Introduction to Pumas

Chris Rackauckas

July 19, 2019

1 Introduction

This is an introduction to Pumas, a software for pharmacometric modeling and simulation.

The basic workflow of Pumas is:

1. Build a model.
2. Define subjects or populations to simulate or estimate.
3. Analyze the results with post-processing and plots.

We will show how to build a multiple-response PK/PD model via the `@model` macro, define a subject with multiple doses, and analyze the results of the simulation. This tutorial is made to be a broad overview of the workflow and more in-depth treatment of each section can be found in the subsequent tutorials and documentation.

1.1 Installation

To install Pumas, first install JuliaPro. Then use the command:

```
using Pkg
Pkg.add("Pumas")
```

Doing it this way, Pumas and its dependencies will install automatically.

1.2 Getting Started

To load the package, use

```
using Pumas
```

1.3 Using the Model Macro

Now let's define a model. A model is defined in an `@model` block. Inside of this block we have a few subsections. The first of which is `@param`. In here we define what kind of parameters we have. For this model we will define a vector parameter θ of size 12:

```
@param begin
  θ ∈ VectorDomain(12)
end
```

Next we define our random effects. The random effects are defined by a distribution from Distributions.jl. For more information on defining distributions, please see the Distributions.jl documentation. For this tutorial, we wish to have a multivariate normal of 11 uncorrelated random effects, so we utilize the syntax:

```
using LinearAlgebra
@random begin
  η ~ MvNormal(Matrix{Float64}(I, 11, 11))
end
```

Notice that here we imported `I` from LinearAlgebra and and said that our Normal distribution's covariance is said `I`, the identity matrix.

Now we define our pre-processing step in `@pre`. This is where we choose how the parameters, random effects, and the covariates collate. We define the values and give them a name as follows:

```
@pre begin
    Ka1     = θ[1]
    CL      = θ[2]*exp(η[1])
    Vc      = θ[3]*exp(η[2])
    Q       = θ[4]*exp(η[3])
    Vp      = θ[5]*exp(η[4])
    Kin     = θ[6]*exp(η[5])
    Kout    = θ[7]*exp(η[6])
    IC50    = θ[8]*exp(η[7])
    IMAX    = θ[9]*exp(η[8])
    γ       = θ[10]*exp(η[9])
    Vmax    = θ[11]*exp(η[10])
    Km      = θ[12]*exp(η[11])
    Resp0   = θ[6]/θ[7]
end
```

Next we define the `@init` block which gives the inital values for our differential equations. Any variable not mentioned in this block is assumed to have a zero for its starting value. We wish to only set the starting value for `Resp`, and thus we use `Resp0` defined in the pre-block:

```
@init begin
    Resp = Resp0
end
```

Now we define our dynamics. We do this via the `@dynamics` block. Differential variables are declared by having a line defining their derivative. For our model, we use:

```
@dynamics begin
    Ev1'    = -Ka1*Ev1
    Cent'   =  Ka1*Ev1 - (CL+Vmax/(Km+(Cent/Vc))+Q)*(Cent/Vc)  + Q*(Periph/Vp)
    Periph' =  Q*(Cent/Vc)  - Q*(Periph/Vp)
    Resp'   =  Kin*(1-(IMAX*(Cent/Vc)^γ/(IC50^γ+(Cent/Vc)^γ)))  - Kout*Resp
end
```

Lastly we utilize the `@derived` macro to define our post-processing. We can output values using the following:

```
@derived begin
```

```
    ev1    = Ev1
    cp     = Cent / θ[3]
    periph = Periph
    resp   = Resp
end
```

The `@model` block is all of these together, giving us the following model:

```
using LinearAlgebra
model = @model begin

    @param begin
      θ ∈ VectorDomain(12)
    end

    @random begin
      η ~ MvNormal(Matrix{Float64}(I, 11, 11))
    end

    @pre begin
        Ka1     = θ[1]
        CL      = θ[2]*exp(η[1])
        Vc      = θ[3]*exp(η[2])
        Q       = θ[4]*exp(η[3])
        Vp      = θ[5]*exp(η[4])
        Kin     = θ[6]*exp(η[5])
        Kout    = θ[7]*exp(η[6])
        IC50    = θ[8]*exp(η[7])
        IMAX    = θ[9]*exp(η[8])
        γ       = θ[10]*exp(η[9])
        Vmax    = θ[11]*exp(η[10])
        Km      = θ[12]*exp(η[11])
        Resp0   = θ[6]/θ[7]
    end

    @init begin
        Resp = Resp0
    end

    @dynamics begin
        Ev1'    = -Ka1*Ev1
        Cent'   =  Ka1*Ev1 - (CL+Vmax/(Km+(Cent/Vc))+Q)*(Cent/Vc)  + Q*(Periph/Vp)
        Periph' =  Q*(Cent/Vc)  - Q*(Periph/Vp)
        Resp'   =  Kin*(1-(IMAX*(Cent/Vc)^γ/(IC50^γ+(Cent/Vc)^γ)))  - Kout*Resp
    end

    @derived begin
        ev1    = Ev1
        cp     = Cent / θ[3]
        periph = Periph
        resp   = Resp
    end
end
```

```
PumasModel
  Parameters: θ
  Random effects: η
  Covariates:
  Dynamical variables: Ev1, Cent, Periph, Resp
  Derived: ev1, cp, periph, resp
```

```
  Observed: ev1, cp, periph, resp
```

1.4 Building a Subject

Now let's build a subject to simulate the model with. A subject defines three components:

1. The dosage regimen
2. The covariates of the indvidual
3. Observations associated with the individual.

Our model did not make use of covariates so we will ignore (2) for now, and (3) is only necessary for fitting parameters to data which will not be covered in this tutorial. Thus our subject will be defined simply by its dosage regimen.

To do this, we use the `DosageRegimen` constructor. It uses terms from the NMTRAN format to specify its dose schedule. The first value is always the dosing amount. Then there are optional arguments, the most important of which is `time` which specifies the time that the dosing occurs. For example,

```
DosageRegimen(15, time=0)
```

```
Pumas.DosageRegimen(1×9 DataFrames.DataFrame. Omitted printing of 2 columns
 Row   time      cmt     amt       evid   ii        addl    rate
       Float64   Int64   Float64   Int8   Float64   Int64   Float64


 1     0.0       1       15.0      1      0.0       0       0.0      )
```

is a dosage regimen which simply does a single dose at time `t=0` of amount 15. If we use arrays, then the dosage regimen will be the grouping of the values. For example, let's define a dose of amount 15 at times `t=0,4,8`, and `12`:

```
regimen = DosageRegimen([15,15,15,15], time=[0,4,8,12])
```

```
Pumas.DosageRegimen(4×9 DataFrames.DataFrame. Omitted printing of 2 columns
 Row   time      cmt     amt       evid   ii        addl    rate
       Float64   Int64   Float64   Int8   Float64   Int64   Float64


 1     0.0       1       15.0      1      0.0       0       0.0
 2     4.0       1       15.0      1      0.0       0       0.0
 3     8.0       1       15.0      1      0.0       0       0.0
 4     12.0      1       15.0      1      0.0       0       0.0      )
```

Let's define our subject to have `id=1` and this multiple dosing regimen:

```
subject = Subject(id=1,evs=regimen)
```

```
Subject
  ID: 1
  Events: 4
```

1.5 Running a Simulation

The main function for running a simulation is `simobs`. `simobs` on a population simulates all of the population (in parallel), while `simobs` on a subject simulates just that subject. If we wish to change the parameters from the initialized values, then we pass them in. Let's simulate subject 1 with a set of chosen parameters:

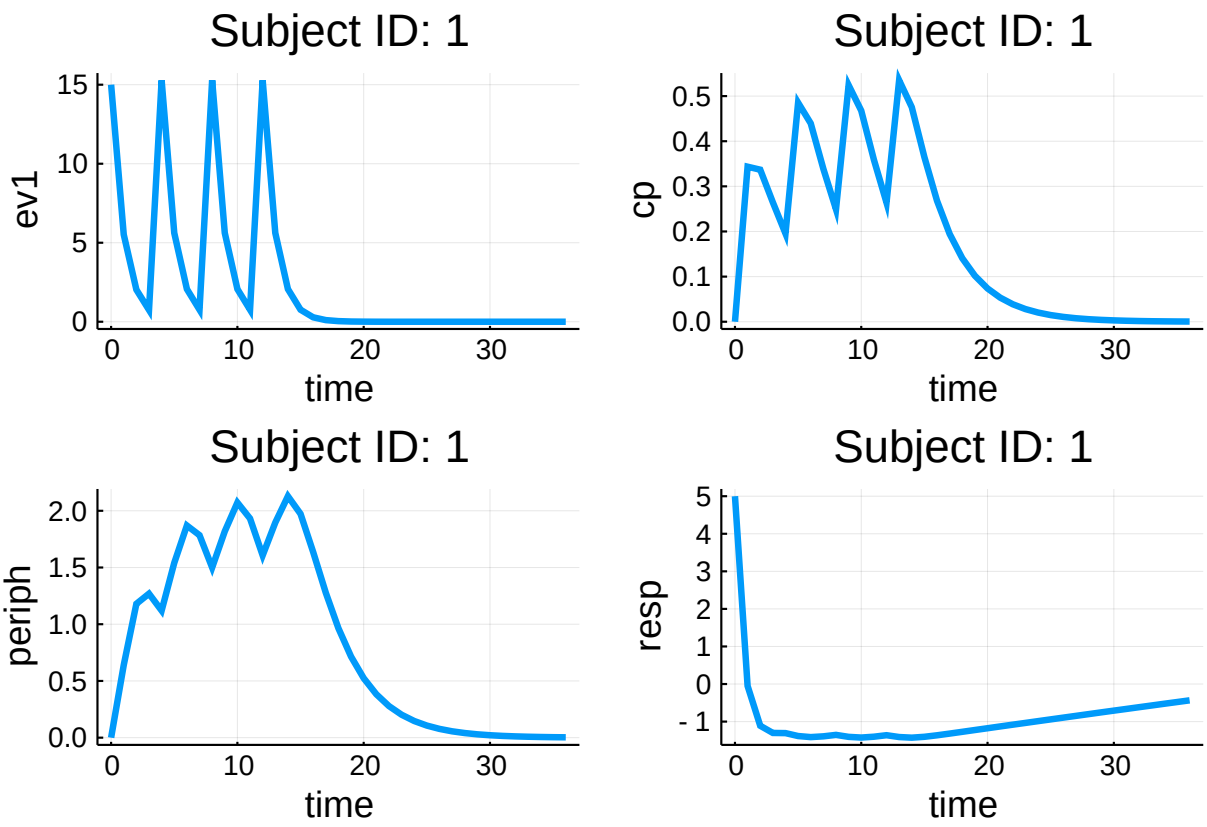
```
fixeffs = (θ = [
          1, # Ka1  Absorption rate constant 1 (1/time)
          1, # CL   Clearance (volume/time)
          20, # Vc   Central volume (volume)
          2, # Q    Inter-compartmental clearance (volume/time)
          10, # Vp   Peripheral volume of distribution (volume)
          10, # Kin  Response in rate constant (1/time)
          2, # Kout Response out rate constant (1/time)
          2, # IC50 Concentration for 50% of max inhibition (mass/volume)
          1, # IMAX Maximum inhibition
          1, # γ    Emax model sigmoidicity
          0, # Vmax Maximum reaction velocity (mass/time)
          2  # Km   Michaelis constant (mass/volume)
          ],)

sim = simobs(model, subject, fixeffs)
```

```
Pumas.SimulatedObservations{Pumas.Subject{Nothing,Nothing,Array{Pumas.Event
,1},Nothing,Pumas.var"#10#11",StaticArrays.SArray{Tuple{1},Float64,1,1}},St
epRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64
}},NamedTuple{(:ev1, :cp, :periph, :resp),NTuple{4,Array{Float64,1}}}}(Subj
ect
  ID: 1
  Events: 4
, 0.0:1.0:36.0, (ev1 = [15.0, 5.518191882720259, 2.030031193420089, 0.74681
19118116573, 15.27474053291821, 5.61935685082457, 2.0672513212755135, 0.760
508188599704, 15.279798816573903, 5.621284991576816  …  4.691042627781331e-
6, 1.7327244969055838e-6, 6.396660736013619e-7, 2.3520384966018455e-7, 8.65
7268811425022e-8, 3.19800937332909e-8, 1.1833094032657302e-8, 4.37001031479
822e-9, 1.6067588670958988e-9, 5.936724019160458e-10], cp = [0.0, 0.3434865
057232563, 0.3369044700634672, 0.2645384450491648, 0.19549055526282713, 0.4
8538938245149676, 0.4395036088307719, 0.3387114255770889, 0.249158047422979
3, 0.5242381515509906  …  0.007753721333481907, 0.005622870257772676, 0.004
077643104304871, 0.0029570672072474604, 0.002144439364387045, 0.00155513335
91996123, 0.0011277760180663698, 0.0008178577012460216, 0.00059310504112172
89, 0.0004301168655387744], periph = [0.0, 0.6399776985024133, 1.1797803103
90396, 1.269406018640501, 1.1200035847595684, 1.540532204433437, 1.86986444
46071975, 1.7849337468671596, 1.4998585409981466, 1.8184769677599533  …  0.
056051853942083 17, 0.04065125551756263, 0.029481108737523305, 0.02137989085
8838832, 0.015504687866634965, 0.011243963653022226, 0.008154098481313578,
0.005913321813402659, 0.004288305601669675, 0.0031098594327216996], resp =
[5.0, -0.050962378984076184, -1.1108564350600163, -1.3022233015837044, -1.3
069236258878125, -1.3847349036119323, -1.4127825391260904, -1.3927682905819
843, -1.3539467866342751, -1.4064935080014027  …  -0.8476682022335202, -0.8
010036862422445, -0.7545050737930822, -0.7084502156445285, -0.6625838858857
511, -0.616656455500657, -0.5706905174235036, -0.5249889560592793, -0.47978
48981437595, -0.434524666043059]))
```

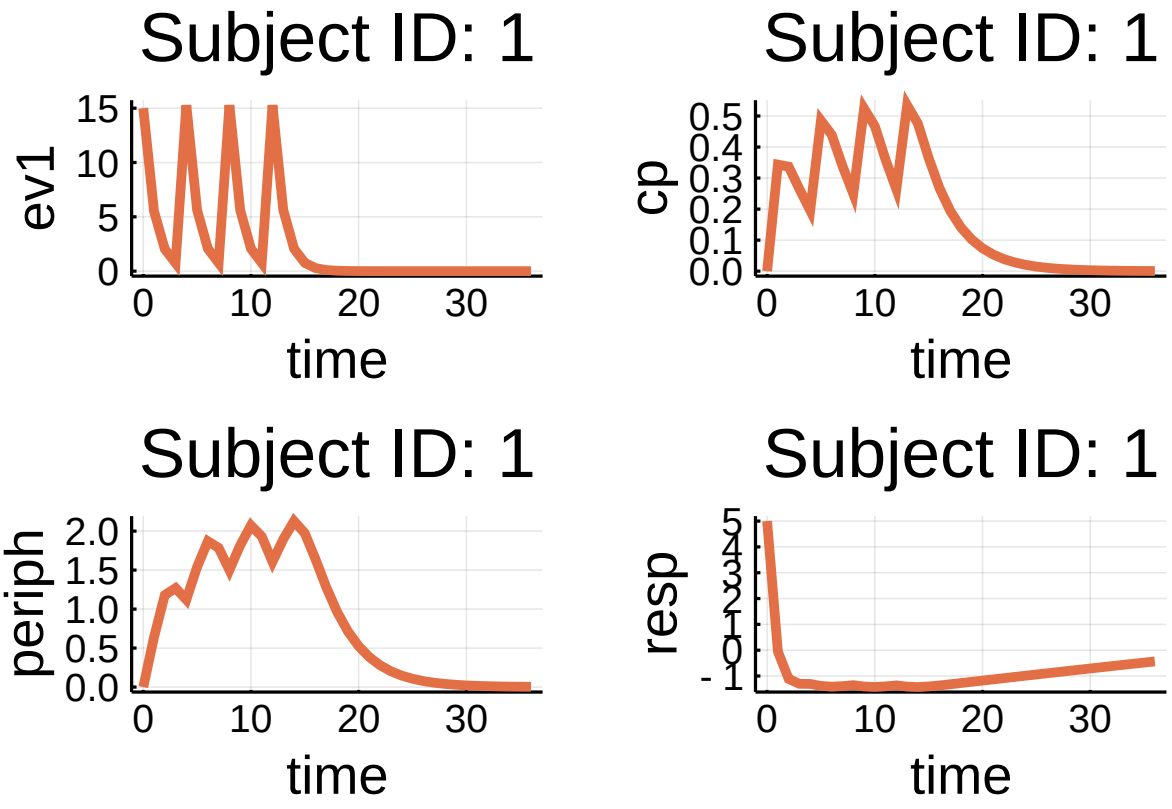
We can then plot the simulated observations by using the `plot` command:

```
using Plots
plot(sim)
```



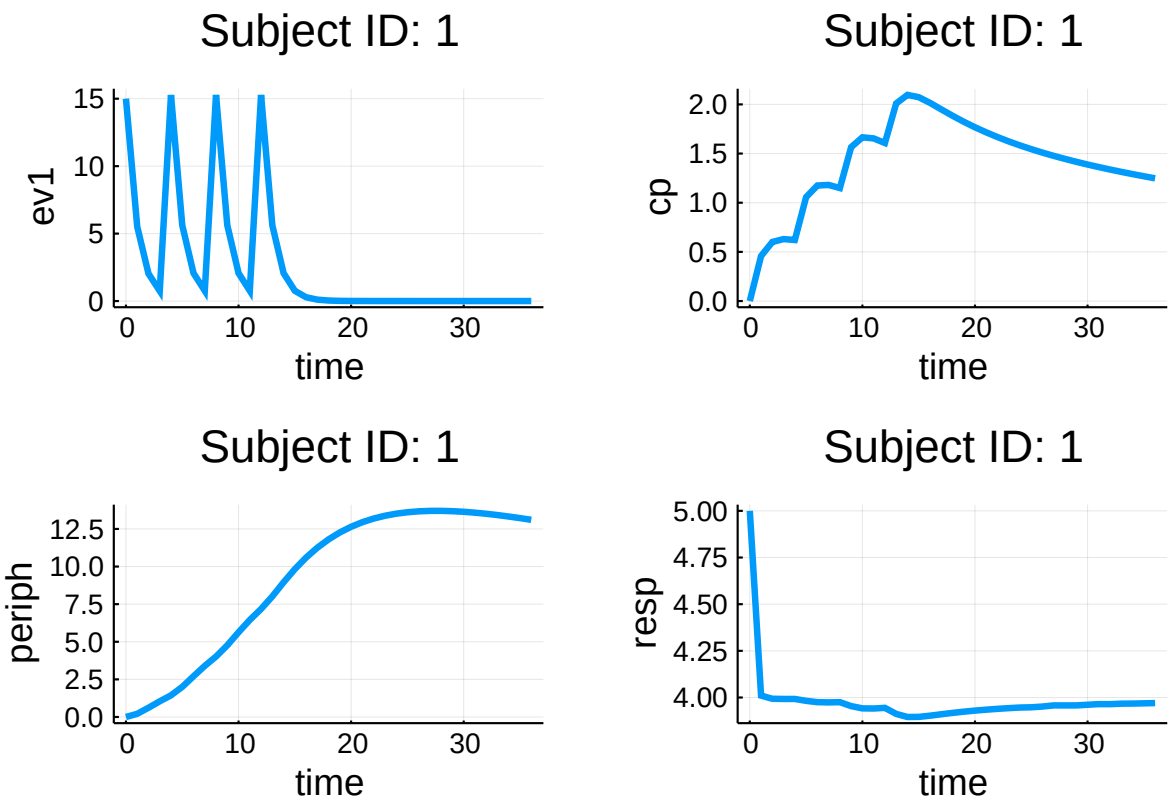
Note that we can use the attributes from `Plots.jl` to further modify the plot. For example,

```
plot(sim,
     color=2,thickness_scaling=1.5,
     legend=false, lw=2)
```



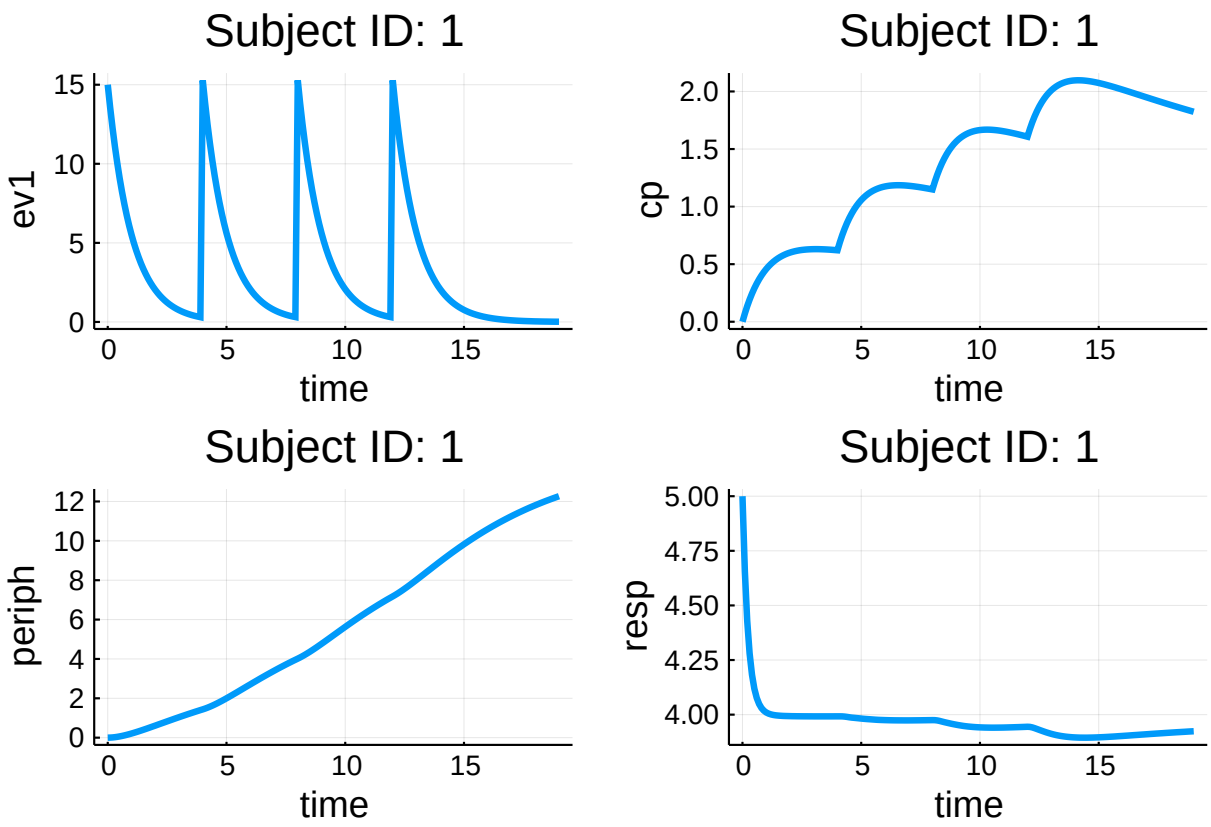
Notice that in our model we said that there was a single parameter θ so our input parameter is a named tuple with just the name θ. When we only give the parameters, the random effects are automatically sampled from their distributions. If we wish to prescribe a value for the random effects, we pass initial values similarly:

```
randeffs = (η = rand(11),)
sim = simobs(model, subject, fixeffs, randeffs)
plot(sim)
```



The points which are saved are by default at once every hour until one day after the last event. If you wish to change the saving time points, pass the keyword argument `obstimes`. For example, let's save at every `0.1` hours and run the simulation for 19 hours:

```
sim = simobs(model, subject, fixeffs, randeffs, obstimes = 0:0.1:19)
plot(sim)
```



1.6 Handling the SimulatedObservations

The resulting `SimulatedObservations` type has two fields. `sim.times` is an array of time points for which the data was saved. `sim.derived` is the result of the derived variables. From there, the derived variabes are accessed by name. For example,

```
sim[:cp]
```

```
191-element Array{Float64,1}:
 0.0
 0.07115034224030281
 0.13509732500097188
 0.19253146696864984
 0.24407750834286404
 0.290300708223127
 0.3317123655904516
 0.3687752581549087
 0.4019077522768325
 0.4314887448944013
 ⋮
 1.872206632248751
 1.865989038435363
 1.8598157051906974
 1.8536873779333063
 1.8476046852652324
 1.8415680699882997
 1.835577781694992
 1.8296340638566782
 1.8237370840709233
```

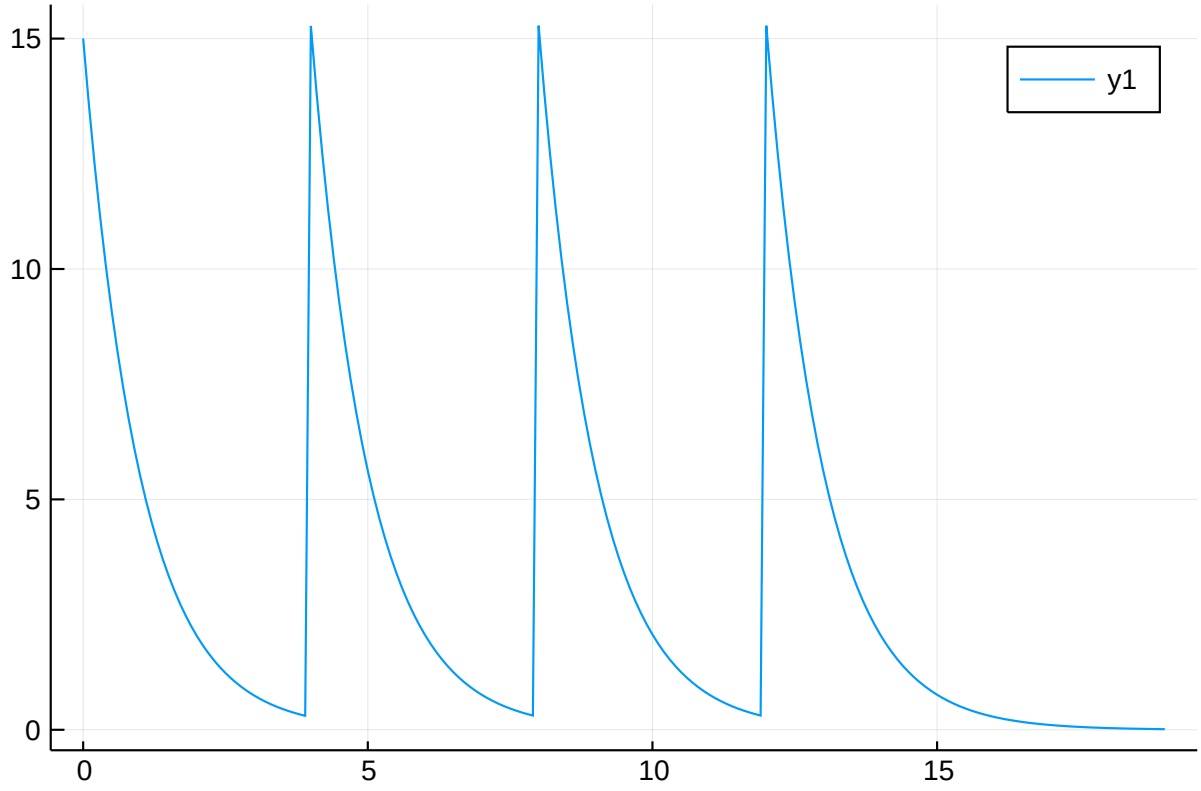
is the array of `cp` values at the associated time points. We can turn this into a DataFrame via using the `DataFrame` command:

```
df = DataFrame(sim)
first(df,6) # Print only the first 6: the DataFrame is huge!
```

	time	ev1	cp	periph	resp	amt	evid	cmt	rate
	Float64	Float64	Float64	Float64	Float64	Float64	Int8	Int64	Float64
1	0.0	15.0	0.0	0.0	5.0	15.0	1	1	0.0
2	0.0	15.0	0.0	0.0	5.0	0.0	0		0.0
3	0.1	13.5726	0.0711503	0.00289886	4.65561	0.0	0		0.0
4	0.2	12.281	0.135097	0.0111686	4.42937	0.0	0		0.0
5	0.3	11.1123	0.192531	0.0242156	4.28073	0.0	0		0.0
6	0.4	10.0548	0.244078	0.0415041	4.18305	0.0	0		0.0

From there, any Julia tools can be used to analyze these arrays and DataFrames. For example, if we wish the plot the result of `ev1` over time, we'd use the following:

```
plot(sim.times,sim[:ev1])
```



Using these commands, a Julia program can be written to post-process the program however you like!

1.7 Conclusion

This tutorial covered basic workflow for how to build a model and simulate results from it. The subsequent models will go into more detail in the components, such as:

1. More detailed treatment of specifying populations, dosage regimens, and covariates.

2. Reading in dosage regimens and observations from NMTRAN data.

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

1.8 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","introduction.jmd")
```

Computer Information:

```
Julia Version 1.3.1
Commit 2d5741174c (2019-12-30 21:36 UTC)
Platform Info:
  OS: Linux (x86_64-pc-linux-gnu)
  CPU: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-6.0.1 (ORCJIT, skylake)
```

Package Information:

```
Status `/home/pkofod/.julia/dev/PumasTutorials/Project.toml`
[336ed68f-0bac-5ca0-87d4-7b16caf5d00b] CSV 0.5.24
[7073ff75-c697-5162-941a-fcdaad2a7d2a] IJulia 1.21.1
[44d3d7a6-8a23-5bf8-98c5-b353f8df5ec9] Weave 0.9.1
[b77e0a4c-d291-57a0-90e8-8db25a27a240] InteractiveUtils nothing
[37e2e46d-f89d-539d-b4ee-838fcccc9c8e] LinearAlgebra nothing
[44cfe95a-1eb2-52ea-b672-e2afdf69b78f] Pkg nothing
```