Metrum Research Group LLC
Phone: 860.735.7043
billgm@metrumrg.com

2 Tunxis Road, Suite 112
Tariffville, CT 06081
metrumrg.com

Torsten

A Prototype Library for Bayesian Pharmacometrics Modeling in Stan

User Manual

Torsten Version 0.83
for Stan Version 2.16.0

August 2017

Contents

Development Team

Bill Gillespie
`billg@metrumrg.com`
Metrum Research Group, LLC

Charles Margossian
`charles.margossian@columbia.edu`
Columbia University, Department of Statistics
(formally Metrum Research Group, LLC)

Acknowledgements

Institutions
We thank Metrum Research Group, Columbia University, and AstraZeneca.

Funding
This work was funded in part by the following organizations:

- Office of Naval Research (ONR) contract N00014-16-P-2039 provided as part of the Small Business Technology Transfer (STTR) program. The content of the information presented in this document does not necessarily reflect the position or policy of the Government and no official endorsement should be inferred.
- Bill & Melinda Gates Foundation

Individuals
We thank the Stan Development Team for giving us guidance on how to create new Stan functions and adding features to Stan's core language that facilitate building ODE-based models.

We also thank Kyle Baron and Hunter Ford for helpful advice on coding in C++ and using GitHub, Curtis Johnston for reviewing the User Manual, and Yaming Su for using Torsten and giving us feedback.

1. Introduction

Stan is an open source probabilistic programing language designed primarily to do Bayesian data analysis [2]. Several of its features make it a powerful tool to specify and fit complex models. Notably, its language is extremely flexible and its No U-Turn Sampler (NUTS), an adaptative Hamiltonian Monte Carlo algorithm, has proven more efficient than commonly used Monte Carlo Markov Chains (MCMC) samplers for complex high dimensional problems [3]. Our goal is to harness these innovative features and make Stan a better software for pharmacometrics modeling. Our efforts are twofold:

(1) We contribute to the development of new mathematical tools, such as functions that support differential equations based models, and implement them directly into Stan's core language.
(2) We develop Torsten, an extension with specialized pharmacometrics functions.

Throughout the process, we work very closely with the Stan Development Team. We have benefited immensely from their mentorship, advice, and feedback. Just like Stan, Torsten is an open source project that fosters collaborative work. Interested in contributing? Shoot us an e-mail and we will help you help us (`billg@metrumrg.com`)!

Torsten is licensed under the BSD 3-clause license.

WARNING: The current version of Torsten is a *prototype*. It is being released for review and comment, and to support limited research applications. It has not been rigorously tested and should not be used for critical applications without further testing or cross-checking by comparison with other methods.

We encourage interested users to try Torsten out and are happy to assist. Please report issues, bugs, and feature requests on our GitHub page: `https://github.com/metrumresearchgroup/stan`.

1.1. Installing Torsten.

Installation files are available on GitHub: `https://github.com/metrumresearchgroup/example-models`

There is currently no mechanism to install Torsten on top of your version of Stan. This is still a work in progress. In the meantime, we offer a version of Stan with Torsten built inside of it. Torsten 0.83 works with Stan 2.16.0. Torsten is built inside the Stan and Stan-math repositories and is agnostic to the interface. We offer support to install Torsten with RStan and CmdStan.

1.1.1. *Intalling Torsten with RStan.*

The easiest way to install the RStan interface with Stan and Torsten is to run the script `R/setupRTorsten.R`. You'll need to make a few minor adjustments, notably by specifying where you wish to install RStan (and its dependency StanHeaders). You may run into difficulty if Torsten is not up to date with Stan's last release[1].

If you already have these packages installed, the script will not automatically overwrite them, which is why you should remove them prior to running `setupRTorsten.R`.

The script currently doesn't install all the dependencies for RStan[2]. This is to prevent the edited Stan-Headers package from getting overwritten by the `install.packages()` procedure. If you wish to install

[1] See issue #3 on our GitHub issue tracker: `https://github.com/charlesm93/example-models/issues/3`
[2] See `https://github.com/stan-dev/rstan/wiki/RStan-Getting-Started` for details on RStan

RStan's dependencies, you could run `install.package("rstan")`, remove the RStan and the StanHeaders packages and then run `SetupRTorsten.R`. We realize this is not optimal and are working on a more elegant solution.

1.1.2. *Installing Torsten with CmdStan.* Similarly, you can install the CmdStan interface with Stan and Torsten using the bash file `setupTorsten.sh`[3].

1.2. **Overview.**

Torsten is a prototype pharmacometrics model library for use in Stan 2.16.0. The current version includes:

- Specific linear compartment models:
  - One compartment model with first order absorption
  - Two compartment model with elimination from and first order absorption into central compartment
- General linear compartment model described by a system of first-order <u>linear</u> Ordinary Differential Equations (ODEs).
- General compartment model described by a system of first order ODEs
- Mix compartment model with forcing a PK function described by a One or Two compartment model, and forced states described by a system of first-order ODEs.

The models and data format are based on NONMEM®[4]/NMTRAN/PREDPP conventions including:

- Recursive calculation of model predictions
  - This permits piecewise constant covariate values
- Bolus or constant rate inputs into any compartment
- Handles single dose and multiple dose histories
- Handles steady state dosing histories
  - Note: The infusion time must be shorter than the inter-dose interval.
- Implemented NMTRAN data items include: TIME, EVID, CMT, AMT, RATE, ADDL, II, SS

In general, all continuous variables may be passed as parameters. A few exceptions apply *to functions which use a numerical integrator* (i.e. the general and the mix compartment models). The below listed cases present technical difficulties, which we expect to overcome in Torsten's next release:

- The RATE and TIME arguments must be fixed
- In the case of a multiple truncated infusion rate dosing regimen:
  - The bioavailability (F) and the amount (AMT) must be fixed.

This library provides Stan language functions that calculate amounts in each compartment, given an event schedule and an ODE system.

1.3. **Implementation details.**

- Stan version 2.16.0

[3]To get the development version of Torsten use `setupTorsten-dev.sh`.

[4]NONMEM® is licensed and distributed by ICON Development Solutions.

- All functions are programmed in C++ and are compatible with the Stan-math automatic differentiation library [4]
- All functions can be called directly in a Stan file in a manner identical to other built-in functions
- One and two compartment models: hand-coded analytical solutions
- General linear compartment models with semi-analytical solutions using the built-in Matrix Exponential function
- General compartment models with numerical solutions using built-in ODE integrators in Stan. The tuning parameters of the solver are adjustable. The steady state solution is calculated using a numerical algebraic solver (expected in Stan 2.17).
- Mix compartment model: the forcing PK function is solved analytically and the forced ODE system is solved numerically.

1.4. **Development plans.**

Our current plans for future development of Torsten include the following:

- Build a system to easily share packages of Stan functions (written in C++ or in the Stan language)
- Allow numerical methods to handle RATE, AMT, TIME, and the bioavailability fraction (F) as parameters in all cases.
- Optimize Matrix exponential functions
  - Function for the action of Matrix Exponential on a vector
  - Hand-coded gradients
  - Special algorithm for matrices with special properties
- Fix issue that arises when computing the adjoint of the lag time parameter (in a dosing compartment) evaluated at $t_{lag} = 0$.
- Make the following arguments optional
  - `biovar` and `tlag`, respectively used for the bioavailability fraction and the lag times in each compartment
  - Tuning parameters of the ODE integrators for the general compartmental function.
- Extend formal tests
  - We want more C++ Google unit tests to address cases users may encounter
  - Comparison with simulations from the R package *mrgsolve* and the software NONMEM®
  - Recruit non-developer users to conduct beta testing

1.5. **Updates since Torsten 0.82.**

- Torsten is now up to date with Stan version 2.16.0
- Add steady state solution for general compartment model, with some limitations
- Add "mixed solver" functions
- Add algebraic solver
- Torsten automatically corrects ODE functors when the rates are non-zero. The code carefully distinguishes cases where the rates are fixed data or parameters.
- Fix bug that prevented Jacobians from being accurately computed for `generalOdeModel_*` when `biovar` was passed as parameters, causing the sampler to dissolve into a quasi-random walk.
- Fix bug that prevented users from using a print statement inside a function that specifies an ODE system.
- Fixed minor bugs and report issues.

2. Using Torsten

The reader should have a basic understanding of how Stan works before reading this chapter. There are excellent resources online to get started with Stan (`http://mc-stan.org/documentation/`).

In this section we go through the different functions Torsten adds to Stan. It will be helpful to apply these functions to a simple example. We have uploaded code and data on `https://github.com/metrumresearchgroup/example-models`.

2.1. **Example 1: Two Compartment Model.**

We model drug absorption in a single patient and simulate plasma drug concentrations:

- Multiple Doses: 1250 mg, every 12 hours, for a total of 15 doses
- PK: plasma concentrations of parent drug (c)
- PK measured at 0.083, 0.167, 0.25, 0.5, 0.75, 1, 1.5, 2, 4, 6, 8, 10 and 12 hours after 1st, 2nd, and 15th dose. In addition, the PK is measured every 12 hours throughout the trial.

The plasma concentration (c) are simulated according to the following equations:

$$\begin{aligned} \log(c) &\sim N\left(\log(\widehat{c}), \sigma^2\right) \\ \widehat{c} &= f_{2cpt}(t, CL, Q, V_2, V_3, k_a) \\ (CL, Q, V_2, V_3, ka) &= \left(5 \text{ L/h}, 8 \text{ L/h}, 20 \text{ L}, 70 \text{ L}, 1.2 \text{ h}^{-1}\right) \\ \sigma^2 &= 0.01 \end{aligned}$$

and the drug concentration is given by $c = y_2/V_2$.

where the mass of drug in the central compartment (y_2) is obtained by solving the system of ordinary differential equations (ODEs):

$$\begin{aligned} y_1' &= -k_a y_1 \\ y_2' &= k_a y_1 - \left(\frac{CL}{V_2} + \frac{Q}{V_2}\right) y_2 + \frac{Q}{V_3} y_3 \\ y_3' &= \frac{Q}{V_2} y_2 - \frac{Q}{V_3} y_3 \end{aligned} \tag{1}$$

The data are generated using the R package *mrgsolve* [?], see `TwoCptModelSimulation.R`. We use this example to demonstrate use of several functions in the Torsten library.

2.2. **Linear One and Two Compartment Model Function.**

The one and two compartment model functions have the form:

```
<model name>(time, amt, rate, ii, evid, cmt, addl, ss,
             theta, biovar, tlag)
```

There is no need to skip a line, but we do so to distinguish between *event* arguments and *model* arguments.

The event arguments describe the event schedule of the clinical trial. `time`, `amt`, `rate`, and `ii` are arrays of real and `evid`, `cmt`, `addl`, and `ss` arrays of integers. All arrays have the same length, which corresponds to the number of events.

Next we have the model arguments: `theta` contains the ODE parameters, `biovar` the bioavailability fraction in each compartment (sometimes denoted as F), and `tlag` the lag time in each compartment. The model arguments may be either one or two dimensional arrays. If they are one dimensional arrays, the parameters are constant for all events. If they are two dimensional arrays then each row contains the parameters for the interval [`time[i-1], time[i]`]. The number of rows should equal the number of events.

The options for *model name* are:

- PKModelOneCpt
- PKModelTwoCpt

which respectively correspond to the one and two compartment model with a first order absorption (figure 1). An array in `theta` is expected to contain parameters CL, V_2, and ka for the one compartment case, and CL, Q, V_2, V_3, and ka for the two compartments case, in this order. Setting ka to 0 eliminates the fist-order absoprtion. `biovar` contains the bioavailability fraction of each compartment (non-effective if set to 1) and `tlag` the lag time in each compartment (non-effective if set to 0).

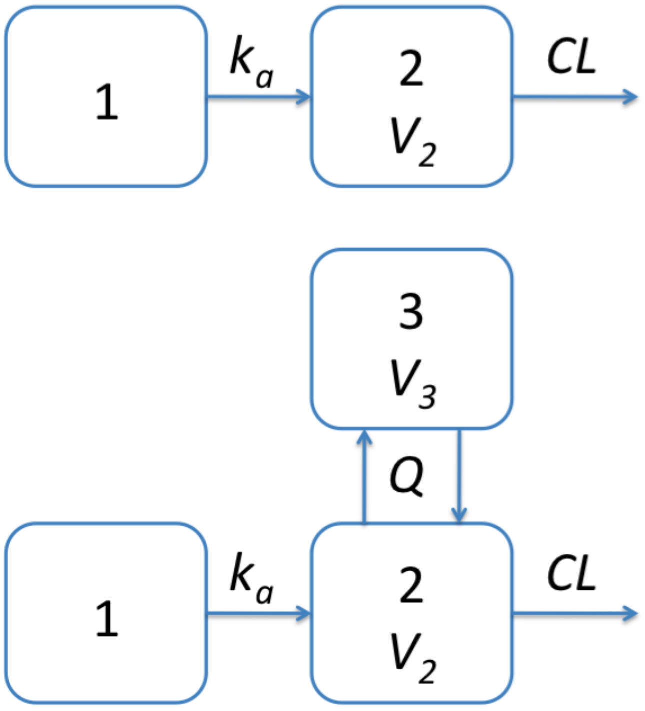

FIGURE 1. One and two compartment models with first order absorption implemented in Torsten.

`PKModelTwoCpt` can be used to fit example 1, see `TwoCptModel.stan`. We are interested in evaluating the ODE parameters, stored in `theta`. The bioavailability fraction and the lag times on the other hand are fixed, and we therefore declare `biovar` and `tlag` in the **transformed data** block. Three MCMC chains of 2000 iterations were simulated. The first 1000 iteration of each chain were discarded. Thus 1000 MCMC samples were used for the subsequent analyses.

```
data {
  int<lower = 1> nt; // number of events
  int<lower = 1> nObs; // number of observation
  int<lower = 1> iObs[nObs]; // index of observation
  int cmt[nt];
  int evid[nt];
  int addl[nt];
  int ss[nt];
  real amt[nt];
  real time[nt];
  real rate[nt];
  real ii[nt];

  vector<lower = 0>[nObs] cObs; // observed concentration (Dependent Variable)
}

transformed data {
                              ⋮
  biovar[1] = 1;
  biovar[2] = 1;
  biovar[3] = 1;

  tlag[1] = 0;
  tlag[2] = 0;
  tlag[3] = 0;


}
                              ⋮
parameters {
  real<lower = 0> CL;
  real<lower = 0> Q;
  real<lower = 0> V2;
  real<lower = 0> V3;
  real<lower = 0> ka;
  real<lower = 0> sigma;
}

transformed parameters {
                              ⋮
  theta[1] = CL;
  theta[2] = Q;
  theta[3] = V2;
  theta[4] = V3;
  theta[5] = ka;

  x = PKModelTwoCpt(time, amt, rate, ii, evid, cmt, addl, ss,
                    theta, biovar, tlag);

  cHat = col(x, 2) ./ V2; // get concentration in the central compartment

  cHatObs = cHat[iObs]; // predictions for observed data records

 }
                              ⋮
```

FIGURE 2. Stan language for fitting a two compartment model using the `PKModelTwoCpt` function (abstract)

Result. The MCMC history plots (figure 3) suggest that the 3 chains have converged to common distributions for all of the key model parameters. The fit to the plasma concentration data (figure 5) are in close agreement with the data, which is not surprising since the fitted model is identical to the one used to simulate the data. Similarly the parameter estimates summarized in Table 1 are consistent with the values used for simulation.

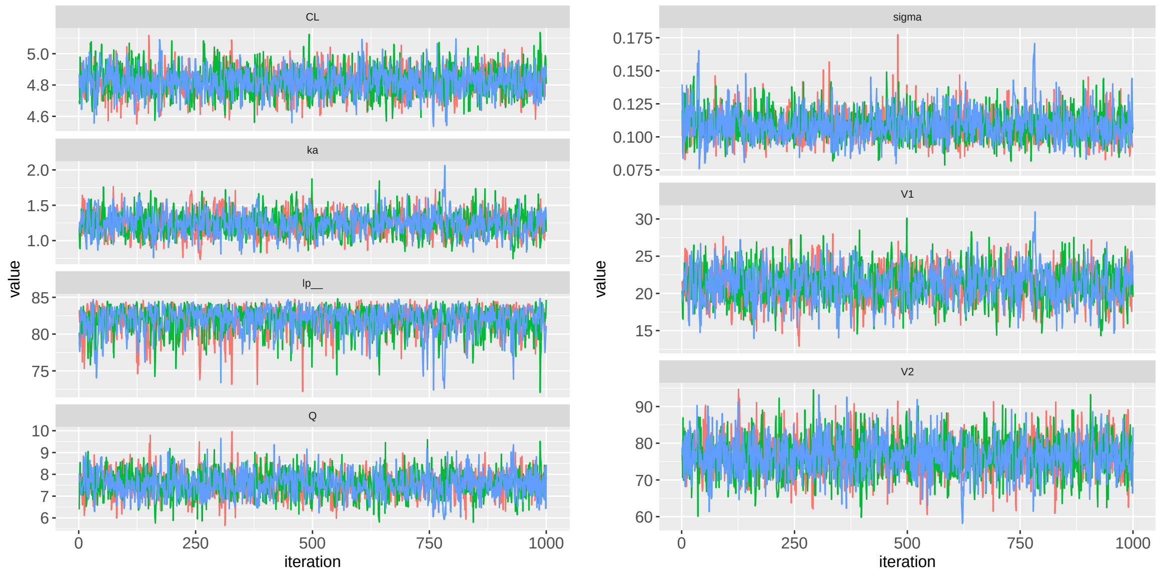

FIGURE 3. MCMC history plots for the parameters of a two compartment model with first order absorption (each color corresponds to a different chain)

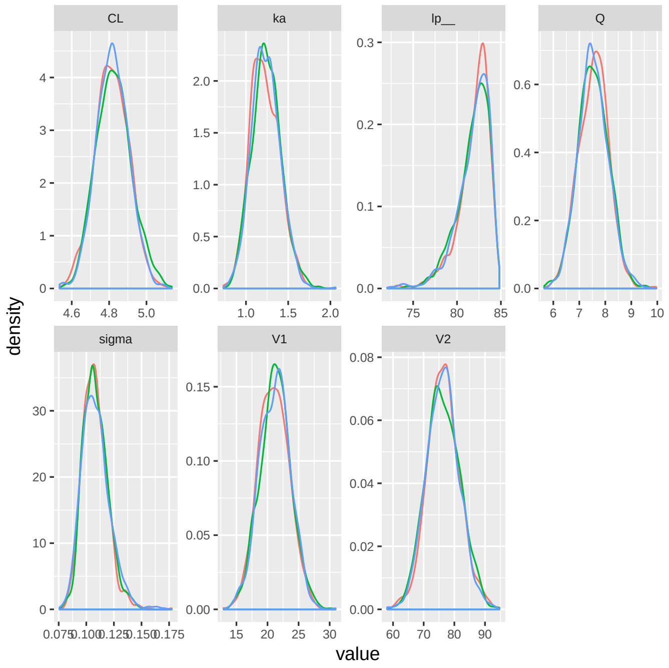

FIGURE 4. Posterior Marginal Densities of the Model Parameters of a two compartment model with first order absorption (each color corresponds to a different chain)

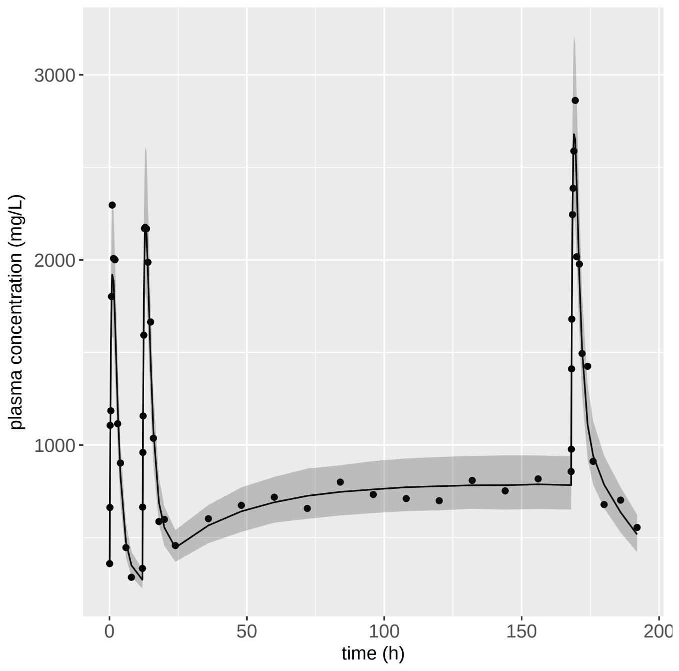

FIGURE 5. Predicted (posterior median and 90 % credible intervals) and observed plasma drug concentrations of a two compartment model with first order absorption

TABLE 1. Summary of the MCMC simulations of the marginal posterior distributions of the model parameters

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
CL	4.82	0.002	0.0901	4.64	4.76	4.82	4.88	5.00	2464.73	1.00
Q	7.54	0.016	0.58	6.43	7.15	7.54	7.92	8.69	1385.75	1.00
V2	21.14	0.069	2.45	16.37	19.44	21.19	22.78	25.89	1245.64	1.00
V3	76.35	0.110	5.35	65.98	72.75	76.26	79.83	87.30	2379.15	1.00
ka	1.23	0.005	0.169	0.923	1.12	1.23	1.35	1.58	1295.01	1.00
sigma	0.108	0.000	0.012	0.0887	0.0999	0.107	0.115	0.135	1973.97	1.00

2.3. **General Linear ODE Model Function.**

A general linear ODE model refers to a model that may be described in terms of a system of first order linear differential equations with (piecewise) constant coefficients, i.e., a differential equation of the form:

$$y'(t) = Ky(t)$$

where K is a matrix. For example K for a two compartment model (equation 1) with first order absorption is:

$$K = \begin{bmatrix} -k_a & 0 & 0 \\ k_a & -(k_{10} + k_{12}) & k_{21} \\ 0 & k_{12} & -k_{21} \end{bmatrix}$$

where $k_{10} = CL/V_2$, $k_{12} = Q/V_2$, and $k_{21} = Q/V_3$.

The linear ODE model function has the form:

```
linOdeModel(time, amt, rate, ii, evid, cmt, addl, ss,
               system, biovar, tlag)
```

`system` can be:

- the matrix `K`, if the constant rate matrix is the same for all events.
- an array of constant rate matrices. The length of the array is the number of events and each element corresponds to the matrix at the interval [`time[i-1], time[i]`].

`system` contains all the ODE parameters, so we no longer need `theta`.

FIGURE 6. Stan language for fitting a two compartment model using the `linOdeModel` function (abstract)

```
transformed parameters {
  matrix[3, 3] K;
  real k10 = CL / V2;
  real k12 = Q / V2;
  real k21 = Q / V3;
  vector<lower = 0>[nTheta] theta[1];
  vector<lower = 0>[nt] cHat;
  vector<lower = 0>[nObs] cHatObs;
  matrix<lower = 0>[nt, 3] x;

  K = rep_matrix(0, 3, 3);

  K[1, 1] = -ka;
  K[2, 1] = ka;
  K[2, 2] = -(k10 + k12);
  K[2, 3] = k21;
  K[3, 2] = k12;
  K[3, 3] = -k21;

  x = linOdeModel(time, amt, rate, ii, evid, cmt, addl, ss,
                  K, biovar, tlag);

  cHat = col(x, 2) ./ V1;

  cHatObs = cHat[iObs]; # predictions for observed data records

}

model{
  logCObs ~ normal(log(cHatObs), sigma);
}
```

2.4. General ODE Model Function.

Torsten may be used to fit models described by a system of first-order ODEs, i.e., differential equations of the form:

$$y'(t) = f(t, y(t))$$

In the case where the rate vector R is non-zero, this equation becomes:

$$y'(t) = f(t, y(t)) + R$$

The general ODE model functions have the form:

```
<model_name>(ODE_system, nCmt,
             time, amt, rate, ii, evid, cmt, addl, ss,
             theta, biovar, tlag,
             rel_tol, abs_tol, max_step)
```

where `ODE_system` specifies $f(t, y(t))$, which the user defines inside the **functions** block (see section 19.2 of the Stan reference manual for details and figure 7 for an example). The user does NOT include the rates in their definition of f. Torsten automatically corrects the derivatives when the rates are non-zero.

`nCmt` is the number of compartments (or, equivalently, the number of ODEs) in the model. `rel_tol`, `abs_tol`, and `max_step` are tuning parameters for the ODE integrator: respectively the relative tolerance, the absolute tolerance, and the maximum number of steps.

The options for `model_name` are:

- generalOdeModel_rk45
- generalOdeModel_bdf

They respectively call the built-in Runge-Kutta 4th/5th order (rk45) integrator, recommended for non-stiff ODEs, and the Backward Differentiation (BDF) integrator, recommended for stiff ODEs. Which value to use for the tuning parameters depends on the integrator and the specifics of the ODE system. Reducing the tolerance parameters and increasing the number of steps make for a more robust integrator but can significantly slow down the algorithm. The following can be used as a starting point: `rel_tol = 1e-6`, `abs_tol = 1e-6` and `max_step = 1e+6` for the rk45 integrator and `rel_tol = 1e-10`, `abs_tol = 1e-10` and `max_step = 1e+8` for the bdf integrator[5]. Users should be prepared to adjust these values. For additional information, see Stan's reference manual (section 19).

A few notable restrictions apply to `generalOdeModel_*`:

- `rate` and `time` cannot be passed as parameters.
- In the case of a multiple truncated infusion rate dosing regimen:
  - The bioavailability (`biovar`) and the amount (`amt`) cannot be passed as parameters.

These restrictions apply to `mixOde#Cpt_*` functions, discussed in the next section.

[5]These are the default tuning parameters for integrate_ode_rk45() and integrate_ode_bdf(). Torsten functions do not have a default values for these parameters. The user must explicitly pass the tuning parameters to generalOdeModel_*().

FIGURE 7. Stan language for fitting a two compartment model using the `generalOdeModel_rk45` function (abstract)

```
functions{
  # define ODE system for two compartment model
  real[] twoCptModelODE(real t,
                            real[] y,
                            real[] theta,
                            real[] dummy_real,
                            int[] dummy_int){
    real Q = theta[1];
    real CL = theta[2];
    real V2 = theta[3];
    real V3 = theta[4];
    real ka = theta[5];
    real k12 = Q / V2;
    real k21 = Q / V3;
    real k10 = CL / V2;
    real y[3];

    dydt[1] = -ka * y[1];
    dydt[2] = ka * y[1] - (k10 + k12)*y[2] + k21*y[3];
    dydy[3] = k12 * y[2] - k21 * y[3];

    return dydt;
  }
}
                              ⋮
transformed parameters {
                              ⋮

  theta[1] = CL;
  theta[2] = Q;
  theta[3] = V1;
  theta[4] = V2;
  theta[5] = ka;

  x = generalCptModel_rk45(twoCptModelODE, 3,
                           time, amt, rate, ii, evid, cmt, addl, ss,
                           theta, biovar, tlag,
                           1e-8, 1e-8, 1e8);
                              ⋮
```

2.5. Mixed ODE Model Function.

In certain cases, an ODE system can be divided in two subsystems:

$$\begin{aligned} y_1' &= f_1(t, y_1) \\ y_2' &= f_2(t, y_1, y_2) \end{aligned}$$

where y_1, y_2, f_1, and f_2 are vector-valued functions, and y_1' is independent of y_2. This structure arises in PK/PD models, where y_1 describes a forcing PK function and y_2 the PD effects. If y_1 has an analytical solution, we can construct a *mixed solver*, which analytically solves y_1 and numerically integrates y_2. This approach leads to an appreciable gain in computational efficiency. In the example of a Friberg-Karlsson semi-mechanistic model [1], we observe an average speedup of $\sim 47 \pm 18\%$ when using the mix solver in lieu of the numerical integrator [**?**].

Torsten supports the mixed solver for cases where y_1 solves the ODEs for a One or Two Compartment model with a first-order absorption.

The mix ODE model functions have the form:

```
<model_name>(reduced_ODE_system, nOde,
             time, amt, rate, ii, evid, cmt, addl, ss,
             theta, biovar, tlag,
             rel_tol, abs_tol, max_step)
```

where `reduced_ODE_system` specifies the system we numerically solve (y_2 in the above discussion, also called the *reduced system*) and `nOde` the number of equations in the reduced system. The function that defines a reduced system has an almost identical signature to that used for a full system, but takes one additional argument: y_1, the PK states, i.e. solution to the PK ODEs (figure 8).

FIGURE 8. Stan language for defining a reduced ODE system

```
functions{
  real[] reducedODE(real t, // time
                    real[] y, // reduced state
                    real[] y1, // PK states
                    real[] theta, // parameters
                    real[] x_r, // data (real)
                    int[] x_int) { // data (integer)
                          ⋮
  }
}
```

Again, the user does not specify the rates. Torsten automatically corrects the derivatives for non-zero rates.

The options for `modelName` are:

- mixOde1CptModel_rk45
- mixOde1CptModel_bdf
- mixOde2CptModel_rk45
- mixOde2CptModel_bdf

These four functions correspond to all the permutations we can obtain when using a forcing One or Two Compartment function, and the Runge-Kutta 4th/5th order (rk45) or Backward Differentiation (BDF) integration method. The mixed ODE functions can be used to compute the steady state solutions supported by the general ODE model functions.

Restrictions regarding which arguments may be passed as parameters for `generalOdeModel_*` also apply to `mixOde#CptModel_*`.

We cannot apply the mixed solver to the Two Compartment example we have been using so far. Instead, we will consider the model which motivated the implementation of the method in the first place.

2.6. **Example 2: Friberg-Karlsson Semi-Mechanistic Model** [1].

In this second example, we add to our two Compartment model a PD effect, described by a system of nonlinear ODEs.

Neutropenia is observed in patients receiving an ME-2 drug. Our goal is to model the relation between neutrophil counts and drug exposure. Using a feedback mechanism, the body maintains the number of neutrophils at a baseline value (figure 9). While in the patient's blood, the drug impedes the production of neutrophils. As a result, the neutrophil count goes down. After the drug clears out, the feedback mechanism kicks in and brings the neutrophil count back to baseline.

Friberg-Karlsson Model for drug-induced myelosuppression (ANC)

$$\begin{aligned}
\log(ANC_i) &\sim N(\log(Circ), \sigma^2_{ANC}) \\
Circ &= f_{FK}(MTT, Circ_0, \alpha, \gamma, c) \\
(MTT, Circ_0, \alpha, \gamma, ktr) &= (125, 5.0, 3 \times 10^{-4}, 0.17) \\
\sigma^2_{ANC} &= 0.001
\end{aligned}$$

where c is the drug concentration in the blood we get from the Two Compartment model, and $Circ$ is obtained by solving the following system of nonlinear ODEs:

$$\begin{aligned}
y'_{\text{prol}} &= k_{\text{prol}} y_{\text{prol}} (1 - E_{\text{drug}}) \left(\frac{Circ_0}{y_{\text{circ}}} \right)^{\gamma} - k_{\text{tr}} y_{\text{prol}} \\
y'_{\text{trans1}} &= k_{\text{tr}} y_{\text{prol}} - k_{\text{tr}} y_{\text{trans1}} \\
y'_{\text{trans2}} &= k_{\text{tr}} y_{\text{trans1}} - k_{\text{tr}} y_{\text{trans2}} \\
y'_{\text{trans3}} &= k_{\text{tr}} y_{\text{trans2}} - k_{\text{tr}} y_{\text{trans3}} \\
y'_{\text{circ}} &= k_{\text{tr}} y_{\text{trans3}} - k_{\text{tr}} y_{\text{circ}}
\end{aligned} \tag{2}$$

where $E_{drug} = \alpha c$.

The ODEs specifying the Two Compartment Model (equation 1) do not depend on the PD ODEs (equation 2) and can be solved analytically by Torsten. We therefore specify our model using a mixed solver function. We do not expect our system to be stiff and use the Runge-Kutta 4th/5th order integrator.

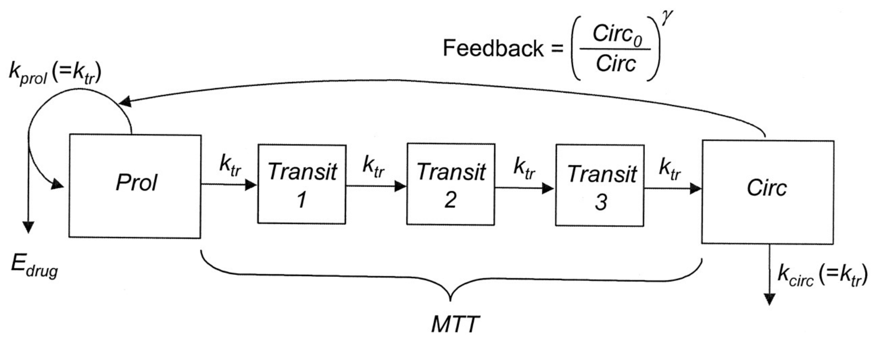

FIGURE 9. Friberg-Karlsson semi-mechanistic Model [1]

FIGURE 10. Stan language to define a reduced ODE system

```
functions{
  # define reduced ODE system for two compartment model
  real[] FK_ODE(real t,
                real[] y,
                real[] y_pk,
                real[] theta,
                real[] dummy_real,
                int[] dummy_int){
    real V2 = theta[3];

    ## PK variables
    real VC = parms[3];

    ## PD variables
    real MTT = parms[6];
    real circ0 = parms[7];
    real alpha = parms[8];
    real gamma = parms[9];
    real ktr = 4 / MTT;
    real prol = y[1] + circ0;
    real transit1 = y[2] + circ0;
    real transit2 = y[3] + circ0;
    real transit3 = y[4] + circ0;
    real circ = fmax(machine_precision(), y[5] + circ0);
    real conc = y_pk[2] / VC;
    real Edrug = alpha * conc;
    real dydt[5];

    conc = y_pk[2] / VC;
    Edrug = alpha * conc;

    dydt[1] = ktr * prol * ((1 - Edrug) * ((circ0 / circ)^gamma) - 1);
    dydt[2] = ktr * (prol - transit1);
    dydt[3] = ktr * (transit1 - transit2);
    dydt[4] = ktr * (transit2 - transit3);
    dydt[5] = ktr * (transit3 - circ);

    return dydt;
  }
}
```

FIGURE 11. Stan language for fitting a Friberg-Karlsson model using `mixOde2CptModel_rk45`

```
functions {
  real[] FK_ODE {
                 ⋮
  }
}

transformed date {
int nOde = 5;
                 ⋮
}

transformed parameters {
  vector[nt] cHat;
  vector[nObsPK] cHatObs;
  vector[nt] neutHat;
  vector<lower = 0>[nObsPD] neutHatObs;
  real theta[nParms]; # ODE parameters
  matrix[nt, nCmt] x;

  theta[1] = CL;
  theta[2] = Q;
  theta[3] = VC;
  theta[4] = VP;
  theta[5] = ka;
  theta[6] = mtt;
  theta[7] = circ0;
  theta[8] = alpha;
  theta[9] = gamma;

  x = mixOde2CptModel_rk45(FK_ODE, nOde,
                           time, amt, rate, ii, evid, cmt, addl, ss,
                           theta, biovar, tlag,
                           1e-6, 1e-6, 1e+6);

  cHat = x[ , 2] / VC;
  neutHat = x[ , 8] + circ0;

  cHatObs = cHat[iObsPK];
  neutHatObs = neutHat[iObsPD];
}
```

Table 2. Summary: Arguments of Torsten functions.

model	function name	argument names	parameters in `theta`
one compartment model with first order absorption	`PKModelOneCpt`	`time`, `amt`, `rate`, `ii`, `evid`, `cmt`, `addl`, `ss`, `theta`, `biovar`, `tlag`	CL, V_2, k_a
two compartment model with first order absorption	`PKModelTwoCpt`	`time`, `amt`, `rate`, `ii`, `evid`, `cmt`, `addl`, `ss`, `theta`, `biovar`, `tlag`	CL, Q, V_2, V_3, k_a
general linear compartment model	`linOdeModel`	`time`, `amt`, `rate`, `ii`, `evid`, `cmt`, `addl`, `ss`, `system`, `biovar`, `tlag`	NA: pass in constant rate matrix instead of `theta`
general compartment models	`genOdeModel_*`	`ODE_system`, `nCmt`, `time`, `amt`, `rate`, `ii`, `evid`, `cmt`, `addl`, `ss`, `theta`, `biovar`, `tlag`, `rel_tol`, `abs_tol`, `max_num_steps`	Parameters that get passed to ODE system
mix 1 compartment model	`mixOde1Cpt_*`	`reduced_ODE_system`, `nOde`, `time`, `amt`, `rate`, `ii`, `evid`, `cmt`, `addl`, `ss`, `theta`, `biovar`, `tlag`, `rel_tol`, `abs_tol`, `max_num_steps`	CL, V_2, k_a, followed by the parameters that get passed to the reduced ODE system
mix 2 compartment model	`mixOde2Cpt_*`	`reduced_ODE_system`, `nOde`, `time`, `amt`, `rate`, `ii`, `evid`, `cmt`, `addl`, `ss`, `theta`, `biovar`, `tlag`, `rel_tol`, `abs_tol`, `max_num_steps`	CL, Q, V_2, V_3, k_a, followed by the parameters that get passed to the reduced ODE system

3. Additional Examples

Code for examples can be found on GitHub: `https://github.com/metrumresearchgroup/example-mode`

All the files to run a model are stored under the directory that bears the model's name. There are four files per example:

- `<model name>.stan`
- `<model name>.data.R`
- `<model name>.init.R`
- `<model name>Simulation.R`

`data.R` contains the data we fit the model to and `init.R` an initial estimate of the parameters. These two files are generated using `Simulation.R`. The `R` folder contains R scripts to compile and run the models, as well as code to output diagnostic plots and statistics.

3.1. **Example 3: Effect Compartment Population Model.**

Let us expand example 1 to a population model fitted to the combined data from phase I and phase IIa studies. The parameters exhibit inter-individual variations (IIV), due to both random effects and to the patients' body weight, treated as a covariate and denoted *bw*:

Population Model for Plasma Drug Concentration (c).

$$\begin{aligned}
\log\left(c_{ij}\right) &\sim N\left(\log\left(\widehat{c}_{ij}\right), \sigma^2\right) \\
\widehat{c}_{ij} &= f_{2cpt}\left(t_{ij}, D_j, \tau_j, CL_j, Q_j, V_{1j}, V_{2j}, k_{aj}\right) \\
\log\left(CL_j, Q_j, V_{ssj}, k_{aj}\right) &\sim N\left(\log\left(\widehat{CL}\left(\frac{bw_j}{70}\right)^{0.75}, \widehat{Q}\left(\frac{bw_j}{70}\right)^{0.75}, \widehat{V}_{ss}\left(\frac{bw_j}{70}\right), \widehat{k}_a\right), \Omega\right) \\
V_{1j} &= f_{V_1} V_{ssj} \qquad V_{2j} = \left(1 - f_{V_1}\right) V_{ssj} \\
\left(\widehat{CL}, \widehat{Q}, \widehat{V}_{ss}, \widehat{k}_a, f_{V_1}\right) &= \left(10\ \text{L/h}, 15\ \text{L/h}, 140\ \text{L}, 2\ \text{h}^{-1}, 0.25\right) \\
\Omega &= \begin{pmatrix} 0.25^2 & 0 & 0 & 0 \\ 0 & 0.25^2 & 0 & 0 \\ 0 & 0 & 0.25^2 & 0 \\ 0 & 0 & 0 & 0.25^2 \end{pmatrix}, \quad \sigma = 0.1
\end{aligned}$$

Furthermore we add a fourth compartment in which we measure a PD effect (figure 12).

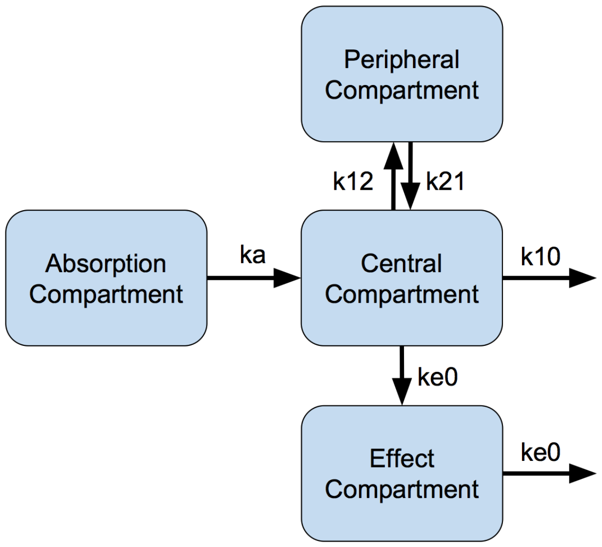

FIGURE 12. Effect Compartment Model

Effect Compartment Model for PD response (R).

$$\begin{aligned}
R_{ij} &\sim N\left(\widehat{R}_{ij}, \sigma_R^2\right) \\
\widehat{R}_{ij} &= \frac{E_{max} c_{eij}}{EC_{50j} + c_{eij}} \\
c'_{e\cdot j} &= k_{e0j}\left(c_{\cdot j} - c_{e\cdot j}\right) \\
\log\left(EC_{50j}, k_{e0j}\right) &\sim N\left(\log\left(\widehat{EC}_{50}, \widehat{k}_{e0}\right), \Omega_R\right) \\
\left(E_{max}, \widehat{EC}_{50}, \widehat{k}_{e0}\right) &= (100, 100.7, 1) \\
\Omega_R &= \begin{pmatrix} 0.2^2 & 0 \\ 0 & 0.25^2 \end{pmatrix}, \quad \sigma_R = 10
\end{aligned}$$

The PK and the PD data are simulated using the following treatment.

- Phase I study
  - Single dose and multiple doses
  - Parallel dose escalation design
  - 25 subjects per dose
  - Single doses: 1.25, 5, 10, 20, and 40 mg
  - PK: plasma concentration of parent drug (c)
  - PD response: Emax function of effect compartment concentration (R)
  - PK and PD measured at 0.083, 0.167, 0.25, 0.5, 0.75, 1, 2, 3, 4, 6, 8, 12, 18, and 24 hours
- Phase IIa trial in patients
  - 100 subjects
  - Multiple doses: 20 mg
  - sparse PK and PD data (3-6 samples per patient)

FIGURE 13. Stan language for fitting an effect compartment model using `linOdeModel` (abstract)

```
transformed parameters {
  for(j in 1:nSubjects){
                      ⋮
  Omega = quad_form_diag(rho, omega);

  for(j in 1:nSubjects){
    CL[j] = exp(logtheta[j, 1]) * (weight[j] / 70)^0.75;
    Q[j] = exp(logtheta[j, 2]) * (weight[j] / 70)^0.75;
    V1[j] = exp(logtheta[j, 3]) * weight[j] / 70;
    V2[j] = exp(logtheta[j, 4]) * weight[j] / 70;
    ka[j] = exp(logtheta[j, 5]);
    ke0[j] = exp(logKe0[j]);
    EC50[j] = exp(logEC50[j]);

    k10 = CL[j] / V1[j];
    k12 = Q[j] / V1[j];
    k21 = Q[j] / V2[j];
    ke0[j] = exp(logKe0[j]);
    EC50[j] = exp(logEC50[j]);

    K = rep_matrix(0, 4, 4);

    K[1, 1] = -ka[j];
    K[2, 1] = ka[j];
    K[2, 2] = -(k10 + k12);
    K[2, 3] = k21;
    K[3, 2] = k12;
    K[3, 3] = -k21;
    K[4, 2] = ke0[j];
    K[4, 4] = -ke0[j];

    x[start[j]:end[j],] = linOdeModel(time[start[j]:end[j]],
                                      amt[start[j]:end[j]],
                                      rate[start[j]:end[j]],
                                      ii[start[j]:end[j]],
                                      evid[start[j]:end[j]],
                                      cmt[start[j]:end[j]],
                                      addl[start[j]:end[j]],
                                      ss[start[j]:end[j]],
                                      K, biovar, tlag);

    cHat[start[j]:end[j]] = 1000 * x[start[j]:end[j], 2] ./ V1[j];
    ceHat[start[j]:end[j]] = 1000 * x[start[j]:end[j], 4] ./ V1[j];
    respHat[start[j]:end[j]] = 100 * ceHat[start[j]:end[j]] ./
      (EC50[j] + ceHat[start[j]:end[j]]);
  }

  cHatObs = cHat[iObs];
  respHatObs = respHat[iObs];
}
                      ⋮
}
```

The model is simultaneously fitted to the PK and the PD data. For this effect compartment model, we construct a constant rate matrix and use `linOdeModel`. Correct use of Torsten requires the user pass the entire event history (observation and dosing events) for an individual to the function. Thus the Stan model shows the call to `linOdeModel` within a loop over the individual subjects rather than over the individual observations.

Results. We use the same diagnosis tools as for the previous example. The MCMC history plots (figure 14) suggest the 4 chains have converged to common distributions. We note some minor auto-correlations for *lp_* (the log posterior) and for IIV parameters: specifically Ω_{ke_0} and ρ. The correlation matrix ρ does not explicitly appear in the model, but it is used to construct Ω, which parametrizes the PK IIV. The fits to the plasma concentration (figure 16) are in close agreement with the data, notably for the sparse data case (phase IIa study). The fits to the PD data (figure 17) look good, though the data is more noisy. The model reflects the noise by producing larger credible intervals. The estimated values of the parameters are consistent with the values used to simulate the data.

Table 3. Summary of the MCMC simulations of the marginal posterior distributions of the model parameters for example 2

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
CLHat	10.523	0.003	0.201	9.712	9.958	10.096	10.231	10.483	4000.000	0.999
QHat	14.867	0.014	0.357	14.182	14.620	14.862	15.106	15.563	678.208	1.007
V1Hat	34.188	0.067	1.089	31.940	33.494	34.214	34.918	36.251	267.748	1.016
V2Hat	103.562	0.076	2.925	98.031	101.600	103.455	105.472	109.583	488.296	1.001
kaHat	1.930	0.004	0.077	1.771	1.880	1.933	1.982	2.076	334.888	1.014
ke0Hat	1.050	0.001	0.044	0.967	1.020	1.051	1.078	1.137	164.741	1.000
EC50Hat	104.337	0.040	2.100	100.169	102.909	104.345	105.768	108.351	744.041	1.000
sigma	0.099	0.000	0.002	0.095	0.097	0.099	0.100	0.103	906.342	1.002
sigmaResp	10.156	0.003	0.197	9.779	10.023	10.154	10.286	10.552	4000.000	1.000
omega[1]	0.270	0.000	0.016	0.241	0.259	0.269	0.280	0.302	4000.000	1.001
omega[2]	0.231	0.001	0.021	0.192	0.217	0.230	0.245	0.275	531.512	1.006
omega[3]	0.219	0.002	0.031	0.158	0.199	0.218	0.238	0.281	158.198	1.017
omega[4]	0.267	0.001	0.026	0.218	0.249	0.266	0.284	0.319	684.870	1.001
omega[5]	0.285	0.002	0.037	0.214	0.259	0.284	0.309	0.361	284.545	1.009
omegaKe0	0.271	0.003	0.047	0.183	0.239	0.271	0.303	0.363	217.350	1.007
omegaEC50	0.213	0.001	0.021	0.174	0.199	0.213	0.227	0.255	190.193	1.000

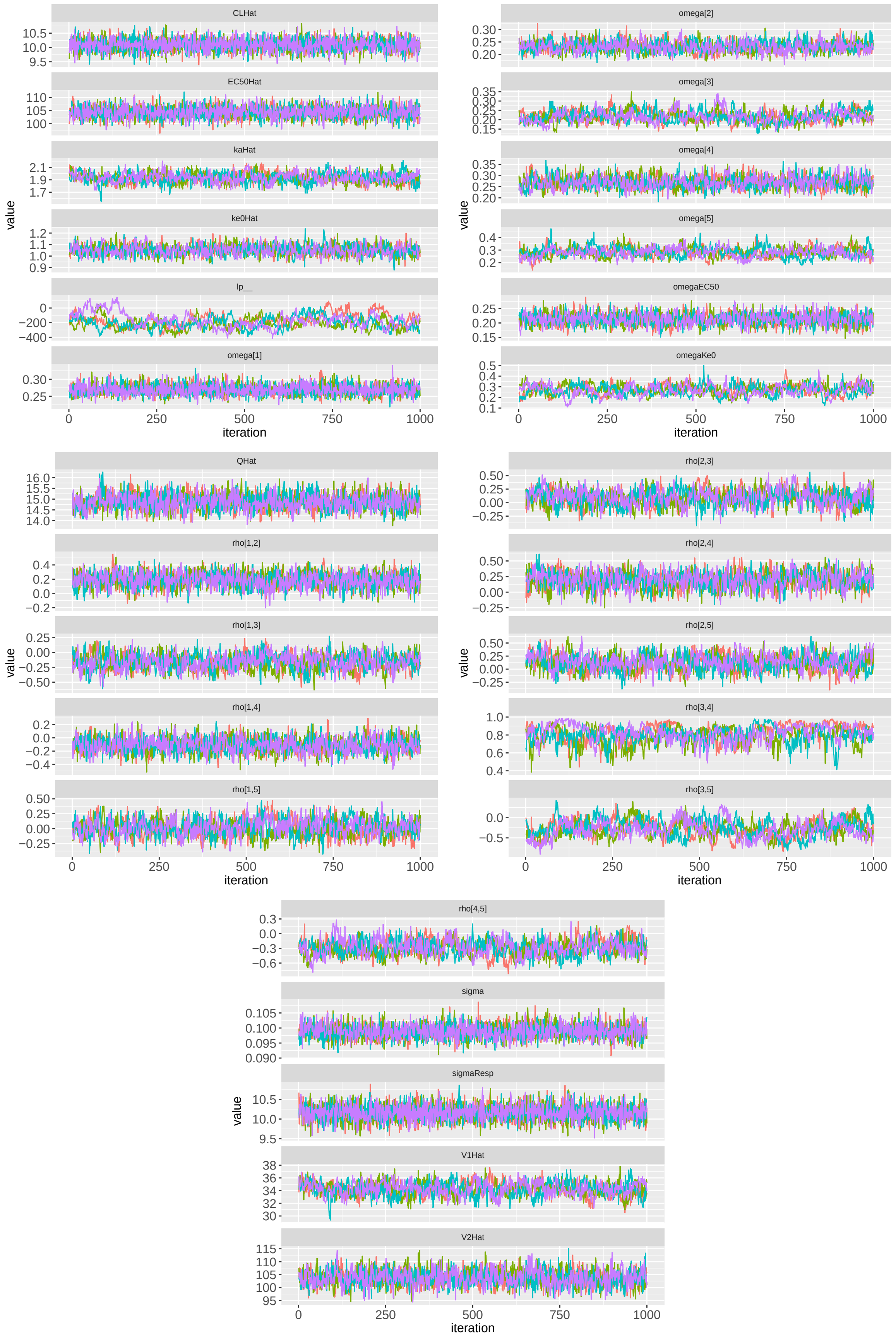

Figure 14. MCMC history plots for the parameters of an Effect Compartment Model (each color corresponds to a different chain) for example 2

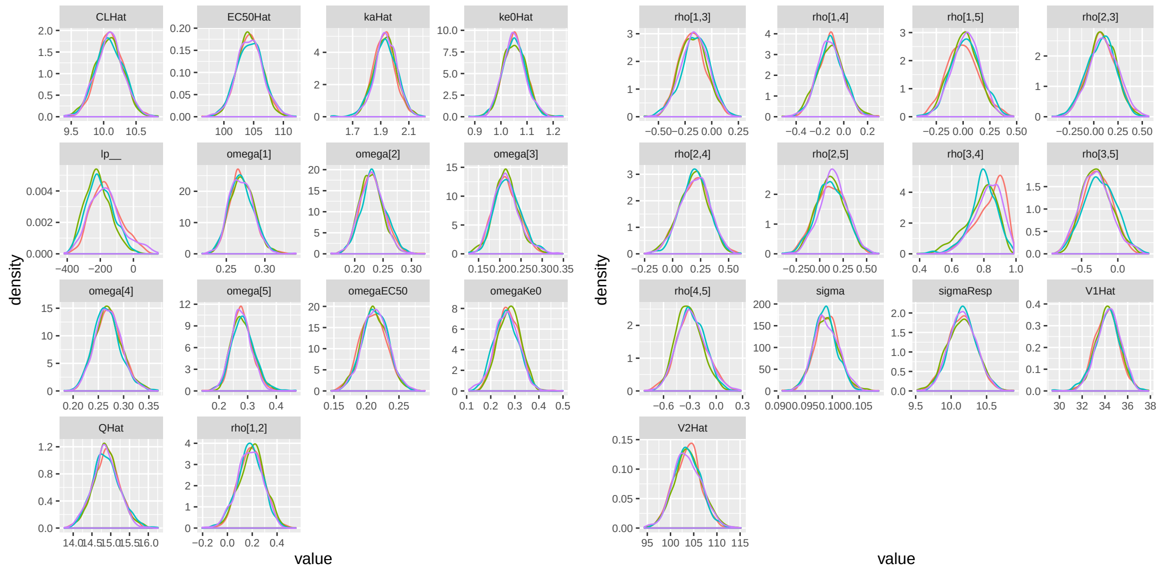

FIGURE 15. Posterior Marginal Densities of the Model Parameters of an Effect Compartment Model (each color corresponds to a different chain) for example 2

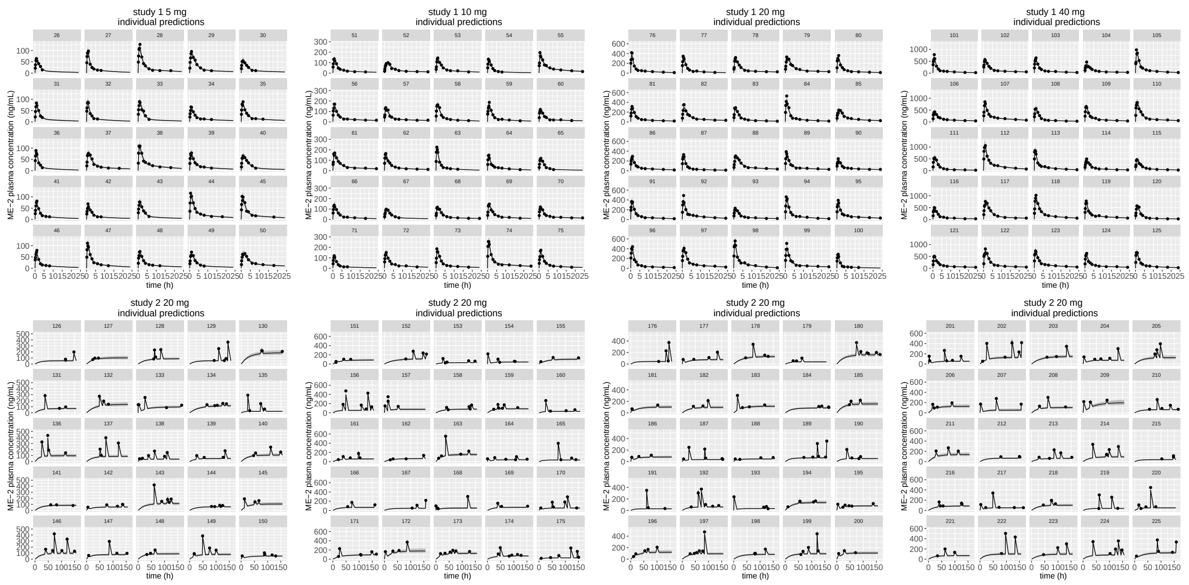

FIGURE 16. Predicted (posterior median and 90 % credible intervals) and observed plasma drug concentrations for example 2 for an Effect Compartment Model

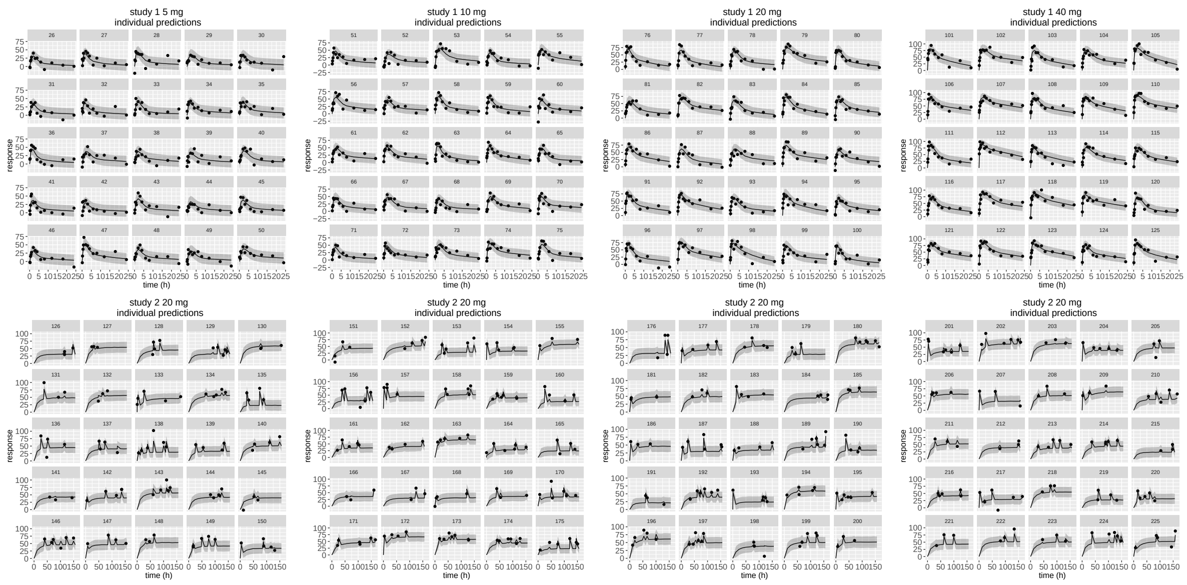

FIGURE 17. Predicted (posterior median and 90 % credible intervals) and observed PD Response for example 2

3.2. **Example 4: Friberg-Karlsson Semi-Mechanistic Population Model.**

We now return to example 2 and extend it to a population model. While we recommend using the mixed solver, for completeness we'll show how to specify the model with the `generalOdeModel_*` function. We leave it as an exercise to the reader to rewrite the model with `mixOde2CptModel_*`.

Friberg-Karlsson Population Model for drug-induced myelosuppression (ANC)

$$\begin{aligned}
\log(ANC_{ij}) &\sim N(Circ_{ij}, \sigma^2_{ANC}) \\
\log\left(MTT_j, Circ_{0j}, \alpha_j\right) &\sim N\left(\log\left(\widehat{MTT}, \widehat{Circ_0}, \widehat{\alpha}\right), \Omega_{ANC}\right) \\
\left(\widehat{MTT}, \widehat{Circ_0}, \widehat{\alpha}, \gamma\right) &= (125, 5, 2, 0.17) \\
\Omega_{ANC} &= \begin{pmatrix} 0.2^2 & 0 & 0 \\ 0 & 0.35^2 & 0 \\ 0 & 0 & 0.2^2 \end{pmatrix}, \quad \sigma_{ANC} = 0.1 \\
\Omega_{PK} &= \begin{pmatrix} 0.25^2 & 0 & a0 & 0 & 0 \\ 0 & 0.4^2 & 0 & 0 & 0 \\ 0 & 0 & 0.25^2 & 0 & 0 \\ 0 & 0 & 0 & 0.4^2 & 0 \\ 0 & 0 & 0 & 0 & 0.25^2 \end{pmatrix}
\end{aligned}$$

The PK and the PD data are simulated using the following treatment.

- Phase IIa trial in patients
  - Multiple doses: 80,000 mg
  - Parallel dose escalation design
  - 15 subjects
  - PK: plasma concentration of parent drug (c)
  - PD response: Neutrophil count (ANC)
  - PK measured at 0.083, 0.167, 0.25, 0.5, 0.75, 1, 2, 3, 4, 6, 8, 12, 18, and 24 hours
  - PD measured once every two days for 28 days.

Once again, we simultaneously fit the model to the PK and the PD data. Note that from a computational perspective, this is a much more difficult problem than the one we dealt with in the previous example. The nonlinear nature of the ODEs forces us to use a numerical solver, which is significantly slower than the linear methods we have employed so far. Because the ODE system of interest is non-stiff, we use the *rk45* version of `genOdeModel`.

It pays off to construct informative priors. For instance, we could fit the PK data first, as was done in example 1, and get informative priors on the PK parameters. The PD parameters are drug independent, so we can use information from the neutropenia literature. In this example, we choose to use weakly informative priors on the PK parameters and strongly informative priors on the PD parameters.

Since it takes a long time to run the model, we only use 100 iterations per chain, and study what we can learn from this less than optimal scenario. It is worth noting that Stan, because of its highly efficient MCMC sampler, still does a reasonable job estimating the posterior distribution.

Figure 18. Stan language for coding an ODE system describing a Friberg-Karlsson Mechanism

```
real[] twoCptNeutModelODE(real t,
               real[] x,
               real[] parms,
               real[] rdummy,
               int[] idummy){
real CL = parms[1];
real Q = parms[2];
real V2 = parms[3];
real V3 = parms[4];
real ka = parms[5];
real mtt = parms[6];
real circ0 = parms[7];
real gamma = parms[8];
real alpha = parns[9];
real k10 = CL / V2;
real k12 = Q / V2;
real k21 = Q / V3;
real ktr = 4 / mtt;
real dxdt[8];
real conc;
real EDrug;
real transit1;
real transit2;
real transit3;
real circ;
real prol;

dxdt[1] = -ka * x[1];
dxdt[2] = ka * x[1] - (k10 + k12) * x[2] + k21 * x[3];
dxdt[3] = k12 * x[2] - k21 * x[3];
conc = x[2]/V1;
EDrug = alpha * conc;
// x[4], x[5], x[6], x[7] and x[8] are differences from circ0.
prol = x[4] + circ0;
transit1 = x[5] + circ0;
transit2 = x[6] + circ0;
transit3 = x[7] + circ0;
circ = fmax(machine_precision(), x[8] + circ0); // Device for implementing a modeled
                                                  // initial condition
dxdt[4] = ktr * prol * ((1 - EDrug) * ((circ0 / circ)^gamma) - 1);
dxdt[5] = ktr * (prol - transit1);
dxdt[6] = ktr * (transit1 - transit2);
dxdt[7] = ktr * (transit2 - transit3);
dxdt[8] = ktr * (transit3 - circ);

return dxdt;
}
```

Results. The MCMC history plots are not as convincing as in the previous examples, mostly because the number of iterations is small (100 versus 1000 in the previous example). It does however look as though the chains are converging to a common distribution, and we see little auto-correlation (in particular, we expect that if we had run the model for 1000 iterations, we would obtain the desired "fuzzy caterpillar" look). The plots of the marginal posterior distributions clearly show that the chains have not (yet) converged to a common distribution, but they do not disagree significantly. Still, the need for more iterations is evident. The model fits the data, and the credible interval reflect the noise in the data. The parameters estimation reflects the real value of the parameters.

Figure 19. Stan language for fitting a Friberg-Karlsson model using `genCptModel_rk45` (abstract)

```
transformed parameters {
                     ⋮
  for(i in 1:nSubjects) {

    parms[1] = thetaM[i, 1] * (weight[i] / 70)^0.75; # CL
    parms[2] = thetaM[i, 2] * (weight[i] / 70)^0.75; # Q
    parms[3] = thetaM[i, 3] * (weight[i] / 70); # V1
    parms[4] = thetaM[i, 4] * (weight[i] / 70); # V2
    parms[5] = kaHat; # ka
    parms[6] = thetaM[i, 5]; # mtt
    parms[7] = thetaM[i, 6]; # circ0
    parms[8] = gamma;
    parms[9] = thetaM[i, 7]; # alpha

    x[start[i]:end[i]] = generalOdeModel_rk45(twoCptNeutModelODE, 8,
                                             time[start[i]:end[i]],
                                             amt[start[i]:end[i]],
                                             rate[start[i]:end[i]],
                                             ii[start[i]:end[i]],
                                             evid[start[i]:end[i]],
                                             cmt[start[i]:end[i]],
                                             addl[start[i]:end[i]],
                                             ss[start[i]:end[i]],
                                             parms, biovar, tlag,
                                             1e-6, 1e-6, 1e6);

    cHat[start[i]:end[i]] = x[start[i]:end[i], 2] / parms[1][3]; # divide by V1
    neutHat[start[i]:end[i]] = x[start[i]:end[i], 8] + parms[1][7]; # Add baseline

  }

  cHatObs = cHat[iObsPK];
  neutHatObs = neutHat[iObsPD];

                     ⋮
```

Table 4. Summary of the MCMC simulations of the marginal posterior distributions of the model parameters for example 3

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
CL	9.986	0.009	0.174	9.641	9.872	9.982	10.107	10.331	400.000	0.997
Q	14.633	0.055	1.106	12.505	13.992	14.623	15.296	16.948	400.000	0.996
V1	32.909	0.174	2.439	28.203	31.186	32.836	34.762	37.750	195.828	1.008
V2	106.631	0.311	6.226	95.234	102.269	106.403	111.000	118.533	400.000	0.999
ka	1.882	0.012	0.175	1.582	1.756	1.871	2.006	2.223	196.052	1.007
sigma	0.106	0.001	0.010	0.089	0.098	0.105	0.112	0.132	259.693	1.009
alpha	3.3E-04	1.4E-06	2.2E-05	2.9E-04	3.2E-04	3.3E-04	3.5E-04	3.8E-04	247	1.01
mtt	132.763	0.515	6.498	120.843	128.082	132.223	136.694	146.845	159.372	1.024
circ0	5.014	0.009	0.172	4.711	4.888	5.000	5.138	5.334	400.000	1.000
gamma	0.190	0.002	0.022	0.153	0.175	0.187	0.202	0.239	139.485	1.025
sigmaNeut	0.092	0.001	0.014	0.068	0.082	0.090	0.100	0.125	161.199	1.010

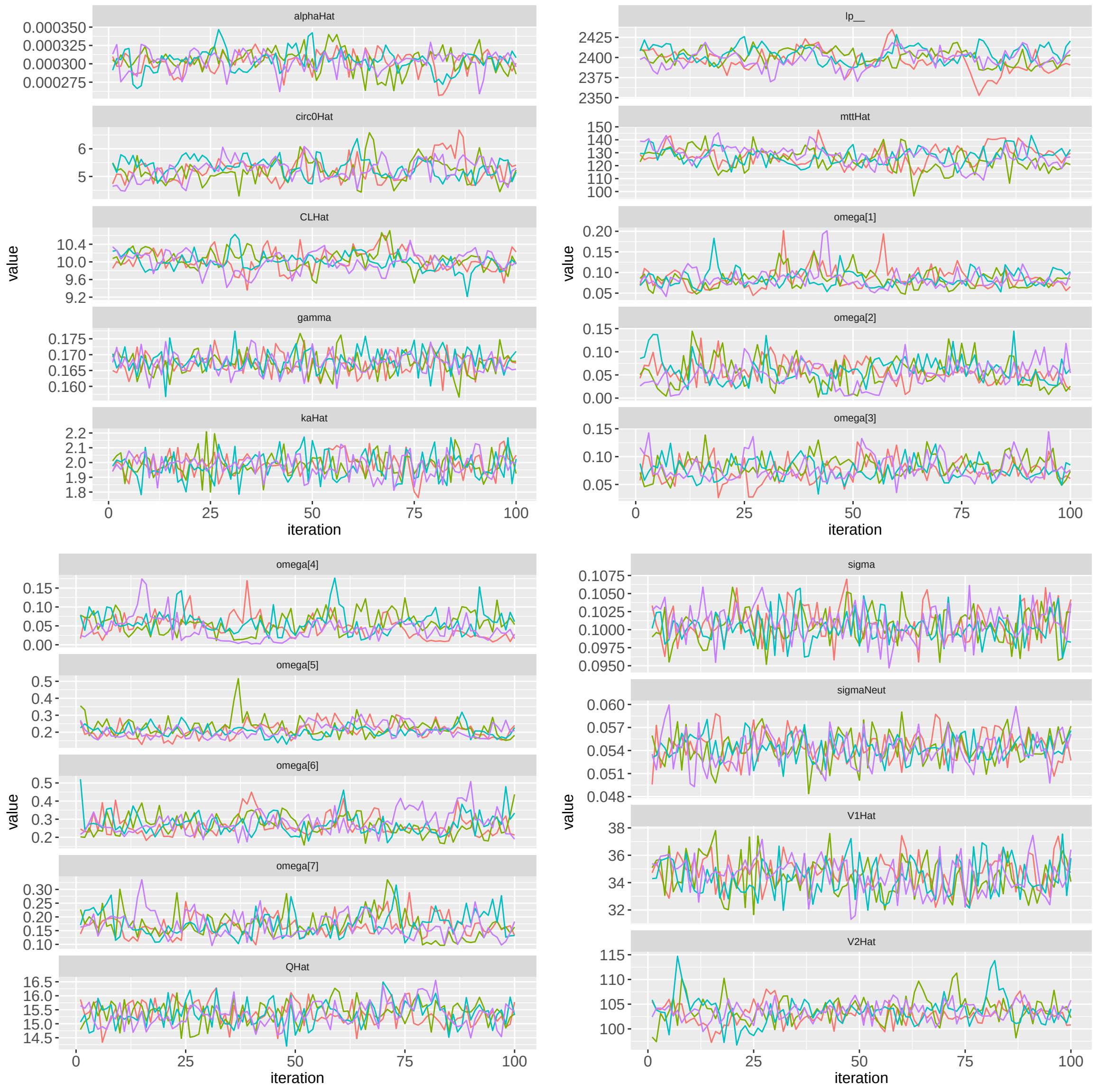

FIGURE 20. MCMC history plots for the parameters of a Friberg-Karlsson semi-mechanistic model (each color corresponds to a different chain) for example 3

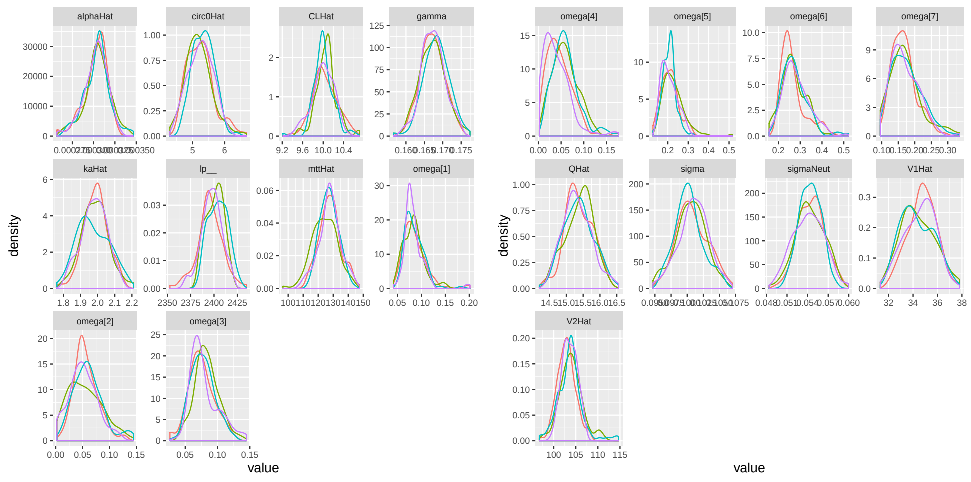

FIGURE 21. Posterior Marginal Densities of the Model Parameters of a Friberg-Karlsson semi-mechanistic model (each color corresponds to a different chain)

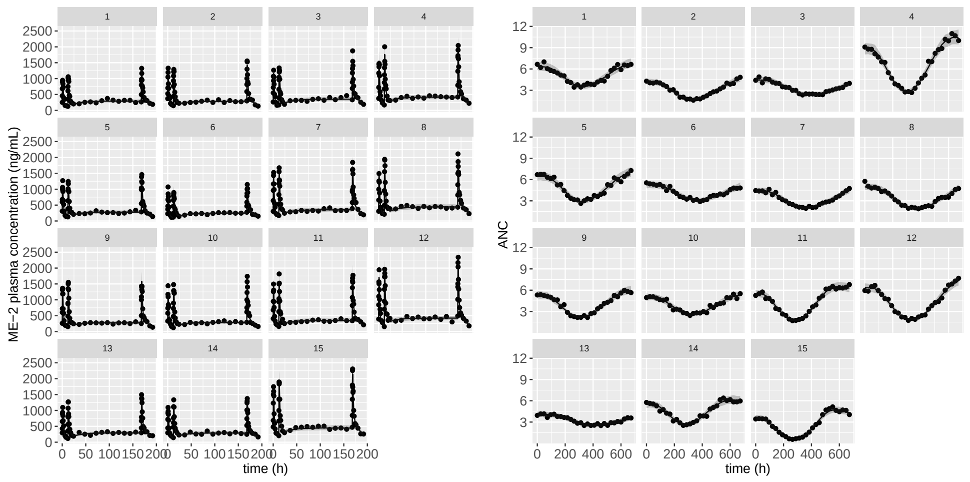

FIGURE 22. Predicted (posterior median and 90 % credible intervals) and observed plasma drug concentrations, and Neutrophil counts, for a Friberg-Karlsson semi-mechanistic model

4. Appendix

(Note: this section is still being worked on and is far from finished)

4.1. **Solving Systems of Algebraic Equations.**

In order to compute a steady state solution for a general compartment model, we need an algebraic solver. Such a solver will be added to Stan's core language[6]. A prototype is available in Torsten's development branch on GitHub.

The `algebra_solver` function uses the Powell hybrid method (also called the "dogleg" method)[7] to find the solution to a system of nonlinear algebraic equations. In other words, we wish to solve:

$$f(x) = 0$$

for x, where f and x are vector valued functions.

The user first defines f, the algebraic system, in the **functions** block, using a pre-specified signature:

```
real[] f(vector x, // unknown
         vector y, // parameters
         real[] dat_r, // data (real)
         int[] dat_int)  // data (integer)
```

The algebraic solver has the form:

```
algebra_solver(f, x, theta, x_r, x_i
               rel_tol, f_tol, max_step)
```

where `f` is the algebraic system defined in the function block, `x` a vector which contains the initial guesses for the solution, `y` a vector of parameters, `dat_r` an array of real data, and `dat_int` an array of integer data.

The last three arguments are optional and specify the tuning parameters of the solver. The relative tolerance (`rel_tol`, default value 1×10^{-10}) sets an upper bound for the error in the computed solution relative to the real solution. Given an estimated solution θ, the function tolerance (`f_tol`, default value 1×10^{-6}) specifies the maximum value of $||f(\theta)||$[7]. The maximum number of steps (`max_num_steps`, default value 1000) is the maximum number of times the solver will compute $f(x)$ before "giving up". While these parameters have default values, the user should be prepared to adjust them.

4.1.1. *Example.*

Let's suppose we wish to solve the following algebraic system:

[6]A pull request has been submitted on Stan's gitHub page. This prototype builds on the Hybrid Nonlinear solver from the C++ *Eigen* library[5]. Their implementation follows closely the one in MINPACK-1[6].

[7]It helps to think geometrically about this. Ideally we would want the point $f(\theta)$ to be at the origin; the norm represents deviations from this ideal. Users should keep in mind the norm scales up with the square-root of x's dimension.

$$\begin{aligned} z_1 &= x_1 - y_1 \\ z_2 &= x_2 - y_2 \end{aligned}$$

where $(y_1, y_2) = (0.9, 1.1)$. We would code this in Stan as follows:

Figure 23. Stan language for solving an algebraic system

```
functions{
 real[] algebraSystem(vector x,
                      vector y,
                      real[ ] dat,
                      int[ ] dat_int) {
   vector[2] f_x;
   f_x[1] = x[1] - y[1];
   f_x[2] = x[2] - y[2];
   return f_x;
 }
}
   ⋮

transformed data {
 vector[2] x;
 vector[2] y;
 real dat[0];
 int dat_int[0];

 x[1] = 1;
 x[2] = 1;
}

transformed parameters{
 vector y[2];
 vector theta[2];

 y_p[1] = 0.9;
 y_p[2] = 1.1;

 theta = algebra_solver(algebra_system, x, y, dat, dat_int);
}
   ⋮
```

4.2. **Piecewise linear interpolation.**

Torsten provides a function for piecewise linear interpolation over a set of x, y pairs. It returns the values of a piecewise linear function at specified values (`xout`) of the first function argument. The function is specified in terms of a set of x, y pairs. The x values must be in increasing order. All 3 arguments may be data or parameters.

The Stan function linear_interpolation implements the following function.

$$y_{\text{out}} = \begin{cases} y_1, & x_{\text{out}} < x_1 \\ y_i + \frac{y_{i+1}-y_i}{x_{i+1}-x_i}\left(x_{\text{out}} - x_i\right), & x_{\text{out}} \in [x_i, x_{i+1}) \\ y_n, & x_{\text{out}} \geq x_n \end{cases}$$

where

$$x = \{x_1, x_2, \dots, x_n\}$$
$$y = \{y_1, y_2, \dots, y_n\}$$
$$x_{i+1} > x_i \;\forall\; i$$

The following function signatures are currently implemented:

```
real linear_interpolation(real xout, real[] x, real[] y)
real[] linear_interpolation(real[] xout, real[] x, real[] y)
```

Use of linear_interpolation is illustrated in a Stan model shown in Figure 24 for fitting a piecewise linear function to a data set consisting of a set of x, y pairs. Complete code for an example using that model is available on GitHub: `https://github.com/metrumresearchgroup/example-models`. The example is named testInterp2.

4.3. **Implementing Torsten.**

Stan's `math` library is written in C++, which offers a great deal of speed and flexibility. The Stan language provides a very handy interface that allows us to focus on statistical modeling and saves us the trouble of doing extensive coding in C++. At run time, a *make* file translates our Stan model into C++, which then gets compiled and executed. Accordingly, there are two steps to add a function to Stan: (1) write the procedure in C++, (2) expose the procedure to the language so users may use it in a Stan file.

The Stan code is open-source and available on GitHub. It is compartmentalized into several repos: `math` contains the mathematical functions, `stan` exposes these functions. Other repos provide code to interface Stan with higher level languages, such as R and Python. Torsten exists as a forked version of `math` and `stan`. Other repos remain unchanged.

Regularly, we merge Stan's latest release into Torsten.

Modifications in math. All Torsten files are located in the `Torsten` directory, under `stan/math`. The code can be found on GitHub: `https://github.com/metrumresearchgroup/math`

Modifications in Stan. We do further modifications in `stan` to expose Torsten's functions. We edit `function_signatures.h` to expose `PKModelOneCpt`, `PKModelTwoCpt`, and `linOdeModel`. The general ODE model functions are higher-order functions (i.e. they take another function as one of their arguments). They are exposed by directly modifying the grammar files, following closely the example of `integrate_ode_rk45` and `integrate_ode_bdf`.

The code can be found on GitHub: `https://github.com/metrumresearchgroup/stan`.

References

[1] Friberg, L.E. and Karlsson, M.O. Mechanistic models for myelosuppression. *Invest New Drugs* **21** (2003):183–194.

FIGURE 24. Stan language model illustrating use of the linear_interpolation function.

```
data{
  int nObs;
  real xObs[nObs];
  real yObs[nObs];
  int nx;
  int nPred;
  real xPred[nPred];
}

transformed data{
  real xmin = min(xObs);
  real xmax = max(xObs);
}

parameters{
  real y[nx];
  real<lower = 0> sigma;
  simplex[nx - 1] xSimplex;
}

transformed parameters{
  real yHat[nObs];
  real x[nx];

  x[1] = xmin;
  x[nx] = xmax;
  for(i in 2:(nx-1))
    x[i] = x[i-1] + xSimplex[i-1] * (xmax - xmin);

  yHat = linear_interpolation(xObs, x, y);
}

model{
  xSimplex ~ dirichlet(rep_vector(1, nx - 1));
  y ~ normal(0, 25);
  yObs ~ normal(yHat, sigma);
}

generated quantities{
  real yHatPred[nPred];
  real yPred[nPred];

  yHatPred = linear_interpolation(xPred, x, y);
  for(i in 1:nPred)
    yPred[i] = normal_rng(yHatPred[i], sigma);
}
```

[2] Carpenter, B., Gelman, A., Hoffman, M., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M.A., Guo, J., Li, P. and Riddel, A. Stan: A probablistic programming language. *Journal of Statistical Software (in press)* (2016).

[3] Hoffman, M.D. and Gelman, A. The no-u-turn sampler: Adaptively setting path lengths in hamiltonian monte carlo. *Journal of Machine Learning Research* (2014):1593–1623.

[4] Carpenter, B., Hoffman, M.D., Brubaker, M.A., Lee, D., Li, P. and Betancourt, M.J. The stan math library: Reverse-mode automatic differentiation in c++. *arXiv 1509.07164.* (2015).

[5] Baron, K.T., Hindmarsh, A.C., Petzold, L.R., Gillespie, W.R., Margossian, C.C. and Pastoor, D. *mrgsolve: Simulate from ODE-Based Population PK/PD and Systems Pharmacology Models*. Metrum Research Group (2017). R package version 0.8.9.
URL `https://cran.r-project.org/package=mrgsolve`

[6] Guennebaud, G., Jacob, B. and et al. Eigen version 3 (2010).
URL `http://eigen.tuxfamily.org`

[7] Jorge J. More, Burton S. Garbow, K.E.H. *User Guide for MINPACK-1*. Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439 (1980).

[8] Powell, M.J.D. A hybrid method for nonlinear equations. *Numerical Methods for Nonlinear Algebraic Equations* **7** (1970):87–114.