66
4	1	64
5	0	65
6	0	70
7	0	57
8	0	73
9	1	64
10	0	61

Now, we add these covariates to the population as below. The `map(f,xs)` will return the result of `f` on each element of `xs`. Let's map a function that build's a subject with the randomly chosen covariates in order to build a population:

```
pop_with_covariates = Population(map(i ->
    Subject(id=i,evs=ev,cvs=choose_covariates()),1:10))
```

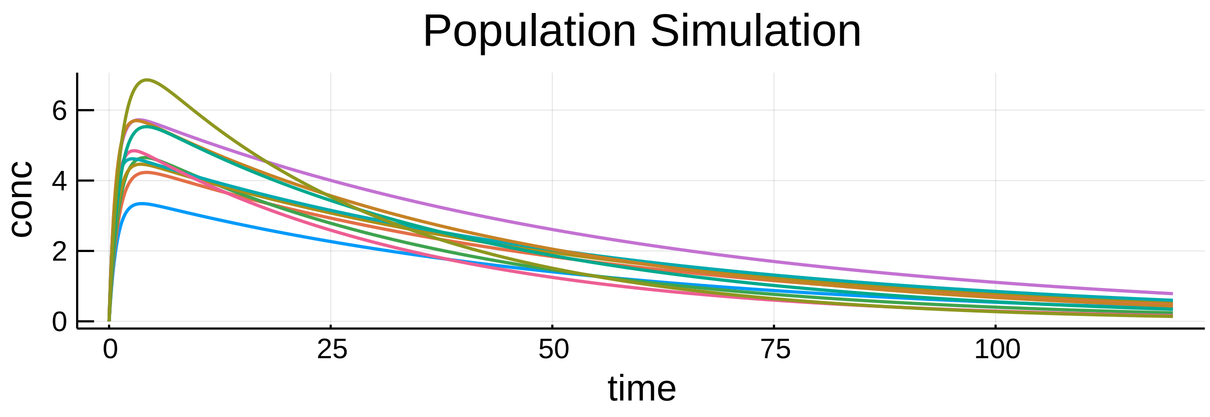
```
Population
  Subjects: 10
  Covariates: isPM, Wt
```

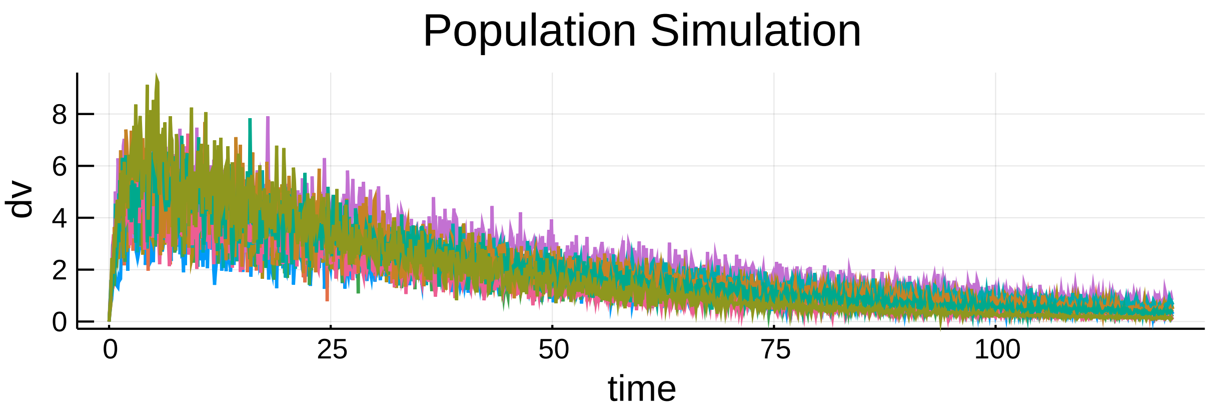
Simulate into the population

```
obs = simobs(model,pop_with_covariates,fixeffs,obstimes=0:0.1:120);
```

and visualize the output

```
plot(obs)
```





1.4 Multiple dose example

The additional dosage regimen controls of the NMTRAN format are available in `DosageRegimen`. For example, `ii` defines the "interdose interval", or the time distance between two doses, while `addl` defines how many additional times to repeat a dose. Thus, let's define a dose of 100 that's repeated 7 times at 24 hour intervals:

```
md =  DosageRegimen(100,ii=24,addl=6)
```

```
Pumas.DosageRegimen(1×8 DataFrames.DataFrame
 Row  time     cmt    amt      evid  ii       addl   rate     ss

      Float64  Int64  Float64  Int8  Float64  Int64  Float64  Int8



 1    0.0      1      100.0    1     24.0     6      0.0      0
 )
```

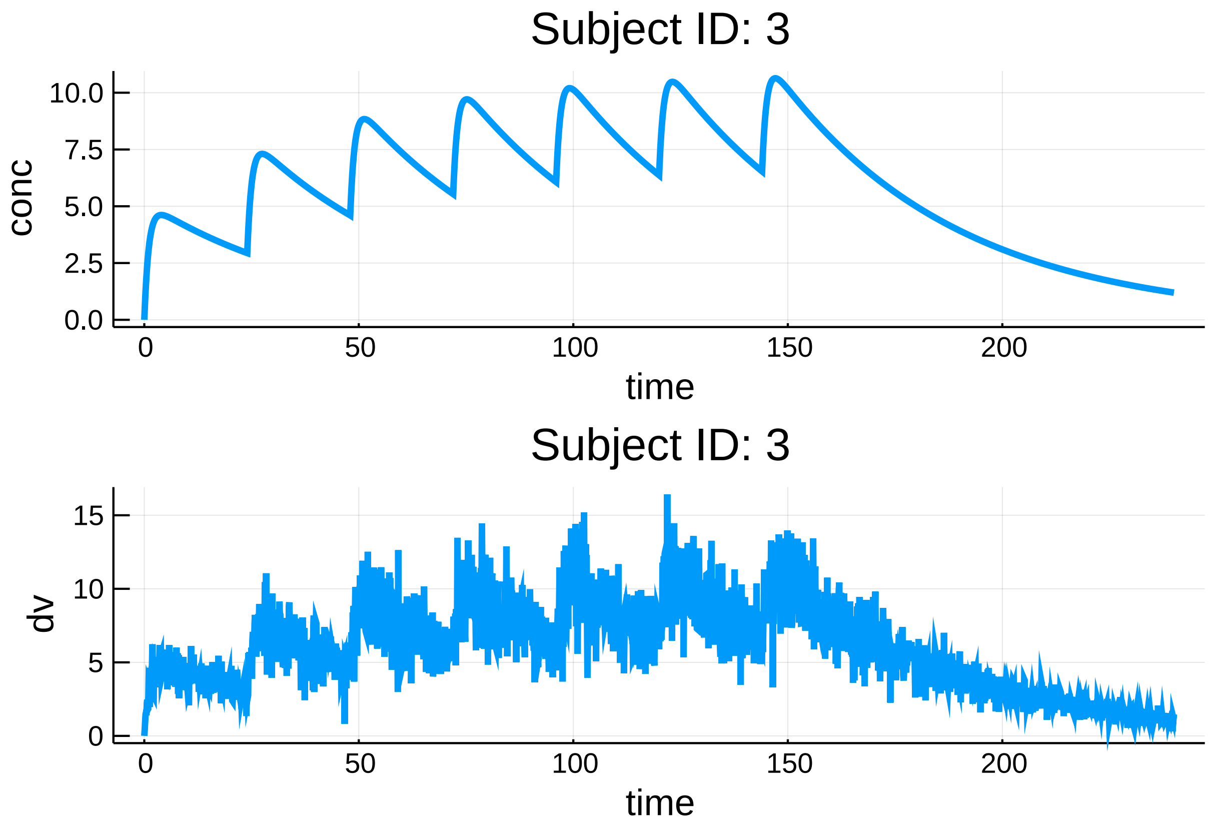
Let's create a new subject, `s3` with this dosage regimen:

```
s3 = Subject(id=3,evs=md, cvs=(isPM=0,Wt=70))
```

```
Subject
  ID: 3
  Events: 7
```

and see the results:

```
obs = simobs(model, s3, fixeffs,obstimes=0:0.1:240)
plot(obs)
```



1.5 Combining dosage regimens

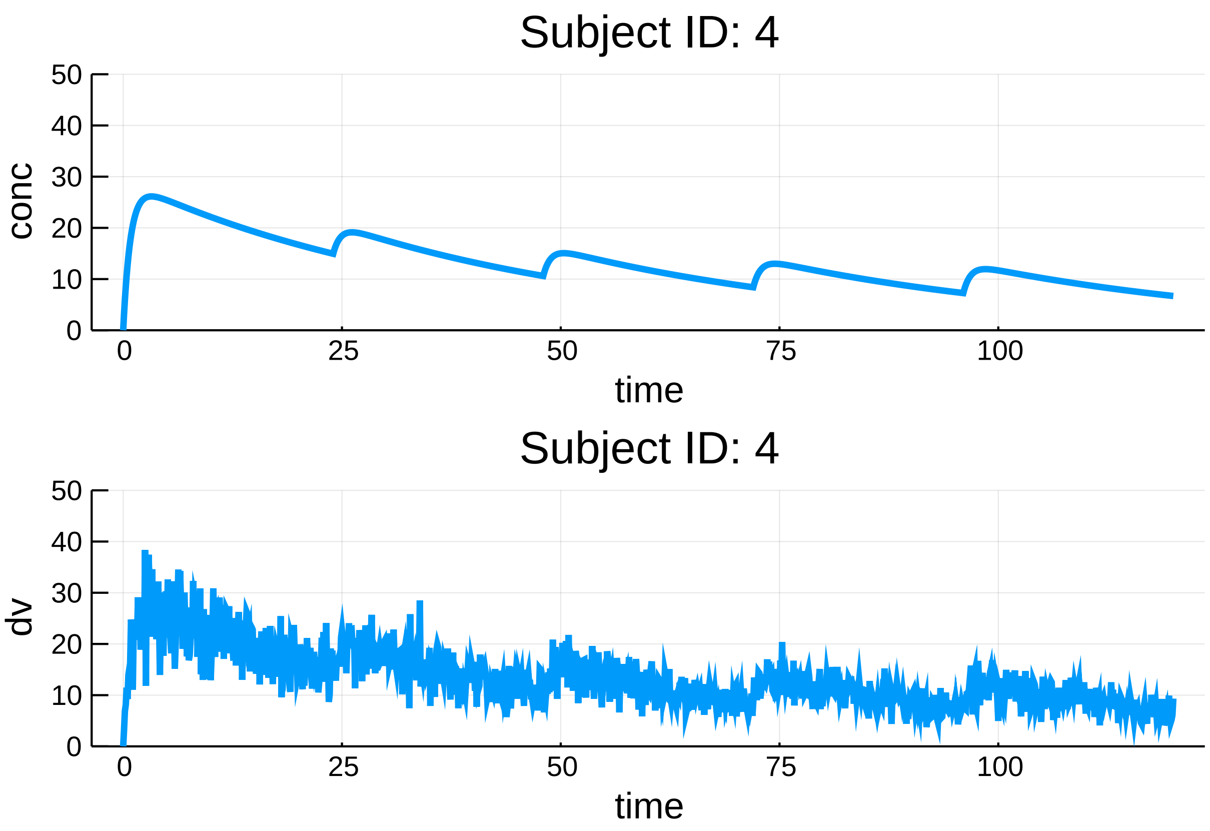
We can also combine dosage regimens to build a more complex regimen. Recall from the introduction that using arrays will build the element-wise combinations. Thus let's build a dose of 500 into compartment 1 at time 0, and 7 doses into compartment 1 of 100 spaced by 24 hours:

```
ldmd = DosageRegimen([500,100],cmt=1, time=[0,24], addl=[0,6],ii=[0,24])
```

```
Pumas.DosageRegimen(2×8 DataFrames.DataFrame
 Row  time     cmt    amt      evid  ii       addl   rate     ss

      Float64  Int64  Float64  Int8  Float64  Int64  Float64  Int8




 1    0.0      1      500.0    1     0.0      0      0.0      0


 2    24.0     1      100.0    1     24.0     6      0.0      0
)
```

Let's see if this result matches our intuition:

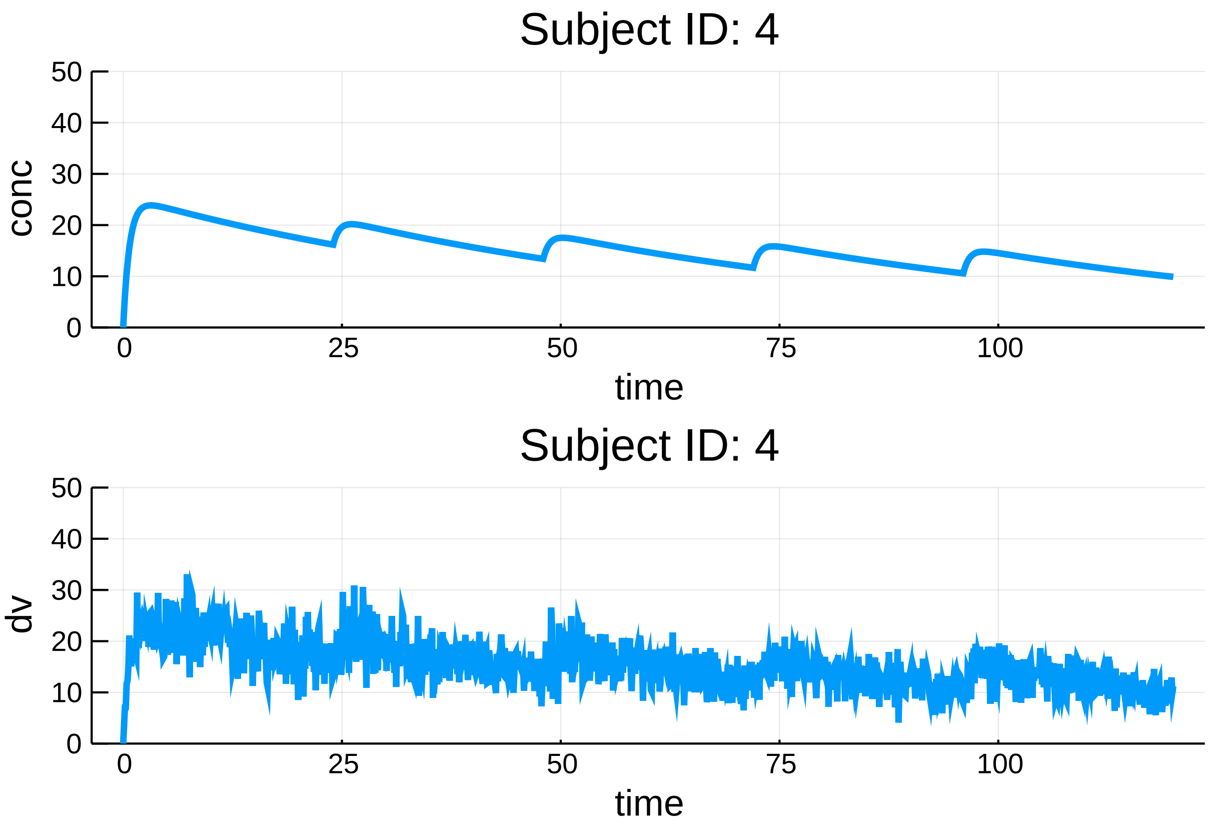
```
s4 = Subject(id=4, evs=ldmd, cvs=(isPM=0,Wt=70))
obs = simobs(model, s4, fixeffs,obstimes=0:0.1:120)
plot(obs, ylims=(0,50))
```



Another way to build complex dosage regiments is to combine previously constructed regimens into a single regimen. For example:

```
e1 = DosageRegimen(500,cmt=1, time=0, addl=0,ii=0)
e2 = DosageRegimen(100,cmt=1, time=24, addl=6,ii=24)
evs = DosageRegimen(e1,e2)
```

```
obs = simobs(model, s4, fixeffs,obstimes=0:0.1:120)
plot(obs, ylims=(0,50))
```



is the same regimen as before.

Putting these ideas together, we can define a population where individuals with different covariates undergo different regimens, and simulate them all together with automatic parallelism:

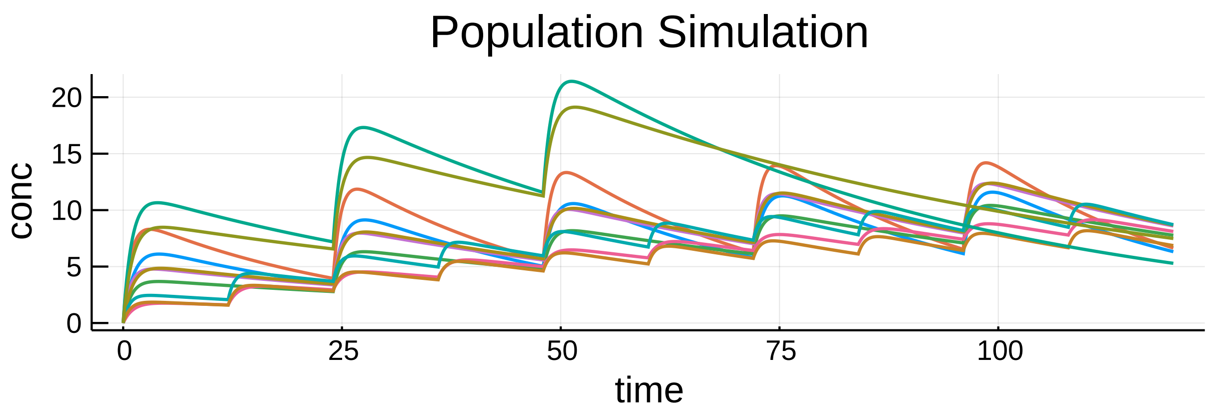
```
e1 = DosageRegimen(100, ii=24, addl=6)
e2 = DosageRegimen(50,  ii=12, addl=13)
e3 = DosageRegimen(200, ii=24, addl=2)
```

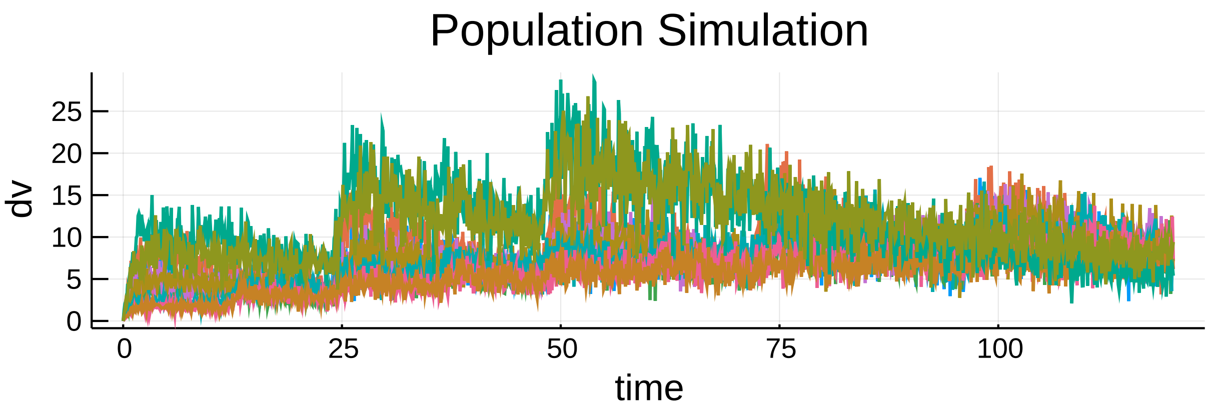
```
Pumas.DosageRegimen(1×8 DataFrames.DataFrame
 Row  time     cmt    amt      evid  ii       addl   rate     ss

      Float64  Int64  Float64  Int8  Float64  Int64  Float64  Int8




 1    0.0      1      200.0    1     24.0     2      0.0      0
 )
```

```
pop1 = Population(map(i -> Subject(id=i,evs=e1,cvs=choose_covariates()),1:5))
pop2 = Population(map(i -> Subject(id=i,evs=e2,cvs=choose_covariates()),6:8))
pop3 = Population(map(i -> Subject(id=i,evs=e3,cvs=choose_covariates()),9:10))
pop = Population(vcat(pop1,pop2,pop3))
```

```
Population
  Subjects: 10
  Covariates: isPM, Wt
```

```
obs = simobs(model,pop,fixeffs,obstimes=0:0.1:120)
plot(obs)
```





1.6 Defining Infusions

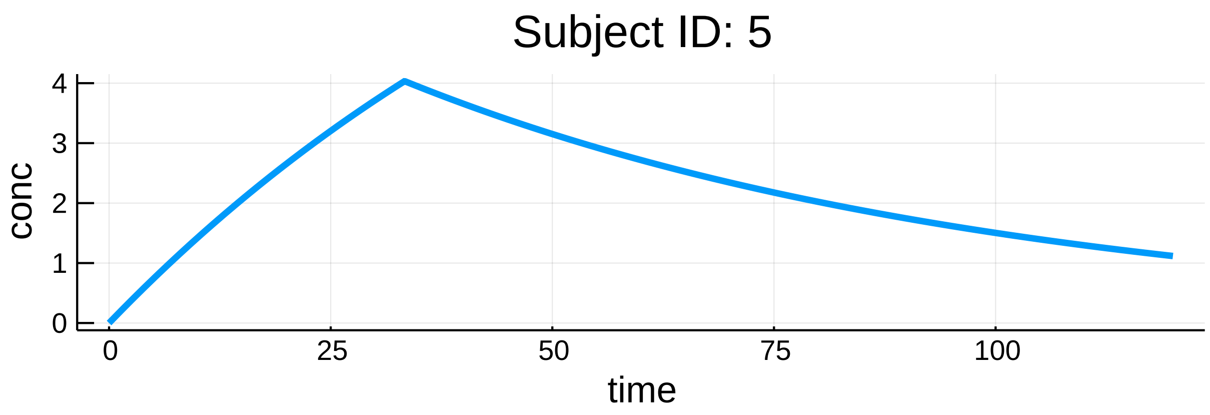
As specified in the NMTRAN format, an infusion is a dosage which is defined as having a non-zero positive rate at which the drug enters the system. Let's define a single infusion dose of total amount 100 with a rate of 3 into the second compartment:

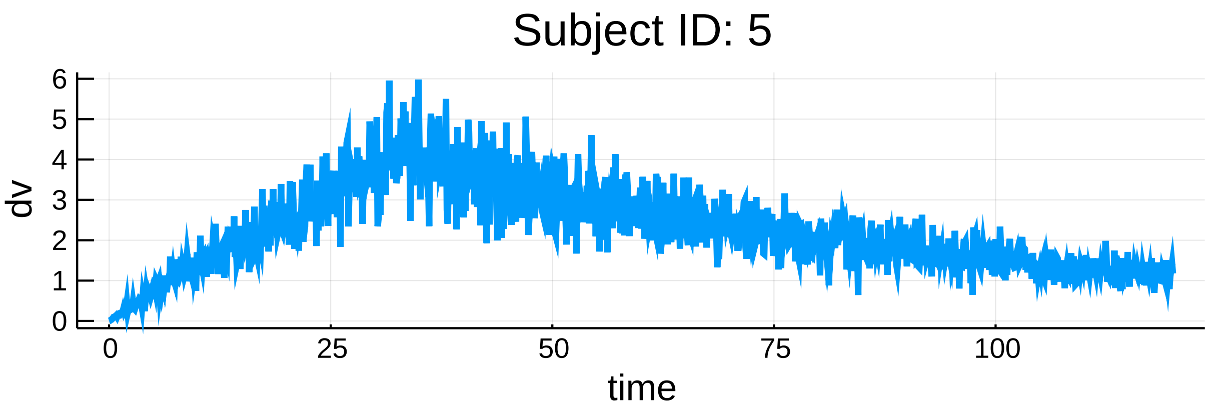
```
inf = DosageRegimen(100, rate=3, cmt=2)
```

```
Pumas.DosageRegimen(1×8 DataFrames.DataFrame
 Row  time     cmt    amt      evid  ii       addl   rate     ss

      Float64  Int64  Float64  Int8  Float64  Int64  Float64  Int8



 1    0.0      2      100.0    1     0.0      0      3.0      0
 )
```

Now let's simulate a subject undergoing this treatment strategy:

```
s5 = Subject(id=5, evs=inf, cvs=(isPM=0,Wt=70))
obs = simobs(model, s5, fixeffs, obstimes=0:0.1:120)
plot(obs)
```





1.7 Final Note on Julia Programming

Note that all of these functions are standard Julia functions, and thus standard Julia programming constructions can be utilized to simplify the construction of large populations. We already demonstrated the use of `map` and a comprehension, but we can also make use of constructs like for loops.

1.8 Conclusion

This tutorial shows the tools for generating populations of infinite complexity, defining covariates and dosage regimens on the fly and simulating the results of the model.

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

1.9 Appendix

These tutorials are part of the PumasTutorials.jl repository, found at: https://github.com/JuliaDiffEq/Di

To locally run this tutorial, do the following commands:

```
using PumasTutorials
PumasTutorials.weave_file("introduction","simulating_populations.jmd")
```

Computer Information:

```
Julia Version 1.1.1
Commit 55e36cc308 (2019-05-16 04:10 UTC)
Platform Info:
  OS: Windows (x86_64-w64-mingw32)
  CPU: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-6.0.1 (ORCJIT, skylake)
Environment:
  JULIA_EDITOR = "C:\Users\accou\AppData\Local\atom\app-1.38.2\atom.exe"  -a
  JULIA_NUM_THREADS = 4
```

Package Information:

```
Status `C:\Users\accou\.julia\environments\v1.1\Project.toml`
[621f4979-c628-5d54-868e-fcf4e3e8185c] AbstractFFTs 0.4.1
[c52e3926-4ff0-5f6e-af25-54175e0327b1] Atom 0.8.8
[f0abef60-9ec0-11e9-27de-db6506a91768] AutoOffload 0.1.0
[6e4b80f9-dd63-53aa-95a3-0cdb28fa8baf] BenchmarkTools 0.4.2
[4ece37e6-a012-11e8-38cd-91247efc2c34] Bioequivalence 0.1.0
[336ed68f-0bac-5ca0-87d4-7b16caf5d00b] CSV 0.5.9
[c5f51814-7f29-56b8-a69c-e4d8f6be1fde] CUDAdrv 3.0.1
[be33ccc6-a3ff-5ff2-a52e-74243cff1e17] CUDAnative 2.2.1
[49dc2e85-a5d0-5ad3-a950-438e2897f1b9] Calculus 0.5.0
[7057c7e9-c182-5462-911a-8362d720325c] Cassette 0.2.5
[34da2185-b29b-5c13-b0c7-acf172513d20] Compat 2.1.0
[3a865a2d-5b23-5a0f-bc46-62713ec82fae] CuArrays 1.1.0
[667455a9-e2ce-5579-9412-b964f529a492] Cubature 1.4.0
[a93c6f00-e57d-5684-b7b6-d8193f3e46c0] DataFrames 0.18.4
[82cc6244-b520-54b8-b5a6-8a565e85f1d0] DataInterpolations 0.2.0
[31a5f54b-26ea-5ae9-a837-f05ce5417438] Debugger 0.5.0
[bcd4f6db-9728-5f36-b5f7-82caef46ccdb] DelayDiffEq 5.9.1
[2b5f629d-d688-5b77-993f-72d75c75574e] DiffEqBase 5.16.3
[ebbdde9d-f333-5424-9be2-dbf1e9acfb5e] DiffEqBayes 1.2.0
[31c91b34-3c75-11e9-0341-95557aab0344] DiffEqBenchmarks 0.1.0
[459566f4-90b8-5000-8ac3-15dfb0a30def] DiffEqCallbacks 2.5.2+
[f3b72e0c-5b89-59e1-b016-84e28bfd966d] DiffEqDevTools 2.13.0
[01453d9d-ee7c-5054-8395-0335cb756afa] DiffEqDiffTools 0.14.0
[aae7a2af-3d4f-5e19-a356-7da93b79d9d0] DiffEqFlux 0.5.2
[071ae1c0-96b5-11e9-1965-c90190d839ea] DiffEqGPU 0.1.0
[c894b116-72e5-5b58-be3c-e6d8d4ac2b12] DiffEqJump 6.1.1+
[8f2b45d5-b17b-5532-9e92-98ae0077e2e3] DiffEqMachineLearning 0.1.0
[78ddff82-25fc-5f2b-89aa-309469cbf16f] DiffEqMonteCarlo 0.15.1
[77a26b50-5914-5dd7-bc55-306e6241c503] DiffEqNoiseProcess 3.3.1
[9fdde737-9c7f-55bf-ade8-46b3f136cc48] DiffEqOperators 3.5.0
[055956cb-9e8b-5191-98cc-73ae4a59e68a] DiffEqPhysics 3.2.0
[a077e3f3-b75c-5d7f-a0c6-6bc4c8ec64a9] DiffEqProblemLibrary 4.3.0
[41bf760c-e81c-5289-8e54-58b1f1f8abe2] DiffEqSensitivity 3.3.0
```

```
[6d1b261a-3be8-11e9-3f2f-0b112a9a8436] DiffEqTutorials 0.1.0
[0c46a032-eb83-5123-abaf-570d42b7fbaa] DifferentialEquations 6.6.0
[31c24e10-a181-5473-b8eb-7969acd0382f] Distributions 0.20.0
[e30172f5-a6a5-5a46-863b-614d45cd2de4] Documenter 0.23.0
[587475ba-b771-5e3f-ad9e-33799f191a9c] Flux 0.8.3
[f6369f11-7733-5829-9624-2563aa707210] ForwardDiff 0.10.3+
[ba82f77b-6841-5d2e-bd9f-4daf811aec27] GPUifyLoops 0.2.5
[c91e804a-d5a3-530f-b6f0-dfbca275c004] Gadfly 1.1.0
[bc5e4493-9b4d-5f90-b8aa-2b2bcaad7a26] GitHub 5.1.1
[7073ff75-c697-5162-941a-fcdaad2a7d2a] IJulia 1.18.1
[42fd0dbc-a981-5370-80f2-aaf504508153] IterativeSolvers 0.8.1
[033835bb-8acc-5ee8-8aae-3f567f8a3819] JLD2 0.1.2
[e5e0dc1b-0480-54bc-9374-aad01c23163d] Juno 0.7.0
[2d691ee1-e668-5016-a719-b2531b85e0f5] LIBLINEAR 0.5.1
[7f56f5a3-f504-529b-bc02-0b1fe5e64312] LSODA 0.4.0
[6f1fad26-d15e-5dc8-ae53-837a1d7b8c9f] Libtask 0.3.0
[c7f686f2-ff18-58e9-bc7b-31028e88f75d] MCMCChains 0.3.10
[33e6dc65-8f57-5167-99aa-e5a354878fb2] MKL 0.0.0
[961ee093-0014-501f-94e3-6117800e7a78] ModelingToolkit 0.5.0
[4886b29c-78c9-11e9-0a6e-41e1f4161f7b] MonteCarloIntegration 0.0.1
[2774e3e8-f4cf-5e23-947b-6d7e65073b56] NLsolve 4.0.0
[8faf48c0-8b73-11e9-0e63-2155955bfa4d] NeuralNetDiffEq 0.1.0
[09606e27-ecf5-54fc-bb29-004bd9f985bf] ODEInterfaceDiffEq 3.3.1
[1dea7af3-3e70-54e6-95c3-0bf5283fa5ed] OrdinaryDiffEq 5.12.0
[65888b18-ceab-5e60-b2b9-181511a3b968] ParameterizedFunctions 4.2.0
[14b8a8f1-9102-5b29-a752-f990bacb7fe1] PkgTemplates 0.6.1
[91a5bcdd-55d7-5caf-9e0b-520d859cae80] Plots 0.25.3
[92933f4c-e287-5a05-a399-4b506db050ca] ProgressMeter 1.0.0
[d7b8c89e-ad89-52e0-b9fd-d0ed321fa021] Pumas 0.1.0
[b7b41870-aa11-11e9-048a-09266ec4a62f] PumasTutorials 0.0.1
[438e738f-606a-5dbb-bf0a-cddfbfd45ab0] PyCall 1.91.2
[d330b81b-6aea-500a-939a-2ce795aea3ee] PyPlot 2.8.1
[1fd47b50-473d-5c70-9696-f719f8f3bcdc] QuadGK 2.1.0
[612083be-0b0f-5412-89c1-4e7c75506a58] Queryverse 0.3.1
[6f49c342-dc21-5d91-9882-a32aef131414] RCall 0.13.3
[731186ca-8d62-57ce-b412-fbd966d074cd] RecursiveArrayTools 0.20.0
[37e2e3b7-166d-5795-8a7a-e32c996b4267] ReverseDiff 0.3.1
[295af30f-e4ad-537b-8983-00126c2a3abe] Revise 2.1.6
[2b6d1eac-7baa-5078-8adc-e6a3e659f14f] SingleFloats 0.1.3
[47a9eef4-7e08-11e9-0b38-333d64bd3804] SparseDiffTools 0.4.0
[90137ffa-7385-5640-81b9-e52037218182] StaticArrays 0.11.0
[4c63d2b9-4356-54db-8cca-17b64c39e42c] StatsFuns 0.8.0
[f3b207a7-027a-5e70-b257-86293d7955fd] StatsPlots 0.11.0
[9672c7b4-1e72-59bd-8a11-6ac3964bc41f] SteadyStateDiffEq 1.5.0
[789caeaf-c7a9-5a7d-9973-96adeb23e2a0] StochasticDiffEq 6.6.0
[c3572dad-4567-51f8-b174-8c6c989267f4] Sundials 3.6.1
[fd094767-a336-5f1f-9728-57cf17d0bbfb] Suppressor 0.1.1
[6fc51010-71bc-11e9-0e15-a3fcc6593c49] Surrogates 0.1.0
[9f7883ad-71c0-57eb-9f7f-b5c9e6d3789c] Tracker 0.2.2
```

```
[fce5fe82-541a-59a6-adf8-730c64b5f9a0] Turing 0.6.18
[1986cc42-f94f-5a68-af5c-568840ba703d] Unitful 0.16.0
[44d3d7a6-8a23-5bf8-98c5-b353f8df5ec9] Weave 0.9.1
[e88e6eb3-aa80-5325-afca-941959d7151f] Zygote 0.3.2
```