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`)!

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 linear 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* [5], 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).

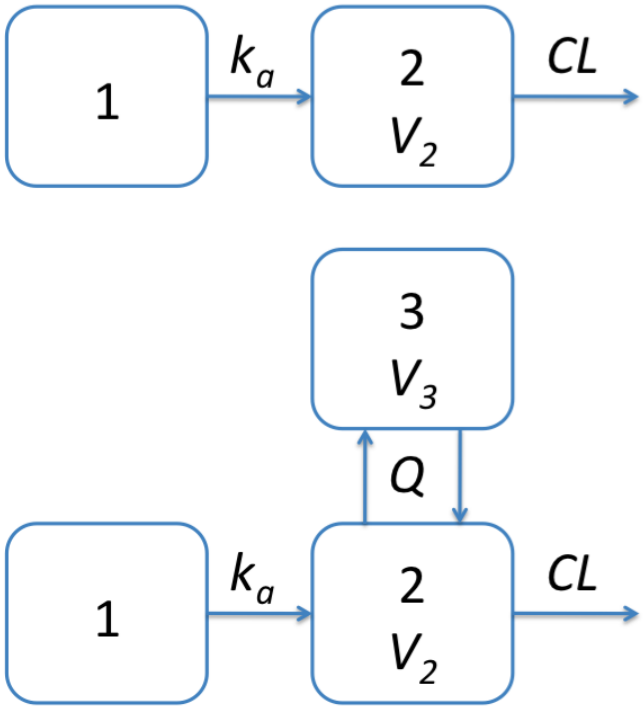

FIGURE 1. One and two compartment models with first order absorption implemented in Torsten.

`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.

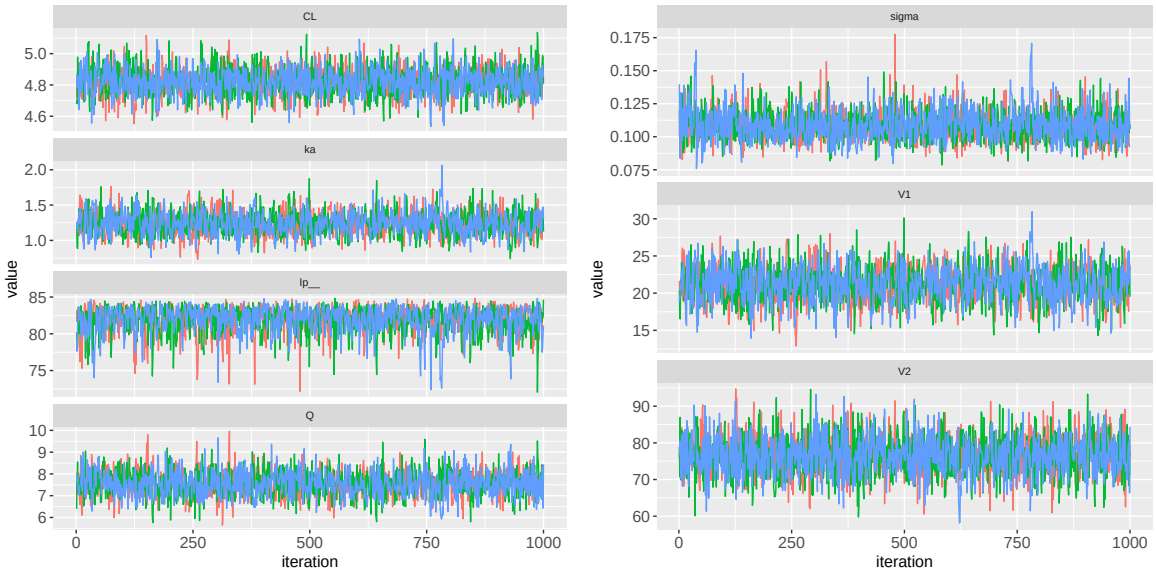

FIGURE 3. MCMC history plots for the parameters of a two compartment model with first order absorption (each color corresponds to a different chain)

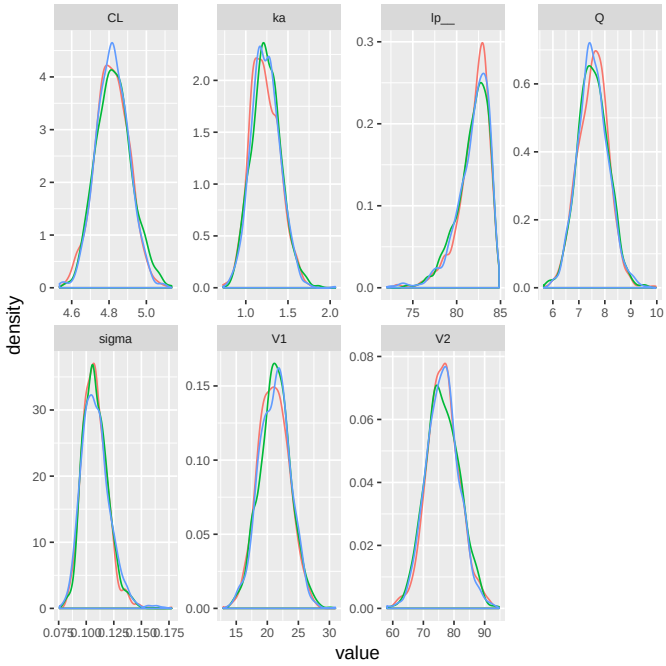

FIGURE 4. Posterior Marginal Densities of the Model Parameters of a two compartment model with first order absorption (each color corresponds to a different chain)

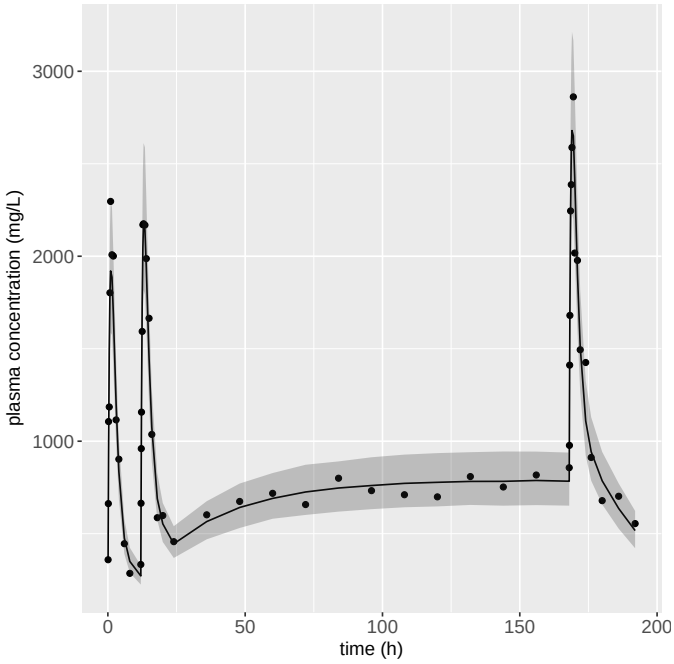

FIGURE 5. Predicted (posterior median and 90 % credible intervals) and observed plasma drug concentrations of a two compartment model with first order absorption

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 [6].

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.

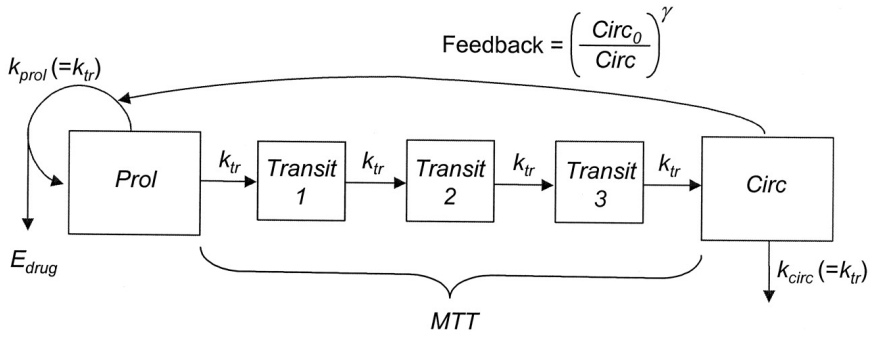

FIGURE 9. Friberg-Karlsson semi-mechanistic Model [1]

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).

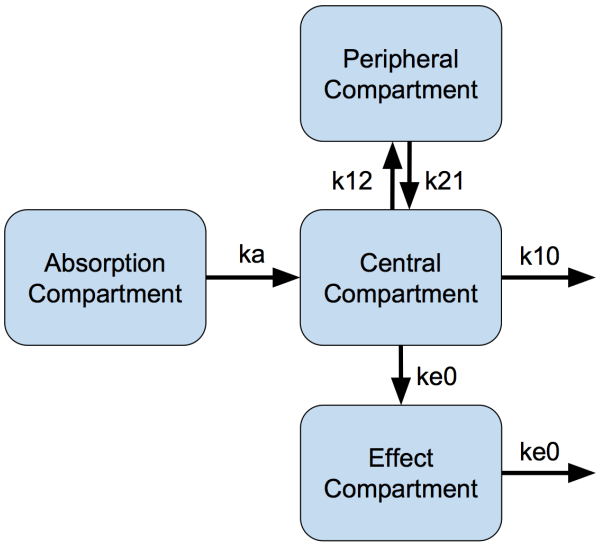

FIGURE 12. Effect Compartment Model

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

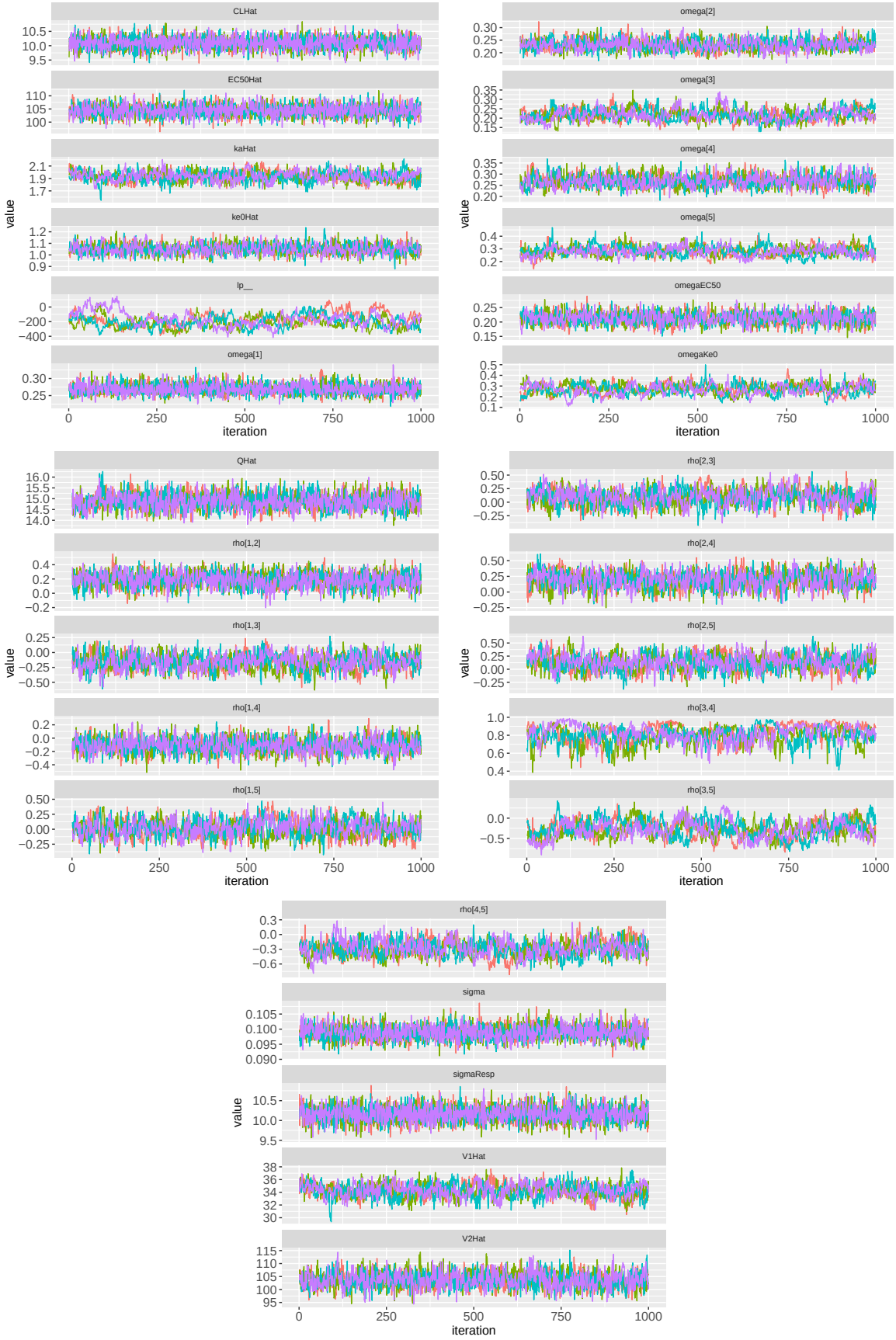

FIGURE 14. MCMC history plots for the parameters of an Effect Compartment Model (each color corresponds to a different chain) for example 2

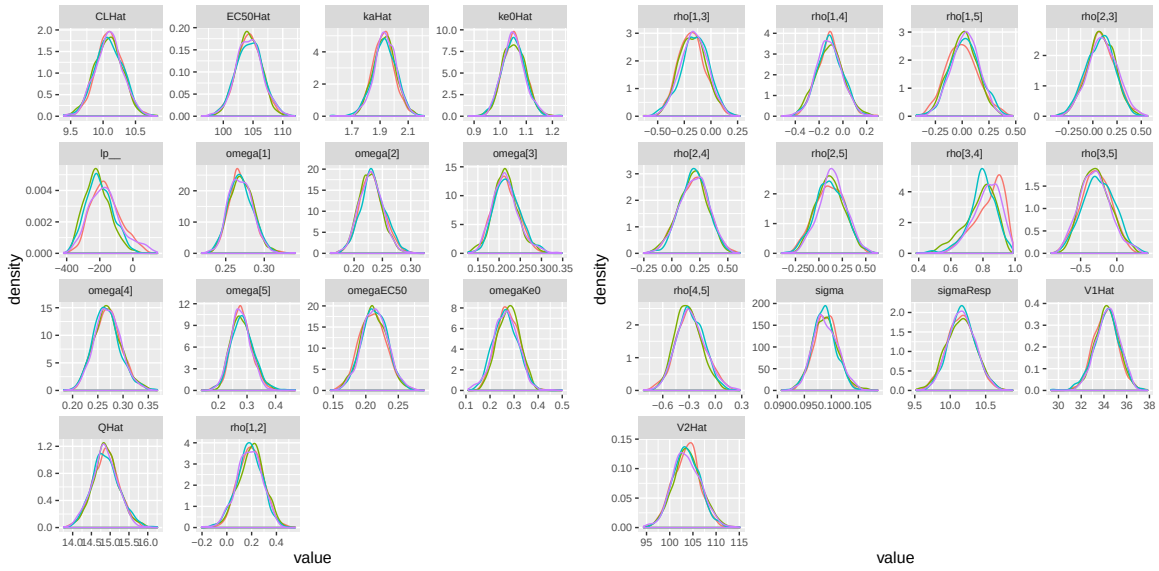

FIGURE 15. Posterior Marginal Densities of the Model Parameters of an Effect Compartment Model (each color corresponds to a different chain) for example 2

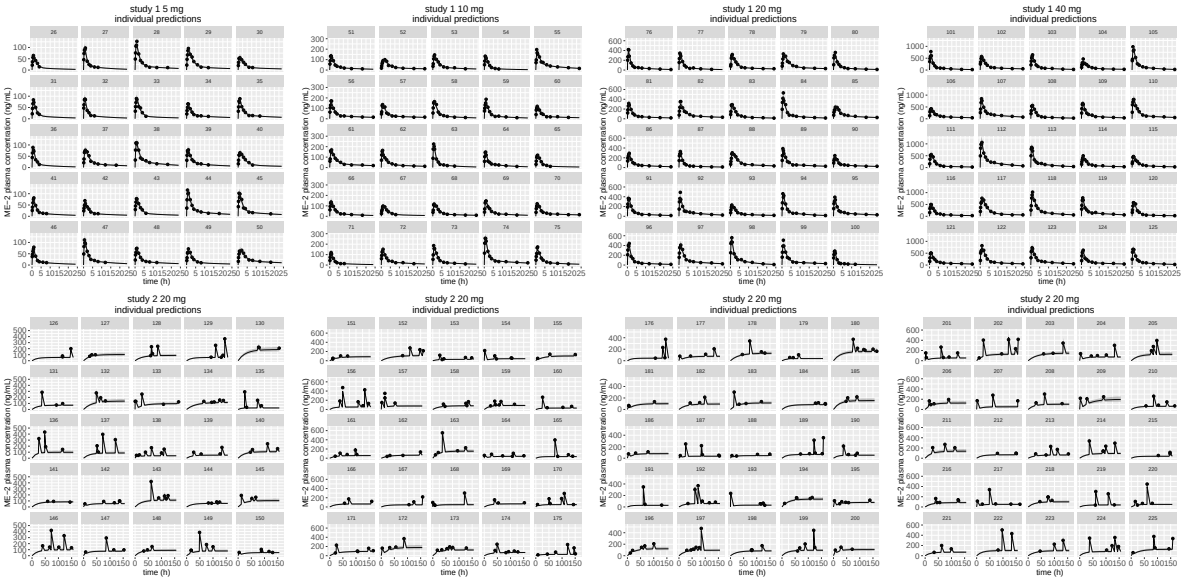

FIGURE 16. Predicted (posterior median and 90 % credible intervals) and observed plasma drug concentrations for example 2 for an Effect Compartment Model

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

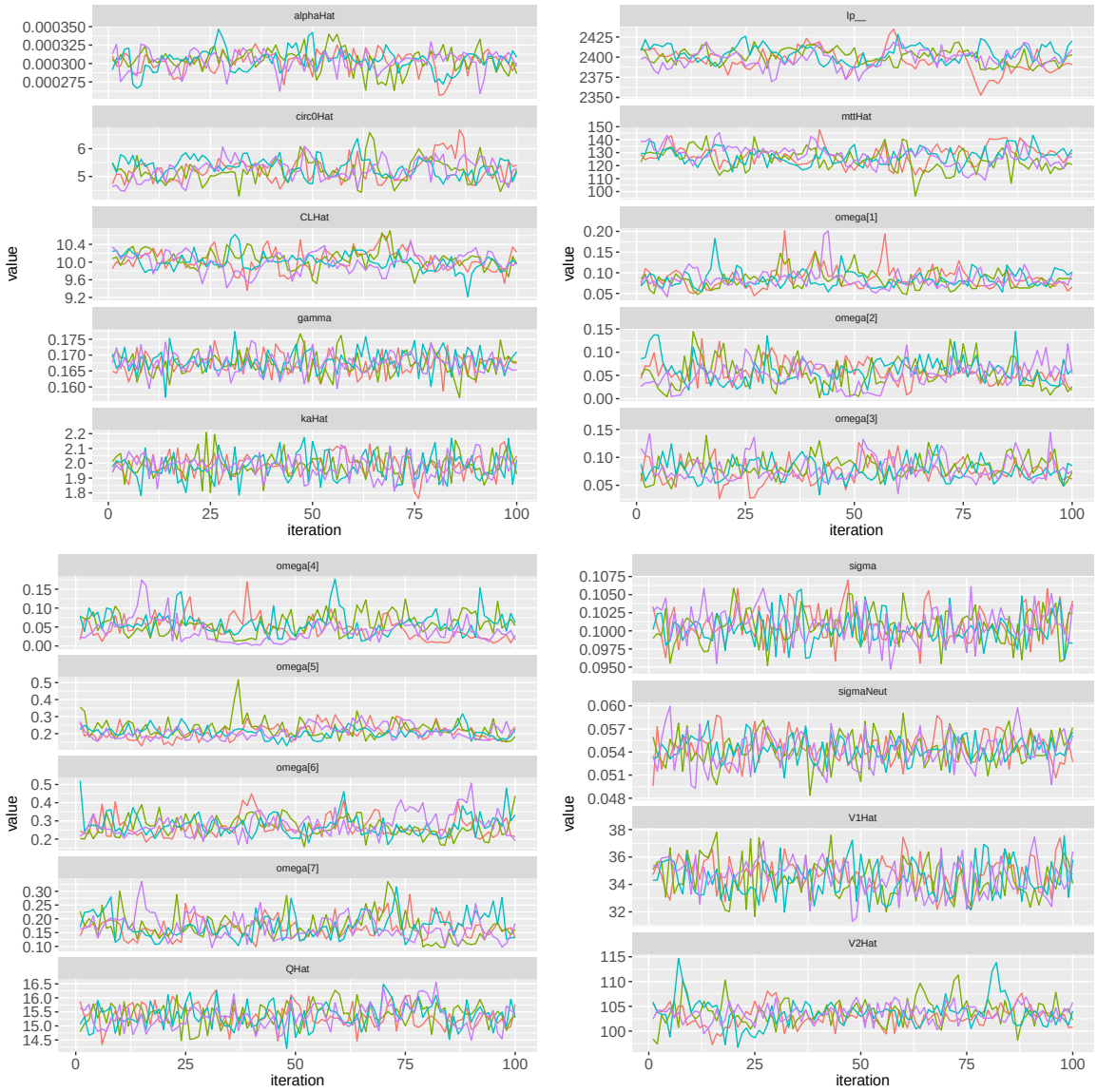

FIGURE 20. MCMC history plots for the parameters of a Friberg-Karlsson semi-mechanistic model (each color corresponds to a different chain) for example 3

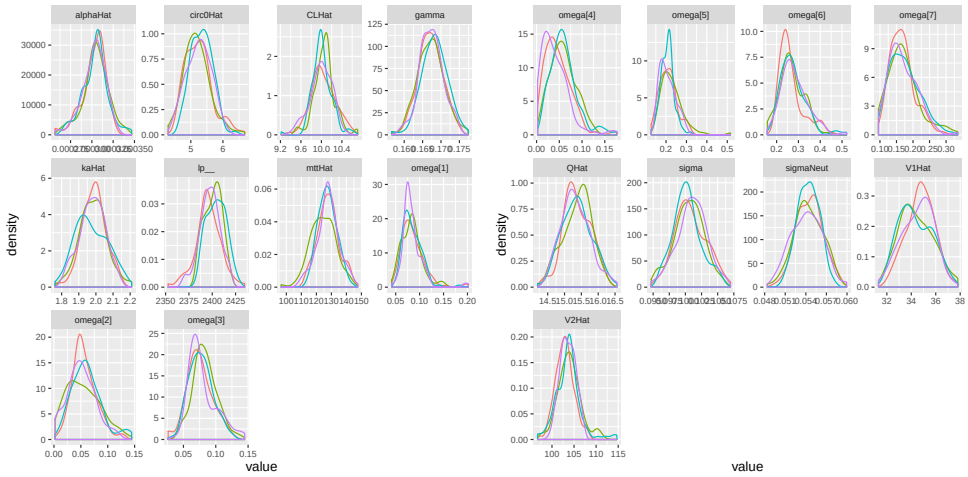

FIGURE 21. Posterior Marginal Densities of the Model Parameters of a Friberg-Karlsson semi-mechanistic model (each color corresponds to a different chain)

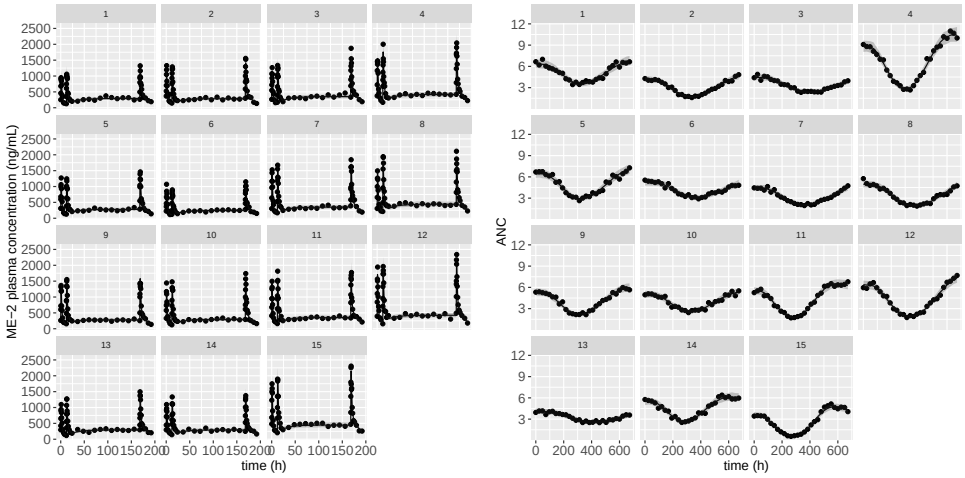

FIGURE 22. Predicted (posterior median and 90 % credible intervals) and observed plasma drug concentrations, and Neutrophil counts, for a Friberg-Karlsson semi-mechanistic model

TECHNICAL APPENDIX

[Pharmacokinetic's] very essence is the use of mathematical formulations based on certain biophysical or biochemical models of the living body, and we must never forget that we are dealing with models, not necessarily realistic facts. [...] And yet, it cannot be denied that classical mathematical pharmacokinetics have been proved to be extremely useful for description, explanation, and even prediction of the distribution of drugs and have helped "optimize" drug administration.

But sometimes pharmacokinetics in its present state is a failure - where the basic assumptions are not valid or are too simplified. In fact, for an oldtimer it is surprising that such an almost naive simple model as the serial multicompartment governed by Fick's diffusion has survived over the years. One way of explaining this is to state that the model may not represent "a diffusional compartment model" at all, but be rather a special case of "sequential chain of first-order reactions," which has an identical mathematical formalism.

Torsten Teorell
Concluding remarks in *Pharmacology and Pharmacokinetics*, 1974 [7]

.

Pharmacometrics is the application of quantitative methods to pharmacology, and has been used to model a broad and diverse range of biomedical processes. The success of quantitative modeling may stem from the fact similar "mathematical formalisms" keep arising again and again; for that matter not only in pharmacometrics, but in all quantitative sciences. This demands care and caution when we make scientific interpretations, but presents many practical advantages. In a way, Torsten, like other pharmacometrics softwares out there, is a tool to make these mathematical formalisms more readily available to modelers. The math, while it may be treated abstractly, is here applied to and always motivated by a scientific problem.

Broadly speaking, pharmacometrics combines biomedicine, biochemistry, mathematics, and statistics. We go a step further when we deal with its software implementation, as we begin to consider problems pertinent to computer science, computational statistics, and numerical analysis. Geometry, it turns out, plays a crutial role and makes some of the theory behind statistical modeling very compelling visually. Mathematical physics, to the delight of one of Torsten's developers, also contributes its fair share.

It is no surprise then that an open-source project, such as Stan or Torsten, brings together scientists with a broad array of backgrounds, which greatly enriches the project, but sometimes makes communication difficult. This appendix attempts to ease conversation between experts in different fields and should be

of interest to advanced users and potential contributors. *The details covered here are not required to use Torsten.*

The focus is on Torsten's design and the algorithms it uses. We point readers to references for more detailed descriptions, and also suggest improvements that can be made to Torsten's current version.

A full understanding of this appendix requires a good knowledge of several topics. Most readers cannot be expected to be experts in all these fields, which is why we review some elementary facts.

Appendix A. Bayesian Data Analysis with Stan

Stan is primarily designed for Bayesian data analysis. It provides users with great flexibility, both for implementing stochastic and deterministic features. This allows users to specify complex mixed-effect models, as the ones we may encounter in pharmacometrics.

Its default algorithm for full bayesian inference is the No-U-Turns Sampler (NUTS) [3]. NUTS, an adaptative Hamiltonian Monte Carlo (HMC) [8] algorithm, has proven more efficient than commonly used random walk Monte Carlo Markov Chains algorithms, such as Metropolis-Hastings and Gibbs sampling, for complex high-dimensional problems [3]. Here, by "complex" we mean non-trivial relationships between the independent and dependent variables, such as ones that involve solving ODEs or hierarchical structures. The dimensionality of a model relates to the number of parameters.

A.1. **Hamiltonian Monte Carlo.**

For this appendix, a quick review will do. For a more detailed introduction on the subject, see Betancourt's *Conceptual Introduction to Hamiltonian Monte Carlo* [9].

The basic idea behind HMC is to treat the Markov chain as a *particle* that moves in the parameter space and the posterior as a *physical potential* that informs the dynamic of that particle. The potential, U, is defined as the negative of the log posterior, π:

$$U = -\log(\pi) \tag{3}$$

Instead of taking a random step, the chain is given a random shove or momentum. The particle's acceleration is given by the negative change in potential. This is exactly what we observe when a ball rolls on a hill under the influence of gravity: it accelerates when it rolls down (i.e as the gravitational potential weakens), but looses speed when it goes up. We replicate this behavior by applying the laws of classical mechanics, elegantly described by Hamilton's equations, to our Markov chain.

Under these circumstances, the chain takes "bigger steps" when the posterior increases, but smaller ones when it moves towards regions of lower density. This results in several desirable properties, that allow the chain to explore the *typical set*, i.e the region with the most density mass, efficiently, especially as we get to high dimensions.

When applying Hamilton's equations to our MCMC problem, we can show the j^{th} component of the chain's momentum, p, varies as the negative partial derivative of the potential with respect to the position in the parameter space, q:

$$p'_j = -\frac{\partial U}{\partial q_j}$$

Recalling our definition of the potential (equation 3), the vectorized expression of the above equation is:

$$p' = -\nabla_q \log(\pi(q)) \tag{4}$$

To simulate Hamiltonian dynamics, we must therefore *compute the gradient of the log posterior with respect to the parameters*. Considering the posterior may have a complex expression, this is not a trivial task.

A.2. **Automatic Differentiation.**

To compute gradients, Stan uses *reverse automatic differentiation*. The posterior, π, gets "translated" into an expression graph and the gradients calculated by applying the chain rule at the nodes of the graph [4]. These nodes represent operators and functions used in π, such as the addition (+) operator or the exponential (`exp`) function.

Let $q \in \mathbb{R}^n$ be parameters and $x \in \mathbb{R}^k$ fixed data. For automatic differentiation to work, a function in Stan, $f : \mathbb{R}^{n+k} \mapsto \mathbb{R}^m$, must compute the output, $f(q, x)$, and the $n \times m$ Jacobian, J, where $J_{i,j} = \partial f_i / \partial q_j$.

This can be done in two ways: either we explicitly code a method to compute J, or we write f as an expression of other Stan functions and use automatic differentiation to calculate J. The latter method is by far the easiest one to code; but it may not always be implementable, nor be the most efficient approach.

At a C++ level, automatic differentiation requires the use of a new class, `var`, which contains (1) the value of the variable and (2) its adjoint with respect to the log posterior[6]. This introduces an important distinction between parameters, with respect to which we need to calculate the gradient of the log posterior and which must therefore be coded as `var`, and data, which are more simply coded as `double`.

It is good practice to allow users to call a function with only fixed data arguments. Most of the time, this means that arguments which may be parameters are templated, and can be passed as either `double` or `var` objects.

[6]The adjoint of x with respect to f is the derivative of f with respect to x.

Appendix B. Ordinary Differential Equations in Pharmacometrics

The work presented here is, in part, adapted from a presentation we gave at the 2017 Stan Conference on *Differential Equation Based Models in Stan* [10] and an informal article on *Complex ODE Based Models*[7] by Margossian.

Ordinary differential equations are a powerful tool to describe physical and biological processes. They persevere across disciplines and keep arising in a remarkable fashion. They are central to pharmacometrics and by extension to Torsten. While they can be given several physical interpretations, their mathematical properties are common to all problems, which allows us to develop general purpose tools.

B.1. **When do ordinary differential equations arise?**

We deal with an ordinary differential equation (ODE) when we want to determine a function $y(t)$ at a specific time but only know the derivative of that function, $\mathrm{d}y/\mathrm{d}t$. In other words, we know the rate at which a quantity of interest changes but not the quantity itself. In many scenarios, the rate depends on the quantity itself.

To get a basic intuition, consider the example of a gas container with a hole in it (figure 23). We can think of the gas as being made of molecules that move randomly in the container. Each molecule has a small chance of leaking through the hole. Thus the more molecules inside the container, the higher the number of escaping molecules per unit time. If there are a large number of molecules and the gas behaves like a continuous fluid, we observe that the more gas in the container, the higher the leakage. This statement can be written as the differential equation:

$$\frac{dy}{dt} = -ky(t)$$

where y is the amount of gas in the container and k is a positive constant. Note that $y' = \mathrm{d}y/\mathrm{d}t$ is an acceptable notation.

In a pharmacokinetic-pharmacodynamic (PK/PD) compartment model, we treat physiological components, such as organs, tissues, and circulating blood, as compartments between which the drug flows and/or in which the drug has an effect. A compartment may refer to more than one physiological component. For example, the central compartment typically consists of the systemic circulation (the blood) plus tissues and organs into which the drug diffuses rapidly.

Just like our leaking gas in a container, the rate at which the quantity of drug changes depends on the drug amount in the various compartments (in the limit where the drug behaves like a continuous substance). Things are slightly more complicated because instead of one box, we now deal with a network of containers. This results in a system of ordinary differential equations.

B.2. **An example: ODE system for the Two Compartment Model.**

Consider the common scenario in which a patient orally takes a drug. The drug enters the body through the gut and is then absorbed into the blood. From there it diffuses into and circulates back and forth

[7] see `https://github.com/stan-dev/stan/wiki/Complex-ODE-Based-Models`.

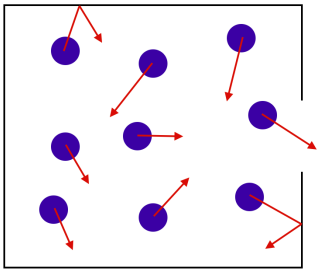

FIGURE 23. Gas container with a hole. *Gas molecules in a box move in a (approximatively) random fashion. In the limit where we have a lot of particles and the gas behaves like a continuous fluid, the rate at which the gas leaks is proportional to the amount of gas in the container. This physical process is described by the differential equation* $\mathrm{d}y/\mathrm{d}t = -ky(t)$*, where* y *is the amount of gas in the container.*

between various tissues and organs. Over time, the body clears the drug, i.e. the drug exits the body (for instance through urine) (figure 24).

Our model divides the patient's body into three compartments:

- **The absorption compartment**: the gut
- **The central compartment**: the systemic circulation (blood) and tissues/organs into which the drug diffuses rapidly
- **The peripheral compartment**: other tissues/organs into which the drug distributes more slowly

We conventionally call this a *Two Compartment Model*, which may seem odd since the model has three compartments. The idea is that the "absorption compartment" doesn't really count. We adopt this convention mostly to agree with the community.

We describe the drug absorption using the following differential equations:

$$\begin{aligned} y'_{\text{gut}} &= -k_a y_{\text{gut}} \\ y'_{\text{cent}} &= k_a y_{\text{gut}} - \left(\frac{CL}{V_{\text{cent}}} + \frac{Q}{V_{\text{cent}}}\right) y_{\text{cent}} + \frac{Q}{V_{\text{peri}}} y_{\text{peri}} \\ y'_{\text{peri}} &= \frac{Q}{V_{\text{cent}}} y_{\text{cent}} - \frac{Q}{V_{\text{peri}}} y_{\text{peri}} \end{aligned} \tag{5}$$

with
y_{gut} : the drug amount in the gut (mg)
y_{cent} : the drug amount in the central compartment (mg)
y_{peri} : the drug amount in the peripheral compartment (mg)
k_a : the rate constant at which the drug flows from the gut to the central compartment (h^{-1})
Q : the clearance at which the drug flows back and forth between the central and the peripheral compartment (L/h)

CL : the clearance at which the drug is cleared from the central compartment (L/h)
V_{cent} : the volume of the central compartment (L)
V_{peri} : the volume of the peripheral compartment (L)

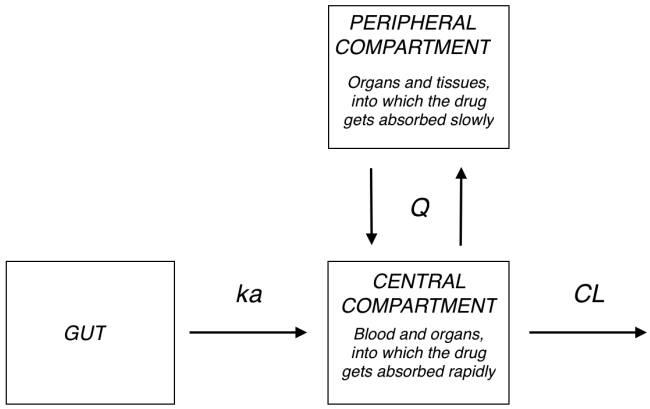

FIGURE 24. Two Compartment Model. *An orally administered drug enters the body through the gut and is then absorbed into the blood, organs, and tissues of the body, before being cleared out.*

B.3. **Overview of Tools for Solving Differential Equations.**

Solving ODEs can be notoriously hard.

In the best case scenario, an ODE system has an analytical solution we can hand-code (as we have done for the one and two compartment models). The vast majority of times, we need to approximate the solution numerically. There exists a very nice technique, involving matrix exponentials, for solving linear ODEs. Nonlinear systems are significantly more difficult but fortunately we can tackle these problems with numerical integrators.

Specialized algorithms for solving ODEs tend be more efficient but have a narrower application; the reverse holds for more general tools. We provide both, thereby allowing users to tackle a broad range of problems and optimize their model when possible (figure 25).

B.4. **The Event Schedule.**

B.4.1. *Handling exterior interventions.*

The ODE system only describes the natural evolution of the patient's system, that is how the drug behaves once it is already in the body. Describing this natural evolution, by solving ODEs, is the task of the *evolution operator*. This operator does not account for exterior interventions during the treatment, such as

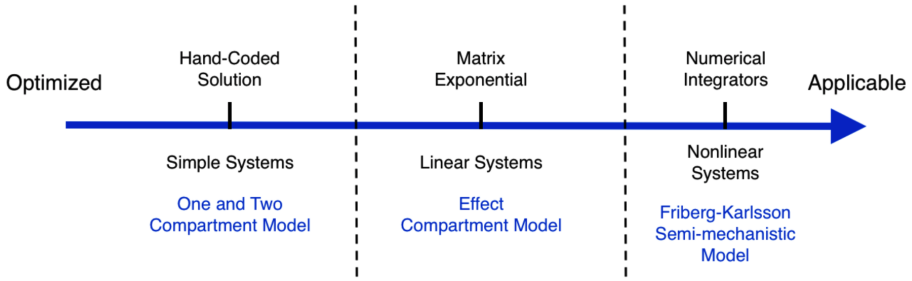

FIGURE 25. The "Optimized-Applicable Spectrum" of tools for solving ODEs. *The top line gives the technique to solve differential equations, the next line the type of ODE system this method should be applied to, and the third line (blue) an example from pharmacometrics, discussed in the user manual. Modelers should prefer the left-most applicable technique.*

the intake of a drug dose. To be accurate, our model must compute these exterior events and solve ODEs in the context of an *event schedule*.

We follow the convention set by NONMEM®[8], which is popular amongst pharmacometricians and which we find acceptable.

An event can either be a change in the state of the system or the measurement of a certain quantity. We distinguish two types of events:

(1) **State Changer**: an (exterior) intervention that alters the state of the system (for example, a bolus dosing, or the beginning of an infusion)
(2) **Observation**: the measurement of a quantity of interest at a certain time

Between two subsequent events, the ODEs fully describe the PK/PD system. Knowing the state y_0 at time t_0 fully defines the solution at finite times. Exploiting this property, Torsten calculates amounts in each compartment from one event to the other. The initial conditions of the ODEs are specified by the previous event and the states evolved from t_{previous} to t_{current}.

The user passes the event schedule using a data table.

NONMEM's convention allows one row to code for multiple events. For example, a single row can specify a patient receives multiple doses at a regular time interval. Consider:

```
TIME = 0, EVID = 1, CMT = 1, AMT = 1500, RATE = 0, ADDL = 4, II = 10, SS = 0
```

This row specifies that a time 0 (TIME = 0), a patient receives a 1500 mg (AMT = 1500) drug dose (EVID = 1) in the gut (CMT = 1), and will receive an additional dose every 10 hours (II = 10) until the patient has taken a total of 5 doses (ADDL = 4, being the number of additional doses, + 1, the original dose). Such an event really corresponds to 5 dosing events. Torsten augments the event schedule accordingly, before solving the ODEs recursively from one event to the other.

In summary, each Torsten function:

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

(1) augments the event schedule to include all state changers
(2) calculates the amounts in each compartment at each event of the augmented schedule by
    (a) integrating the ODEs and computing the *natural* evolution of the system
    (b) computing the effects of state changers
(3) returns the amounts at each event of the original schedule.

B.4.2. *Dosing Events.*

Most state changers in pharmacometrics (though by no means all of them) are due to drug intakes. A drug can be administered using a bolus intake or an infusion over a finite time.

To compute a bolus dose, Torsten simply adds the administered dose amount to the drug mass, $y(t)$, and then resumes integrating the ODEs. Letting $f(y,t) = y'(t)$ be the right-hand side of our ODE system, m be administered amount and τ the time of the dosing, we get:

$$
\begin{aligned}
y(\tau^+) &= y(\tau^-) + m \\
y(t) &= \int_{\tau^+}^{t} f(y,t')dt'
\end{aligned}
\tag{6}
$$

The bolus dosing event introduces a discontinuity with respect to time (figure 26), and can (in theory) lead to issues, especially when the modeler tries to evaluate *lag times*. We still need to characterize how important the resulting error is[9]. We test the computation of the Jacobian by benchmarking automatic differentiation against finite differentiation. As expected, the derivative is not properly evaluated at events which occur exactly at the time of a "dosing", lag time accounted for. Seeing the lag times are continuous variables, this scenario seems unlikely. The test further reveals the Jacobian at other events gets properly evaluated[10].

Users can code an infusion by setting a non-zero rate. For example, {`AMT = 1200, RATE = 150, ...`} corresponds to an infusion of 150 unit mass per unit time that will last 8 unit time (leading to a total of 1200 unit mass being administered). Torsten handles this scenario by creating a new event at the end of the infusion time. For all events between the beginning and end of the infusion, the rate in each compartment is augmented by the infusion rate, R. Letting τ be the beginning, and δ the end of the infusion time, we get:

$$
\begin{aligned}
y(\delta) &= \int_{\tau}^{\delta} f(y,t') + R dt' \\
y(t) &= \int_{\delta}^{t} f(y,t')dt'
\end{aligned}
\tag{7}
$$

Note the upper-bound of the integral, $\delta = m/R$, can be a latent parameter if either m or R is a parameter. In other words, we may need to compute $\partial y/\partial\delta$. This will be relevant to our discussion on numerical integrators.

[9] see issue #3 on `github.com/metrumresearchgroup/math/issues/3`.
[10] There may still be issues with numerical solvers. This has not yet been tested for.

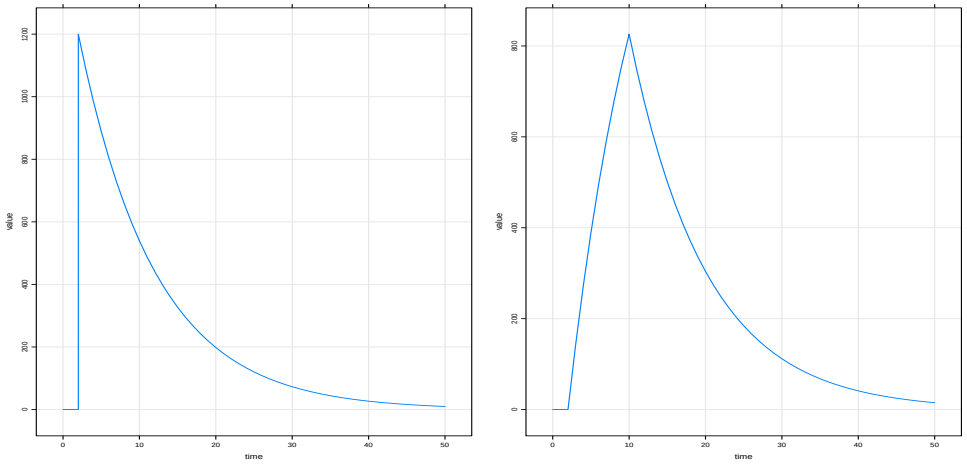

FIGURE 26. Simulated drug mass in a patient's gut after an oral drug administration ***(left)*** *At* $t = 2\text{h}$ *a patient receives a* 1200mg *bolus dose. This creates a discontinuity in the drug mass,* $y(t)$*. This can be an issue when* t_{lag} *is a parameter and thence* t *a latent parameter. As the system evolves, the drug gets naturally cleared out of the gut and absorbed into the blood and other organs (see equation 5).* ***(right)*** *A patient receives an infusion, starting at* $t = 2\text{h}$ *at a rate of* $R = 150\text{mg/h}$*. After 8 hours, the infusion stops and the drug mass decreases. Figures generated with mrgsolve.*

B.4.3. *Model Parameters.*

In addition to passing the event schedule to a torsten function, modelers must specify the *model parameters*:

- θ: parameters which appear in the ODE system
- F: the bioavailability fraction, specified for each compartment
- t_{lag}: lag times, specified for each compartment

All continuous variables, including elements of the event schedule, can be passed as either parameters or fixed data. This demand some caution when we compute sensitivities. Handling θ is straightforward. F and t_{lag} on the other hand modify variables which get passed to the evolution operator and create latent parameters. In particular, F modifies the rate, R, for regular solutions and the amount, m, for steady state solutions. t_{lag} alters the dosing time, t.

Model Parameter	Latent Parameter(s)
F	R (non-steady state case) m (steady state case)
t_{lag}	t

Appendix C. Solving Ordinary Differential Equations

We need to solve differential equations and compute the Jacobian of the solutions with respect to the parameters. In Torsten, this tasks is carried out by the *pred* functors, which act as the *evolution operators* (they compute the evolution of the system from one state to the other between two events).

C.1. **Analytical Solution.**

We hand-code the solution for the One and Two Compartment models with a first-order absorption. The Jacobians are computed using automatic differentiation.

C.2. **The Matrix Exponential.**

Consider a system of linear differential equations:

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

where K is a constant matrix and y a vector function. The solution to the equation is given by

$$y(t) = e^{tK} y_0$$

where y_0 is the initial condition and e is the *matrix exponential*, formally defined by the convergent power series:

$$e^{tK} = \sum_{n=0}^{\infty} \frac{(tK)^n}{n!} = I + tK + \frac{(tK)^2}{2} + \frac{(tK)^3}{3!} + ...$$

Looking at this definition, we clearly see the derivative of e^{tK} is Ke^{tK}.

There exist several methods to compute the matrix exponential. Moler & Van Loan provide an excellent and detailed overview [11], and Rowland & Weisstein a much briefer discussion [12].

The `matrix_exp()` function in Stan computes the matrix exponential. The exponential of a 2×2 matrix is calculated analytically [12], provided that for a matrix:

$$K = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

the condition $(a - d)^2 + 4bc > 0$ is verified.

For the general case, we use the Padé approximation coupled with scaling and squaring. The idea is to approximate the matrix exponential with the finite series:

$$R_{pq}(K) = [D_{pq}(K)]^{-1} N_{pq}(K)$$

where

$$\begin{aligned} N_{pq}(K) &= \sum_{j=0}^{p} \frac{(p+q-j)!p!}{(p+q)!j!(p-j)!} A^j \\ D_{pq}(K) &= \sum_{j=0}^{q} \frac{(p+q-j)!q!}{(p+q)!j!(q-j)!} (-A)^j \end{aligned}$$

In the asymptotic limit, $(p, q) \to (\infty, \infty)$, we recover the definition of the matrix exponential. Moler & Van Loan [11] demonstrate the Padé approximation is more efficient than the Taylor approximation[11], particularly in cases where $||K||$ is small. For reasons which are beyond the scope of this discussion, there are advantages to setting $p = q$ and computing the *diagonal* Padé approximation R_{pp}.

This method fails for large $||K||$ because of round-off errors. This is where the scaling and squaring comes into play. Noting the following property of the matrix exponential:

$$e^K = (e^{K/m})^m$$

we rescale the argument of the Padé approximation and exponentiate the final product by m.

The C++ library *Eigen* [13] contains an unsupported routine to compute the matrix exponential that uses the above-described method. The order of the approximation, i.e. the number of terms in the series, depends on the input matrix. We find Eigen's code promptly goes to the 13^{th} order, except in very simple cases. We edit Eigen's code, so that it may handle `var` types. The Jacobian of `matrix_exp` is approximated by applying automatic differentiation to the Padé approximation.

We can improve the current implementation of `matrix_exp` in several ways. First, we should hand-code the Jacobian, potentially using recommendations by Giles [14]. We suspect computing 13 terms in the Padé approximation is overkill; Sidje recommends 6, which is the default he implements in his package *Expokit* [15]. We also want to take advantage of methods which allow us to directly compute the action of a matrix exponential on a vector. Recall our goal is to compute the solution $e^{Kt} y_0$.

C.3. **Numerical Integrators.**

Stan provides two ODE integrators: a Runge-Kutta $4^{\text{th}}/5^{\text{th}}$ order, `integrate_ode_rk45`, which is relatively fast, and the slower but more robust backward differentiation method, `integrate_ode_bdf`, which can handle stiff systems of equations.

[11] The Taylor approximation computes the formal definition of the matrix exponential with a finite number of terms.

C.3.1. *Nonstiff and stiff systems.*

Stiffness is not a mathematically rigorously defined property but here's the general idea[12]. When we solve an ODE numerically, we construct the solution – the function y – one step at a time using a tangent approximation[13]. If the step is small, we gain in precision but it takes more steps to build the solution. We therefore need to find a balance between precision and computational speed.

In a stiff equation, the rate at which the solution changes with respect to the independent variable (time in many examples) can vary dramatically from one region to the other. This makes it difficult to pick an appropriate step size and requires a specialized algorithm, such as the backward differentiation technique used by `integrate_ode_bdf`.

C.3.2. *Jacobian of the solution.*

Given a vector of parameters, $\alpha \in \mathbb{R}^m$, and the solution function, $y \in \mathbb{R}^n$, we construct an $m \times n$ Jacobian. This is done by augmenting the original system with the state variables:

$$J_{n,m} = \frac{\partial}{\partial \alpha_m} y_n$$

which correspond to the individual elements of the Jacobian. Moreover, we calculate $\frac{\mathrm{d}}{\mathrm{d}t} J_{n,m}$ and add it to the right-hand-side of the ODE system. By integrating the augmented system, we simultaneously solve for the solution, y, and the Jacobian, J. For more details, see section 13 of *Carpenter et al.* [4].

An analysis of the augmented system shows the computational cost scales up significantly with the number of original states in the system; and (to a lesser degree) with the number of parameters [6]. In most cases, the modeler has no control over the dimension of y. We will treat a notable exception in the next section. Keeping the number of parameters at a minimum is a matter of careful coding.

In Stan, the functor which defines an ODE system, must observe a strict signature:

```
real[] foo (real t,        // time
            real[] y,      // array of states
            real[] theta, // array of parameters
            real[] x_r,   // array of real data
            int[] x_i)    // array of integer data
```

This strict signature requires some careful bookkeeping, as we need to sort out which variables are parameters or fixed data. As we saw in section B.4.3, it may not be immediately obvious if a variable is a latent parameter.

[12]The wikipedia article on *stiffness* is actually pretty good: `https://en.wikipedia.org/wiki/Stiff_equation`.

[13]This is, by the way, the reason why numerical integrators are so expensive. For the analytical or matrix exponential solutions, the function can be directly computed at the times of interest, without going through intermediate steps.

C.4. **Mixed Solvers.**

In certain cases, an ODE system can be split into two:

$$y_1(t)' = f_1(y_1, t)$$
$$y_2(t)' = f_2(y_1, y_2, t)$$

where y_1 does not depend on y_2. In this situation, we usually refer to y_1 as the *forcing function*. If y_1 has a closed-form solution, we can use a *mixed solver* to solve the system. The idea is to solve for y_1 analytically and then numerically integrate a *reduced system* to find y_2. This requires we write f_2 in terms of the solution y_1 – often a relatively complex expression – rather than as an expression of the (more simple) derivative y_1'.

The mixed solver thus presents a trade-off: it increases the complexity of the integrant but reduces the number of states we integrate numerically. The benefit turns out to overweight the cost, particularly with automatic differentiation. The gain in efficiency varies on a case-by-case basis. For the example of a Friberg-Karlsson model, the average speed-up is $49 \pm 14\%$ [6].

A proper implementation of the mixed solver requires a hand-coded closed-form solution for y_1 and some careful bookkeeping. In Torsten, we take advantage of the built-in solution for the one and two compartment model. The user is required to specify which PK forcing function he or she wishes to use and codes the reduced system. Torsten then follows the following coding scheme:

```
y_1 = f_1;
y_2 = integrate(f_2);
return y = {y_1, y_2};
```

where `f_1` is the analytical solution and `f_2` a C++ functor which wraps the reduced system the user passes to the Torsten function and observes the strict signature required by Stan's built-in integrators. For the case of a One Compartment model forcing function, the wrapper follows the following coding scheme:

```
real[] foo(real t,
           real[] y
           real[] theta,
           real[] x_r,
           int[] x_i) {
  // Get PK parameters
  thetaPK[0] = theta[0]; // CL
  thetaPK[1] = theta[1]; // VC
  thetaPK[2] = theta[2]; // ka

  // Get initial PK states
  // The last two components of theta should contain the initial PK states
  init_pk[0] = theta[theta.size() - 2];
  init_pk[1] = theta[theta.size() - 1];

  // The last element of x_r contains the initial time
  dt = t - x_r[x_r.size() - 1];
```

```
  y_pk = fOneCpt(dt, thetaPK, init_PK, x_r);
  dydt = f0(dt, y, y_pk, theta, x_r, x_i);

  return dydt;
}
```

where `f0` is the reduced system the user provides. The reduced system accepts an additional argument, which allows us to pass the analytical solution `y_pk`. Torsten takes care of the following bookkeeping tasks under the hood:

- augmenting the array of parameters, `theta`, with the initial conditions for the forcing function.
- augmenting the array of data, `x_r` with the initial time, so that the evolution operator for the forcing function, `fOneCpt`, knows between which times to evolve the forcing system.

These considerations extend to the case of a Two Compartment forcing function. We need to apply similar thinking when we deal with non-zero rates and steady state solutions.

C.5. **Rates.**

The administration of a dose through an infusion alters the *natural* ODE system, by adding a constant rate, R, to the derivative of the solution:

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

The solution to this new ODE is worked out analytically for the One and Two compartment model, and for the general linear compartment model. In particular for the latter:

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

which gives us the solution:

$$y(t) = e^{tK}(K^{-1}R + y_0) - K^{-1}R$$

For the numerical case, the function which gets passed to the integrator is modified, by adding R to the returned vector. If R is an array of fixed data, `rate` gets passed using the argument `x_r`.

If R is an array of parameters, we augment `theta`. Unfortunately, Torsten currently has no mechanism to separate data and parameters within R. Often times, only the rate in the dosing compartment is a parameter, while the others are fixed (and zero, for that matter). This means we pass more parameters to the ODE than necessary. This implementation is "safe" and does not place restrictions on the arguments the user passes to a Torsten function – but it creates inefficiencies.

C.6. **Steady State Solution.**

A steady state is reached when the dose input and the dose clearance cancel each other.

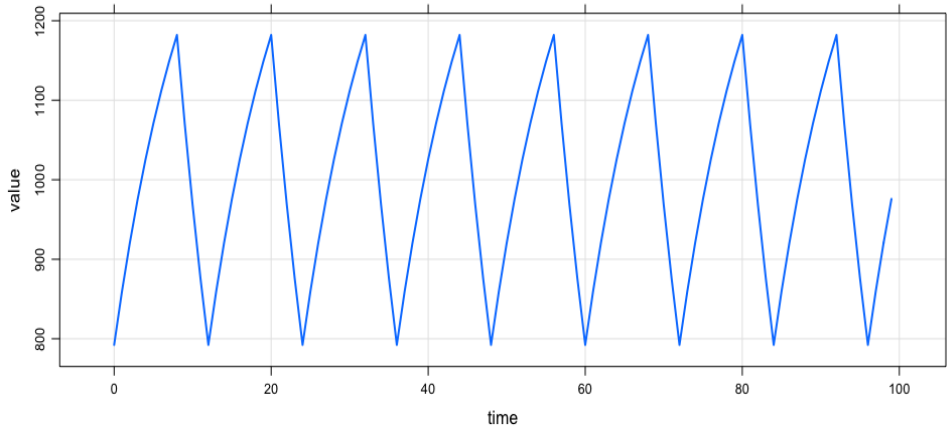

FIGURE 27. Simulated drug mass in a patient's gut at steady state. *Every 12 hours, a patient receives a drug administration. After being under this drug regiment for a certain period of time, the patient's system reaches a steady state. In this particular simulation, the drug administration corresponds to the infusion of 1200 mg over 8 hours. Figure generated with mrgsolve.*

The brute force method to calculate a steady state would be to simulate a dosing regimen for an extended period of time. While this works well in certain settings, it is not very efficient, and sometimes not precise. A better approach is to exploit the definition of the steady state. Let us suppose a treatment repeats itself every τ hours. In figure 27, τ corresponds to the inter-dose interval. Then, once steady state has been reached, we have:

$$y(t_0) = y(t_0 + \tau) \tag{8}$$

where t_0 corresponds to the beginning of a dosing interval. This expression corresponds to an algebraic equation. Our goal is to solve for $y(t_0)$, the drug amount once the patient has reached steady state (at the end of a dosing interval).

For the one and two compartment model, the solution can be worked out analytically. Similarly, in the linear case, we can write a solution using the matrix exponential. In the nonlinear case, we require a numerical algebraic solver.

C.6.1. *Algebraic Solver.*

This next section is adapted from a document Michael Betancourt produced when we were designing the algebraic solver[14].

Stan implements an algebraic solver, which uses Powell's hybrid method [16], and computes the Jacobian of the solution using the implicit function theorem. A prototype solver is available in Torsten 0.83 and a fully tested version will be available in Stan 2.17.

[14]For the original, see `http://discourse.mc-stan.org/t/implicit-function-theorem-math/264`

Solving any algebraic equation can be turned into a root-finding problem, that is we want to solve $f(x^*, \theta) = 0$ for $x^* \in \mathbb{R}^N$ in the neighborhood of some starting point $x \in \mathbb{R}^N$, which often corresponds to an initial guess for the solution. Note we made f explicitly dependent on the model parameters which appear in the algebraic equation, $\theta \in \mathbb{R}^K$. In addition, we need to calculate the sensitivities of θ. This means computing the $N \times K$ Jacobian:

$$J = \frac{\partial x^*}{\partial \theta}(x^*, \theta)$$

Formally, given the function $f : X \times \Theta \mapsto Y$, the implicit function theorem states that if f satisfies some weak regulatory conditions in a neighborhood of the starting point (x, θ) the roots are given by implicit functions, $x^* = h(y)$, where $h : \Theta \mapsto X$ satisfies $f(h(y), y) = 0$. This allows us to define the desired Jacobian as $\partial h(y)/\partial y$.

The implicit function theorem does not give us h – it only tells us that it exists. To do so however, it explicitly constructs the Jacobian of $h(y)$, which is all we need! Defining the two initial Jacobian matrices

$$J_x(x, \theta) = \frac{\partial f}{\partial x}(x, \theta)$$

and

$$J_\theta = \frac{\partial f}{\partial \theta}(x, \theta),$$

then the Jacobian of the roots is given by

$$\frac{\partial h}{\partial \theta}(\theta) = -[J_x(h(\theta), \theta)]^{-1} J_y(h(\theta), \theta),$$

assuming that J_x is invertible.

The Jacobians J_x and J_θ can be computed with automatic differentiation, given the expression for f is known, and Stan provides a method for matrix division.

The invertibility condition for J_x places severe restrictions on the structure of the original problem. In particular, recall $X = \mathbb{R}^N$ and $\Theta = \mathbb{R}^K$, and let $Y = \mathbb{R}^M$. Then J_x is a $M \times N$ matrix and is invertible if and only if $M = N$. In other words, we need as many unknowns as equations in our algebraic system. Even then, J_x can become singular if the roots are not uniquely defined and other pathologies occur.

Appendix D. Code Implementation of 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.

Stan interfaces 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 and mix 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.

[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, B., Margossian, C. and Pastoor, D. *mrgsolve: Simulate from ODE-Based Population PK/PD and Systems Pharmacology Models*. Metrum Research Group, http://mrgsolve.github.io/user_guide/ (2017). R package version 0.8.6.

[6] Margossian, C.C. The mixed solver: Gaining efficiency by combining analytical and numerical methods to solve ordinary differential equations and compute sensitivities. *Manuscript (not yet published)* (2017).

[7] Teorell, T. *Pharmacology and Pharmacokinetics*, chapter Concluding Remarks (Springer, 1974), pages 369 – 371.

[8] Neal, R.M. *MCMC using Hamiltonian Dynamics*. Handbook of Markov Chain Monte Carlo (Chapman & Hall / CRC Press, 2010).

[9] Betancourt, M. A conceptual introduction to hamiltonian monte carlo. *arXiv:1701.02434v1* (2017).

[10] Margossian, C.C. and Gillespie, W.R. Differential equations based models in stan. In *Stan Conference* (http://mc-stan.org/events/stancon2017-notebooks/stancon2017-margossian-gillespie-ode.html, 2017).

[11] Moler, C. and Van Loan, C. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM Review* (2003).

[12] Rowland, T. and Weisstein, E.W. Matrix exponential. MathWorld.
URL `http://mathworld.wolfram.com/MatrixExponential.html`

[13] Guennebaud, G., Jacob, B. *et al.* Eigen v3. `http://eigen.tuxfamily.org` (2010).

[14] Giles, M. An extended collection of matrix derivative results for forward and reverse mode algorithmic differentiation. *The Mathematic Institute, University of Oxford, Eprints Archive* (2008).

[15] Sidje, R.B. Expokit: a software package for computing matrix exponentials. *ACM Transactions on Mathematical Software (TOMS)* **24** (1998):130 – 156.

[16] Powell, M.J.D. A hybrid method for nonlinear equations. In P. Rabinowitz, (ed.) *Numerical Methods for Nonlinear Algebraic Equations* (Gordon and Breach, 1970).