Metrum Research Group LLC
Phone: 860.735.7043
billg@metrumrg.com, yiz@metrumrg.com

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

Torsten

Torsten: A Pharmacokinetic/Pharmacodynamic Model Library for Stan

User's Guide
(Torsten Version 0.88rc, Stan version 2.25.0)

December 18, 2020

Contents

Development team

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

Yi Zhang. yiz@metrumrg.com, Metrum Research Group, LLC

Charles Margossian. charles.margossian@columbia.edu, Columbia University, Department of Statistics(formerly 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.

Introduction

Stan is an open source probabilistic programing language designed primarily to do Bayesian data analysis [3].

Several of its features make it a powerful tool to specify and fit complex models. First, its language is very expressive and flexible. Secondly, it implements a variant of No U-Turn Sampler(NUTS), an adaptative Hamiltonian Monte Carlo algorithm that was proven more efficient than commonly used Monte Carlo Markov Chains (MCMC) samplers for complex high dimensional problems [7, 2]. Our goal is to harness these innovations and make Stan a better software for pharmacometrics modeling. Our efforts are twofold:

(1) We contribute to the development of features, 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? Comment at Torsten repository

https://github.com/metrumresearchgroup/Torsten

or shoot us an e-mail(billg@metrumrg.com, yz@yizh.org)and we will help you help us!

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.

1. Overview

Torsten is a collection of Stan functions to facilitate analysis of pharmacometric data using Stan. 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 PK forcing function described by a linear one or two compartment model.

The models and data format are based on NONMEM®[1]/NMTRAN/PREDPP conventions including:

- Recursive calculation of model predictions
  - This permits piecewise constant covariate values
- Bolus or constant rate inputs into any compartment
- Single dose and multiple dose events
- Handles steady state dosing events

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

- Implemented NMTRAN data items include: TIME, EVID, CMT, AMT, RATE, ADDL, II, SS

In general, all real variables may be passed as model 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:

- 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.

2. Implementation summary

- Current Torsten v0.88 is based on Stan v2.25.0.
- All functions are programmed in C++ and are compatible with the Stan math automatic differentiation library [4]
- One and two compartment models are based on analytical solutions of governing ODEs.
- General linear compartment models are based on semi-analytical solutions using the built-in matrix exponential function
- General compartment models are solved numerically 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.
- A mix compartment model's PK forcing function is solved analytically, and its forced ODE system is solved numerically.

3. 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)
- Optimize Matrix exponential functions
  - Function for the action of Matrix Exponential on a vector
  - Hand-coded gradients
  - Special algorithm for matrices with special properties
- Develop new method for large-scale hierarchical models with costly ODE solving.
- Fix issue that arises when computing the adjoint of the lag time parameter (in a dosing compartment) evaluated at $t_{\text{lag}} = 0$.
- Extend formal tests
  - More unit tests and better CD/CI support.
  - Comparison with simulations from the R package `mrgsolve` and the software NONMEM®
  - Recruit non-developer users to conduct beta testing

Changelog

Version 0.88 *<2020-12-18 Fri>*.

- Added
  - Bioavailability, lag time, ODE real & integer data are optional in PMX function signatures.
  - Support all EVID options from NM-TRAN and mrgsolve.
  - Support steady-state infusion through multiple interdose intervals.
- Changed
  - More efficient memory management of COVDES implenmentation.
  - Update of MPI framework to adapt multilevel paralleism.
  - Update to Stan version 2.25.0.
  - Use cmdstanr as R interface.
  - Stop supporting rstan as R interface.

Version 0.87 *<2019-07-26 Fri>*.

- Added
  - MPI dynamic load balance for Torsten's population ODE integrators
    * `pmx_integrate_ode_group_adams`
    * `pmx_integrate_ode_group_bdf`
    * `pmx_integrate_ode_group_rk45`

    To invoke dynamic load balance instead of default static balance for MPI, issue `TORSTEN_MPI=2` in `make/local`.
  - Support `RATE` as parameter in `pmx_solve_rk45/bdf/adams` functions.
- Changed
  - Some fixes on steady-state solvers
  - Update to rstan version 2.19.2.

Version 0.86 *<2019-05-15 Wed>*.

- Added
  - Torsten's ODE integrator functions
    * `pmx_integrate_ode_adams`
    * `pmx_integrate_ode_bdf`
    * `pmx_integrate_ode_rk45`

    and their counterparts to solve a population/group of subjects governed by an ODE
    * `pmx_integrate_ode_group_adams`
    * `pmx_integrate_ode_group_bdf`
    * `pmx_integrate_ode_group_rk45`
  - Torsten's population PMX solver functions for general ODE models
    * `pmx_solve_group_adams`
    * `pmx_solve_group_bdf`
    * `pmx_solve_group_rk45`
  - Support time step `ts` as parameter in `pmx_integrate_ode_xxx` solvers.
- Changed
  - Renaming Torsten functions in previous releases, the old-new name mapping is
    * `PKModelOneCpt` → `pmx_solve_onecpt`
    * `PKModelTwoCpt` → `pmx_solve_onecpt`
    * `linOdeModel` → `pmx_solve_linode`
    * `generalOdeModel_adams` → `pmx_solve_adams`
    * `generalOdeModel_bdf` → `pmx_solve_bdf`
    * `generalOdeModel_rk45` → `pmx_solve_rk45`
    * `mixOde1CptModel_bdf` → `pmx_solve_onecpt_bdf`
    * `mixOde1CptModel_rk45` → `pmx_solve_onecpt_rk45`
    * `mixOde2CptModel_bdf` → `pmx_solve_twocpt_bdf`
    * `mixOde2CptModel_rk45` → `pmx_solve_twocpt_rk45`

    Note that the new version of the above functions return the *transpose* of the matrix returned by the old versions, in order to improve memory efficiency. The old version are retained but will be deprecated in the future.
  - Update to Stan version 2.19.1.

Version 0.85 *<2018-12-04 Tue>*.

- Added
  - Dosing rate as parameter
- Changed
  - Update to Stan version 2.18.0.

Version 0.84 *<2018-02-24 Sat>*.

- Added
  - Piecewise linear interpolation function.
  - Univariate integral functions.

- Changed
  - Update to Stan version 2.17.1.
  - Minor revisions to User Manual.
  - Bugfixes.

Version 0.83 *<2017-08-02 Wed>*.

- Added
  - Work with TorstenHeaders
  - Each chain has a different initial estimate
- Changed
  - User manual
  - Fix misspecification in ODE system for TwoCpt example.
  - Other bugfixes

Version 0.82 *<2017-01-29 Sun>*.

- Added
  - Allow parameter arguments to be passed as 1D or 2D arrays
  - More unit tests
  - Unit tests check automatic differentiation against finite differentiation.
- Changed
  - Split the parameter argument into three arguments: pMatrix (parameters for the ODEs – note: for `linOdeModel`, pMatrix is replaced by the constant rate matrix K), biovar (parameters for the biovariability), and tlag (parameters for the lag time).
  - bugfixes

Version 0.81 *<2016-09-27 Tue>*.

- Added linCptModel (linear compartmental model) function

Version 0.80a *<2016-09-21 Wed>*.

- Added check_finite statements in pred_1 and pred_2 to reject metropolis proposal if initial conditions are not finite

Installation

Currently Torsten is based on a forked version of Stan. The latest v0.88 is compatible with Stan v2.25.0. Torsten can be accessed from command line for cmdstan interface and =cmdstanr=(https://mc-stan.org/cmdstanr/) for R interface.

1. Command line interface

After downloading the project

- https://github.com/metrumresearchgroup/Torsten

The command line interface `cmdstan` is available to use without installation. The following command builds a Torsten model `model_name` in `model_path`

```
cd Torsten/cmdstan; make model_path/model_name
```

2. R interface

After installing cmdstanr from https://mc-stan.org/cmdstanr/, use the following command to set path

```
cmdstanr::set_cmdstan_path("Torsten/cmdstan")
```

Then one can follow
https://mc-stan.org/cmdstanr/articles/cmdstanr.html
to compile and run Torsten models.

3. MPI support

Torsten's MPI support is of a different flavour than `reduce_sum` found in Stan. To be able to utilize MPI parallelisation, one first needs to ensure an MPI library such as

- https://www.mpich.org/downloads/
- https://www.open-mpi.org/software/ompi/

is available. Torsen's implementation is tested on both `MPICH` and `OpenMPI`.

To use MPI-supported population/group solvers, add/edit `make/local`

```
TORSTEN_MPI=1

# path to MPI headers
CXXFLAGS += -isystem /usr/local/include
# if you are using Metrum's metworx platform, add MPICH3's
# headers with
# CXXFLAGS += -isystem /usr/local/mpich3/include
```

Note that currently `TORSTEN_MPI` and `STAN_MPI` flags conflict on processes management and cannot be used in a same Stan model, and MPI support is only available through `cmdstan` interface.

4. Testing

Models in `example-models` directory are for tutorial and demonstration. The following shows how to build and run the two-compartment model using `cmdstanr`, and use `bayesplot` to examine posterior density of `CL`.

```
library("cmdstanr")
set_cmdstan_path("Torsten/cmdstan")
file.dir <- file.path("Torsten", "example-models", "pk2cpt")
file  <- file.path(file.dir, "pk2cpt.stan")
model <- cmdstan_model(file)
fit <- model$sample(data = file.path(file.dir, "pk2cpt.data.R"),
                    init = file.path(file.dir, "pk2cpt.init.R"),
                    seed = 123,
                    chains = 4,
                    parallel_chains = 2,
                    refresh = 500)
bayesplot::mcmc_dens_overlay(fit$draws("CL"))
```

3

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. The code for the examples can be found at

- https://github.com/metrumresearchgroup/example-models

and also at the `example-models` folder.

Torsten's functions are prefixed with `pmx_`. For some of their arguments we adopt NM-TRAN format for events specification(Table 3.1). All the **`real[]`** arguments above are allowed to be `parameters` in a Stan

Argument Name	Definition	Stan data type
`time`	Time	**`real[]`**
`amt`	Amount	**`real[]`**
`rate`	Infusion rate	**`real[]`**
`ii`	interdose interval	**`real[]`**
`evid`	Event ID	**`int[]`**
`cmt`	Event compartment	**`int[]`**
`addl`	Additionial identical doses	**`int[]`**
`ss`	steady-state dosing flag	**`int[]`**

TABLE 3.1. NM-TRAN compatible event specification arguments. All arrays should have the same length corresponding to the number of events.

model. In addtion, Torsten functions support optional arguments and overloaded signatures. Optional arguments are indicated by surrounding square bracket `[]`. We point out three commonly used PMX model arguments that support overloading. In the rest of this document we assume the convention above unless indicated otherwise.

Argument Name	Definition	Stan data type	Optional
`theta`	Model parameters	**`real[]`** or **`real[ , ]`**	N
`biovar`	bioavailability fraction	**`real[]`** or **`real[ , ]`**	Y (default to 1.0)
`tlag`	Lag time	**`real[]`** or **`real[ , ]`**	Y (default to 0.0)

TABLE 3.2. PMX model parameter overloadings. One can use 1d array **`real[]`** to indicate constants of all events, or 2d array **`real[ , ]`** so that the ith row of the array describes the model arguments for time interval (t_{i-1}, t_i), and the number of the rows equals to the size of **`time`**.

1. One Compartment Model

1.1. Description. Function **`pmx_solve_onecpt`** solves a one-compartment PK model (Figure 3.1). The model obtains plasma concentrations of parent drug $c = y_2/V_2$ by solving for the mass of drug in the

central compartment y_2 from ordinary differential equations(ODEs)

$$y_1' = -k_a y_1, \tag{3.1a}$$

$$y_2' = k_a y_1 - \left(\frac{CL}{V_2} + \frac{Q}{V_2}\right) y_2. \tag{3.1b}$$

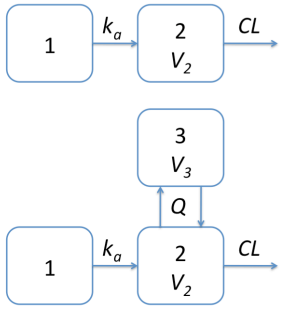

FIGURE 3.1. One and two compartment models with first order absorption implemented in Torsten.

1.2. Usage.

```
matrix = pmx_solve_onecpt(time, amt, rate, ii, evid, cmt, addl, ss, theta [, biovar, tlag ] )
```

1.3. Arguments. See Table 3.1 and Table 3.2.

1.4. Return value. An `ncmt`-by-`nt` matrix, where `nt` is the number of time steps and `ncmt=2` is the number of compartments.

1.5. Note.

- ODE Parameters **`theta`** should consist of CL, V_2, k_a, in that order.
- **`biovar`** and **`tlag`** are optional, so that the following are allowed:

```
pmx_solve_onecpt(..., theta);
pmx_solve_onecpt(..., theta, biovar);
pmx_solve_onecpt(..., theta, biovar, tlag);
```

- Setting $k_a = 0$ eliminates the first-order absorption.

2. Two Compartment Model

2.1. Description. Function **`pmx_solve_twocpt`** solves a two-compartment PK model (Figure 3.1). The model obtains plasma concentrations of parent drug $c = y_2/V_2$ by solving for the mass of drug in the central compartment y_2 from ordinary differential equations(ODEs)

$$y_1' = -k_a y_1 \tag{3.2a}$$

$$y_2' = k_a y_1 - \left(\frac{CL}{V_2} + \frac{Q}{V_2}\right) y_2 + \frac{Q}{V_3} y_3 \tag{3.2b}$$

$$y_3' = \frac{Q}{V_2} y_2 - \frac{Q}{V_3} y_3 \tag{3.2c}$$

2.2. Usage.

```
matrix = pmx_solve_twocpt(time, amt, rate, ii, evid, cmt, addl, ss, theta [, biovar, tlag ] )
```

2.3. Arguments. See Table 3.1 and Table 3.2.

2.4. Return value. An `ncmt`-by-`nt` matrix, where `nt` is the number of time steps and `ncmt=3` is the number of compartments.

2.5. Note.

- ODE Parameters **`theta`** consists of CL, Q, V_2, V_3, k_a.
- **`biovar`** and **`tlag`** are optional, so that the following are allowed:

```
pmx_solve_twocpt(..., theta);
pmx_solve_twocpt(..., theta, biovar);
pmx_solve_twocpt(..., theta, biovar, tlag);
```

- Setting $k_a = 0$ eliminates the first-order absorption.

3. General Linear ODE Model Function

3.1. Description. Function **`pmx_solve_linode`** solves a (piecewise) linear ODEs model with coefficients in form of matrix K

$$y'(t) = Ky(t) \tag{3.3}$$

For example, in a two-compartment model with first order absorption, K is

$$K = \begin{bmatrix} -k_a & 0 & 0 \\ k_a & -(k_{10} + k_{12}) & k_{21} \\ 0 & k_{12} & -k_{21} \end{bmatrix} \tag{3.4}$$

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

3.2. Usage.

```
matrix = pmx_solve_linode(time, amt, rate, ii, evid, cmt, addl, ss, K, biovar, tlag )
```

3.3. Arguments.

- **`K`** System parameters. **`K`** can be either
  - a **`matrix`** for constant parameters in all events, or
  - an array of matrices **`matrix`** `K[nt]` so that the ith entry of the array describes the model parameters for time interval (t_{i-1}, t_i), and the number of the rows equals to the number of event time **`nt`**.
- See Table 3.1 and Table 3.2 for the rest of arguments.

3.4. Return value. An `n`-by-`nt` matrix, where `nt` is the number of time steps and `n` is the number of rows(columns) of square matrix **`K`**.

4. General ODE Model Function

4.1. Description. Function **`pmx_solve_adams`**, **`pmx_solve_bdf`**, and **`pmx_solve_rk45`** solve a first-order ODE system specified by user-specified right-hand-side function **`ODE_rhs`** f

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

In the case where the **`rate`** vector r is non-zero, this equation becomes:

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

4.2. Usage.

```
matrix pmx_solve_[adams || rk45 || bdf](ODE_rhs, int nCmt, time, amt, rate, ii, evid, cmt,
 ↪  addl, ss, theta, [ biovar, tlag, real[,] x_r, int [,] x_i, real rel_tol, real abs_tol,
 ↪  int max_step, real as_rel_tol, real as_abs_tol, int as_max_step ] );
```

Here `[adams || rk45 || bdf]` indicates the function name can use any of the three suffixes. See below.

4.3. Arguments.

- **`ODE_rhs`** ODE right-hand-side f. It should be defined in `functions` block and has the following format

  ```
  real[] = f(real t, real[] y, real[] param, real[] dat_r, int[] dat_i) {...}
  ```

  Here **`t`** is time, **`y`** the unknowns of ODE, **`param`** the parameters, **`dat_r`** the real data, **`dat_i`** the integer data. **`param`**, **`dat_r`**, and **`dat_i`** are from the entry of **`theta`**, **`x_r`**, and **`x_i`** corresponding to **`t`**, respectively. f should not include dosing rates in its definition, as Torsten automatically update f when the corresponding event indicates infusion dosage.
- **`nCmt`** The number of compartments. Equivalently, the dimension of the ODE system.
- **`x_r`** 2d arary real data to be passed to ODE RHS. If specified, its 1st dimension should have the same size as **`time`**.
- **`x_i`** 2d arary integer data to be passed to ODE RHS. If specified, its 1st dimension should have the same size as **`time`**.
- **`rel_tol`** The relative tolerance for numerical integration, default to 1.0E-6.
- **`abs_tol`** The absolute tolerance for numerical integration, default to 1.0E-6.
- **`max_step`** The maximum number of steps in numerical integration, default to 10^6.
- **`as_rel_tol`** The relative tolerance for algebra solver for steady state solution, default to 1.0E-6.
- **`as_abs_tol`** The absolute tolerance for algebra solver for steady state solution, default to 1.0E-6.
- **`as_max_step`** The maximum number of interations in algebra solver for steady state solution, default to 10^2.
- See Table 3.1 and Table 3.2 for the rest of arguments.

4.4. Return value.

An `nCmt`-by-`nt` matrix, where `nt` is the size of **`time`**.

4.5. Note.

- See section 7.5 for different types of integrator and general guidance.
- See section 7.5 for comments on accuracy and tolerance.
- The default values of **`atol`**, **`rtol`**, and **`max_step`** are based on a limited amount of PKPD test problems and should not be considered as universally applicable. We strongly recommend user to set these values according to physical intuition and numerical tests. See also Secion 7.5.
- With optional arguments indicated by square bracket, the following calls are allowed:

```
pmx_solve_[adams || rk45 || bdf](..., theta);
pmx_solve_[adams || rk45 || bdf](..., theta, rel_tol, abs_tol, max_step);
pmx_solve_[adams || rk45 || bdf](..., theta, rel_tol, abs_tol, max_step, as_rel_tol,
 ↪  as_abs_tol, as_max_step);
pmx_solve_[adams || rk45 || bdf](..., theta, biovar);
pmx_solve_[adams || rk45 || bdf](..., theta, biovar, rel_tol, abs_tol, max_step);
pmx_solve_[adams || rk45 || bdf](..., theta, biovar, rel_tol, abs_tol, max_step, as_rel_tol,
 ↪  as_abs_tol, as_max_step);
pmx_solve_[adams || rk45 || bdf](..., theta, biovar, tlag);
pmx_solve_[adams || rk45 || bdf](..., theta, biovar, tlag, rel_tol, abs_tol, max_step);
pmx_solve_[adams || rk45 || bdf](..., theta, biovar, tlag, rel_tol, abs_tol, max_step,
 ↪  as_rel_tol, as_abs_tol, as_max_step);
pmx_solve_[adams || rk45 || bdf](..., theta, biovar, tlag, x_r);
```

```
pmx_solve_[adams || rk45 || bdf](..., theta, biovar, tlag, x_r, rel_tol, abs_tol, max_step);
pmx_solve_[adams || rk45 || bdf](..., theta, biovar, tlag, x_r, rel_tol, abs_tol, max_step,
 ↪  as_rel_tol, as_abs_tol, as_max_step);
pmx_solve_[adams || rk45 || bdf](..., theta, biovar, tlag, x_r, x_i);
pmx_solve_[adams || rk45 || bdf](..., theta, biovar, tlag, x_r, x_i, rel_tol, abs_tol,
 ↪  max_step);
pmx_solve_[adams || rk45 || bdf](..., theta, biovar, tlag, x_r, x_i, rel_tol, abs_tol,
 ↪  max_step, as_rel_tol, as_abs_tol, as_max_step);
```

5. Coupled ODE Model Function

5.1. Description. When the ODE system consists of two subsystems in form of

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

with y_1, y_2, f_1, and f_2 being vector-valued functions, and y_1' independent of y_2, the solution can be accelerated if y_1 admits an analytical solution which can be introduced into the ODE for y_2 for numerical integration. This structure arises in PK/PD models, where y_1 describes a forcing PK function and y_2 the PD effects. In the example of a Friberg-Karlsson semi-mechanistic model(see below), we observe an average speedup of $\sim 47 \pm 18\%$ when using the mix solver in lieu of the numerical integrator. In the context, currently the couple solver supports one- & two-compartment for PK model, and **rk45** & **bdf** integrator for nonlinear PD model.

5.2. Usage.

```
matrix pmx_solve_onecpt_[ rk45 || bdf ](reduced_ODE_system, int nOde, time, amt, rate, ii,
 ↪  evid, cmt, addl, ss, theta, biovar, tlag [, real rel_tol, real abs_tol, int max_step,
 ↪  real as_rel_tol, real as_abs_tol, int as_max_step ] );
matrix pmx_solve_twocpt_[ rk45 || bdf ](reduced_ODE_system, int nOde, time, amt, rate, ii,
 ↪  evid, cmt, addl, ss, theta, biovar, tlag [, real rel_tol, real abs_tol, int max_step,
 ↪  real as_rel_tol, real as_abs_tol, int as_max_step ] );
```

5.3. Arguments.

- **reduced_ODE_rhs** The system 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.

```
real[] reduced_ODE_rhs(real t, real[] y2, real[] y1, real[] theta, real[] x_r,
 ↪  int[] x_i)
```

- **nCmt** The number of compartments. Equivalently, the dimension of the ODE system.
- **rel_tol** The relative tolerance for numerical integration, default to 1.0E-6.
- **abs_tol** The absolute tolerance for numerical integration, default to 1.0E-6.
- **max_step** The maximum number of steps in numerical integration, default to 10^6.
- See Table 3.1 and Table 3.2 for the rest of arguments.

5.4. Return value. An `nPk + nOde`-by-`nt` matrix, where `nt` is the size of **time**, and `nPk` equals to 2 in **pmx_solve_onecpt_** functions and 3 in **pmx_solve_twocpt_** functions.

6. General ODE-based Population Model Function

6.1. Description. All the preivous functions solves for a single sunject. Torsten also provides population modeling counterparts for ODE solutions. The functions solve for a population that share an ODE model but with subject-level parameters and event specifications and have similar signatures to single-subject functions, except that now events arguments **time**, **amt**, **rate**, **ii**, **evid**, **cmt**, **addl**, **ss** specifies the entire population, one subject after another.

6.2. Usage.

```
matrix pmx_solve_group_[adams || rk45 || bdf](ODE_rhs, int nCmt, int[] len, time, amt, rate,
 ↪  ii, evid, cmt, addl, ss, theta, [ biovar, tlag, real[,] x_r, int [,] x_i, real rel_tol,
 ↪  real abs_tol, int max_step, real as_rel_tol, real as_abs_tol, int as_max_step ] );
```

Here `[adams || rk45 || bdf]` indicates the function name can be of any of the three suffixes. See section 7.5.

6.3. Arguments.

- **time**, **amt**, **rate**, **ii**, **evid**, **cmt**, **addl**, **ss** 2d-array arguments that describe data record for the entire population (see also Table 3.1 and Table 3.2). They must have same size in the first dimension. Take **evid** for example. Let N be the population size, then **evid[1** to **evid[n1** specifies events ID for subject 1, **evid[n1 + 1** to **evid[n1 + n2** for subject 2, etc. With n_i being the number of events for subject i, $i = 1, 2, \ldots, N$, the size of **evid**'s first dimension is $\sum_i n_i$.
- **len** The length of data for each subject within the above events arrays. The size of **len** equals to population size N.
- **ODE_rhs** ODE right-hand-side f. It should be defined in `functions` block and has the following format

  ```
  real[] = f(real t, real[] y, real[] param, real[] dat_r, int[] dat_i) {...}
  ```

  Here **t** is time, **y** the unknowns of ODE, **param** the parameters, **dat_r** the real data, **dat_i** the integer data. **param**, **dat_r**, and **dat_i** are from the entry of **theta**, **x_r**, and **x_i** corresponding to **t**, respectively. f should not include dosing rates in its definition, as Torsten automatically update f when the corresponding event indicates infusion dosage.
- **nCmt** The number of compartments. Equivalently, the dimension of the ODE system.
- **x_r** 2d arary real data to be passed to ODE RHS. If specified, its 1st dimension should have the same size as **time**.
- **x_i** 2d arary integer data to be passed to ODE RHS. If specified, its 1st dimension should have the same size as **time**.
- **rel_tol** The relative tolerance for numerical integration, default to 1.0E-6.
- **abs_tol** The absolute tolerance for numerical integration, default to 1.0E-6.
- **max_step** The maximum number of steps in numerical integration, default to 10^6.
- **as_rel_tol** The relative tolerance for algebra solver for steady state solution, default to 1.0E-6.
- **as_abs_tol** The absolute tolerance for algebra solver for steady state solution, default to 1.0E-6.
- **as_max_step** The maximum number of interations in algebra solver for steady state solution, default to 10^2.

6.4. Return value.

An `nCmt`-by-`nt` matrix, where `nt` is the total size of events $\sum_i n_i$.

6.5. Note.

- Similar to single-subject solvers, three numerical integrator are provided:
  - **pmx_solve_group_adams**: Adams-Moulton method,
  - **pmx_solve_group_bdf**: Backward-differentiation formular,
  - **pmx_solve_group_rk45**: Runge-Kutta 4/5 method.
- With optional arguments indicated by square bracket, the following calls are allowed:

```
pmx_solve_group_[adams || rk45 || bdf](..., theta);
pmx_solve_group_[adams || rk45 || bdf](..., theta, rel_tol, abs_tol, max_step);
pmx_solve_group_[adams || rk45 || bdf](..., theta, rel_tol, abs_tol, max_step, as_rel_tol,
 ↪  as_abs_tol, as_max_step);
pmx_solve_group_[adams || rk45 || bdf](..., theta, biovar);
pmx_solve_group_[adams || rk45 || bdf](..., theta, biovar, rel_tol, abs_tol, max_step);
```

```
pmx_solve_group_[adams || rk45 || bdf](..., theta, biovar, rel_tol, abs_tol, max_step,
 ↪  as_rel_tol, as_abs_tol, as_max_step);
pmx_solve_group_[adams || rk45 || bdf](..., theta, biovar, tlag);
pmx_solve_group_[adams || rk45 || bdf](..., theta, biovar, tlag, rel_tol, abs_tol, max_step);
pmx_solve_group_[adams || rk45 || bdf](..., theta, biovar, tlag, rel_tol, abs_tol, max_step,
 ↪  as_rel_tol, as_abs_tol, as_max_step);
pmx_solve_group_[adams || rk45 || bdf](..., theta, biovar, tlag, x_r);
pmx_solve_group_[adams || rk45 || bdf](..., theta, biovar, tlag, x_r, rel_tol, abs_tol,
 ↪  max_step);
pmx_solve_group_[adams || rk45 || bdf](..., theta, biovar, tlag, x_r, rel_tol, abs_tol,
 ↪  max_step, as_rel_tol, as_abs_tol, as_max_step);
pmx_solve_group_[adams || rk45 || bdf](..., theta, biovar, tlag, x_r, x_i);
pmx_solve_group_[adams || rk45 || bdf](..., theta, biovar, tlag, x_r, x_i, rel_tol, abs_tol,
 ↪  max_step);
pmx_solve_group_[adams || rk45 || bdf](..., theta, biovar, tlag, x_r, x_i, rel_tol, abs_tol,
 ↪  max_step, as_rel_tol, as_abs_tol, as_max_step);
```

- The group solvers support paralleisation through Message Passing Interface(MPI). One can access this feature through `cmdstan` or `cmdstanr` interface.

```
# cmdstan interface user need to add "TORSTEN_MPI=1" and
# "TBB_CXX_TYPE=gcc" in "cmdstan/make/local" file. In linux & macos
# this can be done as
echo "TORSTEN_MPI=1" > cmdstan/make/local
echo "TBB_CXX_TYPE=gcc" > cmdstan/make/local # "gcc" should be replaced by user's C compiler
make path-to-model/model_name
mpiexec -n number_of_processes model_name sample... # additional cmdstan options
```

```
library("cmdstanr")
cmdstan_make_local(cpp_options = list("TORSTEN_MPI" = "1", "TBB_CXX_TYPE"="gcc"))  # "gcc"
 ↪  should be replaced by user's C compiler
rebuild_cmdstan()
mod <- cmdstan_model(path-to-model-file, quiet=FALSE, force_recompile=TRUE)
f <- mod$sample_mpi(data = ..., chains = 1, mpi_args = list("n" = number_of_processes),
 ↪  refresh = 200)
```

Here n denotes number of MPI processes, so that N ODE systems (each specified by a same RHS function and subject-dependent events) are distributed to and solved by n processes evenly. Note that to access this feature user must have MPI installed, and some MPI installation may require set additional compiler arguments, such as `CXXLFAGS` and `LDFLAGS`.

7. ODE integrator Function

7.1. Description. Torsten provides its own implementation of ODE solvers that solves

$$y'(t) = f(t, y(t)), \quad y(t_0) = y_0$$

for y. These solvers are customized for Torsten applications and different from those found in Stan. The general ODE PMX solvers in previous sections are internally powered by these functions.

7.2. Usage.

```
real[ , ] pmx_integrate_ode_[ adams || bdf || rk45 ](ODE_rhs, real[] y0, real t0, real[] ts,
 ↪  real[] theta, real[] x_r, int[] x_i [ , real rtol, real atol, int max_step ]);
```

7.3. Arguments.

- **`ODE_rhs`** Function that specifies the right-hand-side f. It should be defined in `functions` block and has the following format

```
real[] = f(real t, real[] y, real[] param, real[] dat_r, int[] dat_i) {...}
```

Here **t** is time, **y** the unknowns of ODE, **param** the parameters, **dat_r** the real data, **dat_i** the integer data.

- **y0** Initial condition y_0.
- **t0** Initial time t_0.
- **ts** Output time when solution is seeked.
- **theta** Parameters to be passed to **ODE_rhs** function.
- **x_r** Real data to be passed to **ODE_rhs** function.
- **x_i** Integer data to be passed to **ODE_rhs** function.
- **rtol** Relative tolerance, default to 1.e-6(**rk45**) and 1.e-8(**adams** and **bdf**).
- **atol** Absolute tolerance, default to 1.e-6(**rk45**) and 1.e-8(**adams** and **bdf**).
- **max_step** Maximum number of steps allowed between neighboring time in **ts**, default to 100000.

7.4. Return value. An n-by-nd 2d-array, where n is the size of **ts** and nd the dimension of the system.

7.5. Note.

- Three numerical integrator are provided:
  - **pmx_integrate_ode_adams**: Adams-Moulton method,
  - **pmx_integrate_ode_bdf**: Backward-differentiation formular,
  - **pmx_integrate_ode_rk45**: Runge-Kutta 4/5 method.

  When not equipped with further understanding of the ODE system, as a rule of thumb we suggest user try **rk45** integrator first, **bdf** integrator when the system is suspected to be stiff, and **adams** when a non-stiff system needs to be solved with higher accuracy/smaller tolerance.
- All three integrators support adaptive stepping. To achieve that, at step i estimated error e_i is calculated and compared with given tolerance so that(roughly speaking)

$$e_i < \|\text{rtol} \times \tilde{y} + \text{atol}\| \tag{3.5}$$

  Here $\tilde{y}$ is the numerical solution of y at current step and $\|\cdot\|$ indicates certain norm. When the above check fails, the solver attempts to reduce step size and retry. The default values of **atol**, **rtol**, and **max_step** are based on Stan's ODE functions and should not be considered as optimal. User should make problem-dependent decision on **rtol** and **atol**, according to estimated scale of the unknowns, so that the error would not affect inference on statistical variance of quantities that enter the Stan model. In particular, when an unknown can be neglected below certain threshold without affecting the rest of the dynamic system, setting **atol** greater than that threshold will avoid spurious and error-prone computation. See [6] and 1.4 of [8] for details.
- With optional arguments indicated by square bracket, the following calls are allowed:

```
pmx_integrate_ode_[adams || rk45 || bdf](..., x_i);
pmx_integrate_ode_[adams || rk45 || bdf](..., x_i, rel_tol, abs_tol, max_step);
```

8. ODE group integrator Function

8.1. Description. All the preivous functions solves for a single ODE system. Torsten also provides group modeling counterparts for ODE integrators. The functions solve for a group of ODE systems that share an ODE RHS but with different parameters. They have similar signatures to single-ODE integration functions.

8.2. Usage.

```
matrix pmx_integrate_ode_group_[adams || rk45 || bdf](ODE_system, real[ , ] y0, real t0,
                                                      int[] len, real[] ts,
```

```
                                                    real[ , ] theta, real[ , ] x_r, int[ ,
                                                     ↪  ] x_i,
                                                    [ real rtol, real atol, int max_step ]
                                                     ↪  );
```

Here `[adams || rk45 || bdf]` indicates the function name can be of any of the three suffixes. See section 7.5.

- **`ODE_rhs`** Function that specifies the right-hand-side f. See Section 7.3.
- **`y0`** Initial condition y_0 for each subsystem in the group. The first dimension equals to the size of the group.
- **`t0`** Initial time t_0.
- **`len`** A vector that contains the number of output time points for each subsystem. The lenght of the vector equals to the size of the group.
- **`ts`** Output time when solution is seeked, consisting of **`ts`** of each subsystem concatenated.
- **`theta`** 2d-array parameters to be passed to **`ODE_rhs`** function. Each row corresponds to one subsystem.
- **`x_r`** 2d-array real data to be passed to **`ODE_rhs`** function. Each row corresponds to one subsystem.
- **`x_i`** 2d-array integer data to be passed to **`ODE_rhs`** function. Each row corresponds to one subsystem.
- **`rtol`** Relative tolerance.
- **`atol`** Absolute tolerance.
- **`max_step`** Maximum number of steps allowed between neighboring time in **`ts`**.

8.3. Return value. An `n`-by-`nd` matrix, where `n` is the size of **`ts`** and `nd` the dimension of the system.

8.4. Note.

- With optional arguments indicated by square bracket, the following calls are allowed:

```
pmx_integrate_group_[adams || rk45 || bdf](..., x_i);
pmx_integrate_group_[adams || rk45 || bdf](..., x_i, rel_tol, abs_tol, max_step);
```

- The group integrators support paralleisation through Message Passing Interface(MPI). One can access this feature through `cmdstan` or `cmdstanr` interface.

```
# cmdstan interface user need to add "TORSTEN_MPI=1" and
# "TBB_CXX_TYPE=gcc" in "cmdstan/make/local" file. In linux & macos
# this can be done as
echo "TORSTEN_MPI=1" > cmdstan/make/local
echo "TBB_CXX_TYPE=gcc" > cmdstan/make/local # "gcc" should be replaced by user's C compiler
make path-to-model/model_name
mpiexec -n number_of_processes model_name sample... # additional cmdstan options
```

```
library("cmdstanr")
cmdstan_make_local(cpp_options = list("TORSTEN_MPI" = "1", "TBB_CXX_TYPE"="gcc"))  # "gcc"
 ↪  should be replaced by user's C compiler
rebuild_cmdstan()
mod <- cmdstan_model(path-to-model-file, quiet=FALSE, force_recompile=TRUE)
f <- mod$sample_mpi(data = ..., chains = 1, mpi_args = list("n" = number_of_processes),
 ↪  refresh = 200)
```

Here n denotes number of MPI processes, so that N ODE systems are distributed to and solved by n processes evenly. Note that to access this feature user must have MPI installed, and some MPI installation may require set additional compiler arguments, such as `CXXLFAGS` and `LDFLAGS`.

9. Univariate integral

```
real univariate_integral_rk45(f, t0, t1, theta, x_r, x_i)
```

```
real univariate_integral_bdf(f, t0, t1, theta, x_r, x_i)
```

Based on the ODE solver capability in Stan, Torsten provides functions calculating the integral of a univariate function. The integrand function f must follow the signature

```
  real f(real t, real[] theta, real[] x_r, int[] x_i) {
    /* ... */
}
```

10. Piecewise linear interpolation

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

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

Torsten also provides function `linear_interpolation` 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. Specifically, `linear_interpolation` implements the following function

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

- The x values must be in increasing order, i.e. $x_i < x_{i+1}$.
- All three arguments may be data or parameters.

Examples

1. Two-compartment model for single patient

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 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.

With the plasma concentration $\hat{c}$ solved from two-compartment ODEs in 2, we simulate c according to:

$$\log(c) \sim N\left(\log(\hat{c}), \sigma^2\right)$$
$$(CL, Q, V_2, V_3, ka) = \left(5 \text{ L/h}, 8 \text{ L/h}, 20 \text{ L}, 70 \text{ L}, 1.2 \text{ h}^{-1}\right)$$
$$\sigma^2 = 0.01$$

The data are generated using the R package `mrgsolve` [1].

Code below shows how Torsten function **`pmx_solve_twocpt`** can be used to fit the above model.

```
data{
  int<lower = 1> nt;  // number of events
  int<lower = 1> nObs;  // number of observation
  int<lower = 1> iObs[nObs];  // index of observation

  // NONMEM data
  int<lower = 1> 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{
  vector[nObs] logCObs = log(cObs);
  int nTheta = 5;  // number of ODE parameters in Two Compartment Model
  int nCmt = 3;  // number of compartments in model
}

parameters{
  real<lower = 0> CL;
  real<lower = 0> Q;
  real<lower = 0> V1;
  real<lower = 0> V2;
  real<lower = 0> ka;
  real<lower = 0> sigma;

}
transformed parameters{
  real theta[nTheta];  // ODE parameters
```

```
  row_vector<lower = 0>[nt] cHat;
  vector<lower = 0>[nObs] cHatObs;
  matrix<lower = 0>[nCmt, nt] x;

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

  // PKModelTwoCpt takes in the NONMEM data, followed by the parameter
  // arrays abd returns a matrix with the predicted amount in each
  // compartment at each event.
  x = pmx_solve_twocpt(time, amt, rate, ii, evid, cmt, addl, ss, theta);

  cHat = x[2, :] ./ V1; // we're interested in the amount in the second compartment

  cHatObs = cHat'[iObs]; // predictions for observed data recors
}

model{
  // informative prior
  CL ~ lognormal(log(10), 0.25);
  Q ~ lognormal(log(15), 0.5);
  V1 ~ lognormal(log(35), 0.25);
  V2 ~ lognormal(log(105), 0.5);
  ka ~ lognormal(log(2.5), 1);
  sigma ~ cauchy(0, 1);

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

Three MCMC chains of 2000 iterations are simulated. The first 1000 iteration of each chain were discarded. Thus 1000 MCMC samples per chain were used for the subsequent analyses. The MCMC history plots(Figure 4.1) 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 4.3) 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 4.1 and Figure 4.2 are consistent with the values used for simulation.

	mean	se_{mean}	sd	2.5%	25%	50%	75%	97.5%	n_{eff}	Rhat
CL	4.823	0.002	0.092	4.647	4.762	4.823	4.883	5.012	2392.155	1.00
Q	7.596	0.013	0.586	6.479	7.201	7.594	7.977	8.785	1923.939	1.00
V1	21.073	0.069	2.573	16.017	19.352	21.046	22.817	26.097	1385.883	1.00
V2	76.365	0.105	5.611	65.805	72.623	76.172	79.916	87.971	2862.184	1.00
ka	1.231	0.004	0.177	0.907	1.107	1.221	1.344	1.599	1581.825	1.00
sigma	0.109	0.000	0.012	0.089	0.100	0.108	0.116	0.134	2560.112	1.00

TABLE 4.1. Summary of the MCMC simulations of the marginal posterior distribu- tions of the model parameters

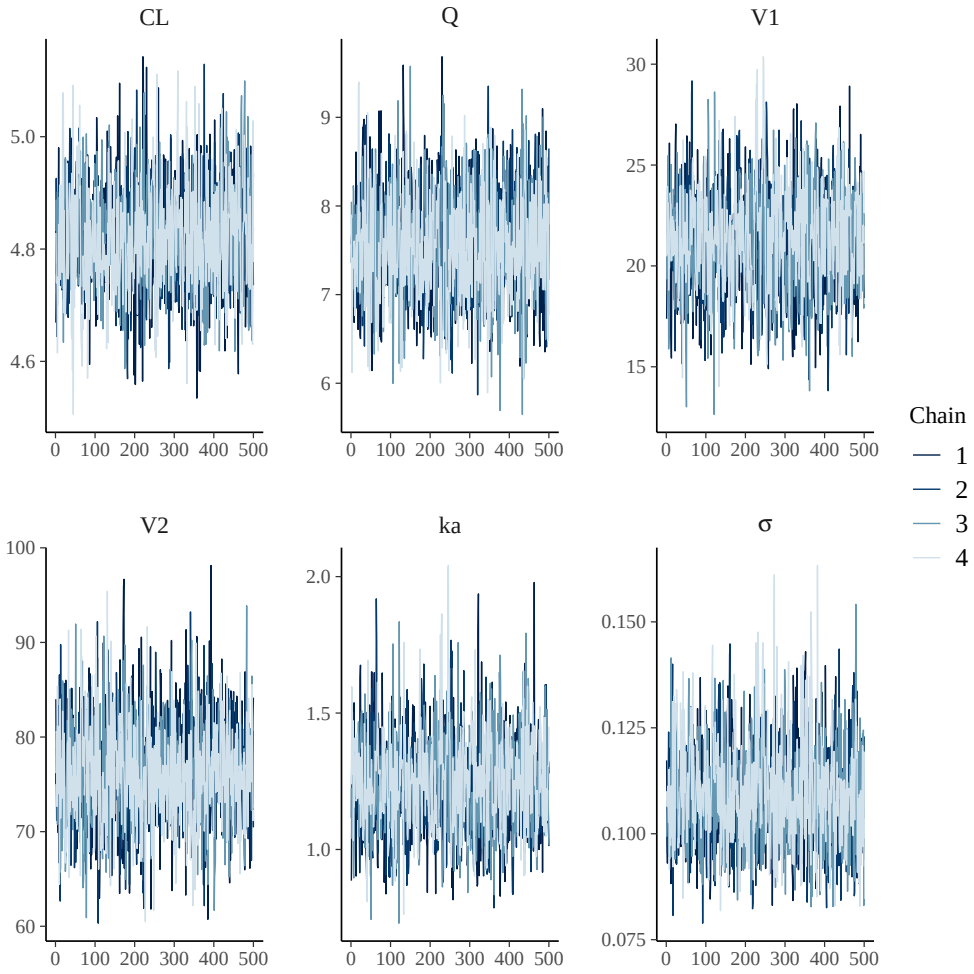

FIGURE 4.1. MCMC history plots for the parameters of a two compartment model with first order absorption (each color corresponds to a different chain)

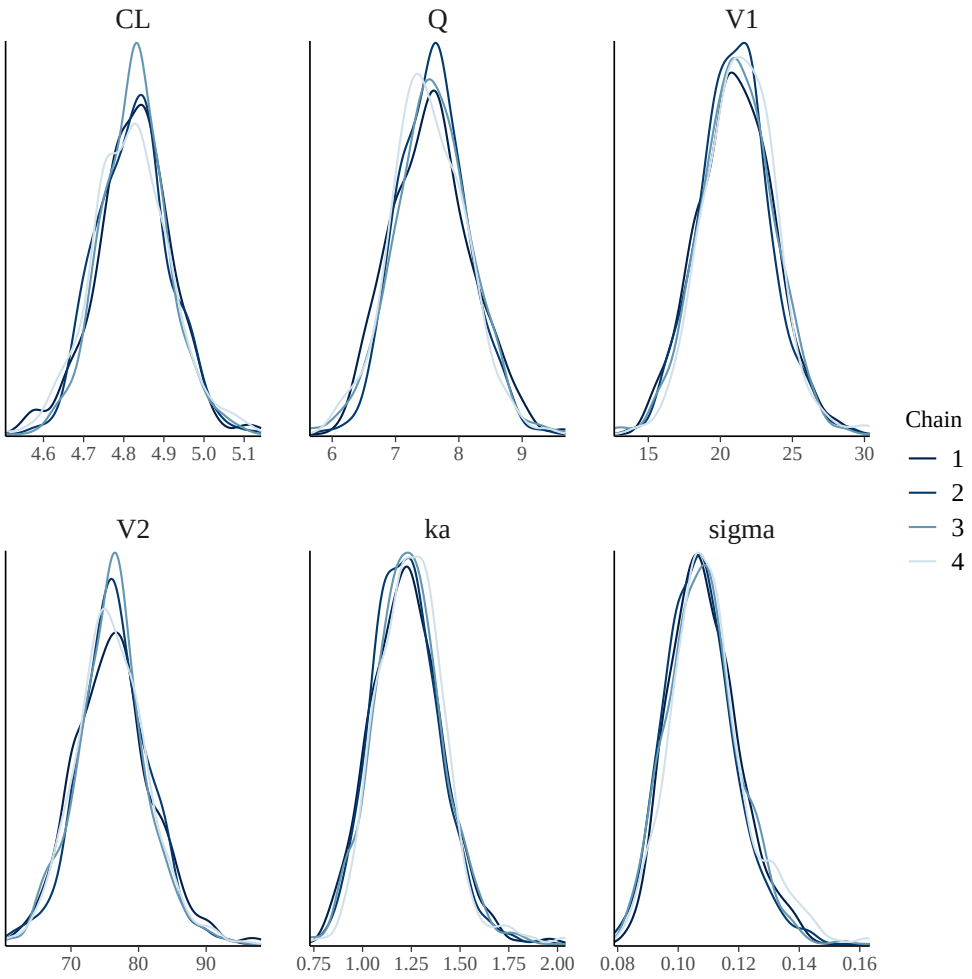

FIGURE 4.2. Posterior Marginal Densities of the Model Parameters of a two compartment model with first order absorption (each color corresponds to a different chain)

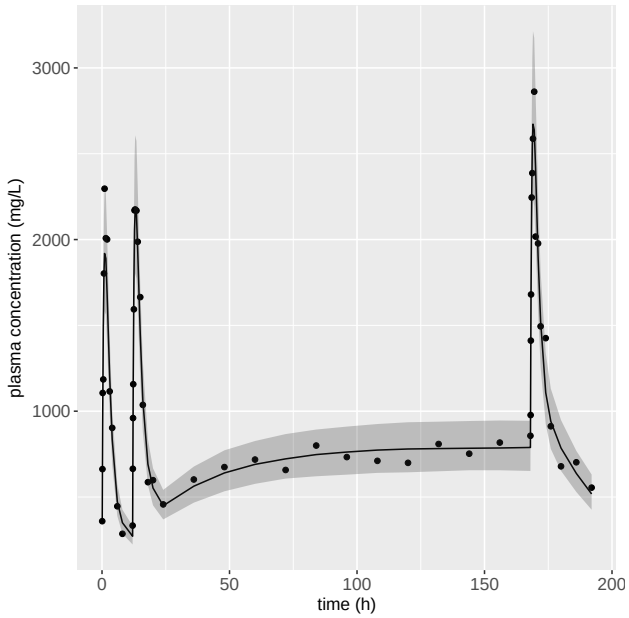

FIGURE 4.3. Predicted (posterior median and 90% credible intervals) and observed plasma drug concentrations of a two compartment model with first order absorption

2. Two-compartment model as a linear ODE model for single patient

Using **pmx_solve_linode**, the following example fits a two-compartment model with first order absorption.

```
// LinTwoCptModelExample.stan
// Run two compartment model using matrix exponential solution
// Heavily anotated to help new users

data{
  int<lower = 1> nt;  // number of events
  int<lower = 1> nObs;  // number of observations
  int<lower = 1> iObs[nObs];  // index of observation

  // NONMEM data
  int<lower = 1> cmt[nt];
  int evid[nt];
  int addl[nt];
  int ss[nt];
  real amt[nt];
  real time[nt];
  real rate[nt];
  real ii[nt];

  row_vector<lower = 0>[nObs] cObs;  // observed concentration (dependent variable)
}

transformed data{
  row_vector[nObs] logCObs = log(cObs);
  int nCmt = 3;
  real biovar[nCmt];
  real tlag[nCmt];

  for (i in 1:nCmt) {
    biovar[i] = 1;
    tlag[i] = 0;
  }
}

parameters{
  real<lower = 0> CL;
  real<lower = 0> Q;
  real<lower = 0> V1;
  real<lower = 0> V2;
  real<lower = 0> ka;
  real<lower = 0> sigma;

}

transformed parameters{
  matrix[3, 3] K;
  real k10 = CL / V1;
  real k12 = Q / V1;
  real k21 = Q / V2;
  row_vector<lower = 0>[nt] cHat;
  row_vector<lower = 0>[nObs] cHatObs;
  matrix<lower = 0>[3, nt] 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;

  // linModel takes in the constant rate matrix, the object theta which
  // contains the biovariability fraction and the lag time of each compartment,
  // and the NONMEM data.
  x = pmx_solve_linode(time, amt, rate, ii, evid, cmt, addl, ss,
                       K, biovar, tlag);

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

  for(i in 1:nObs){
    cHatObs[i] = cHat[iObs[i]];  // predictions for observed data records
  }
}

model{
  // informative prior
  CL ~ lognormal(log(10), 0.25);
  Q ~ lognormal(log(15), 0.5);
  V1 ~ lognormal(log(35), 0.25);
  V2 ~ lognormal(log(105), 0.5);
  ka ~ lognormal(log(2.5), 1);
  sigma ~ cauchy(0, 1);

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

3. Two-compartment model solved by numerical integrator for single patient

Using `pmx_solve_rk45`, the following example fits a two-compartment model with first order absorption. User-defined function `twoCptModelODE` describes the RHS of the ODEs.

```
functions{
  real[] ode_rhs(real t, real[] x, real[] parms, real[] rate, int[] dummy){
    real CL = parms[1];
    real Q = parms[2];
    real V1 = parms[3];
    real V2 = parms[4];
    real ka = parms[5];

    real k10 = CL / V1;
    real k12 = Q / V1;
    real k21 = Q / V2;

    real y[3];

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

    return y;
  }
}

data {
  int<lower = 1> nt;            // number of events
  int<lower = 1> nObs;          // number of observations
  int<lower = 1> iObs[nObs];    // index of observation

  // NONMEM data
```

```
  int<lower = 1> cmt[nt];
  int evid[nt];
  int addl[nt];
  int ss[nt];
  real amt[nt];
  real time[nt];
  real rate[nt];
  real ii[nt];

  row_vector<lower = 0>[nObs] cObs;  // observed concentration (dependent variable)
}

transformed data {
  row_vector[nObs] logCObs = log(cObs);
  int nTheta = 5;   // number of parameters
  int nCmt = 3;   // number of compartments

  // real biovar[nCmt];
  // real tlag[nCmt];

  // for (i in 1:nCmt) {
  //   biovar[i] = 1;
  //   tlag[i] = 0;
  // }
}

parameters {
  real<lower = 0> CL;
  real<lower = 0> Q;
  real<lower = 0> V1;
  real<lower = 0> V2;
  real<lower = 0> ka;
  real<lower = 0> sigma;
}

transformed parameters {
  real theta[nTheta];
  row_vector<lower = 0>[nt] cHat;
  row_vector<lower = 0>[nObs] cHatObs;
  matrix<lower = 0>[3, nt] x;

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

  x = pmx_solve_rk45(ode_rhs, 3,
                     time, amt, rate, ii, evid, cmt, addl, ss,
                     theta, 1e-5, 1e-8, 1e5);

  cHat = x[2, ] ./ V1;

  for(i in 1:nObs){
    cHatObs[i] = cHat[iObs[i]];  // predictions for observed data records
  }
}

model{
  // informative prior
  CL ~ lognormal(log(10), 0.25);
  Q ~ lognormal(log(15), 0.5);
  V1 ~ lognormal(log(35), 0.25);
```

```
  V2 ~ lognormal(log(105), 0.5);
  ka ~ lognormal(log(2.5), 1);
  sigma ~ cauchy(0, 1);

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

4. Joint PK-PD model

A Friberg-Karlsson Semi-Mechanistic model [5] couples a PK model with a PD effect. In the current example, we use the two compartment model in section 2 for PK model.

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 4.4). 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.

$$\log(ANC_i) \sim N(\log(Circ), \sigma^2_{ANC}) \tag{4.1}$$
$$Circ = f_{FK}(MTT, Circ_0, \alpha, \gamma, c) \tag{4.2}$$
$$(MTT, Circ_0, \alpha, \gamma, ktr) = (125, 5.0, 3 \times 10^{-4}, 0.17) \tag{4.3}$$
$$\sigma^2_{ANC} = 0.001 \tag{4.4}$$

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:

$$y'_{\text{prol}} = k_{\text{prol}} y_{\text{prol}} (1 - E_{\text{drug}}) \left(\frac{Circ_0}{y_{\text{circ}}} \right)^{\gamma} - k_{\text{tr}} y_{\text{prol}} \tag{4.5a}$$
$$y'_{\text{trans1}} = k_{\text{tr}} y_{\text{prol}} - k_{\text{tr}} y_{\text{trans1}} \tag{4.5b}$$
$$y'_{\text{trans2}} = k_{\text{tr}} y_{\text{trans1}} - k_{\text{tr}} y_{\text{trans2}} \tag{4.5c}$$
$$y'_{\text{trans3}} = k_{\text{tr}} y_{\text{trans2}} - k_{\text{tr}} y_{\text{trans3}} \tag{4.5d}$$
$$y'_{\text{circ}} = k_{\text{tr}} y_{\text{trans3}} - k_{\text{tr}} y_{\text{circ}} \tag{4.5e}$$

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

The ODEs specifying the Two Compartment Model (Equation (3.2a)) do not depend on the PD ODEs (Equation (4.5)) and can be solved analytically using Torsten's `pmx_solve_twocpt` function. 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.

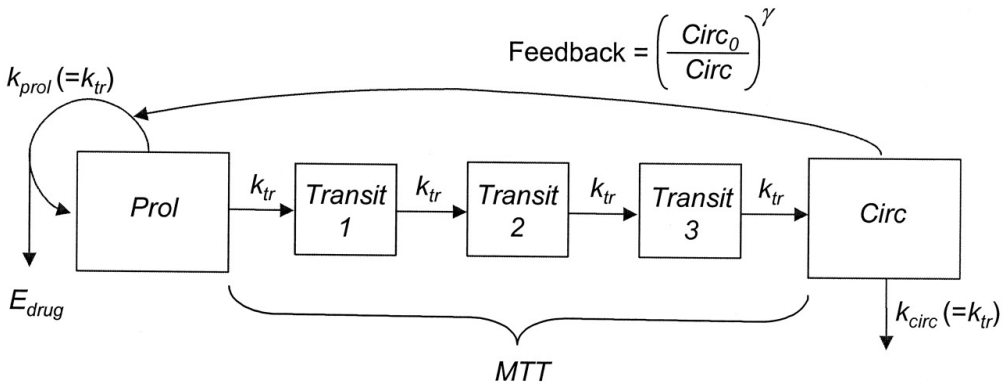

FIGURE 4.4. Friberg-Karlsson semi-mechanistic Model.

```
functions{
  real[] FK_ODE(real t,
                real[] y,
                real[] y_pk,
                real[] theta,
                real[] rdummy,
                int[] idummy){
    /* PK variables */
    real VC = theta[3];

    /* PD variable */
    real mtt      = theta[6];
    real circ0    = theta[7];
    real alpha    = theta[8];
    real gamma    = theta[9];
    real ktr      = 4.0 / 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];

    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;
  }
}

data{
  int<lower = 1> nt;
  int<lower = 1> nObsPK;
  int<lower = 1> nObsPD;
  int<lower = 1> iObsPK[nObsPK];
  int<lower = 1> iObsPD[nObsPD];
  real<lower = 0> amt[nt];
  int<lower = 1> cmt[nt];
  int<lower = 0> evid[nt];
  real<lower = 0> time[nt];
  real<lower = 0> ii[nt];
  int<lower = 0> addl[nt];
  int<lower = 0> ss[nt];
  real rate[nt];
  vector<lower = 0>[nObsPK] cObs;
  vector<lower = 0>[nObsPD] neutObs;

  real<lower = 0> circ0Prior;
  real<lower = 0> circ0PriorCV;
  real<lower = 0> mttPrior;
  real<lower = 0> mttPriorCV;
  real<lower = 0> gammaPrior;
  real<lower = 0> gammaPriorCV;
  real<lower = 0> alphaPrior;
  real<lower = 0> alphaPriorCV;
}
```

```
transformed data{
  int nOde = 5;
  vector[nObsPK] logCObs;
  vector[nObsPD] logNeutObs;
//  int idummy[0];
//  real rdummy[0];

  int nTheta;
  int nIIV;

  int n;                        /* ODE dimension */
  real rtol;
  real atol;
  int max_step;

  n = 8;
  rtol = 1e-8;
  atol = 1e-8;
  max_step = 100000;

  logCObs = log(cObs);
  logNeutObs = log(neutObs);

  nIIV = 7; // parameters with IIV
  nTheta = 9; // number of parameters
}

parameters{

  real<lower = 0> CL;
  real<lower = 0> Q;
  real<lower = 0> VC;
  real<lower = 0> VP;
  real<lower = 0> ka;
  real<lower = 0> mtt;
  real<lower = 0> circ0;
  real<lower = 0> alpha;
  real<lower = 0> gamma;
  real<lower = 0> sigma;
  real<lower = 0> sigmaNeut;

  // IIV parameters
  cholesky_factor_corr[nIIV] L;
  vector<lower = 0>[nIIV] omega;
}

transformed parameters{
  vector[nt] cHat;
  vector<lower = 0>[nObsPK] cHatObs;
  vector[nt] neutHat;
  vector<lower = 0>[nObsPD] neutHatObs;
  real<lower = 0> theta[nTheta];
  matrix[nt, nOde + 3] x;
  real biovar[nTheta];
  real tlag[nTheta];

  for (i in 1:nTheta) {
    biovar[i] = 1.0;
    tlag[i] = 0.0;
  }

  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, rtol, atol, max_step);

  cHat = col(x, 2) / VC;
  neutHat = col(x, 8) + circ0;

  for(i in 1:nObsPK) cHatObs[i]    = cHat[iObsPK[i]];
  for(i in 1:nObsPD) neutHatObs[i] = neutHat[iObsPD[i]];

}

model {

  // Priors
  CL    ~ normal(0, 20);
  Q     ~ normal(0, 20);
  VC    ~ normal(0, 100);
  VP    ~ normal(0, 1000);
  ka    ~ normal(0, 5);
  sigma ~ cauchy(0, 1);

  mtt       ~ lognormal(log(mttPrior), mttPriorCV);
  circ0     ~ lognormal(log(circ0Prior), circ0PriorCV);
  alpha     ~ lognormal(log(alphaPrior), alphaPriorCV);
  gamma     ~ lognormal(log(gammaPrior), gammaPriorCV);
  sigmaNeut ~ cauchy(0, 1);

  // Parameters for Matt's trick
  L ~ lkj_corr_cholesky(1);
  omega ~ cauchy(0, 1);

  // observed data likelihood
  logCObs ~ normal(log(cObs), sigma);
  logNeutObs ~ normal(log(neutObs), sigmaNeut);
}
```

5. Two-compartment population model

Using `pmx_solve_group_bdf`, the following example fits a two-compartment population model.

```
functions{

  // define ODE system for two compartmnt model
  real[] twoCptModelODE(real t,
                        real[] x,
                        real[] parms,
                        real[] rate,  // in this example, rate is treated as data
                        int[] dummy){

    // Parameters
    real CL = parms[1];
    real Q = parms[2];
    real V1 = parms[3];
    real V2 = parms[4];
```

```
    real ka = parms[5];

    // Re-parametrization
    real k10 = CL / V1;
    real k12 = Q / V1;
    real k21 = Q / V2;

    // Return object (derivative)
    real y[3];  // 1 element per compartment of
                // the model

    // PK component of the ODE system
    y[1] = -ka*x[1];
    y[2] = ka*x[1] - (k10 + k12)*x[2] + k21*x[3];
    y[3] = k12*x[2] - k21*x[3];

    return y;
  }
}
data{
  int<lower = 1> np;            /* population size */
  int<lower = 1> nt;  // number of events
  int<lower = 1> nObs;  // number of observations
  int<lower = 1> iObs[nObs];  // index of observation

  // NONMEM data
  int<lower = 1> cmt[np * nt];
  int evid[np * nt];
  int addl[np * nt];
  int ss[np * nt];
  real amt[np * nt];
  real time[np * nt];
  real rate[np * nt];
  real ii[np * nt];

  real<lower = 0> cObs[np*nObs];  // observed concentration (dependent variable)
}

transformed data {
  real logCObs[np*nObs];
  int<lower = 1> len[np];
  int<lower = 1> len_theta[np];
  int<lower = 1> len_biovar[np];
  int<lower = 1> len_tlag[np];

  int nTheta = 5;   // number of parameters
  int nCmt = 3;   // number of compartments
  real biovar[np * nt, nCmt];
  real tlag[np * nt, nCmt];

  logCObs = log(cObs);

  for (id in 1:np) {
    for (j in 1:nt) {
      for (i in 1:nCmt) {
        biovar[(id - 1) * nt + j, i] = 1;
        tlag[(id - 1) * nt + j, i] = 0;
      }
    }
    len[id] = nt;
    len_theta[id] = nt;
    len_biovar[id] = nt;
    len_tlag[id] = nt;
```

```
  }
}

parameters{
  real<lower = 0> CL[np];
  real<lower = 0> Q[np];
  real<lower = 0> V1[np];
  real<lower = 0> V2[np];
  real<lower = 0> ka[np];
  real<lower = 0> sigma[np];
}

transformed parameters{
  real theta[np * nt, nTheta];
  vector<lower = 0>[nt] cHat[np];
  real<lower = 0> cHatObs[np*nObs];
  matrix[3, nt * np] x;

  for (id in 1:np) {
    for (it in 1:nt) {
      theta[(id - 1) * nt + it, 1] = CL[id];
      theta[(id - 1) * nt + it, 2] = Q[id];
      theta[(id - 1) * nt + it, 3] = V1[id];
      theta[(id - 1) * nt + it, 4] = V2[id];
      theta[(id - 1) * nt + it, 5] = ka[id];
    }
  }

  x = pmx_solve_group_bdf(twoCptModelODE, 3, len,
                          time, amt, rate, ii, evid, cmt, addl, ss,
                          theta, biovar, tlag);

  for (id in 1:np) {
    for (j in 1:nt) {
      cHat[id][j] = x[2, (id - 1) * nt + j] ./ V1[id];
    }
  }

  for (id in 1:np) {
    for(i in 1:nObs){
      cHatObs[(id - 1)*nObs + i] = cHat[id][iObs[i]];  // predictions for observed data
       ↪  records
    }
  }
}

model{
  // informative prior
  for(id in 1:np){
    CL[id] ~ lognormal(log(10), 0.25);
    Q[id] ~ lognormal(log(15), 0.5);
    V1[id] ~ lognormal(log(35), 0.25);
    V2[id] ~ lognormal(log(105), 0.5);
    ka[id] ~ lognormal(log(2.5), 1);
    sigma[id] ~ cauchy(0, 1);

    for(i in 1:nObs){
      logCObs[(id - 1)*nObs + i] ~ normal(log(cHatObs[(id - 1)*nObs + i]), sigma[id]);
    }
  }
}
```

6. Lotka-Volterra group model

Using `pmx_integrate_ode_group_rk45`, the following example fits a Lotka-Volterra group model, based on Stan's case study.

```
functions {
  real[] dz_dt(real t,       // time
               real[] z,     // system state {prey, predator}
               real[] theta, // parameters
               real[] x_r,   // unused data
               int[] x_i) {
    real u = z[1];
    real v = z[2];

    real alpha = theta[1];
    real beta = theta[2];
    real gamma = theta[3];
    real delta = theta[4];

    real du_dt = (alpha - beta * v) * u;
    real dv_dt = (-gamma + delta * u) * v;
    return { du_dt, dv_dt };
  }
}
data {
  int<lower = 0> N_subj;      // number of subjects
  int<lower = 0> N;           // number of measurement times
  real ts_0[N];                 // measurement times > 0
  real y0_0[2];     // initial measured populations
  real<lower = 0> y_0[N, 2];    // measured populations
}
transformed data {
  int len[N_subj] = rep_array(N, N_subj);
  real y0[N_subj, 2] = rep_array(y0_0, N_subj);
  real y[N_subj, N, 2] = rep_array(y_0, N_subj);
  real ts[N_subj * N];
  for (i in 1:N_subj) {
    ts[((i-1)*N + 1) : (i*N)] = ts_0;
  }
}

parameters {
  real<lower = 0> theta[N_subj, 4];   // { alpha, beta, gamma, delta }
  real<lower = 0> z_init[N_subj, 2];  // initial population
  real<lower = 0> sigma[N_subj, 2];   // measurement errors
}
transformed parameters {
  matrix[2, N_subj * N] z;
  z = pmx_integrate_ode_group_rk45(dz_dt, z_init, 0, len, ts, theta,
   ↪  rep_array(rep_array(0.0, 0), N_subj), rep_array(rep_array(0, 0),N_subj));
}
model {
  for (isub in 1:N_subj) {
    theta[isub, {1, 3}] ~ normal(1, 0.5);
    theta[isub, {2, 4}] ~ normal(0.05, 0.05);
    sigma[isub] ~ lognormal(-1, 1);
    z_init[isub] ~ lognormal(10, 1);
    for (k in 1:2) {
      y0[isub, k] ~ lognormal(log(z_init[isub, k]), sigma[isub, k]);
      y[isub, , k] ~ lognormal(log(z[k, ((isub-1)*N + 1):(isub*N)]), sigma[isub, k]);
    }
  }
```

7. Univariate integral of a quadratic function

integral of a quadratic function. This example shows how to use `univariate_integral_rk45` to calculate the integral of a quadratic function.

```
functions {
  real fun_ord2(real t, real[] theta, real[] x_r, int[] x_i) {
    real a = 2.3;
    real b = 2.0;
    real c = 1.5;
    real res;
    res = a + b * t + c * t * t;
    return res;
  }
}
data {
  real t0;
  real t1;
  real dtheta[2];
  real x_r[0];
  int x_i[0];
}
transformed data {
  real univar_integral;
  univar_integral = univariate_integral_rk45(func, t0, t1, dtheta,
                          x_r, x_i);
}
/* ... */
```

8. Linear intepolation

This example illustrates how to use `linear_intepolationi` to fit a piecewise linear function to a data set consisting of (x, y) pairs.

```
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);
}
```

9. Effect Compartment Population Model

Here we expand the example in 2 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.

9.1. Population Model for Plasma Drug Concentration c.

$$\log\left(c_{ij}\right) \sim N\left(\log\left(\widehat{c}_{ij}\right), \sigma^2\right),$$

$$\widehat{c}_{ij} = f_{2cpt}\left(t_{ij}, D_j, \tau_j, CL_j, Q_j, V_{1j}, V_{2j}, k_{aj}\right),$$

$$\log\left(CL_j, Q_j, V_{ssj}, k_{aj}\right) \sim N\left(\log\left(\widehat{CL}\left(\frac{bw_j}{70}\right)^{0.75}, \widehat{Q}\left(\frac{bw_j}{70}\right)^{0.75}, \widehat{V}_{ss}\left(\frac{bw_j}{70}\right), \widehat{k}_a\right), \Omega\right),$$

$$V_{1j} = f_{V_1} V_{ssj},$$

$$V_{2j} = \left(1 - f_{V_1}\right) V_{ssj},$$

$$\left(\widehat{CL}, \widehat{Q}, \widehat{V}_{ss}, \widehat{k}_a, f_{V_1}\right) = \left(10 \text{ L/h}, 15 \text{ L/h}, 140 \text{ L}, 2 \text{ h}^{-1}, 0.25\right),$$

$$\Omega = \begin{pmatrix} 0.25^2 & 0 & 0 & 0 \\ 0 & 0.25^2 & 0 & 0 \\ 0 & 0 & 0.25^2 & 0 \\ 0 & 0 & 0 & 0.25^2 \end{pmatrix},$$

$$\sigma = 0.1$$

Furthermore we add a fourth compartment in which we measure a PD effect(Figure 4.5).

9.2. Effect Compartment Model for PD response R.

$$R_{ij} \sim N\left(\widehat{R}_{ij}, \sigma_R^2\right),$$

$$\widehat{R}_{ij} = \frac{E_{max} c_{eij}}{EC_{50j} + c_{eij}},$$

$$c'_{e\cdot j} = k_{e0j}\left(c_{\cdot j} - c_{e\cdot j}\right),$$

$$\log\left(EC_{50j}, k_{e0j}\right) \sim N\left(\log\left(\widehat{EC}_{50}, \widehat{k}_{e0}\right), \Omega_R\right),$$

$$\left(E_{max}, \widehat{EC}_{50}, \widehat{k}_{e0}\right) = \left(100, 100.7, 1\right),$$

$$\Omega_R = \begin{pmatrix} 0.2^2 & 0 \\ 0 & 0.25^2 \end{pmatrix}, \quad \sigma_R = 10.$$

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

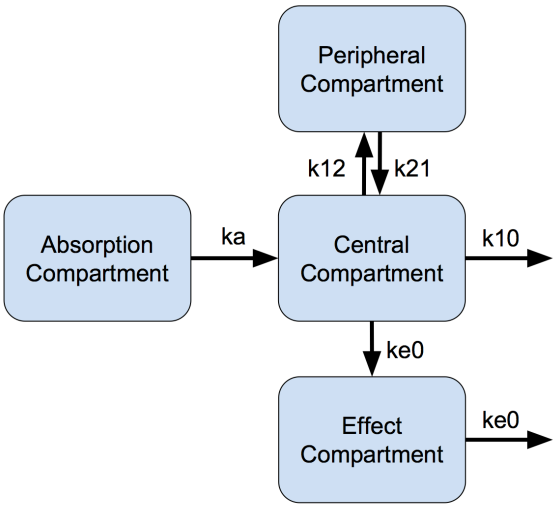

FIGURE 4.5. Effect Compartment Model

- 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)

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 `pmx_solve_linode`. 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 `pmx_solve_linode` within a loop over the individual subjects rather than over the individual observations.

```
transformed parameters{
  vector<lower = 0>[nRandom] thetaHat;
  cov_matrix[nRandom] Omega;
  real<lower = 0> CL[nSubjects];
  real<lower = 0> Q[nSubjects];
  real<lower = 0> V1[nSubjects];
  real<lower = 0> V2[nSubjects];
  real<lower = 0> ka[nSubjects];
  real<lower = 0> ke0[nSubjects];
  real<lower = 0> EC50[nSubjects];
  matrix[nCmt, nCmt] K;
  real k10;
  real k12;
  real k21;
  row_vector<lower = 0>[nt] cHat;
  row_vector<lower = 0>[nObs] cHatObs;
```

```
  row_vector<lower = 0>[nt] respHat;
  row_vector<lower = 0>[nObs] respHatObs;
  row_vector<lower = 0>[nt] ceHat;
  matrix[nCmt, nt] x;

  thetaHat[1] = CLHat;
  thetaHat[2] = QHat;
  thetaHat[3] = V1Hat;
  thetaHat[4] = V2Hat;
  thetaHat[5] = kaHat;

  Omega = quad_form_diag(rho, omega); // diag_matrix(omega) * rho * diag_matrix(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];

    K = rep_matrix(0, nCmt, nCmt);

    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] ] = pmx_solve_linode(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[2, start[j]:end[j]] ./ V1[j];
    ceHat[start[j]:end[j]] = 1000 * x[4, start[j]:end[j]] ./ 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];

}
```

9.3. Results. We use the same diagnosis tools as for the previous examples. The MCMC history plots (Figure 4.6) suggest the 4 chains have converged to common distributions. We note some minor autocorrelations 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 4.8) are in close agreement with the data, notably for

the sparse data case (phase IIa study). The fits to the PD data (Figure 4.9) 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 4.2) and Figure 4.7).

	mean	se_mean	sd	2.5%	25%	50%	75%	97.5%	n_eff	Rhat
lp___	-201.282	10.073	84.189	-333.764	-259.017	-213.416	-154.381	8.549	69.850	1.044
CLHat	10.095	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	103.454	105.472	109.583	1488.296	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	1164.741	1.000
EC50Hat	104.337	0.040	2.100	100.169	102.909	104.345	105.768	108.351	2744.041	1.000
sigma	0.099	0.000	0.002	0.095	0.097	0.099	0.100	0.103	1906.342	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	1190.193	1.000
rho[1,2]	0.194	0.003	0.100	-0.011	0.127	0.195	0.265	0.379	1000.772	1.004
rho[1,3]	-0.157	0.005	0.126	-0.395	-0.243	-0.157	-0.072	0.088	677.709	1.001
rho[2,3]	0.079	0.012	0.155	-0.227	-0.024	0.082	0.181	0.384	180.306	1.021
rho[1,4]	-0.107	0.003	0.112	-0.319	-0.183	-0.110	-0.032	0.118	1081.932	1.002
rho[2,4]	0.194	0.005	0.126	-0.062	0.110	0.199	0.282	0.428	623.035	1.007
rho[3,4]	0.796	0.008	0.094	0.592	0.737	0.808	0.867	0.940	152.112	1.033
rho[1,5]	0.023	0.006	0.135	-0.232	-0.068	0.024	0.115	0.285	564.687	1.003
rho[2,5]	0.119	0.011	0.160	-0.188	0.008	0.118	0.224	0.438	226.174	1.014
rho[3,5]	-0.246	0.018	0.202	-0.663	-0.382	-0.237	-0.105	0.133	119.465	1.021
rho[4,5]	-0.288	0.009	0.155	-0.576	-0.396	-0.291	-0.183	0.014	275.549	1.009

TABLE 4.2. Summary of the MCMC simulations of the marginal posterior distributions of the model parameters for the effect compartment model example.

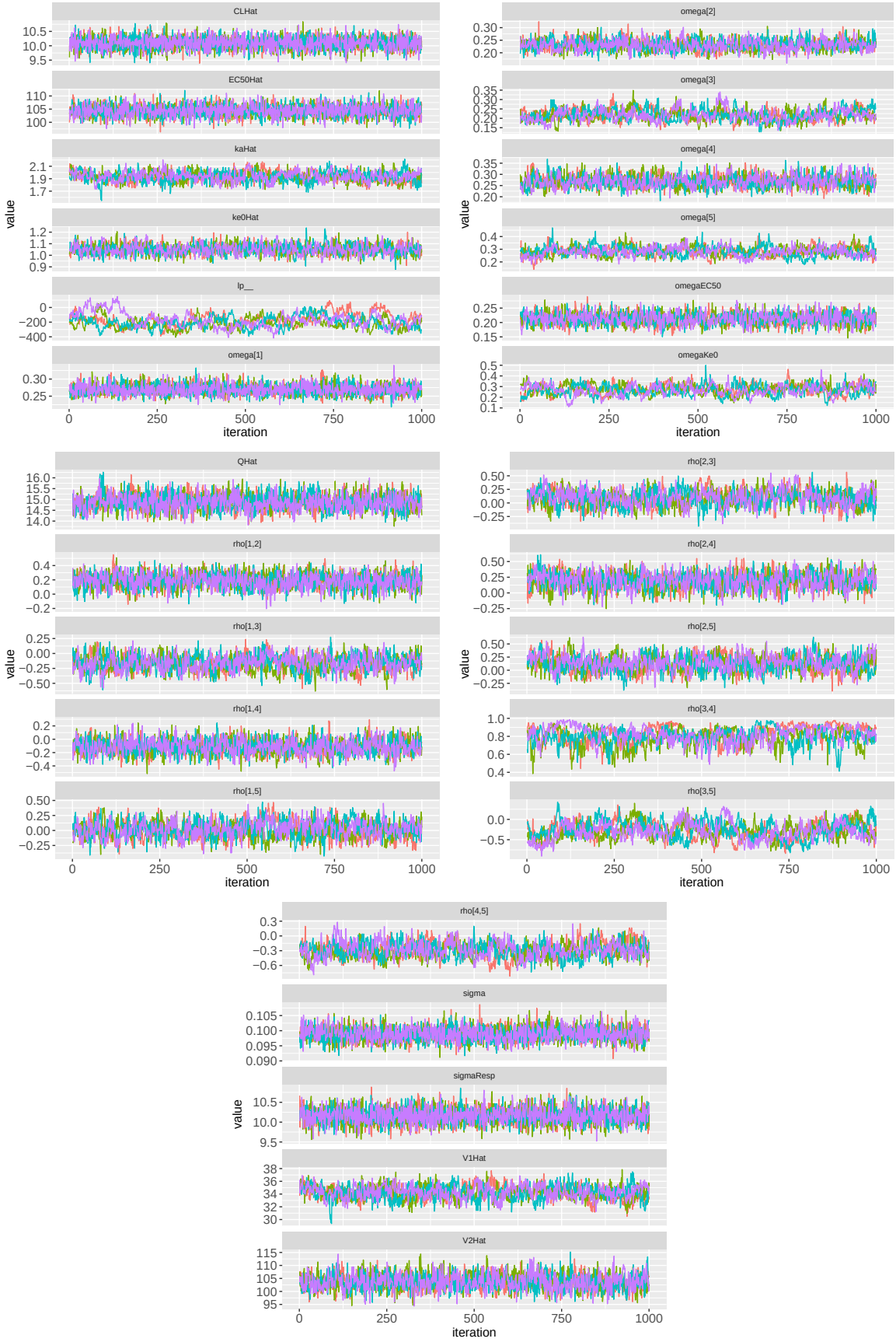

FIGURE 4.6. MCMC history plots for the parameters of an Effect Compartment Model (each color corresponds to a different chain) for example 2

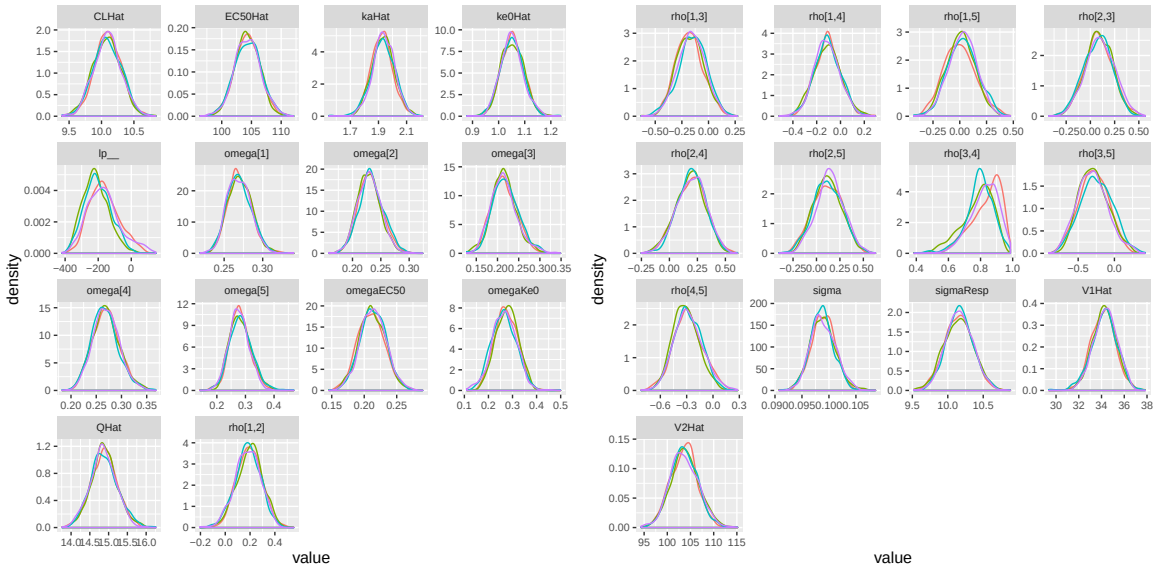

FIGURE 4.7. Posterior Marginal Densities of the Model Parameters of an Effect Compartment Model (each color corresponds to a different chain) for example 2

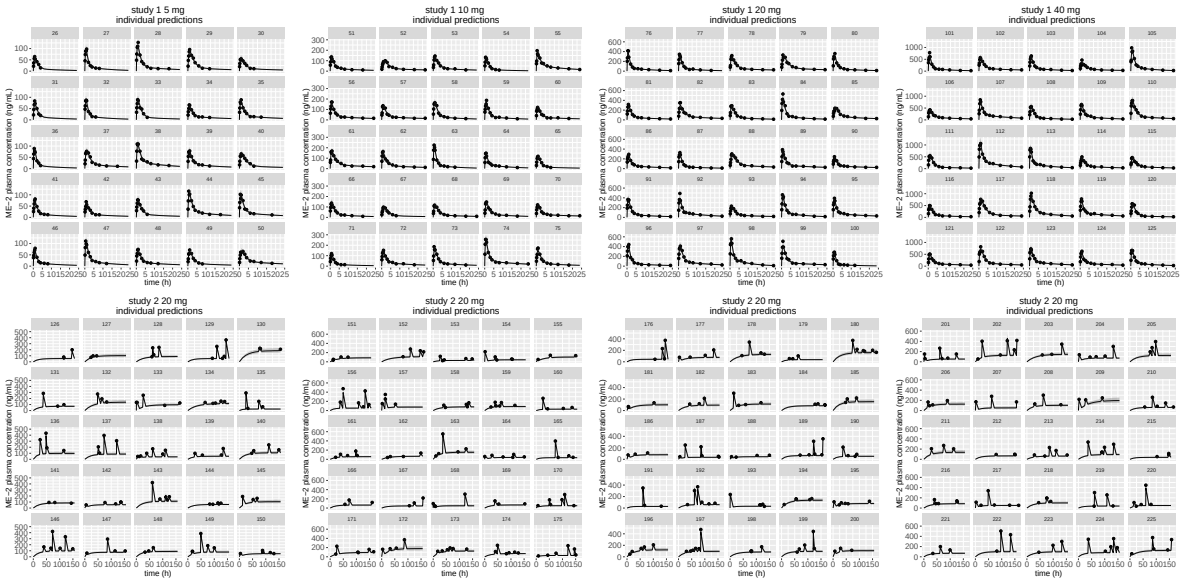

FIGURE 4.8. Predicted (posterior median and 90 % credible intervals) and observed plasma drug concentrations for example 2 for an Effect Compartment Model

FIGURE 4.9. Predicted (posterior median and 90 % credible intervals) and observed PD Response for example 2

10. Friberg-Karlsson Semi-Mechanistic Population Model

We now return to the example in Section 5 and extend it to a population model. While we recommend using the mixed solver, for completeness we 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`.

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

$$\log(ANC_{ij}) \sim N(Circ_{ij}, \sigma^2_{ANC}),$$

$$\log\left(MTT_j, Circ_{0j}, \alpha_j\right) \sim N\left(\log\left(\widehat{MTT}, \widehat{Circ_0}, \widehat{\alpha}\right), \Omega_{ANC}\right),$$

$$\left(\widehat{MTT}, \widehat{Circ_0}, \widehat{\alpha}, \gamma\right) = (125, 5, 2, 0.17),$$

$$\Omega_{ANC} = \begin{pmatrix} 0.2^2 & 0 & 0 \\ 0 & 0.35^2 & 0 \\ 0 & 0 & 0.2^2 \end{pmatrix},$$

$$\sigma_{ANC} = 0.1,$$

$$\Omega_{PK} = \begin{pmatrix} 0.25^2 & 0 & a0 & 0 & 0 \\ 0 & 0.4^2 & 0 & 0 & 0 \\ 0 & 0 & 0.25^2 & 0 & 0 \\ 0 & 0 & 0 & 0.4^2 & 0 \\ 0 & 0 & 0 & 0 & 0.25^2 \end{pmatrix}$$

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 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 `genOdeModel_rk45`,

The two code snippets below show the definition of the ODEs system and the skeleton of the solution process in Stan's `transformed parameters` block.

```
functions{

    real[] twoCptNeutModelODE(real t,
                        real[] x,
                        real[] parms,
                        real[] rdummy,
                        int[] idummy){
                real CL = parms[1];
                real Q = parms[2];
                real V1 = parms[3];
                real V2 = parms[4];
                real ka = parms[5];
                real mtt = parms[6];
                real circ0 = parms[7];
                real gamma = parms[8];
                real alpha = parms[9];

    real k10 = CL / V1;
```

```
    real k12 = Q / V1;
    real k21 = Q / V2;
    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;
  }

}
```

```
transformed parameters{
  row_vector[nt] cHat;
  row_vector[nObsPK] cHatObs;
  row_vector[nt] neutHat;
  row_vector[nObsPD] neutHatObs;
  matrix[nCmt, nt] x;
  real<lower = 0> parms[nTheta]; // The [1] indicates the parameters are constant

  // variables for Matt's trick
  vector<lower = 0>[nIIV] thetaHat;
  matrix<lower = 0>[nSubjects, nIIV] thetaM;

  // Matt's trick to use unit scale
  thetaHat[1] = CLHat;
  thetaHat[2] = QHat;
  thetaHat[3] = V1Hat;
  thetaHat[4] = V2Hat;
  thetaHat[5] = mttHat;
  thetaHat[6] = circ0Hat;
  thetaHat[7] = alphaHat;
  thetaM = (rep_matrix(thetaHat, nSubjects) .*
            exp(diag_pre_multiply(omega, L * etaStd)))';

  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]] = pmx_solve_rk45(twoCptNeutModelODE, nCmt,
                                        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[2, start[i]:end[i]] / parms[3];  // divide by V1
    neutHat[start[i]:end[i]] = x[8, start[i]:end[i]] + parms[7];  // Add baseline

  }

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

}
```

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.

10.2. 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) (Figure 4.10. 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 model fits the data, and the credible interval reflect the noise in the data (Figure 4.12). The parameters estimation reflects the real value of the parameters (Table 4.3 and Figure 4.11).

	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	3.3 (-04)	1.4 (-06)	2.2 (-05)	2.9 (-04)	3.2 (-04)	3.3 (-04)	3.5 (-04)	3.8 (-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

TABLE 4.3. Summary of the MCMC simulations of the marginal posterior distributions of the model parameters for the Friberg-Karlsson model example.

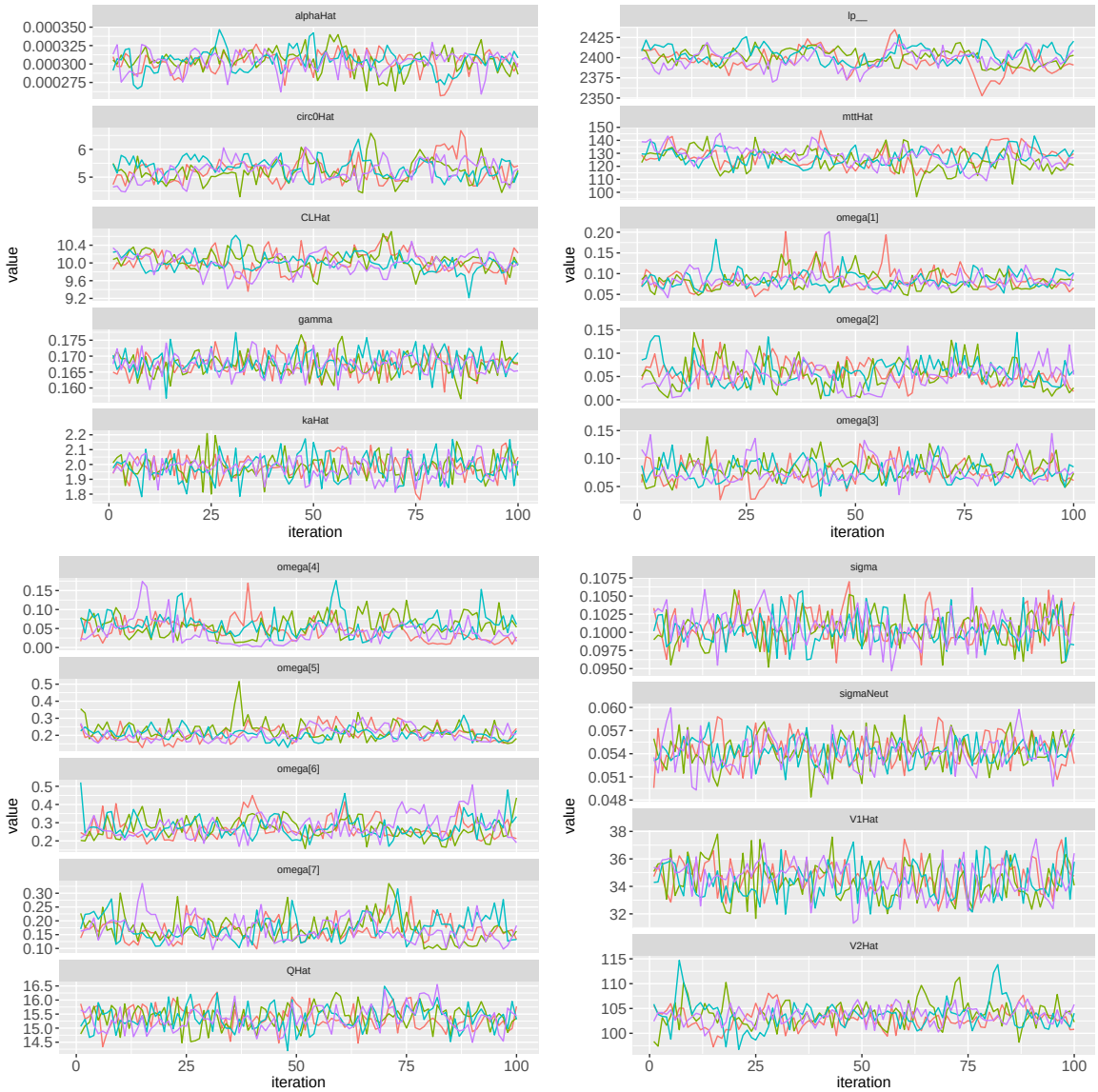

FIGURE 4.10. MCMC history plots for the parameters of a Friberg-Karlsson semi-mechanistic model (each color corresponds to a different chain) for example 3

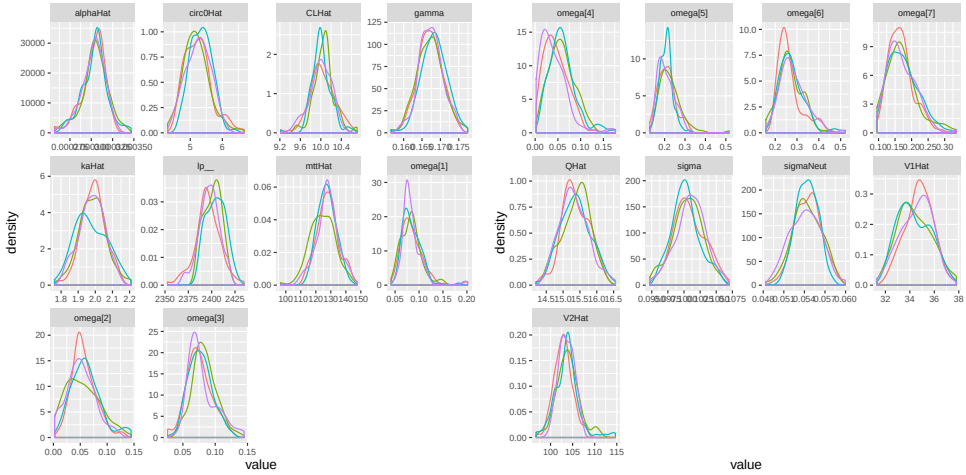

FIGURE 4.11. Posterior Marginal Densities of the Model Parameters of a Friberg-Karlsson semi-mechanistic model (each color corresponds to a different chain)

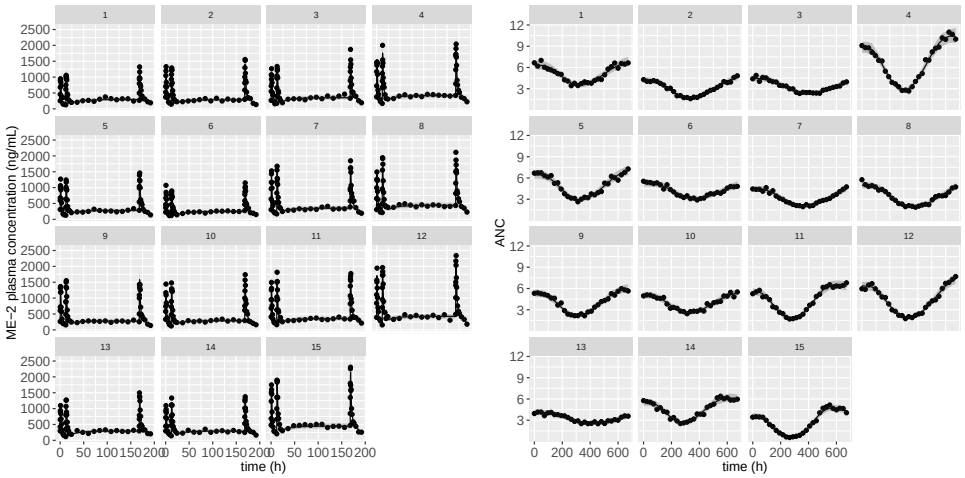

FIGURE 4.12. Predicted (posterior median and 90 % credible intervals) and observed plasma drug concentrations, and Neutrophil counts, for a Friberg-Karlsson semi-mechanistic model

Compiling constants

Several constants are used in Torsten's makefile. These constants can be used in `cmdstan/make/local` file, or use `set_make_local` command in `cmdstanr`.

- **`TORSTEN_MPI=1`** turns on within-chain parallelsation of MPI-enable functions. To use this option one must also point **`TBB_CXX_TYPE`** to proper C compiler. See also section 6.5 and 8.4.
- **`TORSTEN_CVS_JAC_AD=1`** makes BDF and Adams integrator use Stan's automatic differentiation to calculate Jacobian matrix in nonlinear solver [**6**].

Index

Bibliography

[1] Kyle T. Baron and Marc R. Gastonguay. Simulation from ode-based population pk/pd and systems pharmacology models in r with mrgsolve. *Journal of Pharmacokinetics and Pharmacodynamics*, 42(W-23):S84–S85, 2015.

[2] Michael Betancourt. A conceptual introduction to hamiltonian monte carlo. 2018. arXiv: 1701.02434.

[3] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A Probabilistic Programming Language. *Journal of Statistical software*, 76, 2017.

[4] Bob Carpenter, Matthew D. Hoffman, Marcus Brubaker, Daniel Lee, Peter Li, and Michael Betancourt. The Stan Math Library: Reverse-Mode Automatic Differentiation in C++. *arXiv:1509.07164 [cs]*, September 2015. arXiv: 1509.07164.

[5] Lena E. Friberg and Mats O. Karlsson. Mechanistic Models for Myelosuppression. *Investigational New Drugs*, 21(2):183–194, May 2003.

[6] Alan C. Hindmarsh, Radu Serban, Cody J. Balos, David J. Gardner, Carol S. Woodward, and Daniel R. Reynolds. *User Documentation for cvodes v5.4.0*, 2020.

[7] Matthew D. Hoffman and Andrew Gelman. The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. *arXiv:1111.4246 [cs, stat]*, November 2011. arXiv: 1111.4246.

[8] L. F. Shampine, I. Gladwell, Larry Shampine, and S. Thompson. *Solving ODEs with MATLAB*. Cambridge University Press, April 2003.