IVIVC Tutorial

Shubham Maddhashiya

Sep 5, 2019

1 Introduction

This is an introduction and walk-through `IVIVC.jl`, a submodule under software `Pumas.jl`.

In this tutorial, we will establish In Vitro In Vivo Correlation model and predict Plasma Concentration Profile using `IVIVC.jl` for a given ivivc dataset.

The Basic workflow of IVIVC.jl is:

1. Read the data
2. InVitro Modeling
3. Unit Impulse Response (UIR)
4. Deconvolution of InVivo data
5. Establishing InVitro InVivo Correlation Model
6. Prediction of InVivo
7. Validation
8. Automation, Plots and Reports (High level API)

1.1 Installation

This package is under `Pumas.jl`, So if you haven't installed `Pumas.jl`. Please follow the instructions here

1.2 Getting Started

Load the package

```
using Pumas.IVIVC
```

1.3 Read the data

First of all, let's read the given data set, we will have three dataset file for InVitro, Reference InVivo, and InVivo data, one for each.

`read_vitro`, `read_uir` and `read_vivo` functions are to read and parse InVitro, Reference InVivo, and InVivo data csv file, respectively.

These functions take `path` to the data file (or `DataFrame` object of csv file) and accept keyword args for columns headers.

By Default, keyword args for `read_vitro` functions are `id=:id, time=:time, conc=:conc, formulation=:form`, for `read_uir` keyword args are `time=:time, conc=:conc, formulation=:form, dose=:dose`, and for `read_vivo` function, these are `time=:time, conc=:conc, formulation=:form, dose=:dose`.

```
vivo_data = read_vivo("./vivo_data.csv");
```

if you have a different header for any column name, you can pass column name as keyword arg. For example, if we have time column header `conc` for plasma concentration column, then we can do like this.

```
vivo_data = read_vivo("./vivo_data.csv", conc=:conc);
```

These functions return array of subject or batch sorted by their subject-id of batch-id. Each entry in this array is mapping of formulation type to corresponding data for a particular subject or batch of data.

If you have your data already in DataFrame object, you can pass it instead of `path` and rest will follow the same.

```
using CSV
df = CSV.read("./vivo_data.csv")
vivo_data = read_vivo(df);
```

Let's see how the data are packed inside `vivo_data` and take the `fast` formulation (one of the formulation types in our data) data for the first subject.

```
vivo_fast_data = vivo_data[1]["fast"]
```

```
15×5 DataFrames.DataFrame
 Row  id     time     conc     formulation  dose
      Int64  Float64  Float64  String       Int64

 1    1      0.0      0.0      fast         100
 2    1      0.5      7.219    fast         100
 3    1      1.0      39.7     fast         100
 4    1      1.5      69.29    fast         100
 5    1      2.0      92.86    fast         100
 6    1      3.0      118.3    fast         100
 7    1      4.0      107.2    fast         100
 8    1      6.0      68.77    fast         100
 9    1      8.0      45.91    fast         100
 10   1      10.0     28.1     fast         100
 11   1      12.0     18.28    fast         100
 12   1      14.0     12.2     fast         100
 13   1      16.0     8.19     fast         100
 14   1      20.0     4.447    fast         100
 15   1      24.0     2.703    fast         100
```

Time array of this data

```
vivo_fast_data.time
```

```
15-element Array{Float64,1}:
  0.0
  0.5
  1.0
  1.5
  2.0
  3.0
  4.0
  6.0
  8.0
 10.0
 12.0
 14.0
 16.0
 20.0
 24.0
```

Similar syntax follows for `read_uir` and `read_vitro` functions.

1.4 InVitro Modeling

In InVitro data, We have time series data of fraction dissolved amount of drug. Using this data, we have to fit a predefined model, which, of course will be continuous. Standard models available in IVIVC.jl are Emax, Weibull, Double Weibull and Makoid banakar. You can also use custom model by passing function of the model.

`estimate_fdiss` function takes object to vitro data and `Symbol` to model to fit (which are available in the package) or function of the same.

You can see available models using `get_avail_models` function. It returns dictionary of `Symbol` to model.

```
get_avail_models()
```

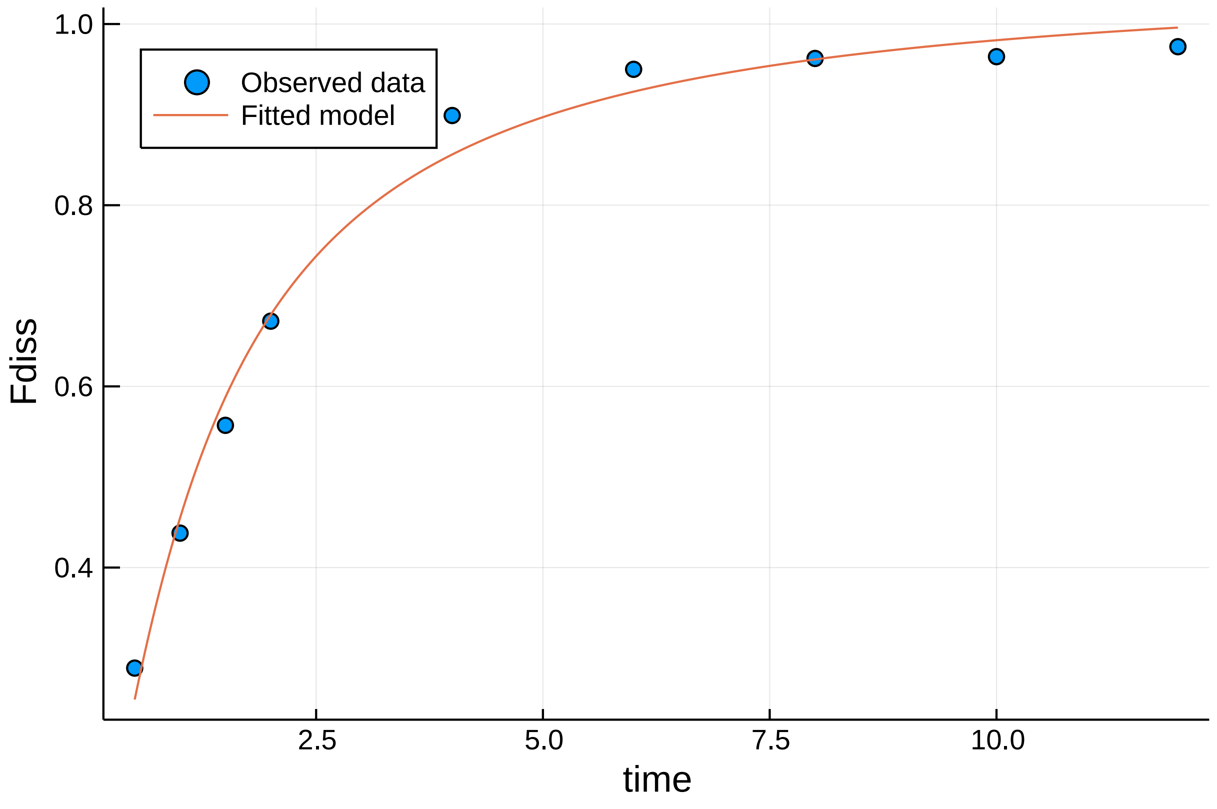
```
Dict{Symbol,Function} with 10 entries:
  :eng      => emax_ng
  :w        => weibull
  :m        => makoid
  :e        => emax
  :d_weibll => double_weibull
  :emax     => emax
  :makoid   => makoid
  :weibull  => weibull
  :emax_ng  => emax_ng
  :dw       => double_weibull
```

Let's fit Emax model to our InVitro data.

```
vitro_data = read_vitro("./vitro_data.csv");
vitro_fast_data = vitro_data[1]["fast"]
estimate_fdiss(vitro_fast_data, :emax);
```

We can call `plot` function on this instance of data to plot the fitted model.

```
using Plots
scatter(vitro_fast_data.time, vitro_fast_data.conc, label="Observed
    data", xlabel="time", ylabel="Fdiss")
plot!(vitro_fast_data, label="Fitted model", legend=:topleft)
```



Note: If you are passing function then you will also have to provide initial estimates of parameters, lower bounds and upper bounds. For example, let's fit Emax model again but passing our own function.

```
Emax_model(t, p) = @. (p[1] * (t ^ p[2])) / (p[3]^p[2] + t^p[2])
p0 = [vitro_fast_data.conc[end], 1.2, vitro_fast_data.time[2]]
lb = [0.0, 1.0, 0.0]
ub = [1.25, Inf, vitro_fast_data.time[end]]
estimate_fdiss(vitro_fast_data, Emax_model, p0=p0, lower_bound=lb, upper_bound=ub);
```

Optimized parameters are stored in `pmin` vector.

```
vitro_fast_data.pmin
```

```
3-element Array{Float64,1}:
 1.0541385268387788
 1.2547575044440769
 1.2461970228752515
```

Model the other formulations also, we will need it in later sections.

```
estimate_fdiss(vitro_data[1]["medium"], :emax)
estimate_fdiss(vitro_data[1]["slow"], :emax);
```

1.5 Unit Impulse Response (UIR)

When the reference vivo data is `Solution` type, then we can fit `bateman` function which is available in the package. In result, we will get estimation of `ka` (absorption rate constant), `ke` (elimination rate constant) and `V` (volume of distribution). We will use this estimated `kel` in Wagner Nelson method. To do this step, use `estimate_uir` function. This function takes the object to uir data, Symbol to model or custom function and fraction of available dose. Default keyword arg is `frac` and is set to 1.

```
uir_data = read_uir("./uir_data.csv")
estimate_uir(uir_data, :bateman);
```

To change fraction of available dose.

```
estimate_uir(uir_data, :bateman, frac=1.0);
```

All the estimates parameters are stored in `pmin` vector, similar to others. Note: the first entry is `ka`, the second is `kel`, and the third one is `V`.

```
ka, kel, V = uir_data.pmin
@show ka

ka = 0.5261486841547705

@show kel

kel = 0.395958080194541

@show V;

V = 0.4005347674226047
```

1.6 Deconvolution of InVivo data

Plasma Concentration profile is consist of absorption and elimination process, and from this, process to get absorption profile is known as deconvolution. In this tutorial, we will use the Wagner Nelson method, one of the traditional methods but is the most frequently used method in IVIVC. In a later tutorial, we will see more advanced methods like numerical deconvolution and Differential Equation based methods. To do this step, we will use `estimate_fabs` function, which takes `conc`, `time`, `kel` and `Symbol` to the deconvolution method. Symbol to Wagner Nelson method is `:wn`. In case of WN method, `estimate_fabs` returns `FAbs` profile and `AUC_Inf` (Area under plasma conc. curve from `time=0` to `time=Inf`).

Let's apply the Wagner Nelson method on `fast` formulation data of the first subject.

```
time = vivo_fast_data.time
conc = vivo_fast_data.conc
fabs, conc_auc = estimate_fabs(conc, time, kel, :wn)
@show fabs

fabs = [0.0, 0.0247988, 0.140846, 0.267062, 0.39091, 0.601105, 0.705957, 0.
803628, 0.87411, 0.910041, 0.936749, 0.955469, 0.968171, 0.987752, 1.0]

@show conc_auc;

conc_auc = 807.9599802139458
```

we will need `FAbs` for all formulation in next step, so let's get it now.

```
all_fabs_dict = Dict(); all_conc_auc = Dict()
for (formulation, profile) in vivo_data[1]
  all_fabs_dict[formulation], all_conc_auc[formulation] = estimate_fabs(profile.conc,
                                                             profile.time, kel, :wn)
end
```

1.7 Establishing InVitro InVivo Correlation Model

Now, we have arrived at the main step, for which we did all the above steps from reading the data to deconvolution. Currently, there are three models available in the package, which includes time and amplitude scaling and shifting parameters.

$$Fabs = AbsScale * Fdiss(t * Tscale) \tag{1}$$
$$Fabs(t) = AbsScale * Fdiss(t * Tscale - Tshift) \tag{2}$$
$$Fabs(t) = AbsScale * Fdiss(t * Tscale - Tshift) - AbsBase \tag{3}$$

Let's establish first IVIVC model, which has two parameters (AbsScale and Tscale) to estimate.

```
ivivc_model = (form, time, x) -> x[1] * vitro_data[1][form](time * x[2])
# initial estimates of parameters, upper_bounds and lower_bounds
p0 = [0.8, 0.5]
ub = [1.25, 1.25]
lb = [0.0, 0.0];
```

Now, we will define our MSE (mean square error) function which we will optimize using `Optim.jl` package.

```
mse(x, y) = sum(abs2.(x .- y))/length(x)
function errfun(x)
  err = 0.0
  for (form, prof) in vivo_data[1]    # (formulation => profile)
    err = err + mse(ivivc_model(form, prof.time, x), all_fabs_dict[form])
  end
  return err
end

errfun (generic function with 1 method)
```

Load the `Optim.jl`. If you don't have it, run this piece of code to install the pkg. `import Pkg; Pkg.add("Optim.jl")`

```
using Optim

od = Optim.OnceDifferentiable(p->errfun(p), p0, autodiff=:finite)
mfit = Optim.optimize(od, lb, ub, p0, Fminbox(LBFGS()))
pmin = Optim.minimizer(mfit)

2-element Array{Float64,1}:
 1.0536918865064493
 0.5991956451927721
```

Estimated `AbsScale` and `Tscale` are

```
AbsScale, Tscale = pmin
@show AbsScale

AbsScale = 1.0536918865064493

@show Tscale;

Tscale = 0.5991956451927721
```

1.8 Prediction of InVivo

Now, We have established IVIVC model; we can predict Plasma Concentration Profile for a formulation. Let's how well our model is to predict plasma conc. profile of fast formulation. Since, we have InVitro data of fast formulation, we first model this data to get a continuous relation between `time` and `Fdiss` which we already did in InVitro Modeling step. Note: for now, we are not predicting on external data.

After this, we plug this `estimated Fdiss` to our IVIVC model and integrate the model over vivo time interval to predicted plasma conc. profile.

Let's go into little bit mathematics of Wagner Nelson method and derive main equation for prediction.

Our Main governing differential equation is (assuming first order elimination)

$$\frac{dc}{dt} = r(t) - kel * c \tag{4}$$

where `r(t)` is input rate.

By Wagner Nelson method, we get the following expression

$$Fabs = \frac{c + kel * \int_0^t c(t)dt}{kel * \int_0^\infty c(t)dt} \tag{5}$$

And by our IVIVC model.

$$Fabs = AbsScale * Fdiss(t * Tscale) \tag{6}$$

Using the above three equations, we can write the following expression

$$\frac{dc}{dt} = kel * AUC_0^\infty * AbsScale * \frac{d(Fdiss(t * Tscale))}{dt} - kel * c \tag{7}$$

Above equation can be further simplified.

$$\frac{dc}{dt} = kel * AUC_0^\infty * AbsScale * (Tscale * g(t * Tscale)) - kel * c \tag{8}$$

where `g(t)` is derivative of vitro model function and derivatives of all available vitro models are also available in the package.

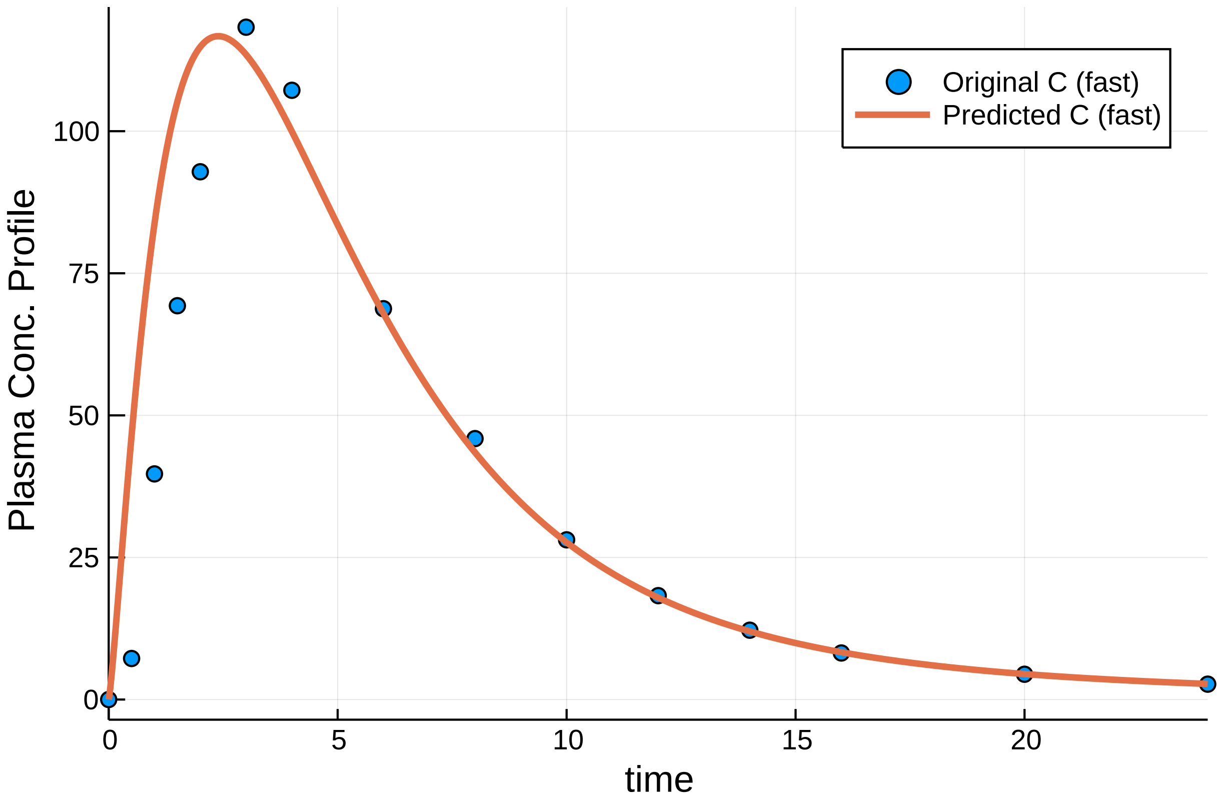
So, to predict `c`, we just need to integrate the above differential equation with the initial value of `c` equals to `0`. We will do the integration using `OrdinaryDiffEq.jl` package. To install it, do this, `import Pkg; Pkg.add("OrdinaryDiffEq.jl")`.

```
using OrdinaryDiffEq
# let grab the derivative of Emax function
import Pumas.IVIVC: e_der
g = e_der
pmin = vitro_data[1]["fast"].pmin
f(c, p, t) = kel * conc_auc * AbsScale * Tscale * g(t * Tscale, pmin) - kel * c
c0 = 0.0
tspan = (vivo_fast_data.time[1], vivo_fast_data.time[end])
prob = ODEProblem(f, c0, tspan)
sol = OrdinaryDiffEq.solve(prob, Tsit5(), reltol=1e-8, abstol=1e-8)
sol
```

```
retcode: Success
Interpolation: specialized 4th order "free" interpolation
t: 170-element Array{Float64,1}:
  0.0
  1.628446313980051e-6
  2.335026408715909e-6
  4.540664750162709e-6
  6.603042173065821e-6
  1.0024046746807039e-5
  1.4194970391299074e-5
  2.0024621855015842e-5
  2.7509726772454322e-5
  3.73783530414721e-5
  ⋮
 21.435549948096465
 21.76133743345786
 22.089953360365502
 22.42146286543184
 22.755930225602135
 23.09341880000814
 23.433990966064517
 23.777708060271102
 24.0
u: 170-element Array{Float64,1}:
 0.0
 7.3939620214305604e-6
 1.1820100647515016e-5
 2.7681165649215754e-5
 4.4491293382634916e-5
 7.535984078787327e-5
 0.00011679641388520761
 0.00018004429941138048
 0.0002683566306503289
 0.0003944041309281985
 ⋮
 3.716774686913834
 3.568707511988569
 3.4279157149007813
 3.2939704451123886
 3.166471187768947
 3.0450437999395357
 2.929338688031049
 2.819029112246563
 2.750929699638005
```

Now, Let's plot the predicted plasma conc. profile and original plasma conc. profile, so that we can visualize the performance of our IVIVC model.

```
scatter(vivo_fast_data.time, vivo_fast_data.conc, label="Original C (fast)")
plot!(sol, label="Predicted C (fast)", xlabel="time", ylabel="Plasma Conc.  Profile")
```



1.9 Validation

Validation is a very important step to see how well our IVIVC model is performing against internal data (known as Internal Validation) and External data (External Validation). We can see the error in predicting `Cmax` and `AUC` of plasma conc. profile. `percentage_prediction_error` function which is available in the pkg, a handy function to calculate `Cmax` and `AUC` error. According to the FDA guidelines, the predictability can be accepted when the average absolute percentage error in `Cmax` and `AUC` are 10% or less and for every formulation, are of 15% or less.

```
cmax_pe, auc_pe = percentage_prediction_error(vivo_fast_data.time, vivo_fast_data.conc,
                                               sol.t, sol.u)    # sol.u is just predicted c
@show cmax_pe
```

```
cmax_pe = 1.3392227186542658
```

```
@show auc_pe;
```

```
auc_pe = 6.177005827057335
```

1.10 Automation, Plots and Reports

We have gone through all the necessary steps to do IVIVC using the package, but now I will tell you `one-liner` to do all the steps from 2 to 5. `IVIVCModel` function will do all the above steps for you just in one line. `IVIVCModel` function takes data object for vitro, uir and vivo, and symbol to vitro model to fit. Others keyword args are `uir_frac=1.0` (fraction of dose available), `deconvo_method=:wn`, and `ivivc_model=:two` (two stands for the number of parameters in ivivc model and this is the first one of three models).

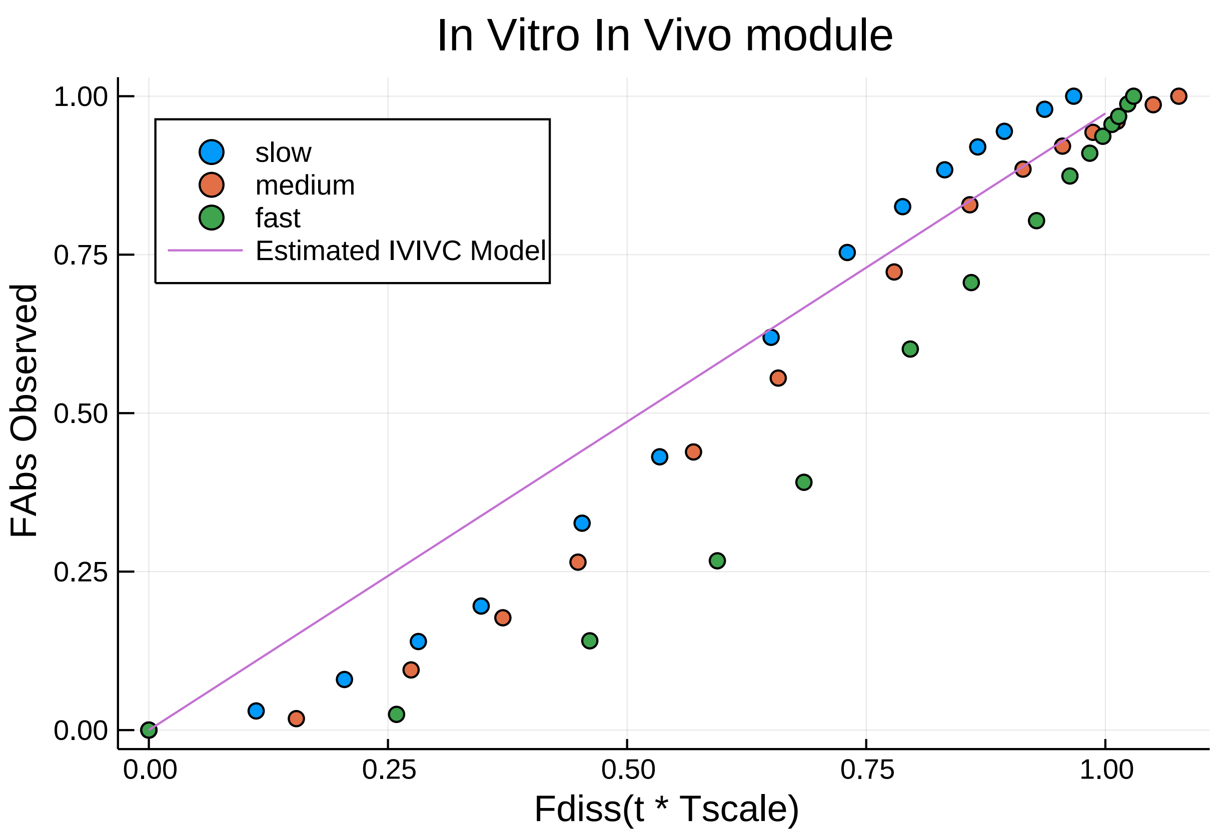
```
model = IVIVCModel(vitro_data, uir_data, vivo_data, vitro_model=:emax, uir_frac=1.0,
    deconvo_method=:wn, ivivc_model=:three);
```

Above line of code is equivalent to what we did the from step 2 to 5.

1.10.1 IVIVC Plot

It is the plot between `Fabs` and `Fdiss` with established IVIVC model. `ivivc_plot` is a convenient plot recipe to plot the ivivc plot.

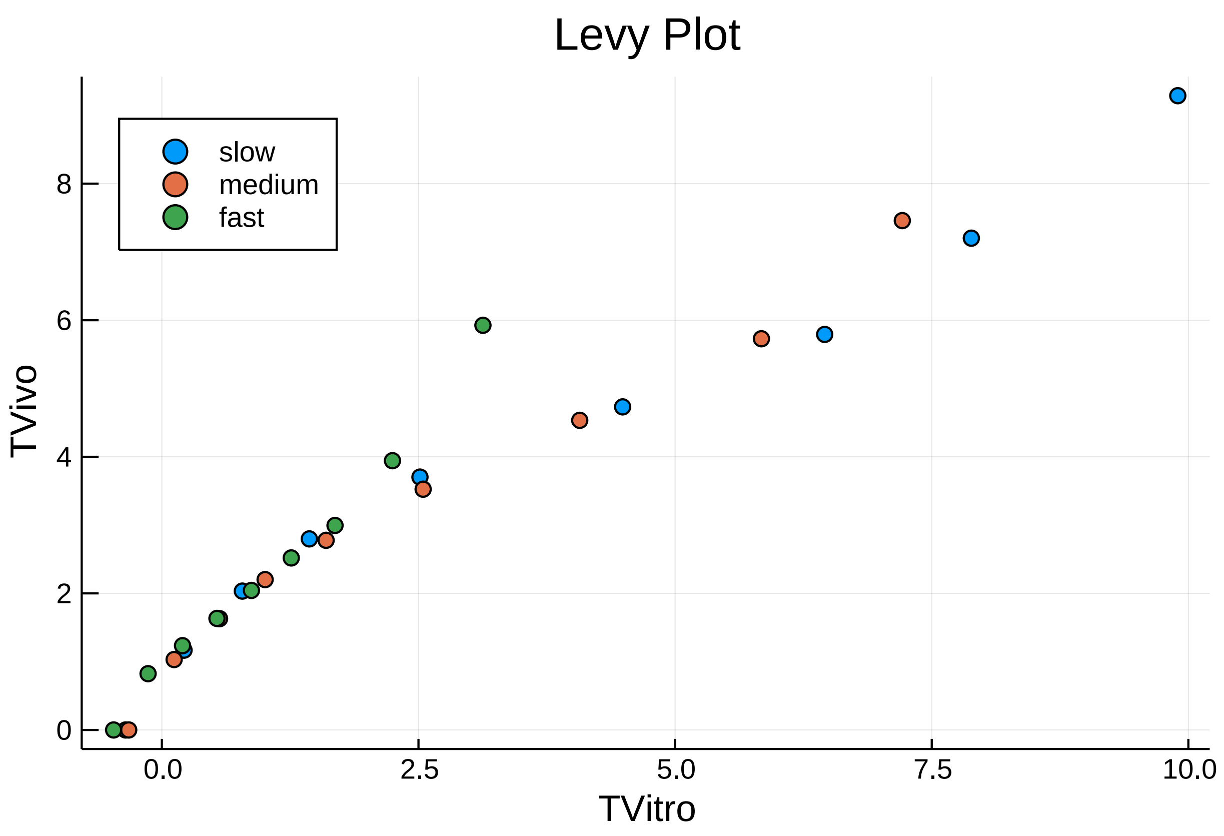
```
ivivc_plot(model)
```



1.10.2 Levy Plot

It is the plot, which is generally used in the inspecting time scale between `Fabs` and `Fdiss` profile. In this plot, we plot `TVitro` on the x-axis and `TVivo` on the y-axis at matched `Fdiss` and `Fdiss`. Note: We take time points at every 1/10th fraction of dissolved or absorbed, missing time points are filled using Linear Interpolation.

```
levy_plot(model)
```



1.10.3 Prediction and Validation

Now, let's predict the plasma conc. profile for fast formulation.

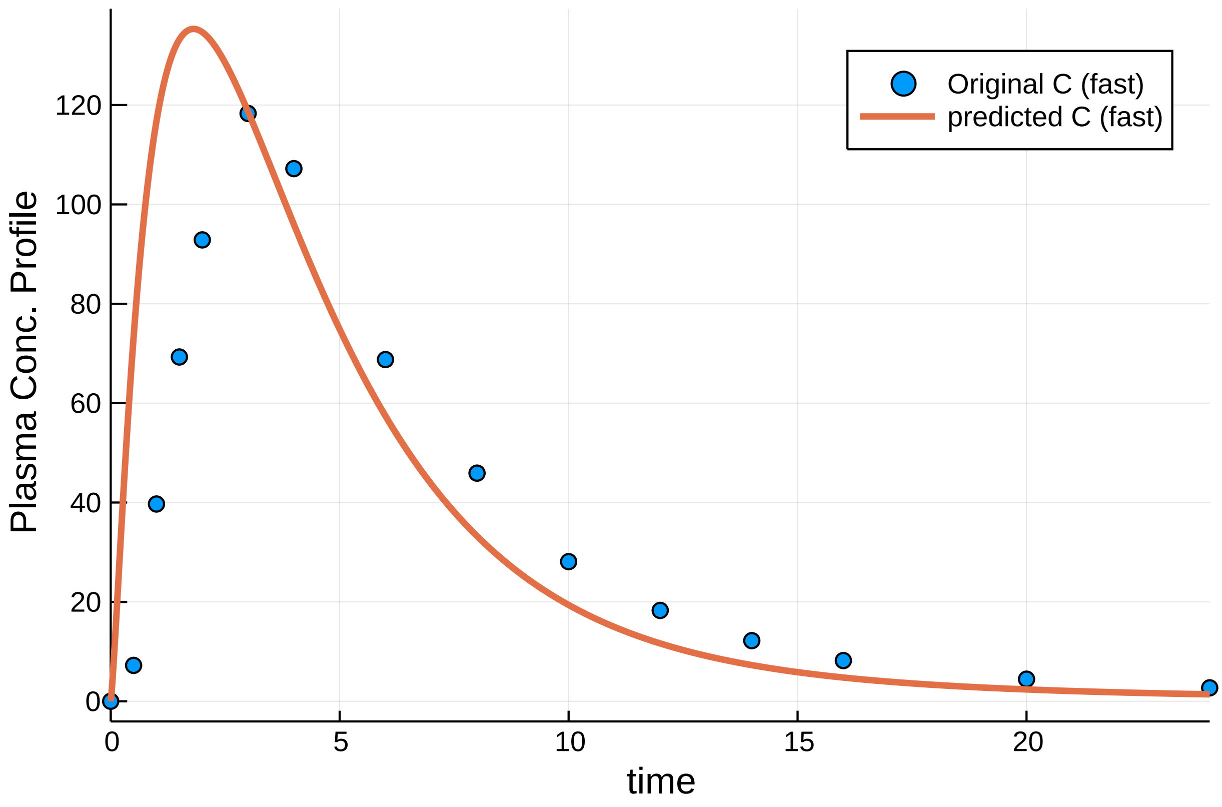
```
sol = predict_vivo(model, "fast");
```

plot the predicted plasma conc. profile vs original plasma conc. profile and `Cmax` and `AUC` predictio error.

```
time = vivo_data[1]["fast"].time
conc = vivo_data[1]["fast"].conc
cmax_pe, auc_pe = percentage_prediction_error(time, conc, sol.t, sol.u)
@show cmax_pe, auc_pe;
```

```
(cmax_pe, auc_pe) = (14.379661179689466, 0.5660411346530778)
```

```
scatter(time, conc, label="Original C (fast)")
plot!(sol, label="predicted C (fast)", xlabel="time", ylabel="Plasma Conc.  Profile")
```

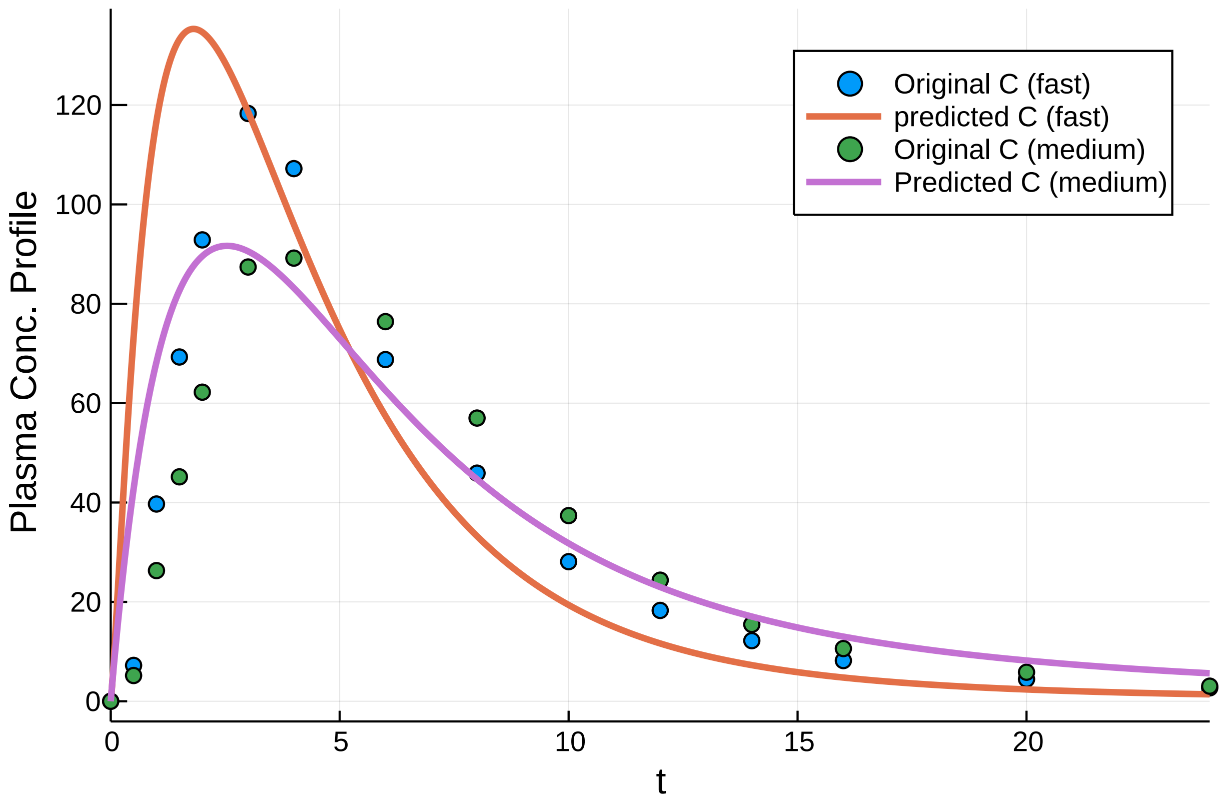


Add the plot for medium formulation and the percentage error in `Cmax` and `AUC`.

```julia
sol = predict_vivo(model, "medium")
time = vivo_data[1]["medium"].time
conc = vivo_data[1]["medium"].conc
cmax_pe, auc_pe = percentage_prediction_error(time, conc, sol.t, sol.u)
@show cmax_pe, auc_pe;
```

```
(cmax_pe, auc_pe) = (2.7580179517701184, 3.9560820996918498)
```

```julia
scatter!(time, conc, label="Original C (medium)")
plot!(sol, label="Predicted C (medium)")
```

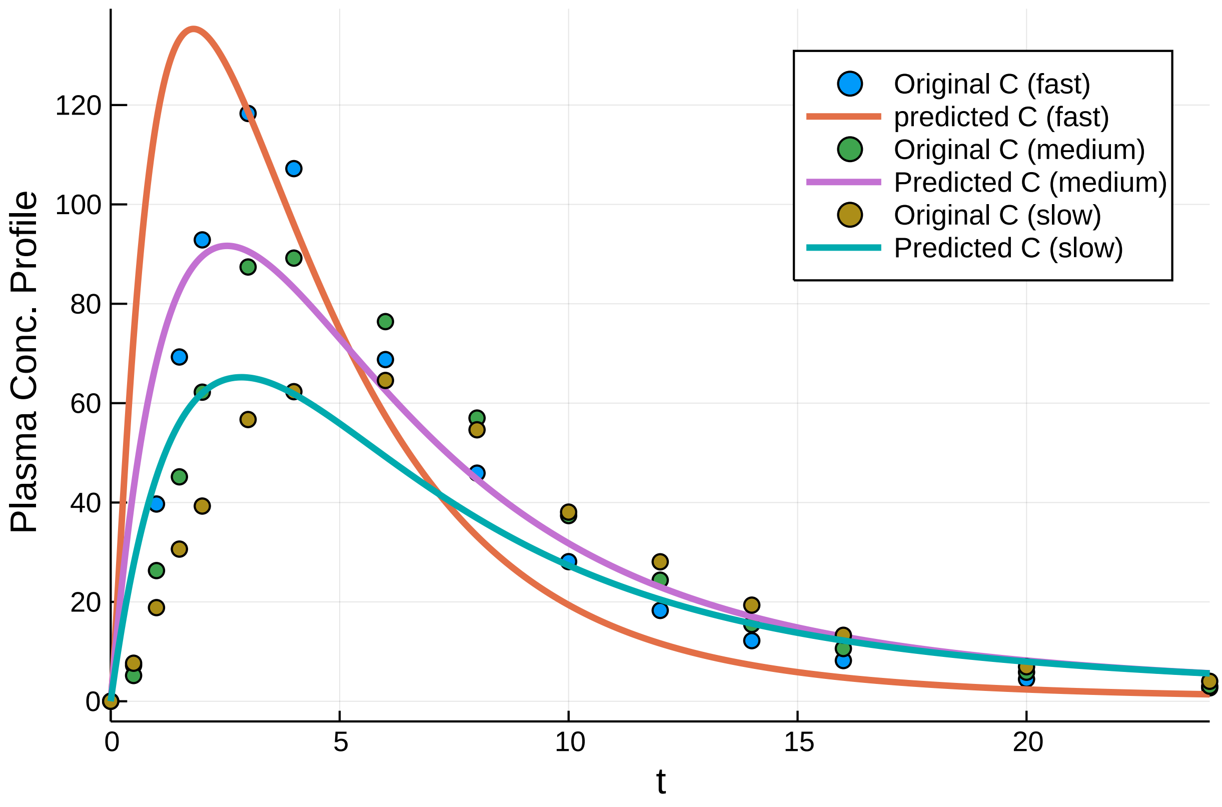


Also, add plot for slow formulation and the percentage error in `Cmax` and `AUC`.

```
sol = predict_vivo(model, "slow")
time = vivo_data[1]["slow"].time
conc = vivo_data[1]["slow"].conc
cmax_pe, auc_pe = percentage_prediction_error(time, conc, sol.t, sol.u)
@show cmax_pe, auc_pe;

(cmax_pe, auc_pe) = (0.9950484042976151, 6.6174326894310145)

scatter!(time, conc, label="Original C (slow)")
plot!(sol, label="Predicted C (slow)")
```



1.10.4 Reports

You can save all the estimated parameters and intermediate results using `to_csv` function to csv files. By default, all the files will be saved to home directory. To save to another location, pass the path to location using the `path` keyword argument.

```
to_csv(model, path=homedir())
```

```
"C:\\Users\\shubham\\ivivc_params_estimates.csv"
```

Note: This package is still in WIP; it might break.