Technical appendix for Torsten

Torsten Development Team

Updated November 2018

Abstract

Stan (mc-stan.org) is a probabilistic programing language, primarily designed to do Bayesian analysis. It is expressive and gives users a lot of flexibility when specifying models. Its main algorithm to compute posterior distributions is the No U-Turn sampler, an adaptive Hamiltonian Monte Carlo sampler. Torsten extends Stan with a suite of functions that facilitate the specification of pharmacokinetic and pharmacodynamic models. Both softwares are open-source. This article serves as an appendix to the Torsten user manual (https://metrumresearchgroup.github.io/Torsten): it reviews various technical components required to construct and fit models in pharmacometrics. It is an under-the-hood exploration of the package, which should be of interest to advance users and potential contributors.

Keywords: Pharmacometrics, Bayesian modeling, Hamiltonian Monte Carlo, Automatic differentiation, Ordinary differential equations, Numerical methods, Stan, Torsten.

Contents

1 Introduction

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

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

It is no surprise then that an open-source project, such as `Stan` or `Torsten`, brings together scientists with a broad array of backgrounds, which greatly enriches the project, but sometimes makes communication difficult. This articles attempts to ease conversation between experts in different fields and should be of interest to advanced users and potential contributors. It is written as an appendix to the *Torsten User Manaual*[1].

Important*: The details covered here are not required to use* `Torsten`*. For that, we direct the reader to the Stan Book*[2] *and the Torsten User Manual.*

The focus is on `Torsten`'s design and the algorithms it uses. We point readers to references for more detailed descriptions, and also suggest improvements that can be made. A full understanding of this appendix requires a good knowledge of several topics. Most readers cannot be expected to be experts in all these fields, which is why we review some elementary facts. We will however assume that readers are familiar with Bayesian analysis, and the use of Markov chains Monte Carlo (MCMC) to sample from a target posterior distribution.

1.1 General Background

`Stan` is a probabilistic programing language, that allows users to specify models in terms of mathematical functions and probability distributions, and provides several inference algorithms to fit these models (Carpenter et al., 2017). It also comes

[1] `https://metrumresearchgroup.github.io/Torsten/`

[2] `http://www.stat.columbia.edu/~gelman/bda.course/_book/`

with a suite of packages to diagnose these models and construct robust frameworks for Bayesian modeling (see for example (Gabry, Simpson, Vehtari, Betancourt, & Gelman, 2017) and (Betancourt, 2018)). Our goal is to make `Stan` more accessible to modelers in Pharmacometrics, and our efforts are two fold:

1. we contribute general purpose tools to the core `Stan` language to support differential equation based models.

2. we develop `Torsten`, an extension with specialized functions for pharmacometrics (Gillespie & Zhang, 2018; Margossian & Gillespie, 2016).

This appendix hence contains details about the development of `Torsten` specific functions and more general tools directly incorporated inside `Stan`. We are for instance aware that the algebraic solver, initially developed to model biomedical steady states, is also used in econometrics. Furthermore, the `Torsten` development team is not the only group that contributes to the development of `Stan` for pharmacometrics. The details we review here are the product of a fruitful collaboration between individuals at various institutions.

2 Bayesian data analysis with `Stan`

`Stan` is primarily designed for Bayesian data analysis. It is a probabilistic programing language that offers users a lot of flexibility when specifying their model. In a Bayesian framework this means writing a joint distribution over the data and the model parameters, a distribution which can often be factored into a prior and a likelihood component. Note however that, in more complicated models, the distinction between prior and likelihood is not always clear or even useful (Gelman, Simson, & Betancourt, 2017).

`Stan`'s default algorithm is based on the No-U-Turn sampler (NUTS) (Hoffman & Gelman, 2014). Modifications to the latter are described in (Betancourt, 2017). NUTS is an adaptive Hamiltonian Monte Carlo (HMC) sampling algorithm, and has proven more efficient than other MCMC algorithms, such as Metropolis-Hasting and Gibbs sampling for complex high-dimensional problems (Hoffman & Gelman, 2014). Here, by “complex” we mean non-trivial relationships between the independent and dependent variables, such as ones that involve solving ordinary differential equations (ODEs) or hierarchical structures. The dimensionality of a model relates to the number of parameters.

2.1 Hamiltonian Monte Carlo sampling

We only do a quick review. More detailed and excellent treatments of the subject include (Neal, 2010) and (Betancourt, 2017). Note that in practice, there are many tuning parameters required to make HMC work effectively. NUTS alleviates this burden by adaptively tuning the parameters during a warm-up phase of the model fitting process.

The basic idea behind HMC is to treat the Markov chain as a *particle* that moves in the parameter space and the posterior as a *physical potential* that informs the dynamic of that particle. Let θ denote the model parameters and y our data. The potential, U, is defined as the negative of the log posterior, $\pi(\theta|y)$:

$$U = -\log \pi(\theta|y) \tag{1}$$

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

Under these circumstances, the chain takes "bigger steps" when the posterior density increases, but smaller ones when it moves towards regions of lower probability density. This results in several desirable properties, that allow the chain to efficiently explore the *typical set*, i.e the region where the vast majority of the probability mass concentrates. The latter phenomenon is termed *concentration of measure* and becomes extremely acute as we get to high dimensions.

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

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

Recalling our definition of the potential (equation 1), the vectorized expression of the above is:

$$p' = -\nabla_\theta \log(\pi(\theta|x)) \tag{2}$$

To simulate Hamiltonian trajectories, we must therefore *compute the gradient of the log joint posterior distribution with respect to the parameters*. Considering how complex such a distribution may be, this is not a trivial task. There is however one crucial simplification. The gradient of the log posterior is the gradient of the log joint. To see this, recall

$$\pi(\theta|x) = \frac{\pi(x, \theta)}{\pi(x)}$$

Thus

$$\log \pi(\theta|x) = \log \pi(x, \theta) + K$$

where K is a constant that does not depend on θ.

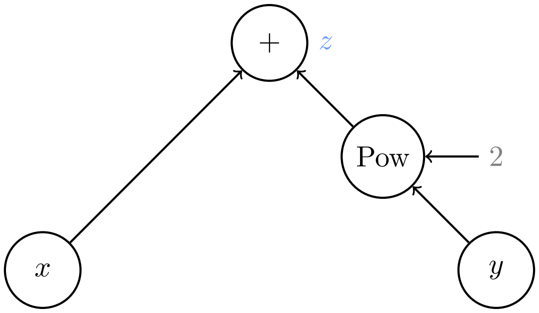

Figure 1: Topological graph for the programing statement `z = x + pow(y, 2)`

2.2 Automatic differentiation

To compute gradients, Stan uses *automatic differentiation* (AD). There is a rich literature on the subject; for an excellent review see (Baydin, Pearlmutter, Radul, & Siskind, 2018); and for a more extensive treatment, see (Griewank & Walther, 2008). AD revolutionized the fields of computational statistics and machine learning by eliminating the burden of manually calculating derivatives. Given the code for a differentiable function, AD numerically evaluates derivatives of that function, at a particular point. This is done by sorting the function into an expression graph. The nodes on this expression graph represent operators such as addition (`+`), power (`pow`), or the exponential (`exp`) function (figure 1). An AD software contains methods to compute the derivatives of these operators (analytically or otherwise), and combines these results, via the chain rule, to produce the gradient of the target function. Extensive details on how this gets done inside of `Stan` can be found in (Carpenter et al., 2015).

We now take the perspective, not of a modeler, but of a developer. Let $q \in \mathbb{R}^n$ be parameters and $x \in \mathbb{R}^k$ fixed data. For AD to work, a function in `Stan`, $f : \mathbb{R}^{n+k} \mapsto \mathbb{R}^m$, must compute the output, $f(q, x)$, and the $n \times m$ Jacobian matrix, J, where $J_{ij} = \partial f_i / \partial q_j$. This can be done in two ways: either we explicitly code a method to compute J, in which case our function behaves as a node on the expression graph; or we express f in terms of other `Stan` functions and directly use AD. The latter method is by far the easiest one to implement. Unfortunately, it can lead to inefficient and unstable code, when recklessly applied to complex mathematical functions. Two relevant examples are numerical solvers for ODEs and algebraic equations.

At a `C++` level, AD (as implemented in `Stan`) requires a new class, `var`, which contains (1) the value of a variable and (2) its adjoint with respect to the log joint distribution. Recall the adjoint of x with respect to f is the derivative of f with respect to x. There is therefore an important distinction between parameters, with respect to which we need to calculate the gradient and which must therefore be coded as var, and data, which are more simply coded as double.

It is good practice to allow users to call a function with only fixed data arguments. Most of the time, this means that arguments which may be parameters are templated, and can be passed as either double or var objects. For more details on how to write new functions in Stan at a C++ level, see the entry *Contributing new Stan functions*, on https://github.com/stan-dev/stan/wiki/.

3 Ordinary differential equations in pharmacometrics

The section is, in part, adapted from a presentation we gave at the 2017 Stan Conference on *Differential Equation Based Models in Stan* (Margossian & Gillespie, 2017a).

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

3.1 When do ordinary differential equations arise?

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

To get a basic intuition, consider the example of a gas container with a hole in it (figure 2). We can think of the gas as being made of molecules that move randomly in the container. Each molecule has a small chance of leaking through the hole. Thus the more molecules inside the container, the higher the number of escaping molecules per unit time. If there are a large number of molecules and the gas behaves like a continuous fluid, we observe that the more gas in the container, the higher the leakage. This statement can be written as the differential equation:

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

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

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

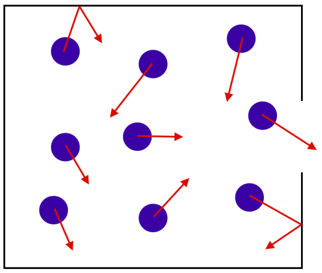

Figure 2: Gas container with a hole. *Gas molecules in a box move in a (approximatively) random fashion. In the limit where we have a lot of particles and the gas behaves like a continuous fluid, the rate at which the gas leaks is proportional to the amount of gas in the container. This physical process is described by the differential equation* $\mathrm{d}y/\mathrm{d}t = -ky(t)$*, where* y *is the amount of gas in the container.*

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

3.2 An example: ODE system for the Two Compartment Model

Consider the common scenario in which a patient orally takes a drug. The drug enters the body through the gut and is then absorbed into the blood. From there it diffuses into and circulates back and forth between various tissues and organs. Over time, the body clears the drug, i.e. the drug exits the body (for instance through urine) (figure 3).

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

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

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

$$\begin{aligned} y'_{\text{gut}} &= -k_a y_{\text{gut}} \\ y'_{\text{cent}} &= k_a y_{\text{gut}} - \left(\frac{CL}{V_{\text{cent}}} + \frac{Q}{V_{\text{cent}}} \right) y_{\text{cent}} + \frac{Q}{V_{\text{peri}}} y_{\text{peri}} \\ y'_{\text{peri}} &= \frac{Q}{V_{\text{cent}}} y_{\text{cent}} - \frac{Q}{V_{\text{peri}}} y_{\text{peri}} \end{aligned} \tag{3}$$

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

3.3 Overview of tools for solving differential equations

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

Specialized algorithms for solving ODEs tend be more efficient but have a narrower application; the reverse holds for more general tools. We provide both, thereby allowing users to tackle a broad range of problems and optimize their model when possible (figure 4).

3.4 The Event schedule

3.4.1 Handling exterior interventions

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

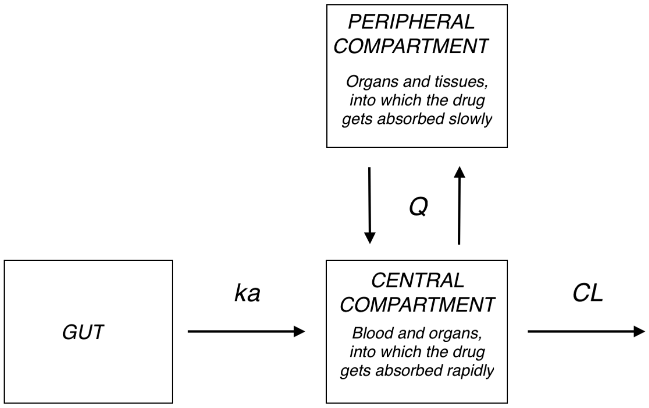
PERIPHERAL COMPARTMENT
Organs and tissues, into which the drug gets absorbed slowly
Q
GUT
ka
CENTRAL COMPARTMENT
Blood and organs, into which the drug gets absorbed rapidly
CL

Figure 3: Two Compartment Model. *An orally administered drug enters the body through the gut and is then absorbed into the blood, organs, and tissues of the body, before being cleared out.*

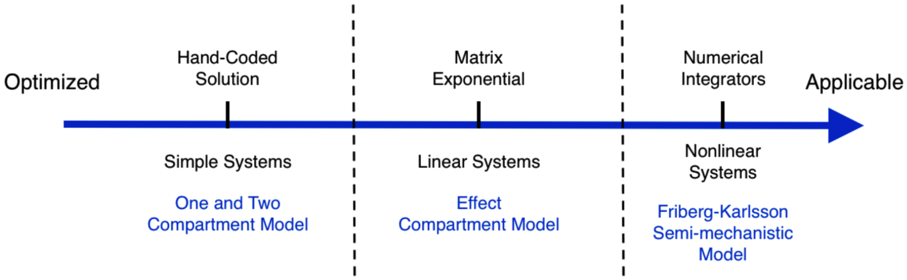

Figure 4: The "Optimized-Applicable Spectrum" of tools for solving ODEs. *The top line gives the technique to solve differential equations, the next line the type of ODE system this method should be applied to, and the third line (blue) an example from pharmacometrics, discussed in the user manual. Modelers should prefer the left-most applicable technique.*

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

We follow the convention set by NONMEM®[3], which is popular amongst pharmacometricians and which we find acceptable.

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

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

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

The user passes the event schedule using a data table. NONMEM's convention allows one row to code for multiple events. For example, a single row can specify a patient receives multiple doses at a regular time interval. Consider:

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

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

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

3.4.2 Dosing events

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

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

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

$$\begin{aligned} y(\tau^+) &= y(\tau^-) + m \\ y(t) &= \int_{\tau^+}^{t} f(y, t') dt' \end{aligned} \tag{4}$$

The bolus dosing event introduces a discontinuity with respect to time (figure 5). As we will discuss in a moment, this can lead to issues.

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

$$\begin{aligned} y(\delta) &= \int_{\tau}^{\delta} f(y, t') + R dt' \\ y(t) &= \int_{\delta}^{t} f(y, t') dt' \end{aligned} \tag{5}$$

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

Let us now go back to bolus doses, and issues they may cause. If τ or any observation time depends on one or more model parameters, the joint distribution of our model may no longer be differentiable at every point of interest. This problem arises, for example, when the modeler tries to estimate *lag times*. We still need to characterize how important the resulting error is[4]. We tested the computation of the Jacobian matrix with AD. As expected, the derivative is not properly evaluated at events which occur exactly at the time of a dosing, lag time accounted for. To be more precise, the derivatives are ill-defined, and can at best be characterized by looking at two one sided limits. The main concern here is then not the value that AD returns, but the fact that it does not identify the derivative as being ill-posed and return an error message to that effect. In that sense, AD is not robust.

But back to our problem in pharmacometrics: lag times are continuous variables, therefore a scenario in which an event occurs exactly at the time of the dosing is unlikely (in many settings, such a scenario has in fact a probability 0). Our

[4] see issue #3 on `github.com/metrumresearchgroup/math/issues/3`.

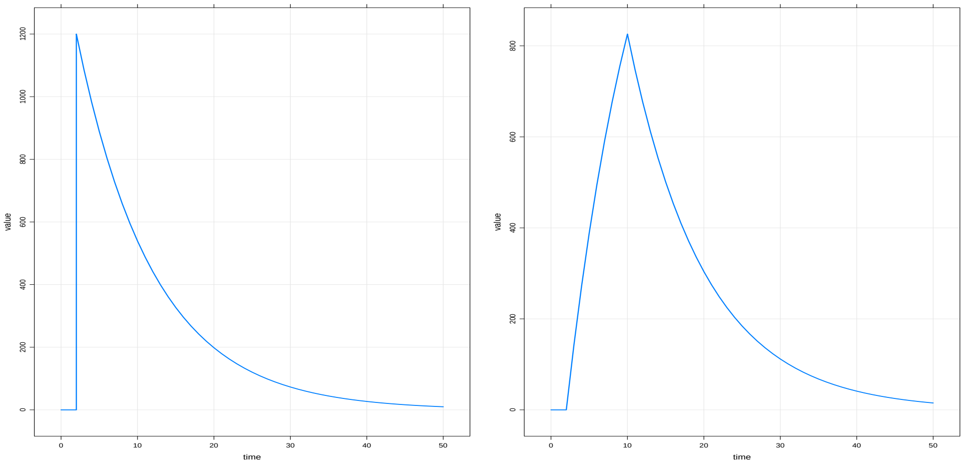

Figure 5: Simulated drug mass in a patient's gut after an oral drug administration ***(left)*** *At* $t = 2\text{h}$ *a patient receives a* 1200mg *bolus dose. This creates a discontinuity in the drug mass,* $y(t)$*. This can be an issue when* t_{lag} *is a parameter. As the system evolves, the drug gets naturally cleared out of the gut and absorbed into the blood and other organs (see equation 3).* ***(right)*** *A patient receives an infusion, starting at* $t = 2\text{h}$ *at a rate of* $R = 150\text{mg/h}$*. After 8 hours, the infusion stops and the drug mass decreases. Figures generated with `mrgsolve`.*

test further reveals the Jacobian matrix at other events is properly evaluated[5]. An additional mitigating factor is that we usually do not make measurements on the absorption compartment but on central compartments. In such compartments, the change in drug mass is not instantaneous. They introduce a discontinuity in derivatives, but this is not an issue.

3.4.3 Model parameters

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

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

All continuous variables, including elements of the event schedule, can be passed as either parameters or fixed data. This demand some caution when we compute

[5] There may still be issues with numerical solvers. This has not yet been tested for.

sensitivities. Handling θ is straightforward. F and t_{lag} on the other hand modify variables which get passed to the evolution operator and create latent parameters. In particular, F modifies the rate, R, for regular solutions and the amount, m, for steady state solutions. t_{lag} alters the dosing time, t.

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

4 Solving ordinary differential equations

Recall we want to compute the log joint distribution and its gradient (equivalently, the gradient of the log posterior). We therefore need to solve differential equations and propagate derivatives through them. This means computing the Jacobian matrix of the solutions with respect to the parameters. In `Torsten`, this tasks is carried out by the *pred* functors, which act as the *evolution operators* (they compute the evolution of the system from one state to the other between two events).

4.1 Analytical solution

We hand-code the solution for the One and Two Compartment models with a first-order absorption. The Jacobians are computed using AD. We could work out these derivatives analytically, but the resulting speed-up may only be minor.

4.2 The Matrix Exponential

Consider a system of linear differential equations:

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

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

$$y(t) = e^{tK} y_0 \tag{6}$$

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

$$e^{tK} = \sum_{n=0}^{\infty} \frac{(tK)^n}{n!} = I + tK + \frac{(tK)^2}{2} + \frac{(tK)^3}{3!} + ... \tag{7}$$

Looking at this definition, we clearly see the derivative of e^{tK} is Ke^{tK}. In practice, it is not feasible to compute an infinite number of terms.

Routine	**Arguments**	**Output**
`matrix_exp()`	A	e^A
`matrix_exp_multiply()`	A, B	$e^A B$
`scale_matrix_exp_multiply()`	A, B, t	$e^{At} B$

Table 1: Routines in `Stan` to compute the matrix exponential.

There exist several methods to compute the matrix exponential. (Moler & Van Loan, 2003) provide an excellent and detailed review. (?, ?) propose an algorithm to efficiently compute the action of the matrix exponential, that is the product of e^A with a vector, such as ${y_0}^6$. Note that when solving a system of linear ODEs, the solution is the action of a matrix exponential. To accommodate the scope of applications of the matrix exponential, `Stan` provides three routines, summarized in table 4.2. We review the algorithm implemented in `Stan` in some level of details to outline certain questions that we, as developers, may ask ourselves.

First we note that for a 2 × 2 matrix

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

there is an analytical solution, provided $(a-d)^2 + 4bc > 0$ (Rowland & Weisstein, n.d.). Naturally, we will want to take advantage of this. For the general case, we use the Padé approximation coupled with scaling and squaring. The idea is to approximate the matrix exponential with the finite series:

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

where

$$\begin{aligned} N_{pq}(K) = & \quad \sum_{j=0}^{p} \frac{(p+q-j)!p!}{(p+q)!j!(p-j)!} A^j \\ D_{pq}(K) = & \sum_{j=0}^{q} \frac{(p+q-j)!q!}{(p+q)!j!(q-j)!} (-A)^j \end{aligned} \tag{8}$$

In the asymptotic limit, $(p,q) \to (\infty, \infty)$, we recover the definition of the matrix exponential. (Moler & Van Loan, 2003) explain the benefits of this method. We

[6] For some details on how this is done in `Stan`, see `https://github.com/stan-dev/math/issues/771`.

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

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

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

All in all, this a rather nasty looking equation, with several tuning parameters, such as the choice of p, q, and m. Theoretical considerations can help us make an informed decision. Fortunately, there exist several packages which already implement this algorithm. The `C++` library `Eigen` (Guennebaud, Jacob, et al., 2010) contains an unsupported routine to compute the matrix exponential, using the above described method. It is readily available to `Stan`, which extensively uses other components of `Eigen`. It would be to look at other routines, such as the one implemented in the package `Expokit` (Sidje, 1998). Tests revealed `Eigen` usually computes 13 terms in the Padé series, whereas `Expokit` typically computes 6, yielding less precise results but faster computation. For our goals, 6 terms may already provide us with all the accuracy we need. More generally, the ideal number of terms depends on the matrix.

Once we have selected a method to calculating the matrix exponential, we need to propagate derivatives by computing the derivatives of the output with respect to the parameter-dependent inputs. This amounts to constructing a Jacobian matrix or tensor. There are clever ways of doing this, but for `matrix_exp()` we can actually get away with brute force AD. Indeed, while equation 8 looks rather intricate, it only involves basic operations which can be sorted into an expression graph. For a basic implementation, it therefore suffices to overload `Eigen`'s routine, such that it can handle `var` types. A word a caution though: in other cases, brute-forcing AD through a numerical method can fail spectacularly, yielding unstable and inefficient code. Two examples are numerical ODE integrators and numerical algebraic solvers. Furthermore, analytical or semi-analytical gradients tend to produce more efficient code.

4.3 Numerical integrators

Numerical integrators fundamentally differ from analytical and matrix exponential methods. Rather than evaluating the solution at certain points of interest, integrators construct the *solution function*, step by step, across a large grid of points. This is the main reason why there are much slower than the methods we have examined so far; they however require much less assumptions about the structure of the solution and apply to a much broader class of problems. Most importantly, they allow us to solve systems of nonlinear ODEs with no closed-form solutions.

Euler's method provides the right intuition on how integrators work. We begin at a point

$$y_0 = y(t_0)$$

and then compute

$$y_1 = y(t_0 + \Delta t)$$

by evaluating the derivative of y with respect to t at y_0 and performing a *tangent approximation*. Recall the ODEs give us the needed derivatives. We then repeat this procedure, starting at point y_1, and computing a tangent there. The output is a piece-wise linear approximation of the true solution. The immediate question is: what step size, Δt, should we use? Again, we want to achieve accuracy, but also minimize computational cost.

This problem is notoriously hard when we deal with *stiff* systems. Stiffness is not a rigorously mathematically defined property but nevertheless captures an important idea. In a stiff equation, the rate at which y changes with respect to t varies dramatically from one region to the other. Hence a given step size may be optimal in one region, but very suboptimal in another region. Tackling this issue requires its own class of algorithms which can adaptively change the step size, as the solution gets computed

`Stan` provides three ODE integrators. The Runge-Kutta 4th/5th order, `integrate_ode_rk45`, is relatively fast but cannot handle stiff ODEs. For stiff systems, modelers should use `integrate_ode_bdf` or `integrate_ode_adams`, which respectively implement the backward differentiation method and the Adams-Moulton schemes. The latter is more stable when we wish to achieve higher-order accuracy. Note that, once again, these methods fall on an optimum - applicable spectrum.

4.3.1 Propagating derivatives through an ODE system

Suppose our solution is $y \in \mathbb{R}^n$. We first need to identify inputs which may depend on model parameters. These can be:

1. certain coefficients in the ODEs, θ
2. initial states, y^0
3. time points, t

Correspondingly, we need to construct the Jacobian matrix

$$J = \begin{bmatrix} \frac{\partial y_1}{\partial \theta_1} & ... & \frac{\partial y_1}{\partial \theta_p} & \frac{\partial y_1}{\partial y_1^0} & ... & \frac{\partial y_1}{\partial y_n^0} & ... & \frac{\partial y_1}{\partial t_1} & ... & \frac{\partial y_1}{\partial t_k} \\ ... & ... & ... & ... & ... & ... & ... & ... & ... & ... \\ \frac{\partial y_n}{\partial \theta_1} & ... & \frac{\partial y_n}{\partial \theta_p} & \frac{\partial y_n}{\partial y_1^0} & ... & \frac{\partial y_n}{\partial y_n^0} & ... & \frac{\partial y_n}{\partial t_1} & ... & \frac{\partial y_n}{\partial t_k} \end{bmatrix}$$

Note this matrix can be split into three components: the derivatives with respect to θ, those with respect to y^0, and those with respect to t. In practice, we may only require one of these components. With careful coding, we only compute the derivatives we care about and keep our code efficient.

Here, a naive implementation of AD fails spectacularly. The main reason is that, in order to compute a solution, a numerical integrator takes thousands, if not millions of steps. The resulting expression graph is therefore extremely large, resulting in slow, memory intensive, and numerically unstable code.

Fortunately, there is structure in our problem which we can take advantage of. Denote $\alpha = (\theta, y^0, t)$ the vector which contains all the elements that require sensitivities. To obtain the needed derivatives, we augment the original ODE system with the state variables:

$$J_{ij} = \frac{\partial}{\partial \alpha_j} y_i$$

which correspond to the individual elements of the Jacobian matrix. The resulting *coupled* ODE system is

$$\begin{aligned}
y_1' &= f_1(y, t, \theta) \\
y_2' &= f_2(y, t, \theta) \\
&\ldots \\
\frac{\mathrm{d}}{\mathrm{d}t} \frac{\partial y_1}{\partial \theta_1} &= f_{1,1}(y, t, \theta) \\
&\ldots \\
\frac{\mathrm{d}}{\mathrm{d}t} \frac{\partial y_n}{\partial \theta_p} &= f_{n,p}(y, t, \theta) \\
&\ldots \\
\frac{\mathrm{d}}{\mathrm{d}t} \frac{\partial y_1}{\partial y_1^0} &= f_{n,p}(y, t, \theta) \\
&\ldots
\end{aligned}$$

By solving this system, we simultaneously produce the desired solution *and* the needed derivatives! Note further that writing this system does not require an analytical form for J_{nm}, but for

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}t} J_{ij} &= \frac{\mathrm{d}}{\mathrm{d}t} \frac{\partial}{\partial \alpha_j} y_i \\
&= \frac{\partial}{\partial \alpha_j} \frac{\mathrm{d}}{\mathrm{d}t} y_i
\end{aligned}$$

Intuitively, we may assume that since $\frac{\mathrm{d}}{\mathrm{d}t} y_i$ is known, we can analytically work out the derivative of the solution with respect to α_i. This requires some algebra; details can be found in section 13 of (Carpenter et al., 2015).

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

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

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

This strict signature requires some careful bookkeeping, as we need to sort out which variables are parameters or fixed data. As we saw in section 3.4.3, it may not be immediately obvious if a variable is a latent parameter.

4.4 Mixed solvers: combining analytical and numerical solutions

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

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

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

The mixed solver thus presents a trade-off: it increases the complexity of the integrant but reduces the number of states we integrate numerically. We conjecture the benefit usually outweights the cost, particularly with AD. The gain in efficiency varies on a case-by-case basis. For the example of a Friberg-Karlsson model, the average speed-up is $49 \pm 14\%$ (Margossian & Gillespie, 2017b).

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

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

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

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

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

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

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

  return dydt;
}
```

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

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

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

4.5 Rates

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

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

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

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

which gives us the solution:

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

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

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

4.6 Computing steady state solutions

This section is in part adapted from (Margossian, 2018), and a presentation we gave at the Stan Con 2018, in Asilomar, California.

At steady state, the dose input and the dose clearance cancel each other. In many cases, a patient reaches a steady state after repeatedly taking a treatment over a long period of time.

The brute force method to calculate a steady state would be to simulate a dosing regimen for an extended period of time. While this works well in certain settings, it is not very efficient, and sometimes not precise. A better approach is to exploit the definition of the steady state. Let us suppose a treatment repeats itself every τ hours. In figure 6, τ corresponds to the inter-dose interval. Then, once steady state has been reached, we have:

$$y(t_0) = y(t_0 + \tau) \tag{9}$$

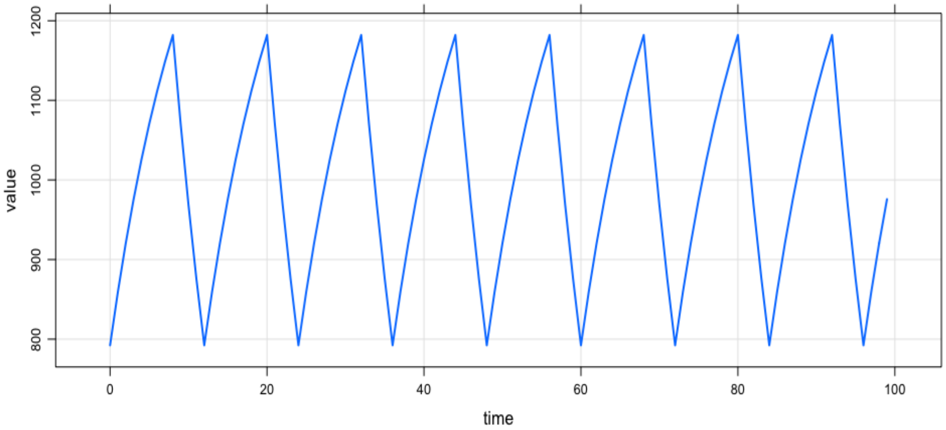

Figure 6: Simulated drug mass in a patient's gut at steady state. *Every 12 hours, a patient receives a drug administration. After being under this drug regiment for a certain period of time, the patient's system reaches a steady state. In this particular simulation, the drug administration corresponds to the infusion of 1200 mg over 8 hours. Figure generated with mrgsolve.*

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

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

4.6.1 Algebraic solver

`Stan` implements an algebraic solver, which uses Powell's hybrid method (Powell, 1970), based on an unsupported routine in `Eigen`. We find the method to work well in pharmacometrics. It is robust, but slow for certain problems. How well it scales in high dimensions is unclear. There is therefore interest in expanding the number tools available to solve algebraic equations, notably by incorporating less robust but faster methods such as Newton's solver.

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

As always, we need to compute the Jacobian matrix of the outputs with respect to the input parameters θ. Because the algebraic solver is an iterative algorithm, using standard AD is neither robust, nor efficient. As with ODEs, we can exploit the structure of the problem, in particular using the implicit function theorem. A standard computer experiment demonstrates the latter method is orders of magnitude faster than using standard AD.

Formally, given the function $f : X \times \Theta \mapsto Y$, the implicit function theorem states that if f satisfies certain regulatory conditions in a neighborhood of the starting point (x, θ) the roots are given by implicit functions, $x^* = h(y)$, where $h : \Theta \mapsto X$ satisfies $f(h(y), y) = 0$. This allows us to define the desired Jacobian as $\partial h(y)/\partial y$. For more details and some generalizations, see (Bell & Burke, 2008).

The implicit function theorem does not give us h, but it explicitly constructs the Jacobian of $h(y)$, which is all we need! Defining the two initial Jacobian matrices.

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

and

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

then the Jacobian of the roots is given by

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

provided J_x is invertible.

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

The invertibility condition for J_x places restrictions on the structure of the original problem. In particular, recall $X = \mathbb{R}^N$ and $\Theta = \mathbb{R}^K$, and let $Y = \mathbb{R}^M$. Then J_x is a $M \times N$ matrix and is invertible if and only if $M = N$. In other words, we need as many unknowns as equations in our algebraic system. Even then, J_x can become singular if the roots are not uniquely defined and other pathologies occur. For additional details, see chapter 17 of the *Stan Book*.

5 Open-source code for `Torsten`

The primary repository for `Torsten`, which includes documentations, examples, and installation files, is `https://github.com/metrumresearchgroup/Torsten`.

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

(1) write the procedure in `C++`, (2) expose the procedure to the language so users may use it in a Stan file. `Stan` interfaces with higher level languages, such as `R` and `Python`. `Torsten` exists as a forked version of `math` and `stan`. Other repos remain unchanged.

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

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

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

References

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2018). Automatic differentiation in machine learning: a survey. *CoRR*, *abs/1502.05767*. Retrieved from `http://arxiv.org/abs/1502.05767`

Bell, B. M., & Burke, J. V. (2008). Algorithmic differentiation of implicit functions and optimal values. , *64*. doi: https://doi.org/10.1007/978-3-540-68942-3_17

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

Betancourt, M. (2018, October). Towards a principled bayesian workflow. Retrieved from `https://betanalpha.github.io/assets/case_studies/principled_bayesian_workflow.html`

Carpenter, B., Gelman, A., Hoffman, M., Lee, D., Goodrich, B., Betancourt, M., . . . Riddel, A. (2017). Stan: A probabilistic programming language. *Journal of Statistical Software*. doi: 10.18637/jss.v076.i01

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

Gabry, J., Simpson, D., Vehtari, A., Betancourt, M., & Gelman, A. (2017). Visualization in bayesian workflow. *Royal Journal of Statistics, section A*, *182*, 1 -14.

Gelman, A., Simson, D., & Betancourt, M. (2017). The prior can generally only be understood in the context of the likelihood. Retrieved from `arXiv:1708.07487` doi: 10.3390/e19100555

Gillespie, W. R., & Zhang, Y. (2018, October). Torsten: Stan functions for pharmacometric (pmx) applications. validation and new functions. , *45*.

Griewank, A., & Walther, A. (2008). Evaluating derivatives: Principles and techniques of algorithmic differentiation. *Society for Industrial and Applied Mathematics (SIAM)*, *2*. doi: https://doi.org/10.1137/1.9780898717761

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

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

Margossian, C. C. (2018, January). Computing steady states with stan's nonlinear algebraic solver. In *Stan conference 2018 california.*

Margossian, C. C., & Gillespie, W. R. (2016, October). Stan functions for pharmacometrics modeling. In *Journal of pharmacokinetics and pharmacodynamics* (Vol. 43).

Margossian, C. C., & Gillespie, W. R. (2017a, January). Differential equations based models in stan. In *Stan conference.* http://mc-stan.org/events/stancon2017-notebooks/stancon2017-margossian-gillespie-ode.html.

Margossian, C. C., & Gillespie, W. R. (2017b, October). Gaining efficiency by combining analytical and numerical methods to solve odes: Implementation in stan and application to bayesian pk/pd. In *Journal of pharmacokinetics and pharmacodynamics* (Vol. 44).

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

Neal, R. M. (2010). *Mcmc using hamiltonian dynamics.* Chapman & Hall / CRC Press.

Powell, M. J. D. (1970). A hybrid method for nonlinear equations. In P. Rabinowitz (Ed.), *Numerical methods for nonlinear algebraic equations.* Gordon and Breach.

Rowland, T., & Weisstein, E. W. (n.d.). *Matrix exponential.* MathWorld. Retrieved from `http://mathworld.wolfram.com/MatrixExponential.html`

Sidje, R. B. (1998, March). Expokit: a software package for computing matrix exponentials. *ACM Transactions on Mathematical Software (TOMS)*, *24*(1), 130 - 156.