Flexible and efficient Bayesian pharmacometrics modeling using Stan and Torsten, Part I

Abstract

Stan is an open-source probabilistic programing language, primarily designed to do Bayesian data analysis. Its main inference algorithm is an adaptive Hamiltonian Monte Carlo sampler, supported by state of the art gradient computation. Stan's strengths include efficient computation, an expressive language which offers a great deal of flexibility, and numerous diagnostics that allow modelers to check whether the inference is reliable. Torsten extends Stan with a suite of functions that facilitate the specification of pharmacokinetic and pharmacodynamic models, and makes it straightforward to specify a clinical event schedule. Part I of this tutorial demonstrates how to build, fit, and criticize simple pharmacokinetic models using Stan and Torsten.

1 Introduction

Bayesian inference offers a principled approach to learn about unknown variables from data using a probabilistic analysis. The conclusions we draw are based on the posterior distribution which, in all but the simplest cases, is intractable. We can however probe the posterior using a host of techniques such as Markov chains Monte Carlo sampling and approximate Bayesian computation. Writing these algorithms is a tedious and error prone endeavor but fortunately modelers can often rely on existing software with efficient implementations.

In the field of pharmacometrics, statistical software such as NONMEM [**?**] and Monolix [**?**] support many routines to specify and analyze pharmacokinetic and pharmacodynamic population models. There also exist more general probabilistic languages such as, to only name a few, BUGS [**?**] and more recently Stan [6], which will be the focus of this tutorial. Stan supports a rich library of probability densities, mathematical functions including matrix operations and numerical solvers for differential equations. These features make for a rich and flexible language, however writing common pharmacometrics models can be tedious. Torsten extends Stan by providing a suite of functions to facilitate the specification of pharmacometrics models. These functions make it straightforward to model the event schedule of a clinical trial and parallelize computation across patients for population models.

1.1 Why Stan?

We believe that Stan, coupled with Torsten, can be an important addition to the pharmacometrician's toolkit, especially for Bayesian data analysis.

The most obvious strength of Stan is its flexibility: it is straightforward to specify priors, systems of ODEs, a broad range of measurement models, missing data models and hierarchies (i.e. population models). Because of this flexibility, various data sources and their corresponding measurement models can be combined into one large model, over which full Bayesian inference can be performed [e.g 27]. There are not many examples in pharmacometrics where the flexibility of Stan would be fully utilized, but we believe this is in part because such tools were not readily available to pharmacometricians in the past. The richness of the Stan language could well open the gate to new types of models.

Secondly, Stan supports state of the art inference algorithms, most notably its adaptive *Hamiltonian Monte Carlo* sampler, a gradient-based Markov chains Monte Carlo algorithm [3] based on the No U-Turn sampler (NUTS) [16], automatic differentiation variational inference (ADVI) [17], and penalized maximum likelihood estimators. Stan's inference algorithms are supported by a cutting edge automatic differentiation library, which efficiently generates the requisite derivatives [7]. It is worth pointing out that algorithms, such as NUTS and ADVI, were first developed and implemented in Stan, before being widely adopted by the applied statistics and machine learning communities. As of the writing of this article, new inference algorithms continue to be prototyped in Stan such as, to take a few recent examples, adjoint-differentiated Laplace approximations [20], cross-chain warmup [29], and path finding for improved chain initialization[1].

Thirdly, Stan runs many *many* diagnostics – including the detection of divergent transitions [3], and the improved computation of effective sample sizes and scale reduction factors, $\hat{R}$ [25], and more – and gives detailed warning messages. This makes it considerably easier to identify issues with our inference and our models. Several of these tools improve commonly used diagnostics which may not detect important problems, in which case our inference fails without us realizing it. Stan fails better: it fails loudly.

Last but not least, both Stan and Torsten are open-source projects, meaning they are free and their source code can be examined and, if needed, scrutinized. The projects are under active development with new features being added regularly.

1.2 Bayesian inference: notation, goals, and comments

Given observed data, $\mathcal{D}$, and latent variables, θ, a Bayesian model is defined by the joint distribution, $p(\mathcal{D}, \theta)$. The latent variables can include model parameters, missing data, and more. In this tutorial, we are mostly concerned with estimating model parameters.

The joint distribution observes a convenient decomposition,

$$p(\mathcal{D}, \theta) = p(\theta)p(\mathcal{D} \mid \theta),$$

with $p(\theta)$ the *prior* distribution and $p(\mathcal{D} \mid \theta)$ the *likelihood*. The prior encodes information about the parameters, usually based on scientific expertise or results from previous analysis. The likelihood tells us how the data is distributed for a fixed parameter value and, per one interpretation, can be thought of as a "story of

[1] Personal communication with the Stan development team.

how the data is generated" [12]. The Bayesian proposition is to base our inference on the *posterior* distribution, $p(\theta \mid \mathcal{D})$.

For typical pharmacometric applications, the full joint posterior density of the model parameters is an unfathomable object which lives in a high dimensional space. Usually we cannot even numerically evaluate the posterior density at any particular point! Instead we must probe the posterior distribution and learn the characteristics that interest us the most. In our experience, this often includes a measure of a central tendency and a quantification of uncertainty, for example the mean and the variance, or the median and the 5^{th} and 95^{th} quantiles. For skewed or multimodal distributions, we may want a more refined analysis which looks at many quantiles. What we compute are estimates of these quantities. Most Bayesian inference involves calculations based on marginal posterior distributions. That typically requires integration over a high number of dimensions—an integration that is rarely tractable by analytic or numerical quadrature. One strategy is to generate *approximate* samples from the posterior distribution and then use sample mean, sample variance, and sample quantiles as our estimators.

Bayes' rule teaches us that

$$p(\theta \mid \mathcal{D}) = \frac{p(\mathcal{D}, \theta)}{p(\mathcal{D})} = \frac{p(\mathcal{D} \mid \theta) p(\theta)}{p(\mathcal{D})}.$$

Typically we can evaluate the joint density in the nominator but not the normalizing constant, $p(\mathcal{D})$, in the denominator. A useful method must therefore be able to generate samples using the *unnormalized* posterior density, $p(\mathcal{D}, \theta)$. Many Markov chains Monte Carlo (MCMC) algorithms are designed to do exactly this. Starting at an initial point, these chains explore the parameter space, Θ, one iteration at a time, to produce the desired samples. Hamiltonian Monte Carlo (HMC) is an MCMC method which uses the gradient to efficiently move across the parameter space [22, 3]. Computationally, running HMC requires evaluating $\log p(\mathcal{D}, \theta)$ and $\nabla_\theta \log p(\mathcal{D}, \theta)$ many times across Θ, i.e. for varying values of θ but fixed values of $\mathcal{D}$. For this procedure to be well-defined, θ must be a continuous variable, else the requisite gradient does not exist. Discrete parameters require a special treatment, which we will not discuss in this tutorial.

A Stan program specifies a method to evaluate $\log p(\mathcal{D}, \theta)$. Thanks to automatic differentiation, this implicitly defines a procedure to compute $\nabla_\theta \log p(\mathcal{D}, \theta)$ [14, 1, 18]. Together, these two objects provide all the relevant information about our model to run HMC sampling and other gradient-based inference algorithms.

1.3 Bayesian workflow

Bayesian inference is only one step of a broader modeling process, which we might call the Bayesian workflow [3, 9, 13]. Once we fit the model, we need to check the inference and if needed, fine tune our algorithm, or potentially change method. And once we trust the inference, we naturally need to check the fitted model. Our goal is to understand the shortcomings of our model and motivate useful revisions. During the early stages of model development, this mostly comes down to troubleshooting our implementation and later this "criticism" step can lead to deeper insights.

All through the tutorial, we will demonstrate how Stan and Torsten can be used to check our inference and our fitted model.

1.4 Setting up Stan and Torsten

Detailed instructions on installing Stan and Torsten can be found on `https://github.com/metrumresearchgroup/Torsten`. At its core, Stan is a C++ library but it can be interfaced with one of many scripting languages, including R, Python, and Julia. We will use cmdStanR, which is a lightweight wrapper of Stan in R, and in addition, the packages Posterior [5] BayesPlot [8], and Loo [10].

The R and Stan code for all the examples are available at `link`.

1.5 Prerequisites and resources

Our aim is to provide a self-contained discussion, while trying to remain concise. The objectives of this tutorial are to describe basic model implementation and execution using Stan and Torsten and subsequent analysis of the results to generate MCMC diagnostics and limited model checking. Thus we illustrate key elements of a Bayesian workflow, but do not present complete Bayesian workflows for each example due to space limitations. We assume the reader is familiar with compartment models as they arise in pharmacokinetic and pharmacodynamic models, and has experience with data that describe a clinical event schedule. For the latter, we follow the convention established by NONMEM/NMTRAN/PREDPP. Exposure to Bayesian statistics and inference algorithms is desirable, in particular an elementary understanding of standard MCMC methods. We expect the reader to know R but we don't assume any background in Stan.

Helpful reads include the *Stan User Manual* [23] and the *Torsten User Manual* [28]. *Statistical Rethinking* by McElreath (2020)[21] provides an excellent tutorial on Bayesian analysis that may be used for self-learning. A comprehensive textbook on Bayesian modeling is *Bayesian Data Analysis* by Gelman et al (2013) [12], with more recent insights on the Bayesian workflow provided by Gelman et al (2020) [13]. Betancourt (2018) offers an accessible discussion on MCMC methods, with an emphasis on HMC [3].

2 Two compartment model

As a starting example, we demonstrate the analysis of longitudinal plasma drug concentration data from a single individual using a linear 2 compartment model with first order absorption. The individual receives multiple doses at regular time intervals and the plasma drug concentration is recorded over time. Our goal is to estimate the posterior distribution of the parameters of the model describing the time course of the plasma drug concentrations in this individual.

2.1 Pharmacokinetic model and clinical event schedule

Let's suppose an individual receives a 1,200 mg dose of a drug every 12 hours, until they have received a total of 14 doses. Drug concentrations are measured in plasma obtained from blood sampled at 0.083, 0.167, 0.25, 0.5, 0.75, 1, 1.5, 2,

Variable	Description
`cmt`	Compartment in which event occurs.
`evid`	Type of event: (0) measurement, (1) dosing.
`addl`	For dosing events, number of additional doses.
`ss`	Steady state indicator: (0) no, (1) yes.
`amt`	Amount of drug administered.
`time`	Time of the event.
`rate`	For dosing by infusion, rate of infusion.
`ii`	For events with multiple dosing, inter-dose interval.

Table 1: Available variables in Torsten to specify an event schedule.

3, 4, 6 and 8 hours following the first, second and last dose, at the time of all remaining doses. and at 12, 18 and 24 hours following the last dose. Let's analyze that data using a linear 2 compartment model with first order absorption, i.e.,

$$\begin{aligned}
\frac{\mathrm{d}y_{\text{gut}}}{\mathrm{d}t} &= -k_a y_{\text{gut}} \\
\frac{\mathrm{d}y_{\text{cent}}}{\mathrm{d}t} &= k_a y_{\text{gut}} - \left(\frac{CL}{V_{\text{cent}}} + \frac{Q}{V_{\text{cent}}}\right) y_{\text{cent}} + \frac{Q}{V_{\text{peri}}} y_{\text{peri}} \\
\frac{\mathrm{d}y_{\text{peri}}}{\mathrm{d}t} &= \frac{Q}{V_{\text{cent}}} y_{\text{cent}} - \frac{Q}{V_{\text{peri}}} y_{\text{peri}}
\end{aligned}$$

with

- $y(t)$: drug mass in each compartment (mg),
- k_a: absorption rate constant (h^{-1}),
- CL: elimination clearance from the central compartment (L / h),
- Q: intercompartmental clearance (L/h),
- V_{cent}: volume of the central compartment (L),
- V_{peri}: volume of the peripheral compartment (L).

Both intervention and measurement events are described by the event schedule, which follows the convention established by NONMEM. This means the user must provide for each event the following variables: `cmt`, `evid`, `addl`, `ss`, `amt`, `time`, `rate`, and `ii`. Table 1 provides an overview of the roll of each variable and more details can be found in the *Torsten User Manual*.

2.2 Statistical model

Given a treatment, x, and the physiological parameters, $\{k_a, Q, CL, V_{\text{cent}}, V_{\text{peri}}\}$, we compute the drug plasma concentration, $\hat{c}$, by solving the relevant ODEs. Our

measurements, y, are a perturbation of $\hat{c}$. This is to account for the fact that our model is not perfect and that our measurements have finite precision. We model this residual error using a lognormal distribution, that is

$$y \mid \hat{c}, \sigma \sim \text{logNormal}(\log \hat{c}, \sigma^2),$$

where σ is a scale parameter we wish to estimate. The deterministic computation of $\hat{c}$ along with the measurement model, defines our likelihood function $p(y \mid \theta, x)$, where $\theta = \{k_a, CL, Q, V_{\text{cent}}, V_{\text{peri}}, \sigma\}$.

It remains to define a prior distribution, $p(\theta)$. Our prior should allocate probability mass to every plausible parameter value and exclude patently absurd values. For example the volume of the central compartment is on the order of ten liters, but it cannot be the size of the Sun. In this simulated example, our priors for the individual parameters are based on population estimates from previous (hypothetical) studies.

$$\begin{aligned}
CL &\sim \text{logNormal}(\log(10), 0.25); \\
Q &\sim \text{logNormal}(\log(15), 0.5); \\
V_{\text{cent}} &\sim \text{logNormal}(\log(35), 0.25); \\
V_{\text{peri}} &\sim \text{logNormal}(\log(105), 0.5); \\
k_a &\sim \text{logNormal}(\log(2.5), 1); \\
\sigma &\sim \text{Half} - \text{Normal}(0, 1);
\end{aligned}$$

Suggestions for building priors can be found in references [9, 2] and at `https://github.com/stan-dev/stan/wiki/Prior-Choice-Recommendations`.

2.3 Specifying a model in Stan

We can now specify our statistical model using a Stan file, which is divided into coding blocks, each with a specific role. From R, we then run inference algorithms which take this Stan file as an input.

2.3.1 Data and parameters block

To define a model, we need a procedure which returns the log joint distribution, $\log p(\mathcal{D}, \theta)$. Our first task is to declare the data, $\mathcal{D}$, and the parameters, θ, using the coding blocks `data` and `parameters`. It is important to distinguish the two. The data is fixed. By contrast, the parameter values change as HMC explores the parameter space, and gradients of the joint density are computed with respect to θ, but not $\mathcal{D}$.

For each variable we introduce, we must declare a type and, for containers such as arrays, vectors, and matrices, the size of the container. In addition, each statement ends with a semi-colon. It is possible to specify constraints on the parameters, using the keywords `lower` and `upper`. If one of these constraints is violated, Stan returns an error message. More importantly, constrained parameters are transformed into unconstrained parameters – for instance, positive variables are put on the log scale – which greatly improves computation.

```
data {
  int<lower = 1> nEvent;
  int<lower = 1> nObs;
  int<lower = 1> iObs[nObs];  // index of events which
                              // are observations.

  // Event schedule
  int<lower = 1> cmt[nEvent];
  int evid[nEvent];
  int addl[nEvent];
  int ss[nEvent];
  real amt[nEvent];
  real time[nEvent];
  real rate[nEvent];
  real ii[nEvent];

  // observed drug concentration
  vector<lower = 0>[nObs] cObs;
}

parameters {
  real<lower = 0> CL;
  real<lower = 0> Q;
  real<lower = 0> VC;
  real<lower = 0> VP;
  real<lower = 0> ka;
  real<lower = 0> sigma;
}
```

2.3.2 model block

Next, the `model` block allows us to modify the variable `target`, which Stan recognizes as the log joint distribution. The following statement increments `target` using the prior on σ, which is a normal density, truncated at 0 to only put mass on positive values.

```
target += normal_lpdf(sigma | 0, 1);
```

The truncation is implied by the fact σ is declared as lower-bounded by 0 in the parameters block. An alternative syntax is the following:

```
sigma ~ normal(0, 1);
```

This statement now looks like our statistical formulation and makes the code more readable. But we should be mindful that this is not a sampling statement, rather instructions on how to increment `target`. We now give the full model block:

```
model {
  // priors
  CL ~ lognormal(log(10), 0.25);
```

```
  Q ~ lognormal(log(15), 0.5);
  VC ~ lognormal(log(35), 0.25);
  VP ~ lognormal(log(105), 0.5);
  ka ~ lognormal(log(2.5), 1);
  sigma ~ normal(0, 1);

  // likelihood
  cObs ~ lognormal(log(concentrationHat[iObs]), sigma);
}
```

The likelihood statement involves a crucial term we have not defined yet: `concentrationHat`. Additional variables can be created using the transformed data and transformed parameters blocks. We will take advantage of these to compute the drug concentration in the central compartment for each event. Note that for the likelihood, we only use the concentration during observation events, hence the indexing `[iObs]`.

2.3.3 Transformed data and transformed parameters block

In `transformed data`, we can construct variables which only depend on the data. For this model, we simply specify the number of compartments in our model (including the gut), `nCmt`, and the numbers of pharmacokinetic parameters, `nTheta`, two variables which will come in handy shortly.

```
  transformed data {
    int nCmt = 3;
    int nTheta = 5;
  }
```

Because the data is fixed, this operation is only computed once. By contrast, operations in the `transformed parameters` block need to be performed (and differentiated) for each new parameter values.

To compute `concentrationHat` we need to solve the relevant ODE within the clinical event schedule. Torsten provides a function which returns the drug mass in each compartment at each time point of the event schedule.

```
matrix<lower = 0>[nCmt, nEvent]
  mass = pmx_solve_twocpt(time, amt, rate, ii, evid,
                          cmt, addl, ss, theta);
```

The first eight arguments define the event schedule and the last argument, `theta`, is an array containing the pharmacokinetic parameters, and defined as follows:

```
real theta[nTheta] = {CL, Q, VC, VP, ka};
```

It is also possible to have `theta` change between events, and specify lag times and bioavailability fractions, although we will not take advantage of these features in the example at hand.

The Torsten function we have chosen to use solves the ODEs analytically. Other routines use a matrix exponential, a numerical solver, or a combination of analytical and numerical methods [19]. It now remains to compute the concentration in the central compartment at the relevant times. The full `transformed parameters` block is as follows:

```
transformed parameters {
  real theta[nTheta] = {CL, Q, VC, VP, ka};
  row_vector<lower = 0>[nEvent] concentrationHat;
  matrix<lower = 0>[nCmt, nEvent] mass;

  mass = pmx_solve_twocpt(time, amt, rate, ii, evid,
                          cmt, addl, ss, theta);

  // Extract mass in central compartment and divide
  // by central volume.
  concentrationHat = mass[2, ] ./ VC;
}
```

The Stan file contains all the coding blocks in the following order: `data`, `transformed data`, `parameters`, `transformed parameters`, `model`. The full Stan code can be found in the Supplementary Material.

2.4 Calling Stan from R

The package CmdStanR allows us to run a number of algorithms on a model defined in a Stan file. An excellent place to get started with the package is `https://mc-stan.org/cmdstanr/articles/cmdstanr.html`.

The first step is to "transpile" the file – call it `twocpt.stan` –, that is translate the file into C++ and then compile it.

```
mod <- cmdstan_model("twocpt.stan")
```

We can then run Stan's HMC sampler by passing in the requisite data and providing other tuning parameters. Here: (i) the number of Markov chains (which we run in parallel), (ii) the initial value for each chain, (iii) the number of warmup iterations, and (iv) the number of sampling iterations.

```
fit <- mod$sample(data = data, chains = n_chains,
                  parallel_chains = n_chains,
                  init = init,
                  iter_warmup = 500,
                  iter_sampling = 500)
```

There are several other arguments we can pass to the sampler and which we will take advantage of throughout the tutorial. For applications in pharmacometrics, we recommend specifying the initial starting points via the `init` argument, as the defaults may not be appropriate. In this tutorial, we draw the initial points from their priors by defining an appropriate R function.

The resulting `fit` object stores the samples generated by HMC from which can deduce the sample mean, sample variance, and sample quantiles of our posterior distribution. This information is readily accessible using `fit$summary()` and summarized in table 2. We could also extract the samples and perform any number of operations on them.

	mean	**median**	**sd**	**mad**	**q5**	**q95**	$\hat{\mathbf{R}}$	$\mathbf{ESS}_{\text{bulk}}$	$\mathbf{ESS}_{\text{tail}}$
CL	10.0	10.0	0.378	0.367	9.39	10.6	1.00	1580	1348
Q	19.8	19.5	4.00	4.01	13.8	26.8	1.00	985	1235
V_{cent}	41.2	40.8	9.71	9.96	25.6	57.7	1.00	732	1120
V_{peri}	124	123	18.0	18.0	97.1	155	1.00	1877	1279
k_a	1.73	1.67	0.523	0.522	1.01	2.68	1.00	762	1108
σ	0.224	0.222	0.0244	0.0232	0.187	0.269	1.01	1549	1083

Table 2: Summary of results when fitting a two compartment model. *The first columns return sample estimates of the posterior mean, median, standard deviation, median absolute deviation,* 5^{th} *and* 95^{th} *quantiles, based on our approximate samples. The next three columns return the* $\hat{R}$ *statistics and the effective sample size for bulk and tail estimates, and can be used to identify problems with our inference.*

2.5 Checking our inference

Unfortunately there is no guarantee that a particular algorithm will work across all the applications we will encounter. We can however make sure that certain necessary conditions do not break. Much of the MCMC literature focuses on estimating expectation values, and we will use these results to develop some intuition.

2.5.1 Central limit theorem

Many common MCMC concepts, such as effective sample size, are amiable to intuitive interpretations. But to really grasp their meaning and take advantage of them, we must examine them in the context of central limit theorems.

For any function f of our latent parameters θ, we define the posterior mean to be

$$\mathbb{E}f = \int_{\Theta} f(\theta)p(\theta \mid \mathcal{D})\mathrm{d}\theta,$$

a quantity also termed the *expectation value*. If we were able to generate exact independent samples,

$$\theta^{(1)}, \theta^{(2)}, ..., \theta^{(n)} \overset{\text{i.i.d}}{\sim} p(\theta \mid \mathcal{D}),$$

one sensible estimator would be the sample mean,

$$\hat{\mathbb{E}}f = \frac{1}{n}\sum_{i=1}^{n} f\left(\theta^{(i)}\right).$$

Then, provided the variance of f is finite, the *central limit theorem* teaches us that the sample mean converges, as the sample size increases, to a normal distribution. We will not dwell on the technical details of the theorem and go straight to its practical implication, namely:

$$\hat{\mathbb{E}}f \overset{\text{approx}}{\sim} \text{Normal}\left(\mathbb{E}f, \frac{\text{Var}f}{n}\right),$$

where the deviation from the approximating normal has order $\mathcal{O}(1/n^2)$. This means that even for a moderate sample size, the approximation tends to be very good. This is a powerful result for two reasons: first it tells us that our estimator is unbiased and more importantly that the expected squared error is driven by the variance of f divided by our sample size, n.

Unfortunately, estimates constructed with MCMC samples will, in general, neither be unbiased, nor will their variance decrease at rate n. For our estimators to be useful, we must therefore check that our samples are unbiased and then use a corrected central limit theorem.

2.5.2 Checking for bias with $\hat{R}$

MCMC samples are biased for several reasons. Perhaps the most obvious one is that Markov chains generate correlated samples, meaning any sample has some correlation with the initial point. If we run the algorithm for enough iterations, the correlation to the initial point becomes negligible and the chain "forgets" its starting point. But what constitutes enough iterations? It isn't hard to construct examples where removing the initial bias in any reasonable time is a hopeless endeavor.

To identify this bias, we run multiple Markov chains, each started at different points, and check that they all convergence to the same region of the parameter space. More precisely, we shouldn't be able to distinguish the Markov chains based on the samples alone. One way to check this is to compute the $\hat{R}$ statistics, for which we provide an intuitive definition:

$$\hat{R} \overset{\text{intuitively}}{=} \frac{\text{Between chain variance}}{\text{Within chain variance}}.$$

If the chains are mixing properly, then $\hat{R} \approx 1.0$, as is the case in table 2. Stan uses an improved $\hat{R}$ statistics described in a recent paper by **(author?)** [25]. We can also visually check that the chains are properly mixing using a trace plot (Figure 1).

If $\hat{R} \gg 1$ and, more generally, if the chains were not mixing, this would be cause for concern and an invitation to adjust our inference method. Even when $\hat{R} = 1$, we should entertain the possibility that all the chains suffer from the same bias. Stan offers additional diagnostics to identify sampling biases, notably by reporting *divergent transitions* of the HMC sampler, a topic we will discuss when we fit more sophisticated models.

2.5.3 Controlling the variance of our estimator

Let's assume that our samples are indeed unbiased. The expected error of our estimator is now determined by the variance. Under certain regularity conditions, our estimator follows an MCMC central limit theorem,

$$\hat{\mathbb{E}}f \overset{\text{approx}}{\sim} \text{Normal}\left(\mathbb{E}f, \frac{\text{Var}f}{n_{\text{eff}}}\right).$$

where the key difference is that $\text{Var}f$ is now divided by the *effective sample size*, n_{eff}, rather than the sample size. This change accounts for the fact our samples are

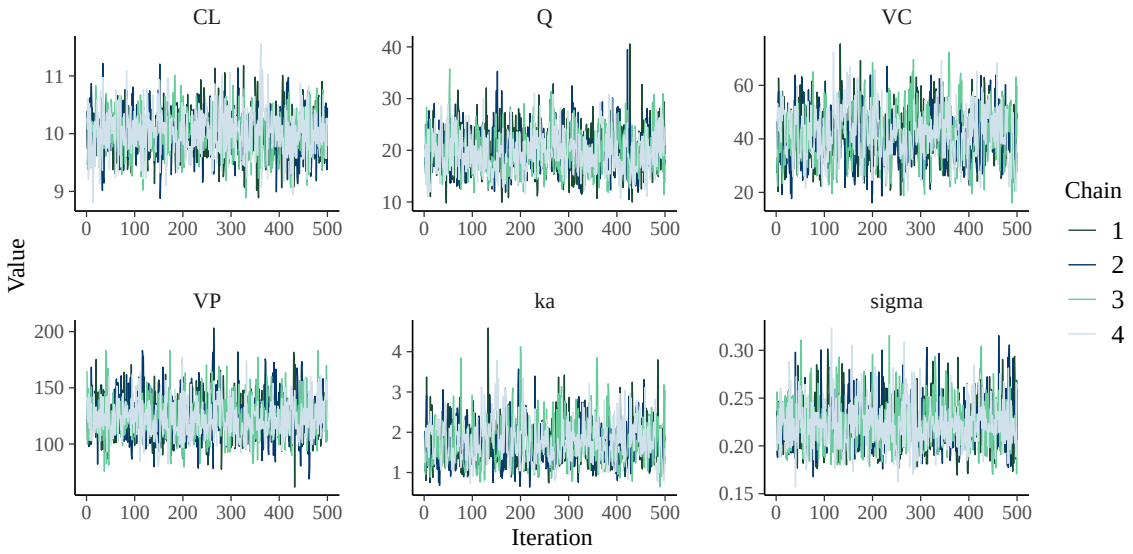


Figure 1: Trace plots. *The sampled values for each parameters are plotted against the iterations during the sampling phase. Multiple Markov chains were initialized at different points. However, once in the sampling phase, we cannot distinguish them.*

not independent: their correlation induces a loss in information, which increase the variance of our estimator. For CL, we have 2,000 samples, but the effective sample size is 1,580 (Table 2). If n_{eff} is low, our estimator may not be precise enough and we should generate more samples.

The effective sample size is only formally defined in the context of estimators for expectation values. We may also be interested in tail quantities, such as extreme quantiles, which are much more difficult to estimate and require many more samples to achieve a desired precision. **(author?)** [25] propose a generalization of the effective sample size for such quantities, and introduce the *tail effective sample size*. This is to be distinguished from the traditional effective sample size, henceforth the *bulk effective sample size*. Both quantities are reported by Stan.

2.6 Checking the model: posterior predictive checks

Once we develop enough confidence in our inference, we still want to check our fitted model. There are many ways of doing this. We may look at the posterior distribution of an interpretable parameter and see if it suggests implausible values. Or we may evaluate the model's ability to perform a certain task, e.g. classification or prediction, as is often done in machine learning. In practice, we find it useful to do *posterior predictive checks* (PPC), that is simulate data from the fitted model and compare the simulation to the observed data [11, chapter 6]. Mechanically, the procedure is straightforward:

1. Draw the parameters from their posterior, $\tilde{\theta} \sim p(\theta \mid y)$.

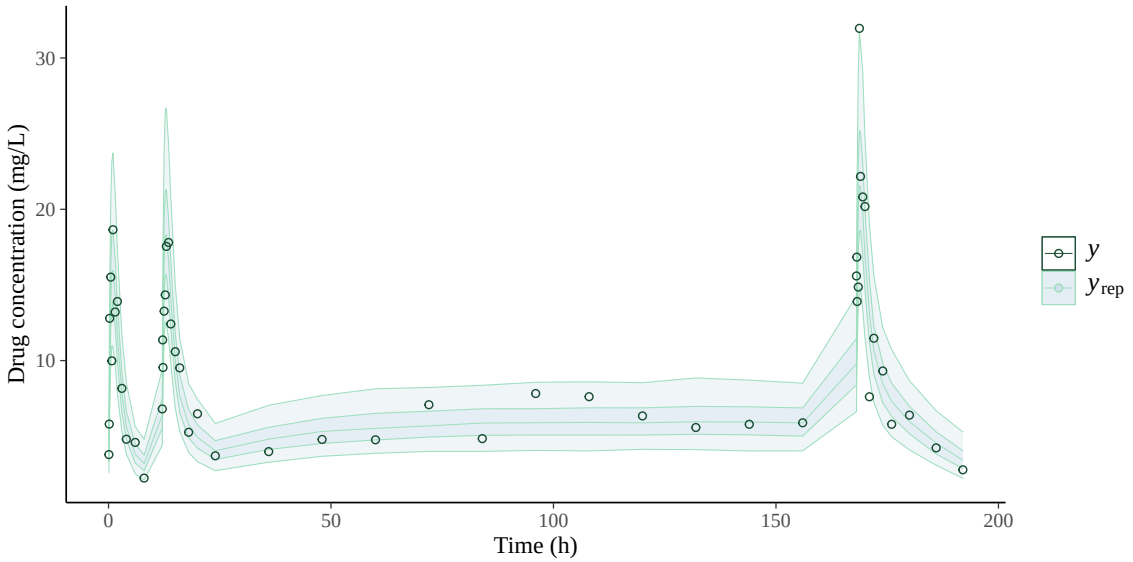


Figure 2: Posterior predictive checks for two compartment model. *The circles represent the observed data and the shaded areas the* 50^{th} *and* 90^{th} *credible intervals based on posterior draws.*

2. Draw new observations from the likelihood, conditional on the drawn parameters, $\tilde{y} \sim p(y \mid \tilde{\theta})$.

This amounts to drawing observations from their posterior distribution, that is $\tilde{y} \sim p(\tilde{y} \mid y)$. The uncertainty due to our estimation and the uncertainty due to our measurement error are then propagated to our predictions.

Stan provides a `generated quantities` block, which allows us to compute values, based on sampled parameters. In our two compartment model example, the following code draws new observations from the likelihood:

```
generated quantities {
  real concentrationObsPred[nObs]
    = lognormal_rng(log(concentrationHat[iObs]), sigma);
}
```

Here, we generated predictions at the observed points for each sampled point, $\theta^{(i)}$. This gives us a sample of predictions and we can use the 5^{th} and 95^{th} quantiles to construct a credible interval. We may then plot the observations and the credible intervals (Figure 2) and see that, indeed, the data generated by the model is consistent with the observations.

2.7 Comparing models: leave-one-out cross validation

Beyond model criticism, we may be interested in model comparison. Continuing our running example, we compare our two compartment model to a one compartment model, which is also supported by Torsten via the `pmx_solve_onecpt` routine. The corresponding posterior predictive checks are shown in Figure 3.

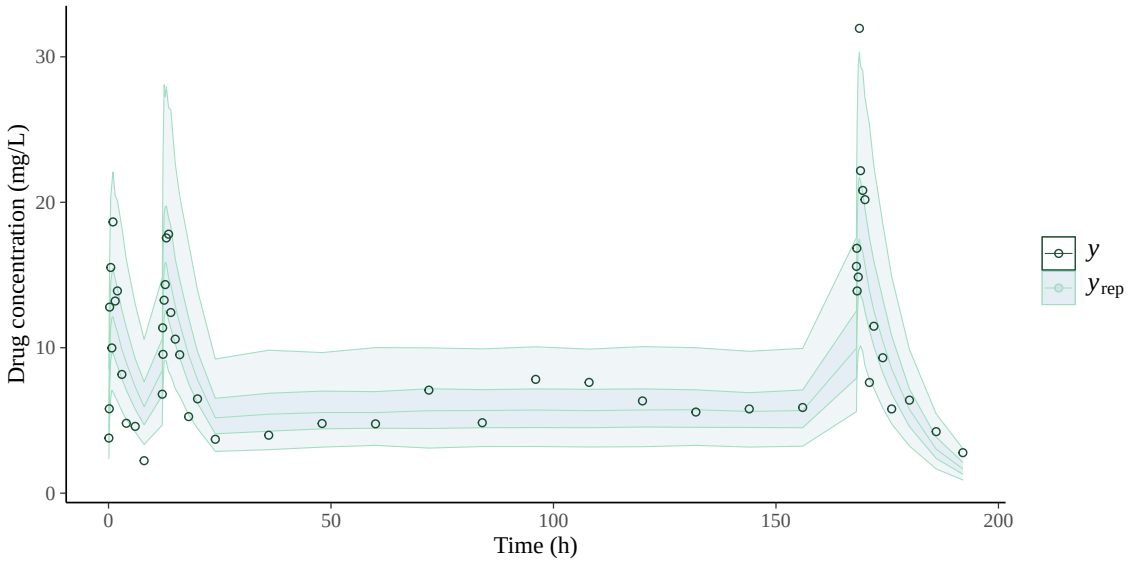


Figure 3: Posterior predictive checks for one compartment model. *The circles represent the observed data and the shaded areas the* 50^{th} *and* 90^{th} *credible intervals based on posterior draws. A graphical inspection suggests the credible interval is wider for the one compartment model than they are for the two compartment model.*

There are several ways of comparing models and which method is appropriate crucially depends on the insights we wish to gain. If our goal is to asses a model's ability to make good out-of-sample predictions, we may consider *Bayesian leave-one-out* (LOO) cross validation. The premise of cross-validation is to exclude a point, (y_i, x_i), from the *training set*, i.e. the set of data to which we fit the model. Here x_i denotes the covariate and in our example, the relevant row in the event schedule. We denote the reduced data set, y_{-i}. We then generate a prediction $(\tilde{y}_i, x_i)$ using the fitted model, and compare $\tilde{y}_i$ to y_i. A classic metric to make this comparison is the squared error, $(\tilde{y}_i - y_i)^2$.

Another approach is to use the *LOO estimate of out-of-sample predictive fit*:

$$\text{elp}_{\text{loo}} := \sum_i^n \log p(y_i \mid y_{-i}).$$

Here, no prediction is made. We however examine how consistent an "unobserved" data point is with our fitted model. Computing this estimator is expensive, since it requires fitting the model to n different training sets in order to evaluate each term in the sum.

Vehtari et al 2016 **(author?)** [26] propose an estimator of elp_{loo}, which uses Pareto smooth importance sampling and only requires a single model fit. The premise is to compute

$$\log p(y_i \mid y)$$

and correct this value, using importance sampling, to estimate $\log p(y_i \mid y_{-i})$. Naturally this estimator may be inaccurate. What makes this tool so useful is

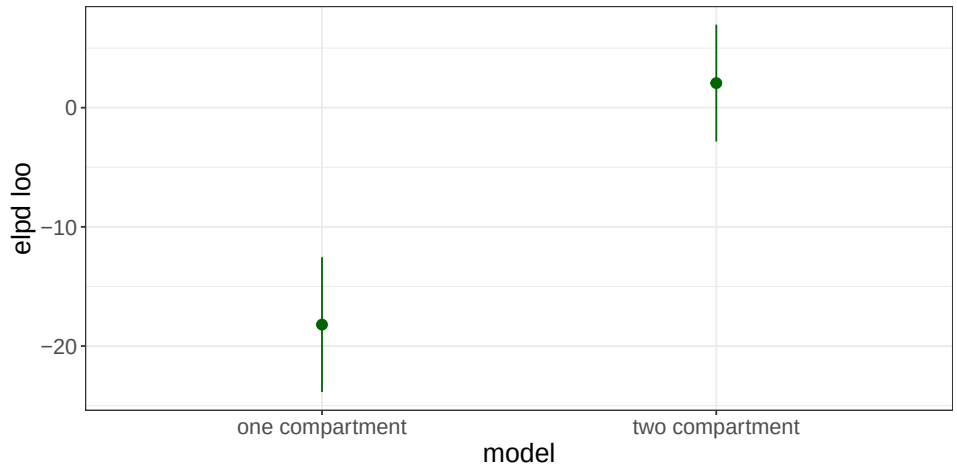


Figure 4: Leave-one-out estimate of out-of-sample predictive fit. *Plotted is the estimate,* elp_{loo}*, for the one and two compartment models. Clearly, the two compartment models has superior predictive capabilities.*

that we can use the Pareto shape parameter, $\hat{k}$, to asses how reliable the estimate is. In particular, if $\hat{k} > 0.7$, then the estimate shouldn't be trusted. The estimator is implemented in the R package Loo [10].

Conveniently, we can compute $\log p(y_i \mid y)$ in Stan's `generated quantities` block.

```
vector[nObs] log_lik;
for (i in 1:nObs)
  log_lik[i] =
   lognormal_lpdf(cObs[i] | log(concentrationHat[iObs]),
                  sigma);
```

These results can then be extracted and fed into Loo to compute elp_{loo}. The file `twoCpt.r` in the Supplementary Material shows exactly how to do this. Figure 4 plots the estimated elp_{loo}, along with a standard deviation, and shows the two compartment model has better out-of-sample predictive capabilities.

3 Two compartment population model

We now consider the scenario where we have data from multiple patients and fit a population model. Population models are a powerful tool to capture the heterogeneity between patients, while also recognizing similarities. Building the right prior allows us to pool information between patients, the idea being that what we learn from one patient teaches us something – though not everything – about the other patients. In practice, such models can frustrate inference algorithms and need to be implemented with care [e.g. 4]. We start with an example where the interaction between the model and our MCMC sampler is well behaved. In Part II of this tutorial, we will examine a more difficult case, for which we will leverage Stan's diagnostic capabilities in order to run reliable inference.

3.1 Statistical model

Let ϑ be the 2D array of body weight-normalized pharmacokinetic parameters for each patient, with

$$\vartheta_j = (CL_{\text{norm},j}, Q_{\text{norm},j}, ka_{\text{norm},j}, V_{\text{cent,norm},j}, V_{\text{peri,norm},j}, k_{a,\text{norm},j}),$$

the parameters for the j^{th} patient. We construct a population model by introducing random variation to describe otherwise unexplained inter-individual variability. In a Bayesian context this is sometimes referred to as a prior distribution for the individual parameters.

$$\log \vartheta_j \sim \text{Normal}(\log \vartheta_{\text{pop}}, \Omega),$$

As before we work on the log scale to account for the fact the pharmacokinetic parameters are constrained to be positive. $\vartheta_{\text{pop}} = (CL_{\text{pop}}, Q_{\text{pop}}, V_{\text{cent,pop}}, V_{\text{peri,pop}}, k_{\text{a,pop}})$ is the population mean and Ω the population covariance matrix. Both ϑ_{pop} and Ω are estimated. In this example, we start with the simple case where Ω is diagonal. For our example we will also use conventional allometric scaling to adjust the clearance and volume parameters for body weight.

$$\begin{aligned}
CL_j &= CL_{\text{norm},j} \frac{\text{weight}}{70}^{.75} \\
Q_j &= Q_{\text{norm},j} \frac{\text{weight}}{70}^{.75} \\
V_{\text{cent},j} &= V_{\text{cent,norm},j} \frac{\text{weight}}{70} \\
V_{\text{peri},j} &= V_{\text{peri,norm},j} \frac{\text{weight}}{70}
\end{aligned}$$

The likelihood remains mostly unchanged, with the caveat that it must now be computed for each patient. Putting this all together, we have the following model, as specified by the joint distribution,

$$\begin{aligned}
\vartheta_{\text{pop}} &\sim p(\vartheta_{\text{pop}}), && \text{(prior on pharmacokinetic parameters)} \\
\Omega &\sim p(\Omega), && \text{(prior on population covariance)} \\
\sigma &\sim p(\sigma) \\
\vartheta \mid \vartheta_{\text{pop}}, \Omega &\sim \text{logNormal}(\vartheta_{\text{pop}}, \Omega), \\
\log y \mid c, \sigma &\sim \text{Normal}(\log c, \sigma).
\end{aligned}$$

3.2 Specifying the model in Stan

We begin by adjusting our parameters block:

```
parameters {
  // Population parameters
  real<lower = 0> CL_pop;
  real<lower = 0> Q_pop;
  real<lower = 0> VC_pop;
  real<lower = 0> VP_pop;
```

```
  real<lower = 0> ka_pop;

  // Inter-individual variability
  vector<lower = 0>[5] omega;
  real<lower = 0> theta[nSubjects, 5];

  // measurement error
  real<lower = 0> sigma;
}
```

The variable, ϑ_{pop} is introduced in `transformed parameters`, mostly for convenience purposes:

```
vector<lower = 0>[nTheta]
  theta_pop = to_vector({CL_pop, Q_pop, VC_pop, VP_pop,
                         ka_pop});
```

The model block reflects our statistical formulation:

```
model {
  // prior on population parameters
  CL_pop ~ lognormal(log(10), 0.25);
  Q_pop ~ lognormal(log(15), 0.5);
  VC_pop ~ lognormal(log(35), 0.25);
  VP_pop ~ lognormal(log(105), 0.5);
  ka_pop ~ lognormal(log(2.5), 1);
  omega ~ lognormal(0.25, 0.1);

  sigma ~ normal(0, 1);

  // hierarchical prior
  for (j in 1:nSubjects)
    theta[j, ] ~ lognormal(log(theta_pop), omega);

  // likelihood
  logCObs ~ normal(log(concentrationObs), sigma);
}
```

It remains to compute `concentrationObs`. There are several ways to do this and, depending on the computational resources available, we may either compute the concentration for each patients sequentially or in parallel. For now, we do the simpler sequential approach. In the upcoming Part II of this tutorial, we examine how Torsten offers easy-to-use parallelization for population models.

Sequentially computing the concentration is a simple matter of bookkeeping. In `transformed parameters` we loop through the patients using a `for` loop. The code is identical to what we used in Section 2.3.3, with the caveat that the arguments to `pmx_solve_twocpt` are now indexed to indicate for which patient we compute the drug mass. For example, assuming the time schedule is ordered by patient, the event times corresponding to the j^{th} patient are given by

```
time[start[j]:end[j]]
```

where `start[j]` and `end[j]` contain the indices of the first and last event for the j^{th} patient, and the syntax for indexing is as in R. The full `for` loop is then

```
for (j in 1:nSubjects) {
  mass[, start[j]:end[j]] =
    pmx_solve_twocpt(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]],
                     theta[j, ]);

  concentration[start[j]:end[j]] =
         mass[2, start[j]:end[j]] / theta[j, 3];
  }
```

Once we have written our Stan model, we can apply the same methods for inference and diagnostics as we did in the previous section.

3.3 Posterior predictive checks

We follow the exact same procedure as in Section 2.6 – using even the same line of code – to create new observations for our patients. Figure 5 plots the results across patients. In addition, we simulate data for new patients by: (i) drawing pharmacokinetic parameters from our population distribution, (ii) solving the ODEs with these simulated parameters and (iii) using our measurement model to simulate new observations. The generated quantities block then looks as follows:

```
generated quantities {
  // Posterior predictive checks for existing patients
  real concentrationObsPred[nObs]
    = exp(normal_rng(log(concentrationObs), sigma));

  // Posterior predictive checks for new patients
  // (here we assume they receive the same treatment
  // as the observed patients)
  real cObsNewPred[nObs];
  matrix<lower = 0>[nCmt, nEvent] massNew;
  real thetaNew[nSubjects, nTheta];
  row_vector<lower = 0>[nEvent] concentrationNew;
  row_vector<lower = 0>[nObs] concentrationObsNew;

  for (j in 1:nSubjects) {
    // (i) simulate pharmacokinetic parameters
    thetaNew[j, ] = lognormal_rng(log(theta_pop),omega);
```

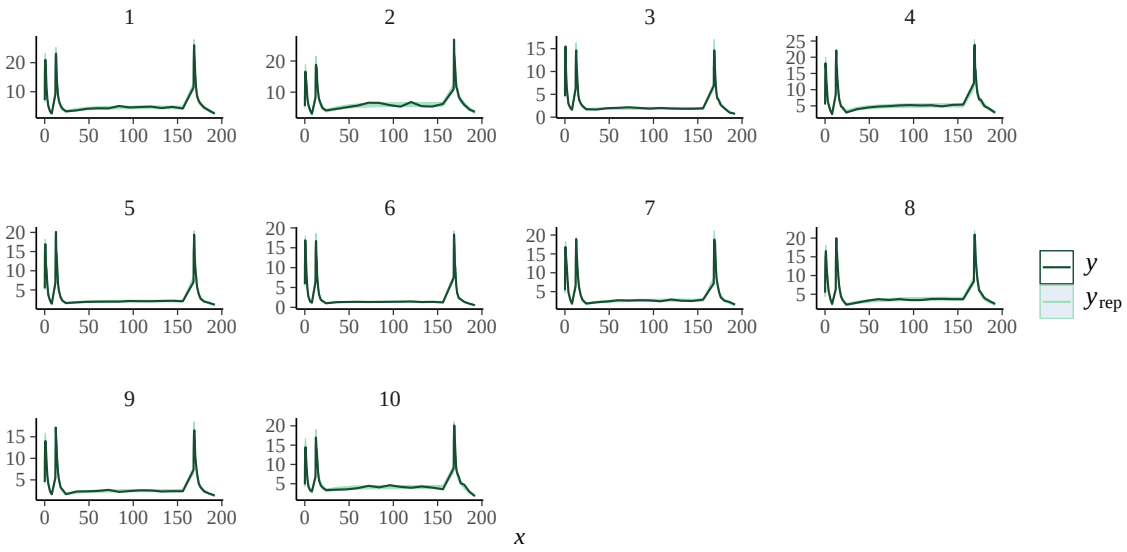


Figure 5: Posterior predictive checks for poulation two compartment model.

```
    // (ii) solve ODEs and compute drug mass
    massNew[, start[j]:end[j]]
      = pmx_solve_twocpt(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]],
                         thetaNew[j, ]);

      concentrationNew[start[j]:end[j]]
        = massNew[2, start[j]:end[j]] / thetaNew[j, 3];

      concentrationObsNew = concentrationNew[iObs];
  }

  // (iii) simulate measurement error
  cObsNewPred = exp(normal_rng(log(concentrationObsNew),
    sigma));
}
```

It is worth noting that the computational cost of running operations in the `generated quantities` is relatively small. While these operations are executed once per iteration, in order to generate posterior samples of the generated quantities, operations in the `transformed parameters` and `model` blocks are run and differentiate multiple times per iterations, meaning they amply dominate the com-

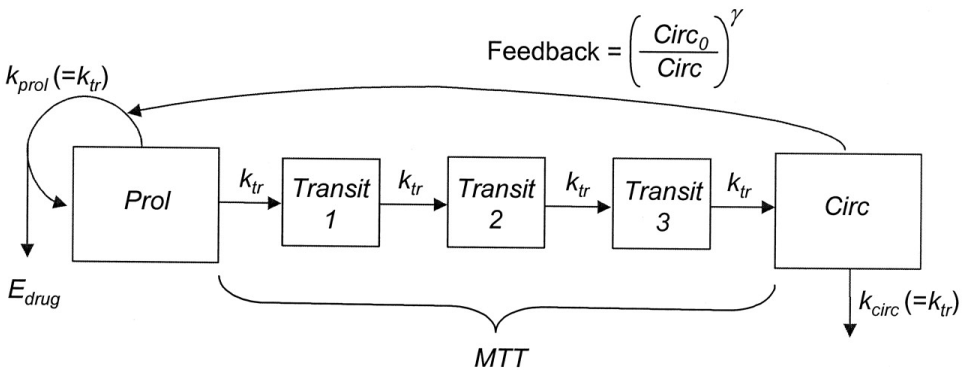


Figure 6: Friberg-Karlsson semi-mechanistic Model.

putation. Hence the cost of doing posterior predictive checks, even when it involves solving ODEs, is marginal. The computational scaling of Stan, notably for ODE-based models, is discussed in the article by **(author?)** [15].

4 Nonlinear pharmacokinetic / pharmacodynamic model

As a starting example, we consider a compartment pharmacokinetic model for a single patient. The patient receives multiple doses at regular time intervals and the drug plasma concentration is recorded over time. Our goal is to infer the physiological parameters of the patient, pertinent to the drug's pharmacokinetics, and the measurement error.

4.1 Nonlinear pharmacokinetic / pharmacodynamic model

For the last example, let us go back to the single patient two-compartment model and append it with a PD model. Specifically, we examine the Friberg-Karlsson semi-mechanistic model for drug-induced myelosuppression [27, 28, 29, 30, 31, 14] with the goal to model the relation between neutrophil counts and drug exposure. It describes a delayed feedback mechanism that keeps the absolute neutrophil count (ANC) at the baseline (Circ_0) in a circulatory compartment (y_{circ}), as well as the drug effect that perturbs this meachanism. The delay between proliferative cells (y_{prol}) and y_{circ} is modeled by three transit compartments with mean transit time

$$\text{MTT} = (3+1)/k_{\text{tr}} \tag{2}$$

where k_{tr} is the transit rate constant.

The PD model can be summarized as

$$\begin{aligned} \log(\text{ANC}) &\sim \text{Normal}(\log(y_{\text{circ}}), \sigma^2_{\text{ANC}}), \\ y_{\text{circ}} &= f_{\text{FK}}(\text{MTT}, \text{Circ}_0, \alpha, \gamma, c), \end{aligned}$$

where $c = y_{\text{cent}}/V_{\text{cent}}$ is the drug concentration calculated from the PK model,

and function f_{FK} represents solving the following nonlinear ODE for y_{circ}

$$\frac{dy_{\text{prol}}}{dt} = k_{\text{prol}} y_{\text{prol}} (1 - E_{\text{drug}}) \left(\frac{\text{Circ}_0}{y_{\text{circ}}} \right)^{\gamma} - k_{\text{tr}} y_{\text{prol}}, \tag{3a}$$

$$\frac{dy_{\text{trans1}}}{dt} = k_{\text{tr}} y_{\text{prol}} - k_{\text{tr}} y_{\text{trans1}}, \tag{3b}$$

$$\frac{dy_{\text{trans2}}}{dt} = k_{\text{tr}} y_{\text{trans1}} - k_{\text{tr}} y_{\text{trans2}}, \tag{3c}$$

$$\frac{dy_{\text{trans3}}}{dt} = k_{\text{tr}} y_{\text{trans2}} - k_{\text{tr}} y_{\text{trans3}}, \tag{3d}$$

$$\frac{dy_{\text{circ}}}{dt} = k_{\text{tr}} y_{\text{trans3}} - k_{\text{tr}} y_{\text{circ}}, \tag{3e}$$

We use $E_{\text{drug}} = \alpha c$ to model the linear effect of drug concentration in central compartment that reduces the proliferation rate or induces cell loss. The entire ODE system is formed by coupling equation (1) and (3). The following `parameters` block summarizes the unknown parameters this model

```
parameters{
  real<lower = 0> CL;
  real<lower = 0> Q;
  real<lower = 0> V1;
  real<lower = 0> V2;
  //  real<lower = 0> ka; // ka unconstrained
  real<lower = (CL / V1 + Q / V1 + Q / V2 +
                sqrt((CL / V1 + Q / V1 + Q / V2)^2 -
                     4 * CL / V1 * Q / V2)) / 2> ka; //
   ka > lambda_1
  real<lower = 0> mtt;
  real<lower = 0> circ0;
  real<lower = 0> alpha;
  real<lower = 0> gamma;
  real<lower = 0> sigma;
  real<lower = 0> sigmaNeut;
}
```

Unlike in Section 2.1, here to solve the nonlinear system we must utilize one of the numerical solvers in Torsten.

4.1.1 Numerical solution of ODE

To solve an ODE numerically in Stan we first need to define its right-hand-side in the `functions` block, a block in which we put all the user-supplied functions.

```
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 = x[2]/V1;
    real EDrug = fmin(1.0, alpha * conc);
    real prol = x[4] + circ0;
    real transit1 = x[5] + circ0;
    real transit2 = x[6] + circ0;
    real transit3 = x[7] + circ0;
    real circ = fmax(machine_precision(), x[8] + circ0);

    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];

    // x[4], x[5], x[6], x[7] and x[8] are differences
   from circ0.
    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;
  }
}
```

One can see that the above function is almost literal translation of Eq. (1) and (3), in that the first three components of `dydt` describes the PK while the rest the PD.

We omit most items in the `data` block but bring reader's attention to a few items. It often helps to keep an index array that points to the observations in the ODE solutions, such as

```
  vector<lower = 0>[nObsPK] cObs;
  vector<lower = 0>[nObsPD] neutObs;
```

which are the indices to the the drug concentration and neutrophils count observations, respectively. Despite Stan/Torsten provides default values, we highly

recommend user define the ODE solver control parameters in the `data` block

```
  real<lower = 0> rtol;
  real<lower = 0> atol;
  int<lower = 0> max_num_step;
```

and judiciously choose these control parameters

- `rtol`: relative tolerance to determine solution convergence,
- `atol`: absolute tolerance to determine solution convergence,
- `max_num_step`: maximum allowed steps.

In particular, 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. For example, 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. For more details, see [24, Chapter 13] and [28, Section 3.7.5] and reference therein.

In `transformed data` block, we define fixed event specification arguments such as `rate`, `ii`, etc, that are not effective in this model. In fact here we also provide bioavailabity fraction `F` and dosing lag time `tLag`, despite the fact that they are defined with default values therefore can be omitted.

```
transformed data{
  real<lower = 0> rate[nt] = rep_array(0.0, nt);
  real<lower = 0> ii[nt] = rep_array(0.0, nt);
  int<lower = 0> addl[nt] = rep_array(0, nt);
  int<lower = 0> ss[nt] = rep_array(0, nt);
  vector[nObsPK] logCObs = log(cObs);
  vector[nObsPD] logNeutObs = log(neutObs);
  int<lower = 1> nCmt = 8;
  real F[nCmt] = rep_array(1.0, nCmt);
  real tLag[nCmt] = rep_array(0.0, nCmt);
}
```

Now we are ready to solve the ODEs. Similar to Stan, Torsten provides a few numerical solvers and in this example we use the Runge-Kutta solver `pmx_solve_rk45`[28, Section 3.4]. This is done in the `transformed parameters` block.

```
transformed parameters{
  vector[nt] cHat;
  vector[nObsPK] cHatObs;
  vector[nt] neutHat;
  vector[nObsPD] neutHatObs;
  matrix[nCmt, nt] x;
  real<lower = 0> parms[9];

  parms = {CL, Q, V1, V2, ka, mtt, circ0, gamma, alpha};
```

```
  x = pmx_solve_rk45(twoCptNeutModelODE, nCmt, time, amt
   , rate, ii, evid, cmt, addl, ss, parms, F, tLag, rtol
   , atol, max_num_step);

  cHat = x[2, ]' / V1;
  neutHat = x[8, ]' + circ0;

  cHatObs = cHat[iObsPK]; // predictions for observed
   data records
  neutHatObs = neutHat[iObsPD]; // predictions for
   observed data records
}
```

One can see that `pmx_solve_rk45` requires a user-defined ODE function as the first argument (similar to how one solve an ODE using Stan function `ode_rk45`). In Stan functions like this are refered as *high–order functions* [23, Chapter 9].

4.1.2 Solve PKPD model as coupled ODE system

The approach in the last section applies to all the models that involve ODE solutions, but we will not use it here. An acute observer may have noticed the PKPD model here exhibits a particular *one-way coupling* structure. That is, the PK (Eq. (1)) and PD (Eq. (3)) are coupled through the proliferation cell count y_{prol} and E_{drug} and the PK can be solved independently from the PD. This is what motivates Torsten's coupled solvers, which analytically solves PK before passing the PK solution to the PD and seeks its numerical solution. Since the dimension of the numerical ODE solution is reduced, in general this coupled strategy is more efficient than the last section's approach of numerically solving a full ODE system. To see it in action, let us apply the coupled solver `pmx_solve_twocpt_rk45` [28, Section 3.5] to the same model. We need only make two changes. First, we modify the ODE function to reflect that only the PD states are to be solved.

```
functions{
  real[] twoCptNeutModelODE(real t, real[] y, real[]
   y_pk, real[] theta, real[] rdummy, int[] idummy){
    /* PK variables */
    real V1 = theta[3];

    /* PD variable */
    real mtt      = theta[6];
    real circ0    = theta[7];
    real gamma    = theta[8];
    real alpha    = 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] / V1;
    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;
  }
}
```

Note that here we pass in PK and PD states as separate arguments y and y_{PK}, and the function describes the ODE for y, while y_{PK} will be solved internally using analytical solution, so user do not need to explicitly call `pmx_solve_twocpt`.

Then we can simply replace `pmx_solve_rk45` with `pmx_solve_twocpt_rk45` call.

```
  x = pmx_solve_twocpt_rk45(twoCptNeutModelODE, nOde,
   time, amt, rate, ii, evid, cmt, addl, ss, parms,
   biovar, tlag, rtol, atol, max_num_step);
```

The `model` block is similar to that in Section 2.1.

```
model{
  CL ~ lognormal(log(CLPrior), CLPriorCV);
  Q ~ lognormal(log(QPrior), QPriorCV);
  V1 ~ lognormal(log(V1Prior), V1PriorCV);
  V2 ~ lognormal(log(V2Prior), V2PriorCV);
  ka ~ lognormal(log(kaPrior), kaPriorCV);
  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);

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

4.1.3 Posterior predictive checks

We hope by now reader has developed the habit of performing PPC on every model. Since we have both PK (drug concentration) and PD (neutrophil count) observations, the PPC should be conducted on both.

```
generated quantities{
  real cObsPred[nt];
  real neutObsPred[nt];

  for(i in 1:nt){
    if(time[i] == 0){
      cObsPred[i] = 0;
    }else{
      cObsPred[i] = exp(normal_rng(log(fmax(
   machine_precision(), cHat[i])), sigma));
    }
    neutObsPred[i] = exp(normal_rng(log(fmax(
   machine_precision(), neutHat[i])), sigmaNeut));
  }
}
```

To add flexibility, `cmdstanr` packages provides a function `generate_quantities` so we can actually run the above code *after* the fitting run is completed. With Stan model variable `mod` that contains the above `generated quantities` block and fitting results `fit`, as in Section 2.4, we can do

```
fit.gq <- mod$generate_quantities(fit)
```

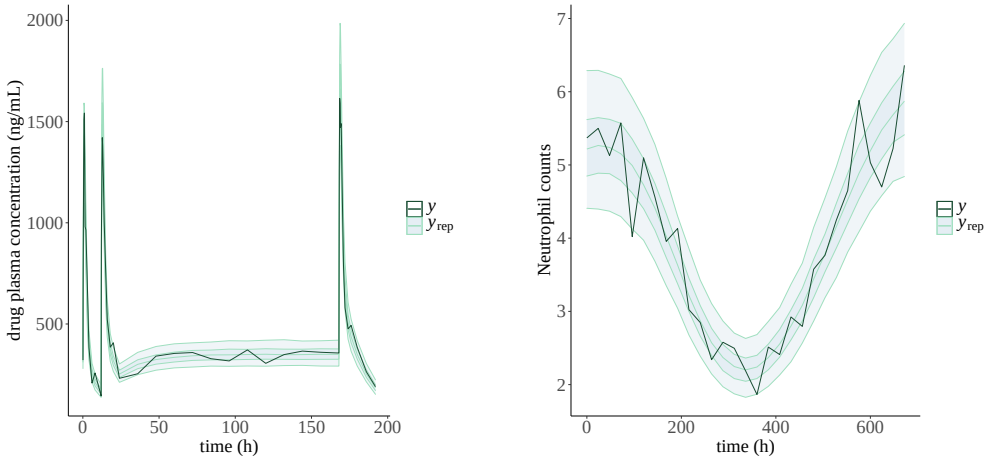
and use the results for PPC. Even better, for such a routine operation in Bayesian data analysis, package `bayesplot` provides numerous plotting functions.

```
c.rep <- fit.gq$draws(variables = c("cObsPred")) %>% as_
   draws_df() %>% as.matrix()
c.rep <- c.rep[ , data$iObsPK]  # use iObsPK from data
   to extract PK predictions corresponding to
   observation time

neut.rep <- fit.gq$draws(variables = c("neutObsPred"))
   %>% as_draws_df() %>% as.matrix()
neut.rep <- neut.rep[ , data$iObsPD]  # use iObsPD from
   data to extract PD predictions corresponding to
   observation time

c.obs <- data$cObs  # PK observations
neut.obs <- data$neutObs # PD observations

# PPC(PK): drug concentration
ppc.pk <- bayesplot::ppc_ribbon(y = c.obs, yrep = c.rep,
    x = data$time[data$iObsPK]) +
    scale_x_continuous(name="time␣(h)") +
```



(a) PK: drug concentration

(b) PD: neutrophil count

Figure 7: Posterior predictive checks for the PKPD model

```
    scale_y_continuous(name="drug␣plasma␣concentration␣(
   ng/mL)") + theme(axis.title=element_text(size=20),
   axis.text=element_text(size=20),legend.text=element_
   text(size=20))

# PPC(PD): neutrophil count
ppc.pd <- bayesplot::ppc_ribbon(y = neut.obs, yrep =
   neut.rep, x = data$time[data$iObsPD]) +
    scale_x_continuous(name="time␣(h)") +
    scale_y_continuous(name="Neutrophil␣counts") + theme
   (axis.title=element_text(size=20),axis.text=element_
   text(size=20),legend.text=element_text(size=20))
```

The code generates Figure 7a and 7b.

5 Conclusion

References

[1] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18:1 – 43, 2018.

[2] Michael Betancourt. Towards a principled bayesian workflow. `https://betanalpha.github.io/assets/case_studies/principled_bayesian_workflow.html`.

[3] Michael Betancourt. A conceptual introduction to Hamiltonian Monte Carlo. *arXiv:1701.02434v1*, 2018.

[4] Michael Betancourt and Mark Girolami. Hamiltonian Monte Carlo for hierarchical models. *arXiv:1312.0906v1*, 2013.

[5] Paul-Christian Bürkner, Jonah Gabry, M. Kay, and Vehtari Aki. Posterior: Tools for working with posterior distributions.

[6] Bob Carpenter, Andrew Gelman, Matt Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus A. Brubaker, Jiqiang Guo, Peter Li, and Allen Riddel. Stan: A probabilistic programming language. *Journal of Statistical Software*, 2017.

[7] Bob Carpenter, Matthew D. Hoffman, Marcus A. Brubaker, Daniel Lee, Peter Li, and Michael J. Betancourt. The Stan math library: Reverse-mode automatic differentiation in C++. *arXiv 1509.07164.*, 2015.

[8] Jonah Gabry and Tristan Mahr. bayesplot: Plotting for bayesian models, 2021. R package version 1.8.0.

[9] Jonah Gabry, Daniel Simpson, Aki Vehtari, Michael Betancourt, and Andrew Gelman. Visualization in bayesian workflow. *Journal of the Royal Statistical Society: Series A (Statistics in Society)*, 182(2):389–402, 2019.

[10] Jonah Gabry, Aki Vehtari, Mans Magnusson, Yuling Yao, Paul-Christian Bürkner, Topi Paananen, Andrew Gelman, Ben Goodrich, Juho Piironen, and Bruno Nocenboim. Efficient leave-one-out cross-validation and waic for bayesian models, 2020.

[11] Andrew Gelman, John B Carlin, Hal S Stern, David B Dunson, Aki Vehtari, and Donald B Rubin. *Bayesian Data Analysis*. Chapman and Hall/CRC, 2013.

[12] Andrew Gelman and Cosma Rohilla Shalizi. Philosophy and the practice of bayesian analysis. *British Journal of Mathematical and Statistical Psychology*, 66, 2013.

[13] Andrew Gelman, Aki Vehtari, Daniel Simpson, Charles C Margossian, Bob Carpenter, Yuling Yao, Lauren Kennedy, Jonah Gabry, Paul-Christian Bürkner, and Martin Modrák. Bayesian workflow. *arXiv preprint arXiv:2011.01808*, 2020.

[14] Andreas Griewank and Andrea Walther. *Evaluating derivatives*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, second edition, 2008.

[15] Léo Grinsztajn, Elizaveta Semenova, Charles C Margossian, and Julien Riou. Bayesian workflow for disease transmission modeling in stan. *arXiv:2006.02985*, February 2021.

[16] Matthew D. Hoffman and Andrew Gelman. The No-U-Turn Sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15:1593–1623, April 2014.

[17] Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David Blei. Automatic differentiation variational inference. *Journal of machine learning research*, 18:1 – 45, 2017.

[18] Charles C. Margossian. A review of automatic differentiation and its efficient implementation. *Wiley interdisciplinary reviews: data mining and knowledge discovery*, 9, 3 2019.

[19] Charles C Margossian and William R Gillespie. Differential equation based models in stan. In *StanCon 2017*, January 2017.

[20] Charles C Margossian, Aki Vehtari, Daniel Simpson, and Raj Agrawal. Hamiltonian monte carlo using an adjoint-differentiated laplace approximation: Bayesian inference for latent gaussian models and beyond. *Advances in Neural Information Processing Systems (NeurIPS)*, 33, October 2020.

[21] Richard McElreath. *Statistical Rethinking: A Bayesian Course with Examples in R and STAN*. CRC Press, second edition, 2020.

[22] Radford M. Neal. MCMC using Hamiltonian dynamics. In *Handbook of Markov Chain Monte Carlo*. Chapman & Hall / CRC Press, 2012.

[23] Stan development team. *Stan reference manual*. 2021.

[24] Stan development team. *Stan user's guide*. 2021.

[25] Aki Vehtari, Andrew Gelman, Daniel Simpson, Bob Carpenter, and Paul-Christian Buerkner. Rank-normalization, folding, and localization: An improved $\hat{r}$ for assessing convergence of MCMC. *Bayesian Analysis*, 2020.

[26] Aki Vehtari, Tommi Mononen, Ville Tolvanen, Tuomas Sivula, and Ole Winther. Bayesian leave-one-out cross-validation approximations for Gaussian latent variable models. *Journal of Machine Learning Research*, 17(103):1–38, 2016.

[27] Sebastian Weber, Andrew Gelman, Daniel Lee, Michael Betancourt, Aki Vehtari, and Amy Racine-Poon. Bayesian aggregation of average data: An application in drug development. *Annals of applied statistics*, 12, 2018.

[28] Yi Zhang, Bill Gillespie, and Charles Margossian. *Torsten user guide*. 2021.

[29] Yi Zhang, William R Gillespie, Ben Bales, and Aki Vehtari. Speed up population bayesian inference by combining cross-chain warmup and within-chain parallelization. *Journal of Pharmacokinetics and Pharmacodynamics*, 47, 2020.