Computing Steady States with Stan's Nonlinear Algebraic Solver

Charles C. Margossian[*]

January 2, 2018

Abstract

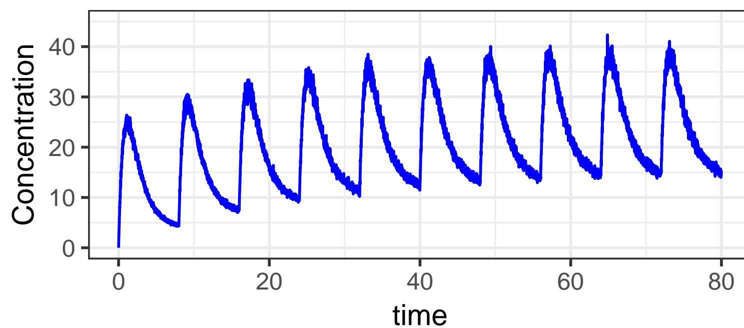
Stan's numerical algebraic solver can be used to solve systems of nonlinear algebraic equations with no closed form solutions. One of its key applications in scientific and engineering fields is the computation of equilibrium states (equivalently steady states). This case study illustrates the use of the algebraic solver by applying it to a problem in pharmacometrics. In particular, I show the algebraic system we solve can be quite complex and embed, for instance, numerical solutions to ordinary differential equations. The code in R and Stan are provided, and a Bayesian model is fitted to simulated data.

This R markdown file runs with rstan 2.17.2.

1 Introduction

In many scientific and engineering fields, we need to compute the state of a system once an equilibrium has been reached. One important case in pharmacometrics is that of a patient who has been under a treatment for an extended period of time. What are the long-term behaviors of a desease, a drug, and its side effects?

We will consider treatments that undergo a cycle, since these are the ones that generate steady states. A typical example is the intake of a drug at a regular time interval. At the beginning of the treatment, the patient may experience changes from one cycle to the other, such as an overall increase in the drug concentration in his or her blood.

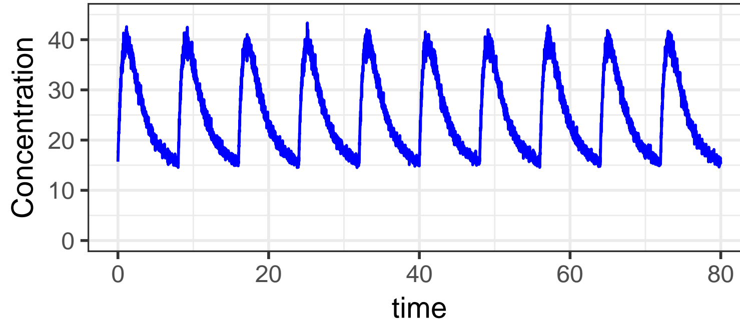


Drug Concentration in the Blood at the start of a treatment

(plot generated by *ggplot2* [1])

However, once a steady state is reached, there is no evolution from one cycle to the other.

[*]Columbia University, Department of Statistics (formally Metrum Research Group, LLC), contact: charles.margossian@columbia.edu



(plot generated by *ggplot2* [1])

Using our physical intuition, we may now formally define a steady state. Let τ be the period of a cycle (in our example the inter-dose interval), and $y(t)$ the function that describes the evolution of a system of interest over time. A steady state is reached when:

$$y(t + \tau) = y(t)$$

A more careful definition accounts for noise in the data and states the above equation holds on average. Since we are building generative models with Stan, we first focus on the determnistic features and assume the above equality to be exact. We will add stochastic components when we formulate the posterior.

Our goal is to evaluate $y(t_0)$, the state at the beginning of a cycle. The brute force approach would be to simulate the treatment, until a steady state is reached. A more elegant method is to solve the above algebraic equation. This can be a formidable problem, especially when y is sophisticated. In particular, the equation could be nonlinear and y could not have an analytical form.

2 Evolving the System over Time

In pharmacometrics y is often the vector solution of a system of ordinary differential equations (ODEs).

The above plotted data was simulated using a *two compartment model with a first-order absorption from the gut*[1], using the R package *mrgsolve* [3]. This model describes how the drug circulates in various compartments of the human body, blood being one of them (or more precisely, part of one of them).

y is the vector solution to the following system of three differential equations:

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

where

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

[1] This is one of the models I discussed at the *Stan Con 2017* [2].

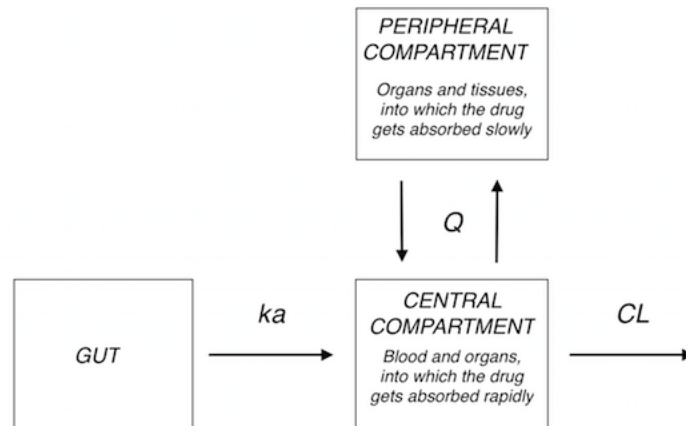


Figure 1: *Two Compartment Model with a First-Order Absorption from the Gut*

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

This system has a closed form solution which involves exponential polynomials.

Solving this equation tells us how the system evolves over time given an initial condition, y_0. This can be thought of as the *natural evolution* of the system. What it fails to account for are exterior interventions such as drug intakes. These need to be computed seperately. If a patient takes a drug orally, there will be a bolus increase in the drug mass in the gut, which we compute by adding the drug amount m to y_0. Schematically, a cycle in our example corresponds to:

1. Start at an intial state: $y(t_0^-)$.
2. Patient takes the drug. Add m to y_0 and get $y(t_0^+)$.
3. Evolve the system by solving the ODEs to get $y(t_0^+ + \tau)$.
4. Start over.

3 Solving Algebraic Equations

We need to solve $y(t_0^-) = y(t_0^+ + \tau)$.

The two compartment model is relatively simple. The ODE system has a closed form solution and the steady state can also be computed analytically. Ideally, we would hand-code the solution, but I would like to illustrate the use of `algebra_solver`, Stan's function to solve nonlinear algebraic equations numerically. This method, while slower, has a much broader application.

Many times y will indeed have no closed-form. It could for instance be the solution to a nonlinear ODE system, which we may well generate by extending our model to describe, in addition to drug circulation, desease progression and/or side-effects (see [2] for some examples). In such a case, we are forced to use a numerical algebraic solver, mostly because of the nonlinearity of the resulting algebraic equation. If the equations are linear, matrix operations will be the more efficient method.

I'll stick to the two compartment model because of its simplicity, and because the model is very fast, which should encourage readers to try and run the code themselves.

3.1 Writing the Algebraic Equation in Stan

Any algebraic equation can be turned into a root-finding problem:

$$f(y) = 0$$

In our example:

$$y(t_0^-) - y(t_0^+ + \tau) = 0$$

where we wish to solve for $y(t_0^-)$, the state of the patient at the beginning of a cycle. Note y is a vector, each element being the drug mass in a compartment. Our first task is to code the left-hand-side of this equation in Stan. We do this inside the **functions** block:

```
vector f(vector y, vector theta, real[] x_r, int[] x_i) {
  real ii = x_r[1];  // interdose interval (or tau)
  real amt = x_r[2];  // dose amount
  int cmt = x_i[1];  // compartment in which the drug is administered
  int evid = 1;

  // return the difference between the evolved and the initial state
  return twoCptModel1(ii, y, theta, amt, cmt, evid) - y;
}
```

Just as for the ODE integrator, the function must observe a strict signature. The first argument must be a `vector` and is the unknowns we wish to solve for. The parameters are passed in the vector `theta`, and real and interger data arrays are passed respectively in `x_r` and `x_i`. Parameters and data can then be "unwrapped" inside the function.

The function `twoCptModel1` is the evolution operator. It is coded as the analytical solution to the above ODEs, and in addition it computes discrete changes due to drug intake:

```
vector twoCptModel1(real dt, vector init, vector theta,
                    real amt, int cmt, int evid) {
  real CL = theta[1];
  real Q = theta[2];
  real V1 = theta[3];
  real V2 = theta[4];
  real ka = theta[5];
  real k10 = CL / V1;
  real k12 = Q / V1;
  real k21 = Q / V2;
  real ksum = k10 + k12 + k21;
  vector[3] alpha;
  vector[3] a;
  vector[3] x = rep_vector(0.0, 3);

  alpha[1] = (ksum + sqrt(ksum * ksum - 4.0 * k10 * k21))/2.0;
  alpha[2] = (ksum - sqrt(ksum * ksum - 4.0 * k10 * k21))/2.0;
  alpha[3] = ka;

  if(init[1] != 0.0){
```

```
    x[1] = init[1] * exp(-alpha[3] * dt);
    a[1] = ka * (k21 - alpha[1]) / ((ka - alpha[1]) * (alpha[2] - alpha[1]));
    a[2] = ka * (k21 - alpha[2]) / ((ka - alpha[2]) * (alpha[1] - alpha[2]));
    a[3] = -(a[1] + a[2]);
    x[2] = init[1] * sum(a .* exp(-alpha * dt));
    a[1] = ka * k12 / ((ka - alpha[1]) * (alpha[2] - alpha[1]));
    a[2] = ka * k12 / ((ka - alpha[2]) * (alpha[1] - alpha[2]));
    a[3] = -(a[1] + a[2]);
    x[3] = init[1] * sum(a .* exp(-alpha * dt));
  }

  if(init[2] != 0){
    a[1] = (k21 - alpha[1]) / (alpha[2] - alpha[1]);
    a[2] = (k21 - alpha[2]) / (alpha[1] - alpha[2]);
    x[2] = x[2] + init[2] * sum(segment(a, 1, 2) .* exp(-segment(alpha, 1, 2) * dt));
    a[1] = k12 / (alpha[2] - alpha[1]);
    a[2] = -a[1];
    x[3] = x[3] + init[2] * sum(segment(a, 1, 2) .* exp(-segment(alpha, 1, 2) * dt));
  }

  if(init[3] != 0){
    a[1] = k21 / (alpha[2] - alpha[1]);
    a[2] = -a[1];
    x[2] = x[2] + init[3] * sum(segment(a, 1, 2) .* exp(-segment(alpha, 1, 2) * dt));
    a[1] = (k10 + k12 - alpha[1]) / (alpha[2] - alpha[1]);
    a[2] = (k10 + k12 - alpha[2]) / (alpha[1] - alpha[2]);
    x[3] = x[3] + init[3] * sum(segment(a, 1, 2) .* exp(-segment(alpha, 1, 2) * dt));
  }

  if(evid == 1) x[cmt] = x[cmt] + amt;

  return x;
}
```

3.2 Calling the Algebraic Solver

We can now call the algebraic solver:

```
y = algebra_solver(f, init_guess, theta, x_r, x_i);
```

where `f` is the function declared above. The second argument is an initial guess. `theta`, `x_r`, and `x_i` are the parameters and data which get passed to `f`.

A good guess increases the speed of the solver and even determines, whether the solver converges or not. In degenerate cases (i.e. when there is more than one solution), the guess can determine which solution the solver returns. Here's a simple example:

$$z_1 = (y_1 - 5)(y_2 - 8)$$
$$z_2 = (y_2 - 5)(y_1 - 8)$$

If the initial guess is $y_{\text{init}} = (1, 10)$, the solver returns $y^* = (8, 8)$. If the initial guess is $y_{\text{init}} = (1, 1)$, the solver returns $y^* = (5, 5)$. In general, the solver looks for a solution in the "neighborhood" of the initial guess, but what this neighborhood corresponds to is not always well defined, when different solutions have comparable scales.

We can also control certain tuning parameters of the algebraic solver: the relative tolerance, the maximum number of iterations, and the function tolerance (the latter measures how far from 0 $f(y^*)$ is). See Stan's user manual, section 20 [3].

In our model, I use the drug mass during an intake in the cycle to scale the initial guess:

```
init_guess[1] = amt[1];
init_guess[2] = amt[1] * 0.5;
init_guess[3] = amt[1] * 0.35;
```

This is a rough guess but it works well. A better approach would be to compute the drug mass in the body at the beginning of the treatment (after one cycle). Such a guess would be parameter dependent. Unfortunately, Stan currently requires the initial guess to be a vector of data, which places severe restrictions. We plan to remove this unnecessary requirement in Stan's next release[2].

The model library *Torsten*, a collection of Stan functions for pharmacometrics [5, 6] (version 0.83) implements the above described method. The *BUGS model library* [7], Torsten's predecessor for WinBUGS [8], does so too and in addition improves the initial guess, by simulating multiple cycles, when the solver fails to converge. This technique can be used in the Stan Math C++ library, though currently not in the Stan language.

In all these cases, we may say the guesses are "naive", as we do not expect them to be spot on. For example, the drug mass after one cycle clearly understimates the mass at steady state. These guesses may however very well capture the scale of the solution, which should be enough the make the solver converge.

4 Stan model

The data we fit our model to is obtained from a simulated clincal trial. A patient has been under a treatment for an extended period of time and has reached steady state. During the trial we monitor four cycles of the treatment, by measuring the plasma drug concentration in the blood. The data is simulated using mrgsolve. Note mrgsolve simulates a steady state by computing the regimen for an extended period of time, rather than using an algebraic solver.

We wish to evaluate the parameters of the two compartment model, i.e. the coefficients in the ODE system (CL, Q, VC, VP, and k_a) and the standard deviation responsible for residual errors (σ).

4.1 R script to run the Stan model

```r
modelName <- "SteadyState"

# Specify the variables for which you want history and density plots
parametersToPlot <- c("CL", "Q", "VC", "VP", "ka", "sigma")

# Additional variables to monitor
otherRVs <- c("cObsPred")

parameters <- c(parametersToPlot, otherRVs)
parametersToPlot <- c("lp__", parametersToPlot)

# initial estimates
init <- function() {
  list(CL = exp(rnorm(1, log(10), 0.2)),
       Q = exp(rnorm(1, log(20), 0.2)),
       VC = exp(rnorm(1, log(70), 0.2)),
```

[2] see https://github.com/stan-dev/math/issues/651

```
       VP = exp(rnorm(1, log(70), 0.2)),
       ka = exp(rnorm(1, log(1), 0.2)),
       sigma = runif(1, 0.5, 2))
}

# The data is simulated with SteadyStateSimulations.R and saved in
# SteadyState.data.R. We'll read in the data into a list.
data <- read_rdump("SteadyState.data.R")
```

```
nChains <- 4
nPost <- 1000  # Number of post-burn-in samples per chain after thinning
nBurn <- 1000  # Number of burn-in samples per chain after thinning
nThin <- 1

nIter <- (nBurn + nPost) * nThin
nBurnin <- nBurn * nThin

fit <- stan(file = file.path(modelDir, paste(modelName, ".stan", sep = "")),
            data = data,
            pars = parameters,
            iter = nIter,
            warmup = nBurnin,
            thin = nThin,
            init = init,
            chains = nChains,
            cores = min(nChains, parallel::detectCores()))

dir.create(outDir)
```

```
## Warning in dir.create(outDir): '/Users/charlesm/Desktop/StanCon2018/model/
## SteadyState' already exists
```

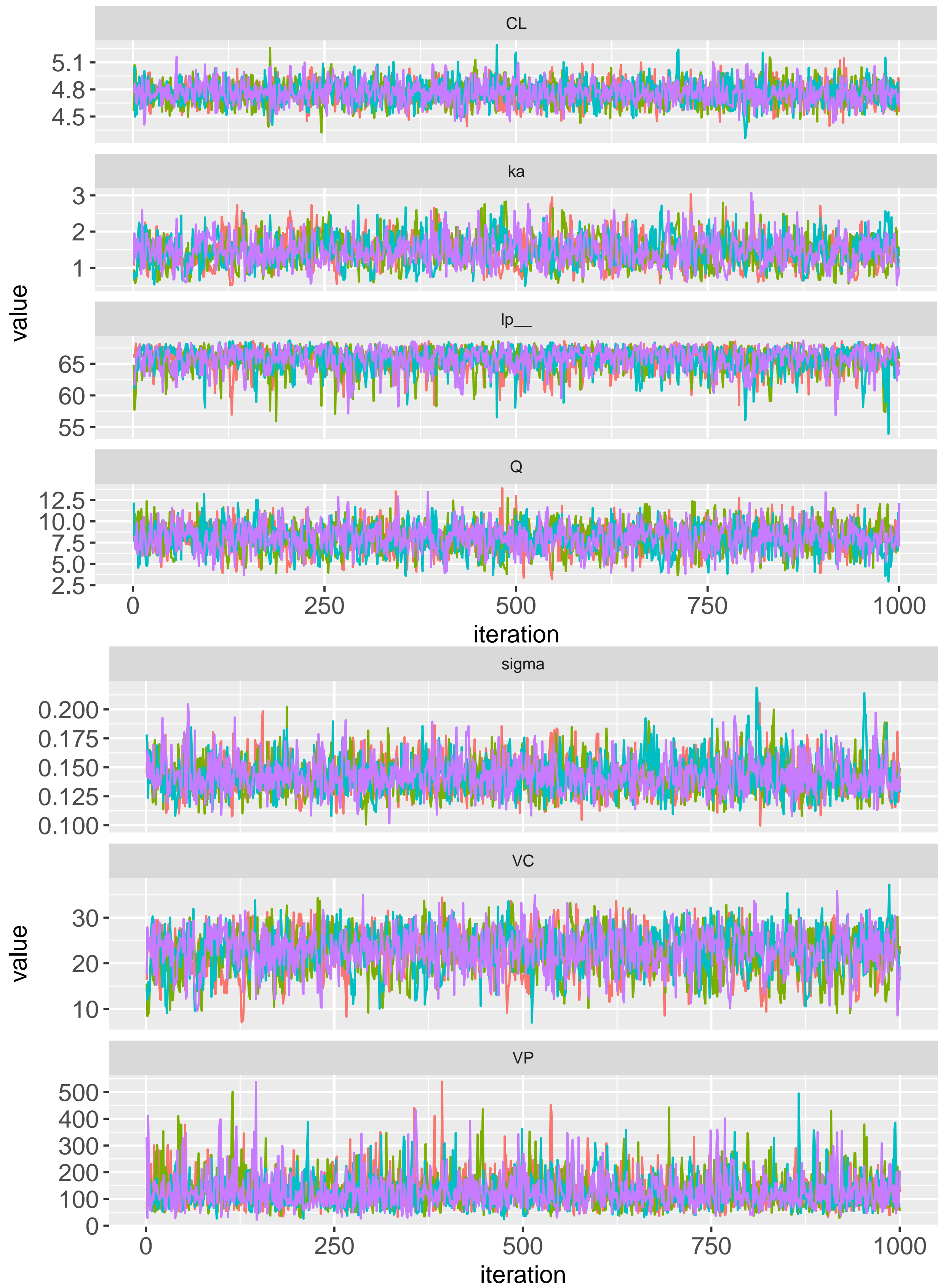
```
save(fit, file = file.path(outDir, paste(modelName, "Fit.Rsave", sep = "")))
```

4.3 Diagnostics

We start with the trace and density plots:

```
mcmcHistory(fit, parametersToPlot)
```

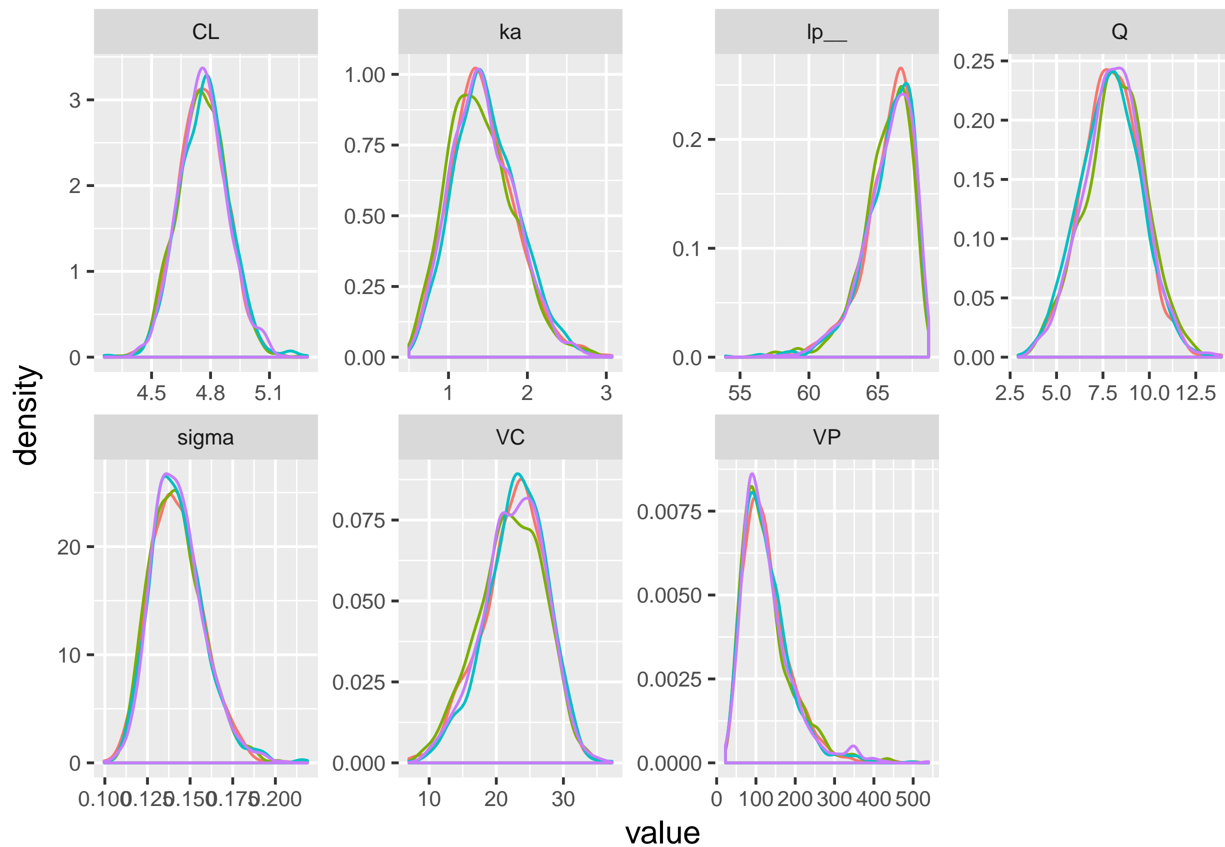
```
## Joining, by = "parameter"
```




```
## NULL
```

```r
mcmcDensity(fit, parametersToPlot, byChain = TRUE)
```

```
## Joining, by = "parameter"
```

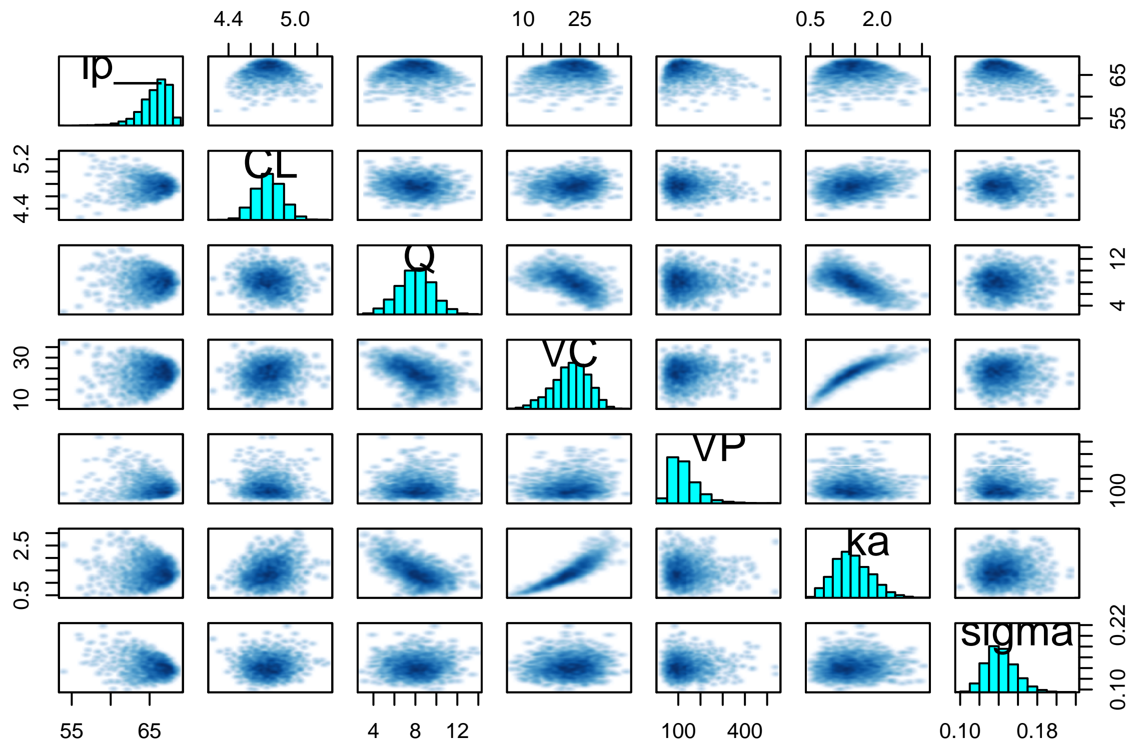



```
## NULL
```

All chains display the desired "fuzzy cartepillar" shape, which indicates low autocorellation and the density plots overlap. This is strong evidence all chains have converged to a common posterior distribution.

Let's take a look at the pairs plot. The pairs plot gives us a nice visual representation of correlation between parameters in the posterior. It can also be used to visualize pathologies during the model fitting, such as divergent transitions (indicated by red dots) or points where the maximum tree depth has been reached (yellow dots). For more details, see the RStan manual on on pairs.stanfit [9].

```r
pairs(fit, pars = parametersToPlot)
```



No red or yellow dots. All looks good.

The results are summarized in the following table:

```
ptable <- parameterTable(fit, parametersToPlot)
ptable
```

```
##              mean      se_mean          sd       2.5%        25%
## lp__   65.7019572 0.0457479662  1.83974263 61.2276732 64.7345815
## CL      4.7626178 0.0021723373  0.12443634  4.5266385  4.6775942
## Q       8.0107238 0.0405489959  1.62215222  4.7554662  6.9290407
## VC     22.5765176 0.1298156416  4.68866049 12.6289149 19.6152436
## VP    127.6915287 1.2308571859 63.63068340 46.7889330 83.4791438
## ka      1.4426514 0.0112669973  0.41048733  0.7325323  1.1453007
## sigma   0.1424544 0.0003288461  0.01552679  0.1161060  0.1312575
##               50%         75%       97.5%    n_eff      Rhat
## lp__   66.0473592  67.0588663  68.1448396 1617.225 1.0004003
## CL      4.7612551   4.8450759   5.0136173 3281.255 1.0005986
## Q       8.0407653   9.1123453  11.1806004 1600.380 1.0021023
## VC     22.9124519  25.9408490  30.6210437 1304.498 1.0027995
## VP    114.6826206 155.7989260 289.8225649 2672.501 0.9996507
## ka      1.4043252   1.7143634   2.3166737 1327.343 1.0023262
## sigma   0.1411216   0.1519403   0.1763881 2229.349 1.0010919
```

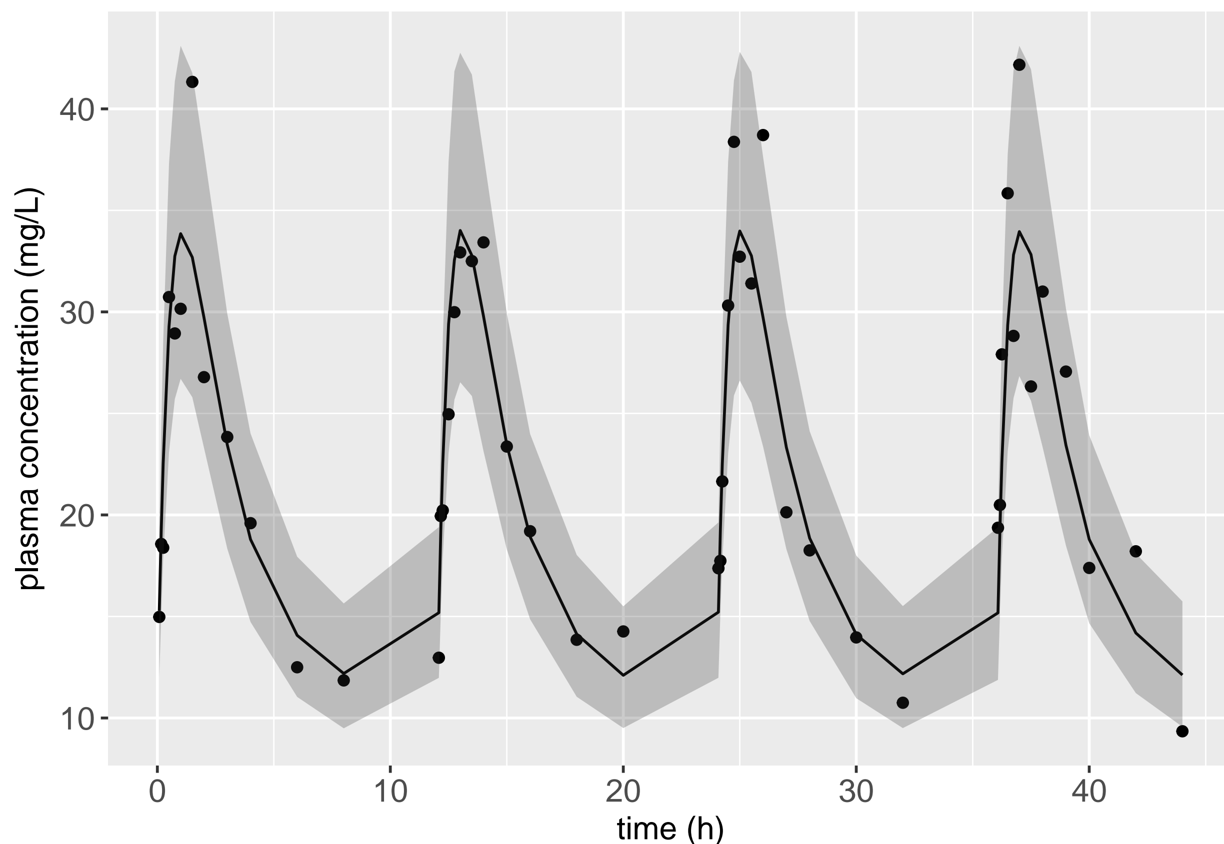
4.4 Posterior Predictive Checks

Using the **generated quantities** block, we generate data from our model which we can then compare to the original data:

```
data <- data.frame(data$cObs, data$time[data$iObs])
data <- plyr::rename(data, c("data.cObs" = "cObs", "data.time.data.iObs." = "time"))
```

```r
pred <- as.data.frame(fit, pars = "cObsPred") %>%
  gather(factor_key = TRUE) %>%
  group_by(key) %>%
  summarize(lb = quantile(value, probs = 0.05),
            median = quantile(value, probs = 0.5),
            ub = quantile(value, probs = 0.95)) %>%
  bind_cols(data)

p1 <- ggplot(pred, aes(x = time, y = cObs))
p1 <- p1 + geom_point() +
  labs(x = "time (h)", y = "plasma concentration (mg/L)") +
  theme(text = element_text(size = 12), axis.text = element_text(size = 12),
        legend.position = "none", strip.text = element_text(size = 8))
p1 + geom_line(aes(x = time, y = median)) +
  geom_ribbon(aes(ymin = lb, ymax = ub), alpha = 0.25)
```



The model fit and the data are in agreement.

5 Alternative Model Specification: Solving an ODE numerically inside an Algebraic Equation

To illustrate the applicability of the algebraic solver, let's rewrite the model without using the analytical solution to the two compartment model ODEs. Instead the ODE will be solved numerically. In practice, we should priviledge analytical methods, but numerical techniques are much more generalizable and make for a more pedagogical example.

The ODE system is coded in Stan as followa:

```
real[] twoCptModelODE(real t,
          real[] x,
          real[] parms,
          real[] x_r,
          int[] x_i){
  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;
}
```

and the algebraic equation becomes:

```
vector f(vector y, vector theta, real[] x_r, int[] x_i) {
  real amt = x_r[2];
  int cmt = x_i[1];
  real y_ii[3] = to_array_1d(y);

  y_ii[cmt] = y_ii[cmt] + amt;
  y_ii = integrate_ode_rk45(twoCptModelODE, y_ii, 0, rep_array(x_r[1], 1),
                            to_array_1d(theta), rep_array(0.0, 1),
                            rep_array(0, 1))[1];

  // return the difference between evolved and initial state
  return to_vector(y_ii) - y;
}
```

The task of evolving the system is now carried out by the integrator, rather than by the function `twoCptModel1`. Notice the dose input gets computed before we call the numerical integrator.

In order to save time when knitting this file, I won't do a full Bayesian analysis. Instead, I'll do a determinisitic test. That is, I'll fix the parameters and make sure the data I simulate with Stan agrees with what I get from mrgsolve. The parameter values are set to:

$$\begin{aligned} CL &= 5\text{L/h} \\ Q &= 8\text{L/h} \\ V_{\text{cent}} &= 20\text{L} \\ V_{\text{peri}} &= 70\text{L} \\ ka &= 1.2\text{h}^{-1} \\ \sigma &\approx 0 \end{aligned}$$

Let's first generate the data with Stan:

```
# Use fixed values for parameters (since we're doing a fixed parameter test)
init <- function () { list(CL = 5,
       Q = 8,
       VC = 20,
       VP = 70,
       ka = 1.2,
       sigma = 0.00001)
}

data <- read_rdump("SteadyStateODE.data.R")

################################################################################################
## run Stan
nChains <- 1
nPost <- 1 ## Number of post-burn-in samples per chain after thinning
nBurn <- 0 ## Number of burn-in samples per chain after thinning
nThin <- 1

nIter <- nPost * nThin
nBurnin <- nBurn * nThin

fit <- stan(file = file.path(modelDir, paste(modelName, ".stan", sep = "")),
            algorithm = "Fixed_param",
            data = data,
            pars = parameters,
            iter = nIter,
            warmup = nBurnin,
            thin = nThin,
            init = init,
            chains = nChains,
            cores = min(nChains, parallel::detectCores()))
```

```
##
## SAMPLING FOR MODEL 'SteadyStateODE' NOW (CHAIN 1).
## Iteration: 1 / 1 [100%]  (Sampling)
##
##  Elapsed Time: 0 seconds (Warm-up)
##                0.002593 seconds (Sampling)
##                0.002593 seconds (Total)
```

```
dir.create(outDir)
```

```
## Warning in dir.create(outDir): '/Users/charlesm/Desktop/StanCon2018/model/
## SteadyStateODE' already exists
```

```
save(fit, file = file.path(outDir, paste(modelName, "Fit.Rsave", sep = "")))
```

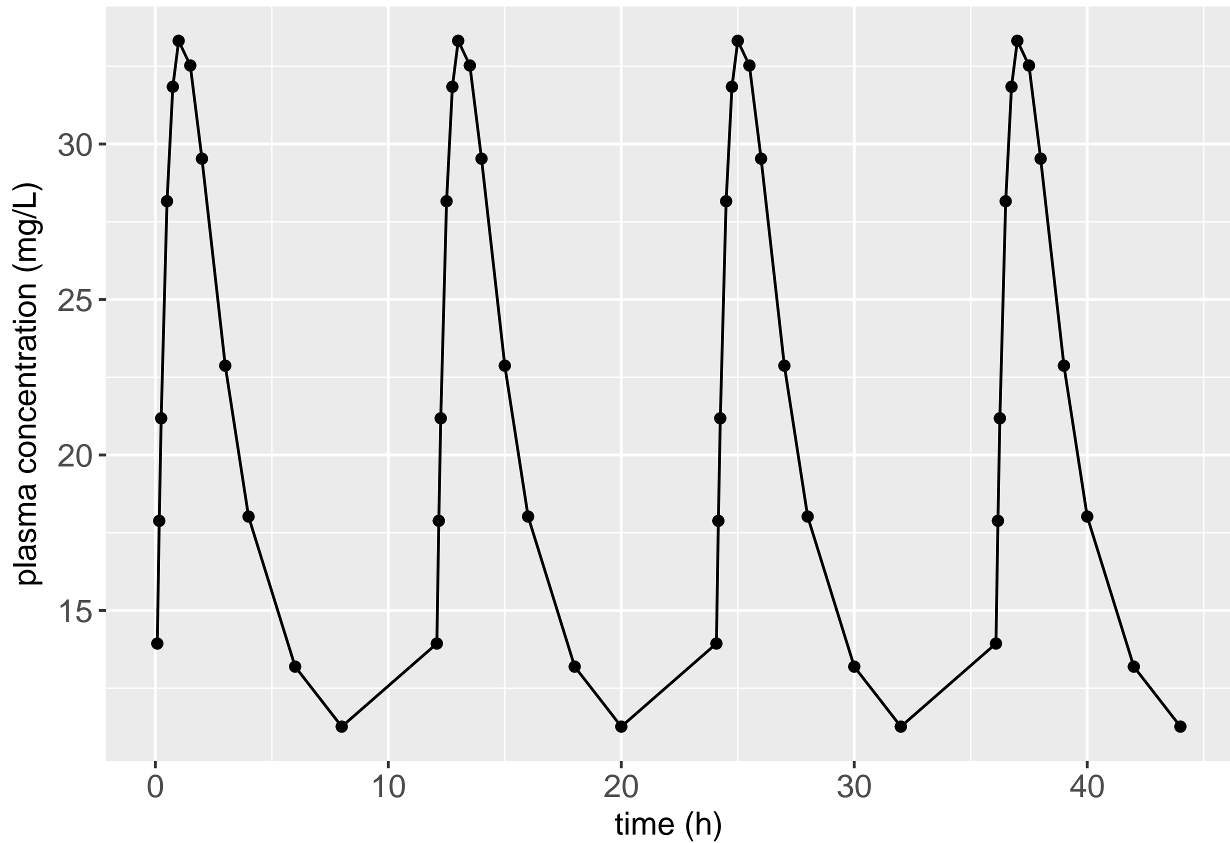
Next, we compare it to what we produced with mrgsolve:

```
data <- data.frame(data$cObs, data$time[data$iObs])
data <- plyr::rename(data, c("data.cObs" = "cObs", "data.time.data.iObs." = "time"))

pred <- as.data.frame(fit, pars = "cObsPred") %>%
  gather(factor_key = TRUE) %>%
  group_by(key) %>%
```

```r
  summarize(lb = quantile(value, probs = 0.05),
            median = quantile(value, probs = 0.5),
            ub = quantile(value, probs = 0.95)) %>%
  bind_cols(data)

p1 <- ggplot(pred, aes(x = time, y = cObs))
p1 <- p1 + geom_point() +
  labs(x = "time (h)", y = "plasma concentration (mg/L)") +
  theme(text = element_text(size = 12), axis.text = element_text(size = 12),
        legend.position = "none", strip.text = element_text(size = 8))
p1 + geom_line(aes(x = time, y = median)) +
  geom_ribbon(aes(ymin = lb, ymax = ub), alpha = 0.25)
```



The data simulated with Stan agrees with that from mrgsolve, demonstrating the algebraic solver's ability to find the root of a relatively complicated function.

Readers familiar with some of the algorithms Stan uses under the hood will also appreciate the fact the solver not only produces a solution but also the partial derivatives of the solution with respect to auxiliary parameters. Moreover, automatic differentiation allows us to combine the Jacobian matrices of complex expressions (such as the numerical solutions to algebraic and differential equations) to compute the gradient of the log posterior.

Acknowledgements

Institutions: I 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: Thank you to the StanCon organizing committee, my colleagues on the Stan development team, and Bill Gillespie from Metrum Research Group.

References

[1] Wickham, H. *ggplot2: Elegant Graphics for Data Analysis.* 978-0-387-98140-6 (Springer-Verlag New York, http://ggplot2.org, 2009).

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

[3] Baron, K.T., Hindmarsh, A.C., Petzold, L.R., Gillespie, B., Margossian, C. and Pastoor, D. *mrgsolve: Simulate from ODE-Based Population PK/PD and Systems Pharmacology Models.* Metrum Research Group, http://mrgsolve.github.io/user guide/ (2017). R package version 0.8.7.

[4] Stan Development Team. Stan Modeling Language Users Guide and Reference Manual. http://mc-stan.org, version 2.16.0 edition (2017).

[5] Margossian, C.C. and Gillespie, W.R. Stan functions for pharmacometrics modeling. In Journal of Pharmacokinetics and Pharmacodynamics, volume 43 (2016).

[6] Margossian, C.C. and Gillespie, W.R. Torsten: User Manual, version 0.83. Metrum Research Group, https://github.com/metrumresearchgroup/example-models/blob/torsten- 0.83/torstenManual.pdf (2017).

[7] Gillespie, W.R. *Prototype PKPD Model Library for WinBUGS*, version 1.2. Metrum Institute, https://bitbucket.org/metrumrg/bugsmodellibrary/wiki/Home

[8] Lunn, D.J. and Thomas, A. and Best, N. and Spiegelhalter, D. *WinBUGS* - A Bayesian modelling framework: Concepts, structure, and extensibility, Statistics and Computing, Volume 10, Number 4, pages 325 - 337, October 2000

[9] Stan Development Team (2017). *RStan*: The R interface of Stan. R package version 2.17.2, http://mc-stan.org/.