PuMaS for Multiple Response Pk/Pd

Chris Rackauckas

22nd October 2018

1 Introduction

This is an introduction to PuMaS, a software for pharmacometric modeling and simulation. In this tutorial we will show how to simulate from a multiple response PK/PD model defined by ordinary differential equations, and how to extract information about the solution.

1.1 Installation

Because PuMaS is still unregistered, you will need to give the Git repository in order to add the package. To do this, use the command `]add https://github.com/UMCTM/PuMaS.jl`. Doing it this way, PuMaS and its dependencies will install automatically. If one cannot authenticate for this command (since the repository is currently private!), then first clone the repository how you please, use `]dev path/to/package`, then then do `]build PuMaS`. Using the build command will download and install the dependencies.

1.2 Getting Started

To load the package, use

```
using PuMaS
```

1.3 Our Example Model

1.4 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])
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:

```
@init begin
    Resp = θ[6]/θ[7]
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])
    end

    @init begin
        Resp = θ[6]/θ[7]
    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
```

1.5 Grabbing Data

In this tutorial we will utilize an example dataset in NMTRAN form. To bring in data we utilize the `process_nmtran` function. The first argument is the path to the dataset. The path to our example data is `example_nmtran_data("event_data/data23")`. Next we give it the name of the covariate from the dataset. Our example data has no covariates, so we note this as `[]`. Lastly, we declare the names of the columns which correspond to the observation variables. In this dataset, we have observations for all of `[:ev1,:cp,:periph,:resp]`. Together, this gives us the command:

```
pop = process_nmtran(example_nmtran_data("event_data/data23"),[],
    [:ev1,:cp,:periph,:resp])
```

That gives us a whole population. If we wish to grab a subject out of the population, we simply index the population. Let's grab the first subject:

```
subject = pop[1]
```

1.6 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:

```
x0 = (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, x0)
```

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:

```
y0 = (η = zeros(11),)
sim = simobs(model, subject, x0, y0)
```

The points which are saved automatically match the observations from the dataset. If you wish to change the saving time points, or did not have observations in your dataset, 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, x0, y0, obstimes = 0:0.1:19)
```

1.7 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.derived.cp
```

```
191-element Array{Float64,1}:
 0.0
 0.47221914014817434
```

```
0.892563879689724
1.2661629995273587
1.5976533425197088
1.8912202652117394
2.1506525123265425
2.379365921095179
2.5804446918538138
2.7566848994999047
⋮
1.7208030796955243
1.7151514165326058
1.709525592732175
1.7039252684684807
1.698350113429305
1.6927998068159575
1.6872740373432795
1.6817725009423412
1.6762949074483278
```

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

```
DataFrame(sim.derived)
```

	ev1	cp	periph	resp
1	0.0	0.0	0.0	5.0
2	90.4837	0.472219	0.0478097	4.90282
3	81.8731	0.892564	0.182936	4.68846
4	74.0818	1.26616	0.393897	4.42606
5	67.032	1.59765	0.6704	4.15103
6	60.6531	1.89122	1.00324	3.88186
7	54.8811	2.15065	1.38416	3.62842
8	49.6585	2.37937	1.80581	3.3952
9	44.9329	2.58044	2.26161	3.18391
10	40.657	2.75668	2.74568	2.99469
11	36.7879	2.91061	3.2528	2.82656
12	33.287	3.04448	3.77834	2.67809
13	30.1194	3.16035	4.31818	2.54768
14	27.2532	3.26008	4.86862	2.43359
15	24.6596	3.34535	5.42643	2.33411
16	22.3129	3.41765	5.98878	2.2477
17	20.1896	3.47835	6.55316	2.17288
18	18.2684	3.52868	7.11734	2.10831
19	16.5299	3.56976	7.67934	2.0527
20	14.9567	3.60258	8.2375	2.00499
21	13.5333	3.62802	8.79038	1.96429
22	12.2456	3.6469	9.33671	1.92974
23	11.0804	3.65996	9.87538	1.90057
24	10.026	3.66786	10.4054	1.87608
25	9.07168	3.67119	10.926	1.85568
26	8.20821	3.67048	11.4366	1.8389
27	7.42708	3.6662	11.9366	1.82534
28	6.72043	3.65877	12.4255	1.81459
29	6.08107	3.64859	12.903	1.80631
30	5.50248	3.63602	13.3688	1.80018
31	4.97873	3.62137	13.8226	1.7959
32	4.50471	3.60492	14.2643	1.79328
33	4.07588	3.5869	14.6939	1.79214
34	3.68801	3.56754	15.1113	1.79231
35	3.33719	3.54703	15.5165	1.79363
36	3.01982	3.52557	15.9095	1.79595
37	2.73259	3.50331	16.2904	1.79915
38	2.47251	3.48041	16.6593	1.80311
39	2.23701	3.457	17.0162	1.80774
40	2.02392	3.43317	17.3614	1.81297
41	1.8312	3.40902	17.695	1.81873
42	1.65695	3.38463	18.0173	1.82496
43	1.49941	3.3601	18.3283	1.83159
44	1.35692	3.33548	18.6283	1.83857
45	1.22797	3.31086	18.9175	1.84586
46	1.11117	3.28627	19.1961	1.8534
47	1.00531	3.26179	19.4642	1.86116
48	0.909441	3.23743	19.7222	1.8691
49	0.822713	3.21324	19.9703	1.87721
50	0.744316	3.18923	20.2087	1.88546
51	0.673482	3.16544	20.4377	1.89383
52	0.609488	3.14188	20.6575	1.9023
53	0.551654	3.11858	20.8683	1.91085

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:

```
using Plots
plot(sim.times,sim.derived.ev1)
```