NCA Tutorial

Yingbo Ma

November 30, 2018

1 Introduction

This is an introduction to `NCA.jl`, a software for noncompartmental analysis (NCA). In this tutorial we will show how to use `NCA.jl` to analysis data.

1.1 Installation

Currently, `NCA.jl` is a submodule in `PuMaS.jl`, so you only need to install `PuMaS.jl`, and everything will be ready to go.

1.2 Getting Started

To load the package, use

```
using PuMaS.NCA
```

First, let's load the example NCA data inside `PuMaS.jl`. This data have 24 individuals, and each of them has 16 data points. Here, we are creating two functions `concs(i)` and `times(i)` which give the concentration timeseries of individual `i`.

```
using PuMaS, CSV

const example_nca_data = CSV.read(example_nmtran_data("nca_test_data/dapa_IV"))
concs(i) = Float64.(example_nca_data[:CObs])[16(i-1)+1:16*i]
times(i) = Float64.(example_nca_data[:TIME])[16(i-1)+1:16*i]
```

2 Efficient Computation of Multiple NCA Diagnostics

2.1 AUC and AUMC

We can compute the area under the curve (AUC) from the first observation time to infinity. Below we are accessing the concentration and corresponding time array for the first individual.

```
auc(concs(1), times(1))
```

```
263.792662196049
```

```
auc(concs(1), times(1), method=:linuplogdown)
```

```
257.8586273987722
```

the keyword argument `method` can be `:linear`, `:linuplogdown`, or `:linlog`, and it defaults to `:linear`. This is a simple interface, however it is not efficient if you want to compute many quantities. The recommended way is to create an `NCAdata` object first and then call the respective NCA diagnostic on the data object:

```
nca = NCAdata(concs(1), times(1))
auc(nca)
```

```
263.792662196049
```

Here, `nca` is a lazy data structure and actual computations are not performed. When we are instantiating `NCAdata`, it only performs data checking and cleaning.

Let's create `NCAdata(concs(i),times(i))` for each individual `i` from 1 to 24. To do that, we would broadcast the whole expression over the individual indices we'd want to use, like:

```
ncas = @. NCAdata(concs(1:24), times(1:24));
```

`ncas` is an array of `NCAdata`s and its `i`th entry is the data for the `i`th subject.

`AUClast` is the area under the curve from the first observation to the last observation. To compute `AUClast` on the first individual, one would do:

```
ncas = @. NCAdata(concs(1:24), times(1:24), dose=5000.)
auc(ncas[1], auctype=:AUClast)
```

```
(auc = 246.932184, auc_dn = 0.0493864368)
```

Note that if `dose` is provided, `auc` will return a tuple, which is in the form of (AUC, normalized AUC).

Or to compute the AUC on every individual, once would do:

```
@. auc(ncas, auctype=:AUClast)
```

```
24-element Array{NamedTuple{(:auc, :auc_dn),Tuple{Float64,Float64}},1}:
 (auc = 246.932184, auc_dn = 0.0493864368)
 (auc = 302.24594, auc_dn = 0.060449188)
 (auc = 288.579555, auc_dn = 0.05771591100000001)
 (auc = 333.80397999999997, auc_dn = 0.066760796)
 (auc = 129.06118400000003, auc_dn = 0.025812236800000006)
 (auc = 291.9510375, auc_dn = 0.0583902075)
 (auc = 333.99432249999995, auc_dn = 0.06679886449999999)
 (auc = 259.9668375, auc_dn = 0.0519933675)
 (auc = 233.64285700000002, auc_dn = 0.0467285714)
 (auc = 242.71946250000002, auc_dn = 0.048543892500000005)
 ⋮
 (auc = 196.6721735, auc_dn = 0.039334434700000004)
 (auc = 319.29748499999994, auc_dn = 0.06385949699999999)
 (auc = 185.39912850000002, auc_dn = 0.037079825700000006)
 (auc = 403.216205, auc_dn = 0.080643241)
 (auc = 202.639537, auc_dn = 0.0405279074)
 (auc = 222.76995, auc_dn = 0.04455399)
 (auc = 364.75571750000006, auc_dn = 0.07295114350000001)
 (auc = 265.6632825, auc_dn = 0.05313265649999999)
 (auc = 156.80645950000002, auc_dn = 0.031361291900000005)
```

One can also compute AUC on a certain interval. To compute AUC on the interval $[10, \infty]$ on the first individual

```
auc(ncas[1], interval=(10,Inf))
```

```
(auc = 27.824427196048966, auc_dn = 0.005564885439209793)
```

One can also specify multiple intervals

```
auc(ncas[1], interval=[(10,Inf), (10, 15)])
```

```
2-element Array{NamedTuple{(:auc, :auc_dn),Tuple{Float64,Float64}},1}:
 (auc = 27.824427196048966, auc_dn = 0.005564885439209793)
 (auc = 4.6593795, auc_dn = 0.0009318759)
```

In many cases, the AUC commands may need to extrapolate in order to cover the desired interval. To see the percentage of extrapolation ($\frac{\text{extrapolated AUC}}{\text{Total AUC}} \cdot 100$), you can use the command:

```
auc_extrap_percent(ncas[1])
```

```
6.391564517256502
```

Area under the first moment of the concentration (AUMC) is

$$\int_{t_0}^{t_1} t \cdot \text{concentration}(t) dt. \tag{1}$$

The interface of computing AUMC is exactly the same with AUC, and one needs to change `auc` to `aumc` for calculating AUMC or related quantities. For instance,

```
aumc_extrap_percent(ncas[1])
aumc(ncas[1])
```

```
(aumc = 1411.6198735770822, aumc_dn = 0.2823239747154164)
```

2.2 Terminal Rate Constant (λz)

The negative slope for concentration vs time in log-linear scale is the terminal rate constant, often denoted by λz. To compute λz, one can call

```
lambdaz(ncas[1])
```

```
(lambdaz = 0.03876710923615265, points = 5, r2 = 0.7444781209375907)
```

`lambdaz` returns a tuple in the form of $(\lambda z, \text{the number of data points used}, r^2)$, where r^2 is the coefficient of determination from the linear regression of the slope determination. By default, it checks last 10 or less data points, one can change it by providing the keyword `threshold`, e.g.

```
lambdaz(ncas[1], threshold=15)
```

```
(lambdaz = 0.03876710923615265, points = 5, r2 = 0.8033543979137503)
```

One can also specify the exact data points by passing their indices

```
lambdaz(ncas[1], idxs=[10, 15, 16])
```

```
(lambdaz = 0.10617388957053892, points = 3, r2 = 0.9455398464554603)
```

You can also pass their time points

```
lambdaz(ncas[1], slopetimes=[1,2,3])
```

```
(lambdaz = 0.5387479621404708, points = 3, r2 = 0.9870369564177864)
```

2.3 Simple functions

`T_max` is the time point at which the maximum concentration (`C_max`) is observed, and they can be computed by:

```
tmax(ncas[1])
cmax(ncas[1])
cmax(ncas[1], interval=(20, 24))
cmax(ncas[1], interval=[(20, 24), (10, 15)])
```

```
2-element Array{NamedTuple{(:cmax, :cmax_dn),Tuple{Float64,Float64}},1}:
 (cmax = 0.653632, cmax_dn = 0.0001307264)
 (cmax = 1.10532, cmax_dn = 0.000221064)
```

Note that `cmax` returns `C_max` and normalized `C_max` if `dose` is provided. If `dose` is provided in the `NCAdata`, that `dose` will be used by all computations where dose can be used.

`T_last` is the time of the last observed concentration value above the lower limit of quantization (LLQ), and the corresponding concentration value is (`C_last`). They can be computed by the command

```
tlast(ncas[1])
clast(ncas[1])
```

```
0.653632
```

By default, LLQ is 0, and concentrations that are below LLQ (BLQ) are dropped. If you wish to keep BLQ data and change LLQ to 0.5, you can do

```
NCAdata(concs(1), times(1), llq=0.5, concblq=:keep)
```

```
NCAdata:
  concentration: Float64
  time:          Float64
  auc:           Float64
  aumc:          Float64
  λz:            Float64
  dose:          Nothing
```

The half-life can be computed by:

```
thalf(ncas[1])
```

```
1.2865889604594312
```

One may need to interpolate or to extrapolate the concentration-time data. For example, if you wanted to interpolate the concentration at $t = 12$ using linear interpolation, you would do:

```
NCA.interpextrapconc(ncas[1], 12., interpmethod=:linear)
```

```
0.911367
```

`interpmethod` can be `:linear`, `:linuplogdown`, or `:linlog`.