Journal of Statistical Software

March 2019, Volume 88, Issue 10. *doi: 10.18637/jss.v088.i10*

Dynamic Modeling, Parameter Estimation, and Uncertainty Analysis in R

Daniel Kaschek
University of Freiburg

Wolfgang Mader
University of Freiburg

Mirjam Fehling-Kaschek
University of Freiburg

Marcus Rosenblatt
University of Freiburg

Jens Timmer
University of Freiburg

Abstract

In a wide variety of research fields, dynamic modeling is employed as an instrument to learn and understand complex systems. The differential equations involved in this process are usually non-linear and depend on many parameters whose values determine the characteristics of the emergent system. The inverse problem, i.e., the inference or estimation of parameter values from observed data, is of interest from two points of view. First, the existence point of view, dealing with the question whether the system is able to reproduce the observed dynamics for any parameter values. Second, the identifiability point of view, investigating invariance of the prediction under change of parameter values, as well as the quantification of parameter uncertainty.

In this paper, we present the R package **dMod** providing a framework for dealing with the inverse problem in dynamic systems modeled by ordinary differential equations. The uniqueness of the approach taken by **dMod** is to provide and propagate accurate derivatives computed from symbolic expressions wherever possible. This derivative information highly supports the convergence of optimization routines and enhances their numerical stability, a requirement for the applicability of sophisticated uncertainty analysis methods. Computational efficiency is achieved by automatic generation and execution of C code. The framework is object-oriented (S3) and provides a variety of functions to set up ordinary differential equation models, observation functions and parameter transformations for multi-conditional parameter estimation.

The key elements of the framework and the methodology implemented in **dMod** are highlighted by an application on a three-compartment transporter model.

Keywords: dynamic models, parameter estimation, code generation, maximum likelihood, uncertainty analysis.

1. Introduction

Dynamic systems modeled by ordinary differential equations (ODEs) are found in several research fields, such as physics, biology or finance. In all these fields, models link theoretical concepts and empirical evidence. Single mechanisms or single processes of a complex system are represented by respective terms in the equations of a dynamic model. Parameter estimation can then identify those processes which are crucial to explaining the observation. In that sense, parameter estimation can be employed as an instrument to *understand* complex systems. Once the link between observation and model has been established by the estimated parameters, questions about their identifiability arise. The parameter space needs to be explored in order to analyze whether the estimate is unique and to determine confidence bounds.

Although the problem of parameter estimation in non-linear ODE models is highly relevant and at the heart of statistical computing, there are currently not more than four R packages published on the topic on the Comprehensive R Archive Network (CRAN), namely **FME** (Soetaert and Petzoldt 2010), **nlmeODE** (Tornoe 2012), **mkin** (Ranke, Lindenberger, and Lehmann 2019) and **scaRabee** (Bihorel 2014). All packages have in common that they are built upon the **deSolve** package (Soetaert, Petzoldt, and Setzer 2010). Taking a broader perspective on the topic of dynamic modeling and inference, we find more packages, dealing with, e.g., discrete-time and continuous-time stochastic systems (Hooker, Ramsay, and Xiao 2016) or the statistical inference of partially observed Markov processes (King *et al.* 2017).

The packages **mkin** and **FME** support ODEs defined by compiled code while **mkin** also provides tools to autogenerate the C code and compile it using the **inline** package (Sklyar, Murdoch, Smith, Eddelbuettel, François, and Soetaert 2018). For model fitting and uncertainty analysis, **mkin** fully resorts to the functionality of **FME**. Concerning model fitting, **FME** and **nlmeODE** (with **nlme** in the background) use deterministic derivative-based optimizers by default, i.e., either Levenberg-Marquardt or Newton methods provided by `nls.lm()`, `optim()` or `nlmimb()`. Although all these optimizers support gradient or Hessian information as input, only the **nlmeODE** package provides an option to augment the ODE by its sensitivity equations to generate derivates for the residuals. By default, sensitivities are computed by finite differences. Last but not least, the **scaRabee** package uses the Nelder-Mead optimization algorithm which is derivative-free but generally needs more iterations until convergence compared to derivative-based methods. In all packages, uncertainty analysis is by default based on the variance-covariance matrix, i.e., non the inverse Hessian matrix of the least squares function. For non-linear models, this method provides a good approximation only if parameters are identifiable and the data is highly informative. If these conditions are not met, more sophisticated methods like, e.g., Markov chain Monte Carlo (MCMC) sampling, implemented in **FME**, are required. The strength of **scaRabee** and especially **nlmeODE** is multi-conditional fitting. This means that the same model with the same parameters but different forcings to reflect experimental conditions is fitted simultaneously to the condition-specific data sets. In the context of mixed-effects modeling as provided by **nlmeODE**, parameters can be grouped in fixed effects (parameters are the same in all conditions) and random effects (parameters are different between conditions).

In this paper we present **dMod**, an R package on dynamic modeling and parameter estimation. The package has grown over the past years and reflects many developments and lessons learned from our research projects with time-resolved experimental data from systems biology. They

	dMod	FME	nlmeODE	scaRabee	mkin
Facilitated set-up of ODE models with automated C code generation for fast simulation of model predictions and model sensitivities	+	(+)[1]	−	−	(+)[1]
Flexible set-up of general parameter transformations (explicit or implicit) and observation functions, allowing for the implementation of multiple experimental conditions similar to mixed-effects modeling	+	−	(+)[2]	(+)[2]	−
Parameter estimation based on trust-region optimization of the negative log-likelihood, making use of the sensitivity equations of the dynamic system and of symbolic derivatives of the observation- and parameter transformation functions	+	−	−	−	−
Identifiability and uncertainty analysis based on the profile-likelihood method to determine confidence intervals for parameters and predictions	+	(−)[3]	−	−	−

Table 1: Overview of the **dMod** core functionality and comparison with other packages. 1 = Only ODE, no sensitivity equations generated. 2 = Mixed-effects modeling allows to define parameters in a condition-specific manner. 3 = Other methods used, e.g., the collinearity method (Brun *et al.* 2001).

form the basis of the development of **dMod** and are found as references in the following paragraphs. The aim and core functionality of **dMod** is summarized in Table 1, first column. A comparison with the other packages is shown in the remaining columns.

The **dMod** package deals with noise in the observation but not with noise in the dynamics. Therefore, the application of **dMod** is restricted to systems that are described or can be approximated by a deterministic set of differential equations. The core functionality is extended by two symbolic methods implemented in Python and interfaced via the **rPython** package: identifiability and observability analysis based on Lie-group symmetries (Merkt, Timmer, and Kaschek 2015) and steady-state constraints for parameter estimation (Rosenblatt, Timmer, and Kaschek 2016).

The implementation of capabilities to generate and propagate derivatives on a compiled-code level distinguishes **dMod** from other modeling frameworks as discussed above. Most of the standard optimization routines implemented in R need derivative information. However, the computation of derivatives in the context of ODE models holds some pitfalls. The accuracy of sensitivities obtained from finite differences can be insufficient because the step control of the integrator presents an additional source of numeric inaccuracy. Even for numeric methods circumventing this problem, e.g., complex-step derivatives (Squire and Trapp 1998), the use of sensitivity equations is still beneficial judging from the accuracy vs. computational cost ratio. See Raue *et al.* (2013b) for a comparison of methods.

Another distinguishing feature of **dMod** is the handling of non-identifiability of parameters,

a phenomenon that occurs frequently in the context of parameter estimation in dynamic systems. In some cases, non-identifiability has structural reasons. The differential equations bear certain symmetries which can or cannot be broken, depending on the structure of the observation. A functional relationship between parameters that leaves the observation invariant is the consequence of the latter. In other cases, the data allows the determination of a unique optimum but other solutions, although worse, cannot be statistically rejected. The **dMod** package deals with parameter identifiability and parameter uncertainty by the profile-likelihood method (Murphy and Van der Vaart 2000; Raue *et al.* 2009; Kreutz, Raue, Kaschek, and Timmer 2013). This method has proven especially useful in the case of non-identifiable parameters where results obtained from both the quadratic approximation by the variance-covariance matrix and MCMC sampling can be misleading (Raue, Kreutz, Theis, and Timmer 2013a). Besides parameter uncertainties, the profile likelihood method allows the estimation of prediction uncertainty (Kreutz, Raue, and Timmer 2012). It therefore supports the planning of new informative experiments (Raue, Kreutz, Maiwald, Klingmüller, and Timmer 2011) or suggests possible model reductions (Maiwald *et al.* 2016).

The key methods implemented in **dMod** are illustrated in great detail on a dynamic model of bile acid flow. The example is a showcase of how modeling *is* a dynamic process, using the analysis tools implemented in **dMod** to predict, plan new experiments, combine data from different experiments and include non-linear parameter constraints to improve parameter identifiability. In this way, we demonstrate that dMod is a fully developed, flexible modeling environment, which is not only fast (thanks to compiled code), but also reliable (thanks to symbolic derivatives) and accurate (thanks to advanced statistical methods).

The paper is organized as follows: Section 2 introduces the mathematical set-up of dynamic modeling, symmetries in dynamic systems, parameter estimation by the maximum-likelihood method and the profile likelihood. Section 3 discusses the implementation and design principles behind the **dMod** software. The functionality of **dMod** is presented in Section 4 on the example of bile acid flow in a three-compartment model. Finally, Section 5 discusses the two Python extensions shipped with **dMod**.

The **dMod** package is available on the Comprehensive R Archive Network at `http://CRAN.R-project.org/package=dMod`. The project is hosted on GitHub at `https://github.com/dkaschek/dMod`, where more information about system requirements and installation is available. **dMod** is licensed under the GPL-3 license.

2. Theoretical background

2.1. Dynamic models and model sensitivities

Dynamic models describe systems with states x, usually quantifying the involved species, their interaction and evolution over time. The time evolution of the states is expressed via a set of ODEs, $\dot{x} = f(x)$. Although constituting quite a special class of dynamic systems, chemical reaction networks formulated by the law of mass action as considered here allow for surprisingly general applications. Typical ODE examples derived from the law of mass action are:

- $\dot{x}_A = k_p$, constant production of x_A ($\emptyset \to A$)
- $\dot{x}_C = k_c\, x_A\, x_B = -\dot{x}_A = -\dot{x}_B$, complex formation ($A + B \to C$)

- $\dot{x}_C = -k_d\, x_C$, proportional degradation of x_C $(C \to \emptyset)$.

A general chemical reaction

$$\nu_1 X_1 + \cdots + \nu_n X_n \longrightarrow \nu_1' X_1 + \cdots + \nu_n' X_n \tag{1}$$

for species X_j, $j = 1, \ldots, n$, and stoichiometric coefficients $\nu_j, \nu_j' \in \mathbb{N}$ translates into the ODE

$$\dot{x}_j = \underbrace{k_0 \left(\prod x_i^{\nu_i} \right)}_{w_0(x,k_0)} (\nu_j' - \nu_j), \tag{2}$$

where k_0 is the reaction rate. When several reactions $1, \ldots, r$ are involved, all contributions have to be summed up which can be expressed by the matrix equation

$$\dot{x} = Sw(x, k). \tag{3}$$

Here, S denotes the $n \times r$ stoichiometry matrix where each column corresponds to one reaction and the coefficients within one column are the values $\nu_j' - \nu_j$ of the corresponding reaction. The vector $w(x, k)$ denotes the vector of r reaction fluxes where the vectors x and k correspond to states and rate constants, respectively. Equation 3 also holds for general ODEs that do not follow from mass action kinetics. Moreover the system may be explicitly time-dependent and may contain forcings $u(t)$ which, e.g., describe the external stimulation of the system. A general form of the dynamic model is therefore given by

$$\dot{x} = f(x, k, u, t) = Sw(x, k, u, t), \qquad \text{with } x(t = 0) = x_0. \tag{4}$$

The solution $x(t, \theta)$ for a given parameter vector $\theta = (k, x_0)$ is called model prediction. It is generally assumed that the stoichiometry matrix S of the system is known. Besides the model predition itself, also the sensitivity of the prediction to changes in the parameter values is of interest. The sensitivities $s_i = \frac{\partial x}{\partial \theta_i}$ satisfy the *sensitivity equations*

$$\dot{s}_i = \frac{\partial f}{\partial x} s_i + \frac{\partial f}{\partial \theta_i}, \qquad \text{with } s_i(t = 0) = \frac{\partial x_0}{\partial \theta_i}, \tag{5}$$

a system of ordinary differential equations that in general depend on the states x and, therefore, need to be solved jointly with the original ODE.

Often, the experimentally observed quantities y do not directly correspond to the species x described by the ODE, but are obtained via an observation function $g : \mathbb{R}^n \to \mathbb{R}^m$,

$$y = g(x, c) \tag{6}$$

with the observation parameters c. Examples for observation functions are scaling and offset transformations, $y = c_s \cdot x + c_o$, or the measurement of a superposition of species, $y_i = \sum_j c_j x_j$. The model prediction for the observed states is obtained by evaluating the observation function on the solution of the ODE. Following the chain rule of differentation also the sensitivities of the observed states $\frac{\partial y}{\partial \theta_i} = \frac{\partial g}{\partial x} \frac{\partial x}{\partial \theta_i} + \frac{\partial g}{\partial \theta_i}$ are obtained. Here, the parameter vector θ has been augmented by the observation parameters, $\theta = (k, x_0, c)$.

The estimation of the parameters θ given the observation $y(t)$ is addressed in the next section.

2.2. Maximum-likelihood method

Parameter estimation is a common task in statistics. It describes the process of inferring parameter values or parameter ranges of a statistical model based on observed data. Over decades, appropriate estimators have been developed for different problem classes. The principle of maximum likelihood allows to derive an estimator which is especially suited for applications where the distribution of the measurement noise is known. This knowledge about the structure of the noise constitutes additional information that makes the maximum-likelihood estimator (MLE) *efficient*, i.e., from all unbiased estimators the MLE has the lowest variance. Other properties of maximum-likelihood estimation are *consistency*, i.e., the estimated parameter value approaches the true parameter value in the limit of infinite data-sample size, and *asymptotic normality*, i.e., in the limit of infinite data the MLE follows a multivariate normal distribution. See Azzalini (1996) for an introduction to likelihood theory.

Maximum-likelihood estimation is based on the maximization of the likelihood function $L(\theta) = \phi(y^D|\theta)$. Here, ϕ is the joint probability density for a vector of observations y, c.f. Equation 6, evaluated at the point $y = y^D$, the vector of data points. The distribution ϕ depends parametrically on model parameters θ. The maximum-likelihood estimator $\hat{\theta}$ is defined as

$$\hat{\theta} := \arg\max_{\theta} L(\theta),$$

meaning that $\hat{\theta}$ is an extremum estimator. Depending on the model class and the probability distribution ϕ, maximization of the likelihood can be a challenge beyond the scope of analytical methods. Numerical optimization methods help to solve the maximization problem in practical applications.

The easiest and one of the most frequent situations is when ϕ follows a normal distribution, $\phi \sim \mathcal{N}(y(\theta), \Sigma)$. When measurements are statistically independent, the variance-covariance matrix $\Sigma = \mathrm{diag}(\sigma_1^2, \ldots, \sigma_N^2)$ is diagonal and ϕ factorizes. The likelihood function reads

$$L(\theta) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{(y_i(\theta)-y_i^D)^2}{2\sigma_i^2}}. \tag{7}$$

Taking twice the negative logarithm, Equation 7 turns into

$$l(\theta) = -2\log L(\theta) = \sum_i \left[\left(\frac{y_i(\theta) - y_i^D}{\sigma_i} \right)^2 + \log(2\pi\sigma_i^2) \right], \tag{8}$$

thus converting the maximization of $L(\theta)$ to a minimization of $l(\theta)$. Assuming that the data uncertainties σ_i are known, Equation 8 is the weighted least-squares function shifted by a constant. For unknown σ_i, Equation 8 can be optimized with respect to θ and σ jointly, yielding the MLE $\hat{\theta}' = (\hat{\theta}, \hat{\sigma})$.

2.3. Non-linear optimization

Numerical optimization is a diverse field with as many algorithms as there are optimization problems around. The **dMod** package supports derivative-based methods, and in particular the Newton method.

Optimization by the Newton method attempts to iteratively find the root of the gradient $\nabla l(\theta)$ by the recursion

$$\theta^{(n+1)} = \theta^{(n)} - Hl(\theta)^{-1}\nabla l(\theta), \tag{9}$$

where $Hl(\theta)$ denotes the Hessian, i.e., the matrix of second derivatives of $l(\theta)$. The fact that for normally distributed noise the log-likelihood is a least squares function, has a big advantage: gradient and (approximate) Hessian can be computed from first-order derivatives (Press, Teukolsky, Vetterling, and Flannery 1996). Since the second-order contributions to the Hessian can be neglected, gradient and Hessian read

$$\nabla l(\theta) = 2r(\theta)J(\theta) \qquad Hl(\theta) \approx 2J(\theta)^T J(\theta), \tag{10}$$

with the weighted residual vector $r_i(\theta) = \frac{y_i(\theta) - y_i^D}{\sigma_i}$ and its first derivative, the Jacobian matrix

$$J_{ij}(\theta) = \frac{\partial r_i}{\partial \theta_j}(\theta) = \frac{1}{\sigma_i}\frac{\partial y_i}{\partial \theta_j}(\theta). \tag{11}$$

As indicated by eqs. (10)-(11), first order model sensitivities $\frac{\partial y_i}{\partial \theta_j}$ are sufficient to compute gradient and Hessian in good approximation.

The Newton recursion, Equation 9, converges in one iteration if $l(\theta)$ is a quadratic form. However, $l(\theta)$ is only quadratic if the model $y(\theta)$ is linear in θ which is usually not the case for ODE models. On the other hand, each smooth function, no matter if quadratic or not, can be approximated by a quadratic function based on its Taylor series. Thus, $l(\theta)$ can be approximated by a quadratic function in at least a small region around θ. The idea of trust-region optimization is to confine a Newton step to a region, the *trust region*, where the quadratic approximation holds (Wright and Nocedal 1999). In each iteration, the trust-region radius is adjusted and the optimization problem restricted to the trust region is solved. For parameter estimation in ODE models this has the additional advantage that parameter changes are constricted and optimization steps into parameter regions where the ODE cannot be numerically solved any more become less frequent. This makes trust-region optimization the method of choice for **dMod**.

2.4. Parameter uncertainty analysis

Parameter estimates $\hat{\theta}$ are obtained by non-linear optimization of the log-likelihood function $l(\theta)$. Although being a function of the parameters, the log-likelihood depends on the data, too. Consequently, "equivalent" random realizations of the data would lead to different parameter estimates $\hat{\theta}$. Given $\hat{\theta}$ for one random realization of the data, the question is, at which significance level we can reject other parameters θ. This leads to the related question of parameter convidence intervals.

A useful tool to derive confidence intervals beyond the scope of Fisher Information Matrix is the profile likelihood (Venzon and Moolgavkar 1988; Murphy and Van der Vaart 2000). Consider a parameter of interest, θ_i, being one of the parameters in the vector θ. Furthermore, let $c_\tau(\theta) = \theta_i - \tau = 0$ be a parameter constraint. Then, extending the likelihood with the constraint via a Lagrange multiplier λ yields

$$\begin{aligned} \mathrm{pl}_i: \quad & \mathbb{R} \longrightarrow \mathbb{R} \\ & \tau \longmapsto \min_{(\theta,\lambda)} \left[l(\theta) + \lambda c_\tau(\theta) \right] \end{aligned} \tag{12}$$

which is called the profile likelihood of the i^{th} parameter. The path $\tau \mapsto (\hat{\theta}_\tau, \hat{\lambda}_\tau)$ along the minimizing parameters is called the profile likelihood path. Hence, the profile likelihood pl_i returns the minimal log-likelihood under the constraint $\theta_i \stackrel{!}{=} \tau$. By construction, the inequality $D_i := \text{pl}_i(\tau) - \text{pl}_i(\hat{\theta}_i) \geq 0$ holds for all τ, assuming equality at least for $\tau = \hat{\theta}_i$. Interpreting the unconstrained model as the null model, which we assume to be true, and that with θ_i fixed to τ as the alternative model, the value D is twice the log of the likelihood-ratio between those. Hence, a likelihood-ratio test can be performed to accept or reject the alternative model at a given confidence level. To compute the corresponding threshold, we assume a sufficiently large sample size to apply Wilks' theorem according to which the thresholds are the quantiles of the χ^2 distribution with one degree of freedom. Consequently, the 68%/90%/95%-confidence intervals of θ_i are those values of τ for which $D_i(\tau) \leq 1$, 2.71 and 3.84, respectively.

The reason to formulate the profile likelihood by Lagrangian multipliers is that we can directly use Equation 12 to derive the profile likelihood path $(\hat{\theta}_\tau, \hat{\lambda}_\tau)$ with respect to τ:

$$\frac{\mathrm{d}}{\mathrm{d}\tau}\begin{pmatrix}\hat{\theta}_\tau \\ \hat{\lambda}_\tau\end{pmatrix} = \begin{pmatrix} Hl(\hat{\theta}_\tau) & \nabla c_\tau(\hat{\theta}_\tau) \\ \nabla c_\tau(\hat{\theta}_\tau)^T & 0 \end{pmatrix}^{-1} \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \tag{13}$$

Equation 13 forms the basis of what is implemented in **dMod** for computing the profile likelihood.

The profile likelihood approach can be applied to compute confidence intervals for model predictions, too, see Kreutz *et al.* (2012); Hass, Kreutz, Timmer, and Kaschek (2016).

3. Implementation and design principles

The guiding idea behind the implementation of **dMod** is to provide a class structure for models, predictions, observations and parameter transformations that allows a flexible combination of many experimental conditions in one objective function to fit models to data. This flexibility is achieved by two concepts: (1) concatenation of functions by the `"*"` operator and (2) stacking of functions representing different conditions by the `"+"` operator. The handling and propagation of derivatives is part of the classes and happens in the background. Methods for the generic `print()`, `plot()` and `summary()` functions are implemented for most outputs.

3.1. Model formulation

A system of ODEs $\dot{x} = f(x, p, u, t)$ is represented by a named `character` vector of symbolic expressions, the right-hand side of the ODE, involving symbols for the states x, the parameters p, the forcings u and the keyword `time` for t. The names of the vector are the state names. The class provided by **dMod** for such objects is the `eqnvec` class which checks whether the character can be parsed and thus, can be interpreted as an equation.

dMod provides several ways to define the differential equations. An `eqnvec` of equations can be explicitly formulated, analogously to a `c()` command. Especially for chemical reactions it can be tedious to keep track of all gain and loss terms. Therefore, **dMod** provides also the `eqnlist` class which encodes the ODE by a list of (1) state names, (2) rate expressions, (3) compartment volumes and (4) the stoichiometric matrix. Each reaction flux, c.f. Equation 3, occurs only once and the gain and loss is represented by the coefficients in the stoichiometric matrix. **dMod** supports the read-in of a csv file with the stoichiometric matrix and rate expressions and provides a function `addReaction()` to construct an `eqnlist` object step-by-step or add further reactions to an already existing `eqnlist`. An `eqnlist` is converted

to (differential) equations, i.e., an `eqnvec`, by `as.eqnvec()`. The conversion does all the bookkeeping of gain and loss terms and volume ratios due to compartment transitions.

dMod makes use of derivatives wherever possible. It is one of the core functionalities of the **cOde** package (Kaschek 2019) upon which **dMod** is based to augment a system of ODEs by its sensitivity equations. If r parameters are involved and the system consists of n states, the number of equations grows as quick as $n^2 + nr$. Solving the equations can be considerably accelerated by utilizing compiled code. **dMod** provides the `odemodel` class. An object of class `odemodel` is generated from an `eqnlist` or `eqnvec`. In the background, C code for the ODE and the combined system of ODE and sensitivity equations is generated, written to the working directory and compiled. The `odemodel` object keeps track of the shared objects. It is the basis of a prediction function $x(t, p)$ generated by the `Xs()` command.

3.2. Prediction functions

dMod seeks to stay close to the mathematical formulation, i.e., `x <- Xs(myodemodel)` will indeed return an R function, an object of class `prdfn`, which expects arguments `times` and `pars` and turns them into a model prediction. The letter `"s"` in `"Xs"` refers to sensitivities, i.e., the solution $x(t)$ is returned, the sensitivities $\frac{\partial x}{\partial p}(t)$ are returned, too.

Besides parameters, the prediction might also depend on forcings $u(t)$ and sudden events, e.g., setting states to 0 at a predefined point in time. Since forcings and events are by default fixed, they are defined together with the prediction function, `x <- Xs(odemodel, forcings = myforcings, events = myevents)`. The definitions of both, forcings and events, are compatible with the way they are defined in **deSolve**.

3.3. Observation functions

An observation function is a function $g(x, p_{\text{obs}}, t)$ that evaluates the solution $x(t)$ together with additional (observation) parameters p_{obs}. The function can explicitly depend on time t. Similar to the ODE case, an observation function can be expressed as a character vector with names corresponding to the names of the observables and equations involving symbols for the states, parameters and the keyword `time`. An observation function is defined as an `eqnvec` and turned into an R function via `Y()`.

It is one of the fundamental concepts of **dMod** to allow concatenation of functions via the `"*"` operator. The mathematical formulation $y = g \circ x$, i.e., $y(t, p) = (g \circ x)(t, p) := g(x(t, p), p, t)$ becomes `y = g * x` in R. To obtain derivatives $\frac{\partial y}{\partial p}$, the chain rule is applied: $\frac{\partial y}{\partial p} = \frac{\partial g}{\partial x}\frac{\partial x}{\partial p} + \frac{\partial g}{\partial p}$. This means that `Y()` needs to be informed which symbols are states and parameters to generate the corresponding expressions $\frac{\partial g}{\partial x}$ and $\frac{\partial g}{\partial p}$. The observation function `g` takes care of computing theses derivatives from the symbolic expressions and doing the matrix multiplication with the sensitivities $\frac{\partial x}{\partial p}$ from the prediction function. Evaluation of symbolic expressions can become inefficient in R. Therefore, the observation function is usually translated into a C code and compiled.

When observation- and prediction functions are concatenated, the result is a prediction function, e.g., `y = g * x` is the R function computing values $y(t)$ from the arguments `times` and `pars` via evaluation of the ODE and subsequent evaluation of the observation function. Two observation functions can be concatenated, too, again yielding an observation function.

3.4. Parameter transformations

Parameter transformations are the key element of **dMod** to formulate different kinds of constraints and allow the combination of several experimental/modeling conditions in one parameter vector.

In principle, a parameter transformation $p = \Phi(\theta)$ is a (differentiable) function connecting **inner** parameters p with **outer** parameters θ. The rationale behind the distinction of inner and outer parameters is that the vector p usually desribes those parameters defined in the model equations. The outer parameters θ refer to a convenient parameterization by which the model parameters are computed. Examples are a log-transform of the inner parameters, $\theta = \log(p) \Leftrightarrow p = \Phi(\theta) = e^\theta$, or parameter constraints like $(p_1, p_2) = (\theta_1, \theta_1 + \theta_2)$.

An R function of class `parfn` is produced by the `P()` command. Transformations can either be formulated explicitly or implicitly. In the explicit case, the function $p = \Phi(\theta)$ corresponds to an `eqnvec` whose names are the names of the inner parameters and entries are equations with symbols for the outer parameters. An implicit transformation has the form $f(p = \Phi(\theta), \theta) = 0$. In this case, f is expressed by an `eqnvec` with equations containing symbols for p and θ and the names of the `eqnvec` are the symbols for p.

Similar to prediction- and observation functions, parameter functions not only return parameter values but the Jacobian of the transformation, too. Exploiting the chain rule, the derivatives are propagated, allowing to define `y = g * x * p` which is a function returning $y(t, \theta)$ and $\frac{\partial y}{\partial \theta}(t)$, i.e., `g * x * p` is a prediction function.

Let `g`, `x`, `p1` and `p2` be an observation-, a prediction- and two parameter transformation functions. Then `p1 * p2` is a parameter transformation function, `x * p1` is a prediction function and `g * p1` is an observation function.

3.5. Multi-conditional prediction

A set of parameter values, forcings and events captures a certain condition in which we find the modeled system. Manipulating the system, single model parameters, forcings or events need to be changed to account for the manipulation. It is a typical approach in systems inference to systematically perturb small parts of a modeled system to reveal information about the processes.

The aim of **dMod** is to allow for "simultaneous" predictions under several conditions and compare these predictions to the corresponding experimental data sets to estimate model parameters. Different experimental conditions are typically expressed by the fact that some parameters are different between conditions whereas others are common to all conditions. This situation occurs, e.g., if perturbation experiments are performed, affecting only few parts of a system. Mathematically speaking, we want to construct a parameter transformation

$$\begin{aligned} \Phi: \quad \mathbb{R}^p &\longrightarrow \bigoplus_{i=1}^{n} \mathbb{R}^q \\ \theta &\longmapsto (\Phi_1(\theta), \ldots, \Phi_n(\theta)) \end{aligned} \tag{14}$$

where $i = 1, \ldots, n$ corresponds to the different conditions. If all parameters are shared throughout all conditions, then $p = q$. If, however, all parameters are distinct, then $p = n \cdot q$. Perturbation experiments correspond to a situation where $p \gtrsim q$.

The **dMod** package allows to define transformations Φ, see Equation 14, by the `"+"` operator:[1]

```
p <- P(eqnvec1, condition = "one") + P(eqnvec2, condition = "two")
```

All symbols from `eqnvec1` and `eqnvec2` are collected and their union constitutes the symbols of θ. The evaluation `p(theta)` returns a list of length n (in the example $n = 2$) with inner parameters. The `"+"` operator can be applied consecutively to add conditions.

Prediction- and observation functions are generalized to multiple conditions by the `"+"` operator, too. Mathematically speaking, they become

$$
\begin{aligned}
x: \quad & \mathbb{R} \times \bigoplus_{i=1}^{n} \mathbb{R}^q \longrightarrow \bigoplus_{i=1}^{n} \mathbb{R}^m \\
& (t, p_1, \ldots, p_n) \longmapsto (x_1(t, p_1), \ldots, x_n(t, p_n)) \\
g: \quad & \bigoplus_{i=1}^{n} \mathbb{R}^m \times \bigoplus_{i=1}^{n} \mathbb{R}^q \times \mathbb{R} \longrightarrow \bigoplus_{i=1}^{n} \mathbb{R}^s \\
& (x_1(t), \ldots, x_n(t), p_1, \ldots, p_n, t) \longmapsto (g_1(x_1(t), p_1, t), \ldots, g_n(x_n(t, p_n), p_n, t)).
\end{aligned}
$$

In words, if prediction- or observation functions are defined for different conditions then they expect condition-specific inputs which are evaluated by the matching functions. Examples for condition-specific prediction functions typically involve different forcings or events. Observation functions can, e.g., differ between different measurement techniques. In all these cases, the different prediction functions $x_1, \ldots, x_n$ or observation functions $g_1, \ldots, g_n$ are defined, referencing the condition, and combined by the `"+"` operator analogously to `p`.

All commands, `P()`, `Xs()`, `Y()`, etc. can be executed with `condition = NULL`. In that case, the corresponding returned function is generic and, if called for different conditions, the same identical function is evaluated with the condition-specific input. The other way round, if a function, say, `x` is defined for several conditions, its prediction can be evaluated only for a subset of conditions by `x(times, pars, conditions = myconditions)`.

3.6. The data structure

In **dMod** different experimental conditions are handled by lists. Parameter transformations, prediction- and observation functions stacked by the `"+"` operator return list objects. On the other hand, `data.frame`s as they are used for linear modeling, mixed-effects modeling or plotting with **ggplot2** are highly convenient to organize the data. The class `datalist` provides the interface between **dMod**'s list structures and `data.frame` objects. A `datalist` is a list of data frames with identical structure: observable names, time points, measured values and measurement uncertainty.

Objects of class `datalist` are usually generated by the `as.datalist()` command from a `data.frame`. The factor variables in the data frame to be used as generators for the unique condition names can be passed by the `split.by` argument. The resulting list of data frames has an additional attribute `"condition.grid"`, a translation table between the condition names and the original factor variables which can be used for specification of parameter transformations or augmentation of predictions by descriptive columns.

[1] We chose the `"+"` sign in analogy to the $\oplus$ formulation used for the direct sum of vector spaces in Equation 14.

3.7. Objective function

The aim of **dMod** is parameter estimation. The objective function to be minimized for this purpose is twice the negative log-likelihood derived from the normal distribution, see Equation 8. The objective function is produced by the command `normL2(data, prdfn)` where `data` is a `datalist` object and `prdfn` is a prediction function. By default, it is expected that the datalist contains non-zero measurment uncertainty values in the `sigma` column corresponding to the σ_i in Equation 8. However, it is also possible to use Equation 8 to estimate σ. To this end, an error model needs to be defined as an `eqvec` object. The names are expected to be a subset of the obervable names and the equations define the expected σ of the observable as a function of error model parameters and the observables themselves. The `eqnvec` object is translated into an "observation function" by the `Y()` command. Finally, the objective function is generated by `normL2(data, prdfn, errfn)` with the datalist and prediction function as before and an object `errfn` of class `prdfn` being the error model function.

The objective function is the final link connecting the chain of parameter transformations, prediction- and observation functions to observations. It collects all derivative information and besides the objective value it also computes gradient and Hessian. The standard optimizer employed within **dMod** is the `trust()` optimizer from the **trust** package.

Objective functions can be added by the `"+"` operator, meaning the objective values, gradients and Hessians are accumulated by summation. Besides the standard function `normL2()`, **dMod** provides several other functions returning objective functions, e.g., `constraintL2()` or `datapointL2()`. They allow to define quadratic parameter priors or treat data points as parameters, respectively, as shown in Section 4. Thus, a typical objective function used for parameter estimation could be

```
obj <- normL2(data, g * x * (p1 + p2 + p3)) + constraintL2(mu, sigma)
```

4. Three-compartment model of bile acid transport

The following model is a simplified dynamic model on bile acid transport published by Kaschek, Sharanek, Guillouzo, Timmer, and Weaver (2017). Bile acids are produced in the liver. They are necessary for the digestion of fat and oil. In the liver, bile acids are taken up in hepatocytes (liver cells) by specific transporter molecules. Clearance occurs either via canalicular or sinusoidal export, i.e., export to the bile transportation system of bile canaliculi or the intercellular space.

To study bile clearance, experiments with hepatocytes or hepatocyte-derived cell lines are performed *in-vitro*. Cells in a Petri dish stick together to a monolayer of cells, forming bile canaliculi in between cells. They adhere to the dish and are surrounded by a buffer providing the cells with nutrients. A radioactive label allows to measure the bile acid *taurocholic acid* (TCA).

For the mathematical description of bile flow, a three-compartment differential equation model is used. TCA is pipetted into the buffer compartment (`TCA_buffer`) from where it is transported into the cells (`TCA_cell`). Intracellular TCA is exported back to the buffer or the canalicular compartment (`TCA_cana`). Finally, canalicular TCA flows back into the buffer compartment.

A

$$\frac{d}{dt}\text{TCA\_buffer} = -\text{import} * \text{TCA\_buffer} + \text{export\_sinus} * \text{TCA\_cell} + \text{reflux} * \text{TCA\_cana}$$

$$\frac{d}{dt}\text{TCA\_cana} = \text{export\_cana} * \text{TCA\_cell} - \text{reflux} * \text{TCA\_cana}$$

$$\frac{d}{dt}\text{TCA\_cell} = \text{import} * \text{TCA\_buffer} - \text{export\_sinus} * \text{TCA\_cell} - \text{export\_cana} * \text{TCA\_cell}.$$

B

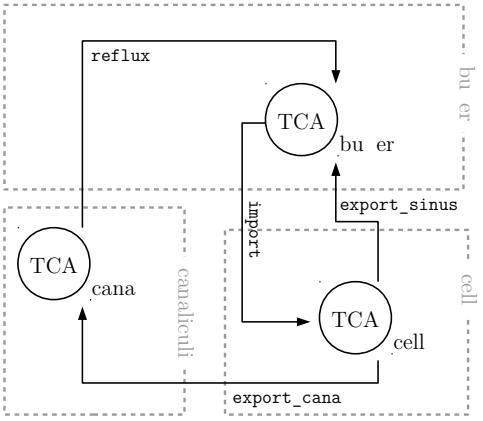

Figure 1: Differential equations and flowchart of the reaction network. Taurocholic acid, TCA, is transported between three compartments by four different processes. (A) Assuming mass-action kinetics, the three dynamic states satisfy a set of coupled differential equations. (B) The equations are visualized in a flowchart.

In the following sections, the bile acid example will be used to illustrate key elements of ODE modeling and how they are implemented using **dMod**. We will start with the implementation of the model itself and show simulation results for the model and model sensitivities. Next, we will introduce the observation function based on radioactive labeling of TCA and will use the model to simulate experimental data. The data will be fitted by the model several times following a multi-start strategy. Parameter identifiability is discussed based on the profile-likelihood method. Finally, different ways to include steady-state constraints in the parameter estimation are discussed and the prediction uncertainty will be assessed.

4.1. Simulation and prediction

These processes involved in bile acid transport give rise to the differential equations and corresponding flowchart presented in Figure 1. Each transportation process is modeled by mass-action kinetics. A possible implementation in **dMod** is:

```
R> reactions <- NULL
R> reactions <- addReaction(reactions, "TCA_buffer", "TCA_cell",
+    rate = "import*TCA_buffer", description = "Uptake")
R> reactions <- addReaction(reactions, "TCA_cell", "TCA_buffer",
+    rate = "export_sinus*TCA_cell", description = "Sinusoidal export")
R> reactions <- addReaction(reactions, "TCA_cell", "TCA_cana",
+    rate = "export_cana*TCA_cell", description = "Canalicular export")
R> reactions <- addReaction(reactions, "TCA_cana", "TCA_buffer",
+   rate = "reflux*TCA_cana", description = "Reflux into the buffer")
```

The reaction list is translated into an ODE model object:

```
R> mymodel <- odemodel(reactions, modelname = "bamodel")
```

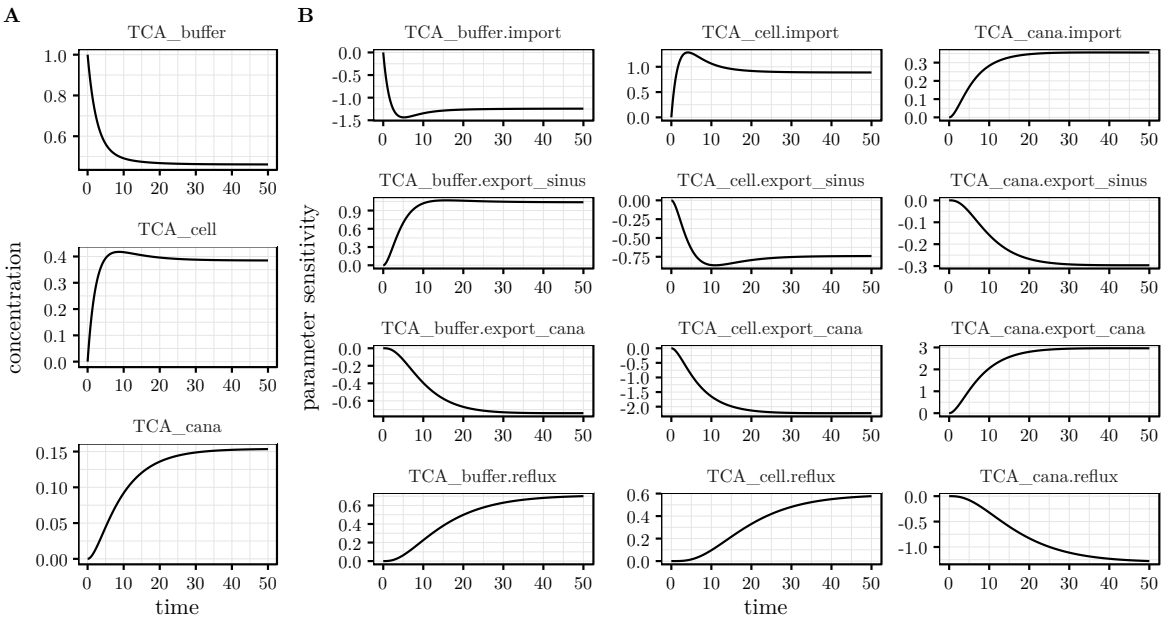

Figure 2: Output of the prediction function. (A) Prediction of the TCA states. (B) Sensitivities of the three TCA states for the rate parameters only.

Finally, the prediction function is generated:

```
R> x <- Xs(mymodel, condition = NULL)
```

The reactions are collected in an `eqnlist` object. The `odemodel()` command composes the single reactions to an ODE system and auto-generates the C code which is used by the **deSolve** package to evaluate the ODE. Prediction functions are generated by the `Xs()` command. The usage of the prediction function is illustrated by the following code chunk. Time points are defined between 0 and 50, numeric values are assigned to all model parameters. Note that the first three parameters correspond to initial state values.

```
R> times <- seq(0, 50, 0.1)
R> pars <- c(TCA_buffer = 1, TCA_cell = 0, TCA_cana = 0, import = 0.2,
+    export_sinus = 0.2, export_cana = 0.04, reflux = 0.1)
```

Finally the prediction function is called and both, the prediction and the sensitivities, are plotted, shown in Figure 2.

```
R> out <- x(times, pars)
R> plot(out)
R> outSens <- getDerivs(x(times, pars[4:7], fixed = pars[1:3]))
R> plot(outSens)
```

Figure 2A shows the uptake of `TCA_buffer` in the cell and canaliculi, saturating around $t = 50$. The prediction parametrically depends on initial values and rate parameters. Figure 2B shows the model sensitivities $\frac{\partial x}{\partial p}$ for the rate parameters.

4.2. Observation function and simulated data

In experiments, the three dynamic states, `TCA_buffer`, `TCA_cell` and `TCA_cana` cannot be directly measured. Rather, the radioactivity can only be measured separately for two compartments, namely the buffer and the cellular compartment, where the latter contains cells and canaliculi. This translates into the following relation between the radioactive counts and the dynamic states of our ODE model:

$$\begin{aligned} \texttt{buffer} &= \texttt{s * TCA\_buffer} \\ \texttt{cellular} &= \texttt{s * (TCA\_cana + TCA\_cell)} \end{aligned} \tag{15}$$

The scaling factor `s` translates amounts of TCA into radioactive counts. The observation function is expressed in **dMod** as follows.

First, observables are defined by an `eqnvec` object from which an observation function `g` is generated by the `Y()` command. The `Y()` command needs to be informed which of the symbols are variables (dynamic states) or parameters. Conveniently, `Y()` can retrieve this information from the prediction function `x` it is built on, or parse an `eqnvec` or `eqnlist` such as `reactions`.

```
R> observables <- eqnvec(buffer = "s*TCA_buffer",
+    cellular = "s*(TCA_cana + TCA_cell)")
R> g <- Y(observables, f = x, condition = NULL, compile = TRUE,
+    modelname = "obsfn")
```

Observation functions link internal to observable states. Thus, providing values for the model parameters, the observation function can be used to simulate the outcome of an experiment. Adding noise to the prediction, experimental data is simulated. In the following, we will simulate the outcome of an *efflux experiment*. The experiment starts with all TCA concentrations in steady state, such as shown in Figure 2 after $t = 50$. To initiate the efflux, the buffer is replaced by TCA-free buffer, i.e., `TCA_buffer = 0`. This translates into the following initial parameter values:

```
R> pars["TCA_cell"] <- 0.3846154
R> pars["TCA_cana"] <- 0.1538462
R> pars["TCA_buffer"] <- 0
R> pars["s"] <- 1e3
```

The predicted dynamics of the system's internal and observable states is obtained by evaluation of the concatenated prediction function $g \circ x$, formulated as `g * x` in **dMod**. The scaling parameter `s` is set to 1000.

```
R> out <- (g * x)(times, pars, conditions = "standard")
```

Since `g` and `x` have been generated as generic functions, i.e., `condition = NULL`, we can assign the output to a condition of our choice, in this case "standard". The predicted noiseless observation is obtained by considering the observable states only at the time point of observation `timesD`.

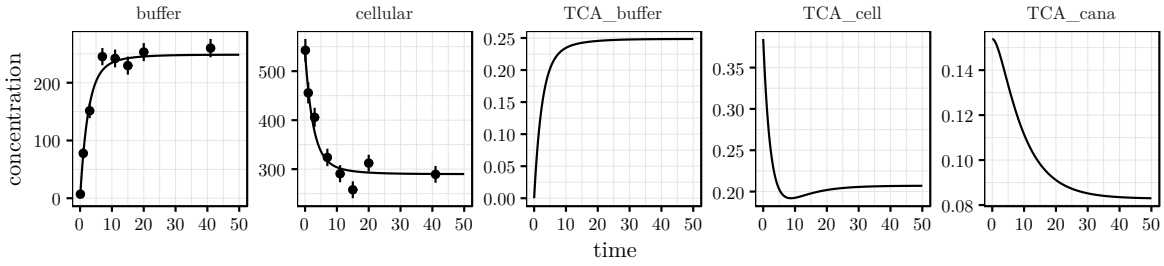

Figure 3: Model prediction of the observable and internal states. Simulated data is shown as dots with error bars.

```
R> timesD <- c(0.1, 1, 3, 7, 11, 15, 20, 41)
R> datasheet <- subset(as.data.frame(out),
+    time %in% timesD & name %in% names(observables))
```

Data uncertainties σ are derived by the Poisson nature of radioactive count experiments, i.e., $\sigma_x = \sqrt{x}$. To avoid division by 0, the minimal σ-value is set to 1. Random values are added to the predicted values to simulate observation noise. In the end, the `data.frame` is converted into a `datalist` object.

```
R> datasheet <- within(datasheet, {
+    sigma <- sqrt(value + 1)
+    value <- rnorm(length(value), value, sigma)
+  })
R> data <- as.datalist(datasheet)
R> plot(out, data)
```

Both, the simulated data and the model prediction from which the data is derived are shown in Figure 3. The data reflects a typical time course of an efflux experiment, showing decreasing cellular TCA levels and increasing levels of TCA in the buffer.

4.3. Parameter transformation

Parameter transformations play a crucial role in the set-up of **dMod**. They can have several purposes such as fixing paramer values, implementing parameter bounds, including steady-state constraints or mapping parameters to different conditions. While being conceptually the same, it might be worth noting that parameters can be distinguished in two classes. The first class of parameters are *initial values* for dynamic states, such as `TCA_buffer`. Parameters of the second class, such as rate parameters, have no accompanying dynamic state.

First, we use parameter transformations to constrain all parameters to be positive or zero because all our parameters are either amounts or rate parameters. The parameter transformation is generated by the `P()` command taking an `eqnvec` object. Parameter transformations explicitly state the relation between the **inner** parameters, i.e., the parameter values that are evaluated within the model, and the **outer** parameters, i.e., the parameter values provided by the user or by an optimizer. In our case, we imply positivity of inner parameters using the `exp()` function on outer parameters. The corresponding code reads:

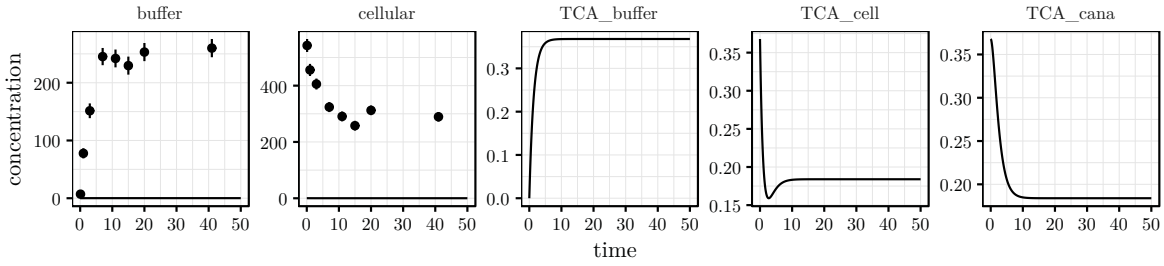

Figure 4: Prediction of internal and observed states. All values of the outer parameters have been set to −1. Simulated data points are shown as dots with error bars.

```
R> p <- P(
+    trafo = eqnvec(TCA_buffer = "0", TCA_cell = "exp(TCA_cell)",
+      TCA_cana = "exp(TCA_cana)", import = "exp(import)",
+      export_sinus = "exp(export_sinus)", export_cana = "exp(export_cana)",
+      reflux = "exp(reflux)", s = "exp(s)"),
+   condition = "standard")
R> outerpars <- getParameters(p)
R> pouter <- structure(rep(-1, length(outerpars)), names = outerpars)
R> plot((g * x * p)(times, pouter), data)
```

The vector `outerpars` is the collection of all symbols on the right-hand side of `trafo`. It coincides with `names(pars)` except for `TCA_buffer`, which is fixed to a constant expression, here 0, by the transformation. However, the interpretation of the parameters has changed since now their values are on a log-scale. All three functions, the observation function, prediction function and parameter transformation can be concatenated to one new prediction function, `g * x * p` which takes times and values of the outer parameters to predict internal and observable states. The model prediction generated by `pouter` is shown in Figure 4.

4.4. Objective function and model fitting

The objective of model fitting is to find parameter values such that the corresponding model prediction matches the observation. In Figure 4, the graphs of `buffer` and `cellular` should match the observation within the error. For normally distributed measurement noise, maximum-likelihood estimation is equivalent to least-squares estimation. A least-squares objective function can be generated by the `normL2()` command which requires a `datalist` object, in our case `data`, and a prediction function, in our case `g * x * p`.

If prior knowledge for parameter values is available, this can be incorporated by calling `constraintL2()`. The function penalizes distances between arbitrary prior values and parameter values with a quadratic function weighted by the strength of the prior. Thereby, parameters are treated like observations with normally distributed error. The objective functions returned by `normL2()` and `constraintL2()` are objects of class `objfn` and can be added by the `"+"` operator.

Frequently, non-identifiable parameters are encountered in non-linear dynamic systems. In this case, adding a weak prior to all parameters prevents the optimizer from selecting extreme

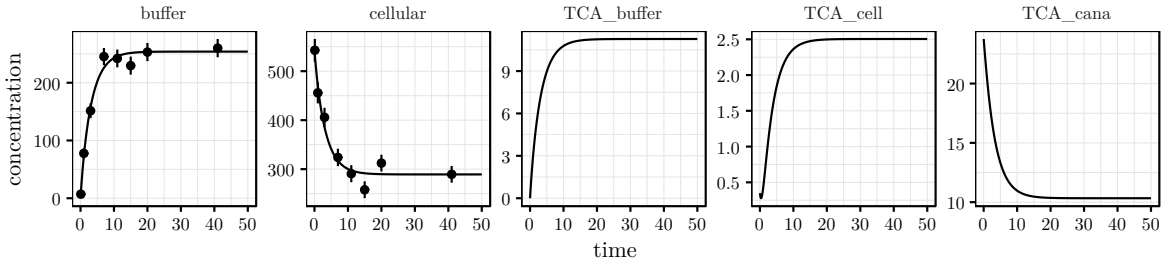

Figure 5: Prediction of internal and observed states after optimization of the objective function. Simulated data points are shown as dots with error bars.

parameter values for which the ODE solver aborts. This situation is to be distinguished from the prior knowledge case. A general prior is only used during model development to facilitate parameter estimation and should be dropped in the end. To distinguish prior knowledge contributions from a general prior in the objective value, the `attr.name` argument of `constraintL2()` can be used. Numeric attributes from objective functions added by the `"+"` operator are collected and values from equally named attributes are added. Setting `attr.name = "data"` causes the prior to be combined with the data likelihood whereas `attr.name = "prior"` distinguishes it from the data.

The following code illustrates the implementation of the objective function and how it is used with the `trust()` optimizer from the **trust** package (Geyer 2015) to obtain a model fit, shown in Figure 5.

```
R> obj <- normL2(data, g * x * p) + constraintL2(pouter, sigma = 10)
R> myfit <- trust(obj, pouter, rinit = 1, rmax = 10)
R> plot((g * x * p)(times, myfit$argument), data)
```

Besides non-identifiability of parameters, local optima constitute another pit-fall when optimizing non-linear functions. The `trust()` optimizer employs derivative information and therefore, if starting within a certain region around a local optimum, is very efficient in finding it back. Once an optimum is found, we can be confident that there is no deeper point around. However, to be confident that an optimum is the globally best solution, we might want to scatter starting points for optimization runs all over the parameter space. The **dMod** package provides the `mstrust()` function based on `trust()` to do a multi-start search:

```
R> out_mstrust <- mstrust(obj, pouter, rinit = 1, rmax = 10, iterlim = 500,
+    sd = 4, cores = 4, fits = 50)
R> myframe <- as.parframe(out_mstrust)
R> plotValues(myframe, tol = 0.01, value < 100)
R> plotPars(myframe, tol = 0.01, value < 100)
```

Here, we have searched according to $\vec{p}_i = \vec{p}_0 + \Delta\vec{p}_i$ where $\vec{p}_0$ is the center, in our case `pouter`, $\Delta\vec{p}_i \sim N(0, \sigma^2)$ is a random parameter vector taken from a normal distribution, in our case $\sigma = 4$, and the index i runs from 1 to `fits = 50`. The `mclapply()` command from the standard **parallel** package (R Core Team 2019) is used internally to run fits in parallel, here `cores = 4`. The result of `mstrust()` is a list of all returned values of `trust()`. To extract the

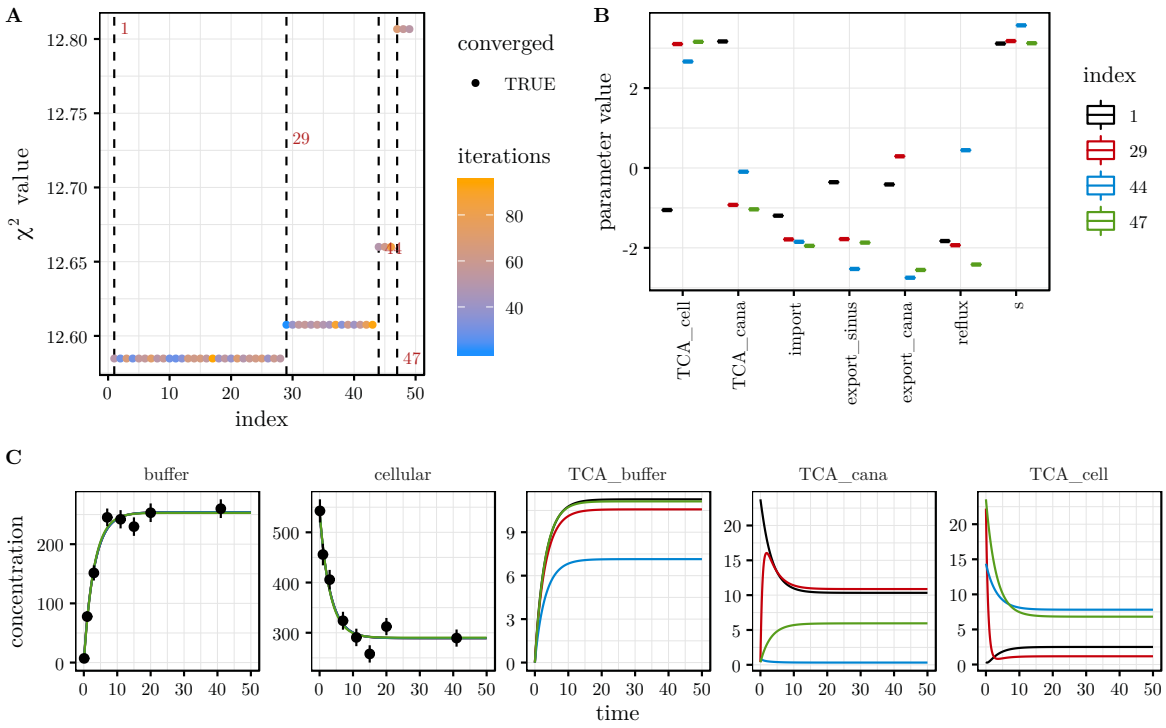

Figure 6: Result of multi-start fitting procedure. (A) Fits have been sorted by increasing objective value. Four optima were found with almost identical objective value. (B) The parameter values for different optima are shown in different colors. (C) Each local optimum corresponds to a different model prediction, shown in different colors. The observed states are pracitically undistinguishable although the internal states show different behavior.

final objective value, parameter values, convergence information and the number of iterations, `as.parframe()` is used. The multi-start approach identifies four local optima, see Figure 6, which yield almost the same objective value, Figure 6A. Despite the similar objective value, the optima are not close to each other in parameter space, as being illustrated by Figure 6B, and lead to different predictions, Figure 6C. Figure 6B suggests that the two initial values `TCA_cell` and `TCA_cana` are connected in the sense that if one takes a large value, the other takes a small value and vice versa. This is not surprising because the observed cellular TCA amount is the sum of both. A new experiment needs to be designed to distinguish one situation from another.

4.5. Working with several conditions

In practice, the canaliculi only form a closed compartment if Ca^{2+}/Mg^{2+} ions are present in the buffer. Therefore, if the experiment is repeated with Ca^{2+}/Mg^{2+}-free efflux buffer, the contents of the canaliculi escapes quickly into the buffer compartment. Under this condition, the `buffer` measurement reflects what was formerly the total TCA content in buffer and canaliculi whereas the `cellular` measurement reflects what was fomerly the TCA content of the cells. Mathematically, two experimental conditions which differ only by the `reflux` parameter need to be combined in one objective function.

Like before, we simulate a data set. Then, a parameter transformation for the additional condition is set up and the parameter space is explored by a multi-start fit.

To simulate the new experimental condition, the `"reflux"` parameter is modified. The new data set is combined with the original data by the `"+"` operator.

```
R> pars["reflux"] <- 1e3
R> out <- (g*x)(times, pars, conditions = "open")
R> datasheet <- subset(as.data.frame(out),
+    time %in% timesD & name %in% names(observables))
R> datasheet <- within(datasheet, {
+    sigma <- sqrt(value + 1)
+    value <- rnorm(length(value), value, sigma)
+  })
R> data <- data + as.datalist(datasheet)
```

To add a condition to the parameter transformation function, we use the equations of the standard condition as a template for the "open" condition. Parameter transformation functions for different conditions are combined by the `"+"` operator.

```
R> trafo <- getEquations(p, conditions = "standard")
R> trafo["reflux"] <- "exp(reflux_open)"
R> p <- p + P(trafo, condition = "open")
```

Both transformations "standard" and "open" now possess the outer parameters `reflux` and `reflux_open`. However, the value of `reflux` is mapped to an inner parameter only by transformation "standard". Accordingly, transformation "open" only uses `reflux_open`. Thus, both transformations return the same values for all but the `reflux` parameter. The prediction function `g * x` is generic in the sense that `condition = NULL` whereas the concatenation `g * x * p` has the conditions "standard" and "open", evaluating the identical function `g * x` on two parameter vectors.

We define an updated objective function:

```
R> outerpars <- getParameters(p)
R> pouter <- structure(rep(-1, length(outerpars)), names = outerpars)
R> obj <- normL2(data, g * x * p) + constraintL2(pouter, sigma = 10)
```

Then we start 50 fits around `pouter`. The list of fits is simplified to a `parframe` and by the `as.parvec()` function, the parameter vector (of the best fit) is extracted from the `parframe`. The best fit is used to make a prediction which is plotted together with the simulated data. All results are shown in Figure 7.

```
R> out_mstrust <- mstrust(obj, pouter, rinit = 1, rmax = 10, iterlim = 500,
+    sd = 4, cores = 4, fits = 50)
R> myframe <- as.parframe(out_mstrust)
R> plotValues(myframe, tol = 0.1, value < 100)
R> plotPars(myframe, tol = 0.1, value < 100)
R> bestfit <- as.parvec(myframe)
R> plot((g * x * p)(times, bestfit), data)
```

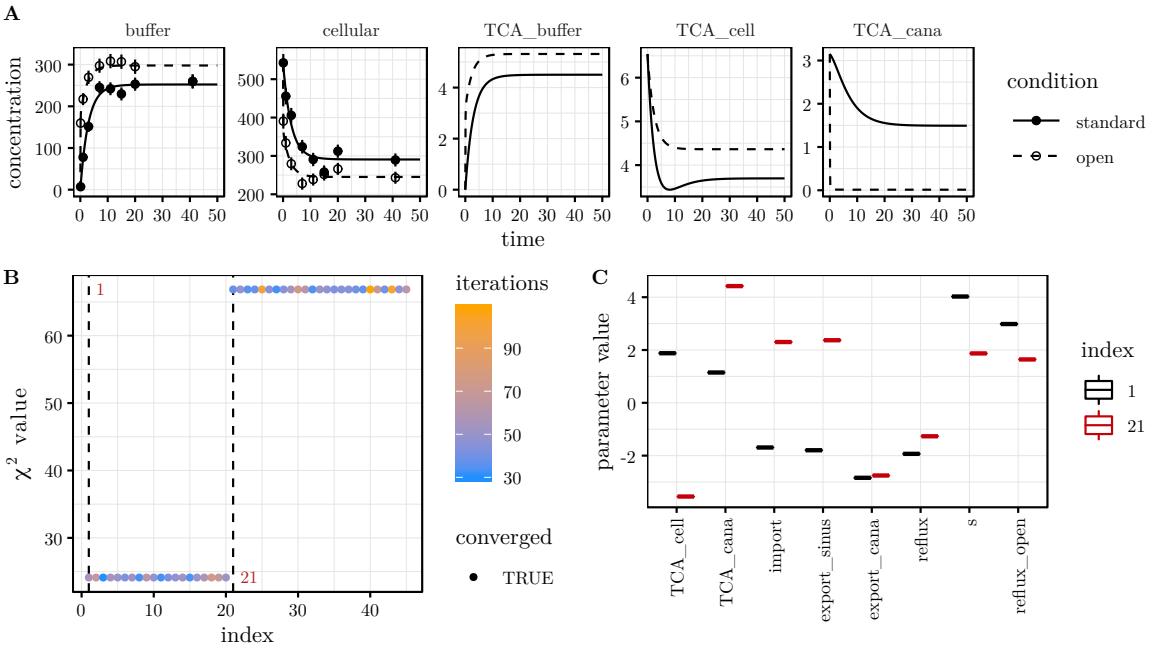

Figure 7: Result of multi-start fitting procedure with two experimental conditions. (A) Model prediction of the best fit and the simulated data are shown in different colors. (B) Fits have been sorted by increasing objective value. The lowest value clearly separates from the second plateau. (C) Plotting the parameter values for each of the fits reveals that the second plateau consists of two optima. The lowest plateau however corresponds to a unique optimum.

Interestingly, with the new experiment the best optimum becomes unique. The log-likelihood difference, i.e., half the difference between the objective values, is more than 20 between the lowest and the second plateau, which is highly significant. The uniqueness of the lowest plateau is confirmed by Figure 7C which shows no scattering of the black circles.

4.6. Parameter uncertainty and identifiability

One might wonder why the optimum is unique as for any choice of the scaling parameter `s` we find appropriate values of the TCA initial value parameters that give rise to exactly the same prediction of the observables. The reason for the uniqueness is the parameter L_2-constraint that we have added to the objective function. Nonetheless, we will see, that the non-identifiability is still visible in the profile likelihood.

The profile likelihood is computed by the `profile()` command. There are several options to control the step-size and accuracy. For convenience the `method` option can be used to select between the presets `"integrate"` and `"optimize"`.

The code

```
R> profiles <- profile(obj, bestfit, names(bestfit), limits = c(-5, 5),
+    cores = 4)
R> plotProfile(profiles)
R> plotPaths(profiles, whichPar = "s")
```

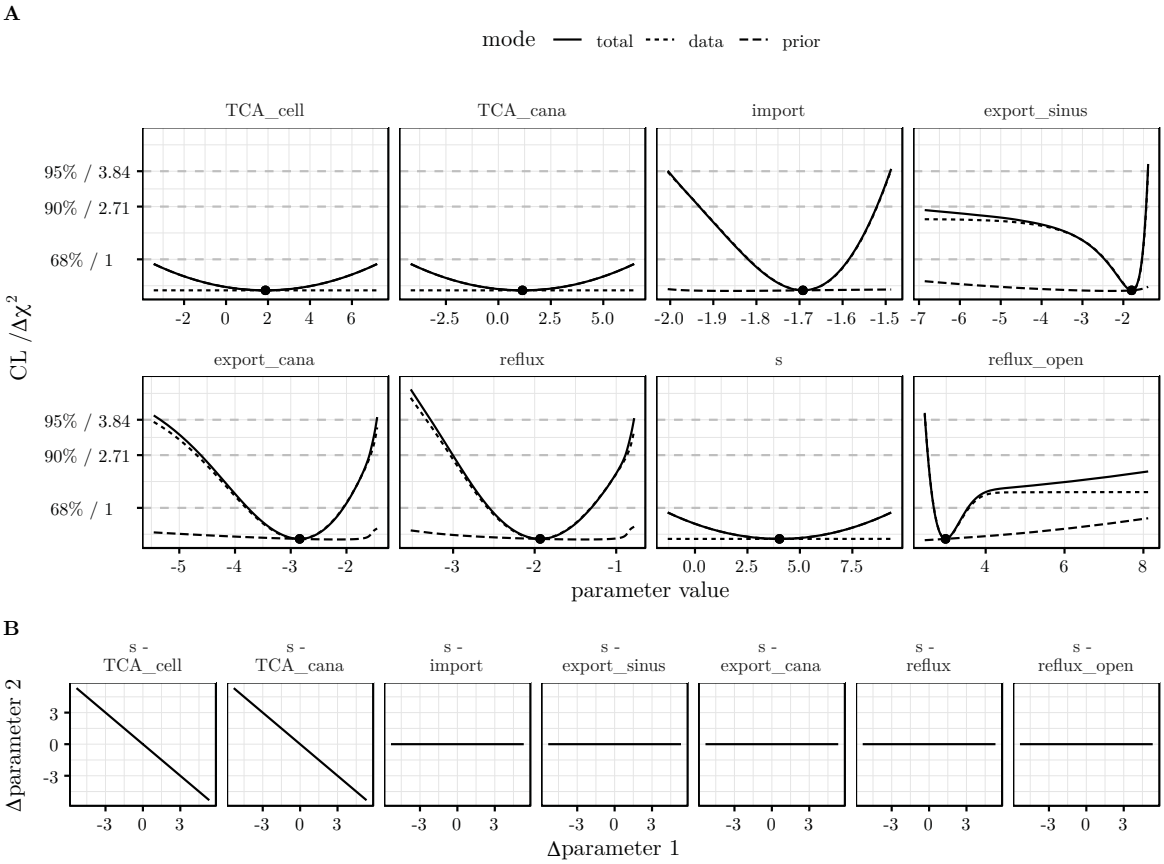

Figure 8: Profile likelihood. (A) Profiles of all parameters. Data- and prior contribution to the total objective value are distinguished by line-type. (B) Parameter paths for the scaling parameter `s`.

gives us the result shown in Figure 8. Computing the profile likelihood, the sum of data contribution, `normL2`, and prior contribution, `constraintL2`, are optimized under the constraint of a given parameter value for the profiled parameter. In the optimum, data and prior contribution are evaluated separately giving rise to the dashed and dotted lines in Figure 8A. As we had expected, the data contribution to the initial value parameters `TCA_cell`, `TCA_cana` and the scaling parameter `s` is constantly zero. The parameters are structurally non-identifiable.

Each profile corresponds to a certain path in parameter space. The path for the profile of the non-identifiable scaling parameter `s` is shown in Figure 8B. It shows a clear coupling of the scaling parameter `s` and the initial value parameters `TCA_cell` and `TCA_cana`: both initial value parameters have to be decreased by the same extent as the scaling parameter is increased to keep the prediction unchanged.

The profiles of the parameters `export_sinus` and `reflux_open` exceed the 95% confidence threshold only to one side. Given the data, the `export_sinus` parameter could equally be $-\infty$ (corresponding to an export rate of 0) without changing the likelihood significantly for the worse. A similar statement holds for the `reflux_open` parameter which could equally be ∞ meaning that we could assume instantaneous draining of the canaliculi for the `"open"`

condition. The two parameters are practically non-identifiable.

Finally, the parameters `import`, `export_cana` and `reflux` exceed the 95% confidence threshold in both directions meaning that the parameters have finite confidence intervals. However, the confidence intervals are rather large and we might ask if there is further information that we could use to improve parameter identifiability without generating new data.

4.7. Steady-state constraints and implicit transformations

So far we have estimated both initial concentrations, `TCA_cell` and `TCA_cana`, independently. However, we know that the efflux experiment was just started after completion of the uptake process. Our system runs into a steady state, the buffer is exchanged and the measurement begins. Hence, we can use the steady-state condition as an additional information for the modeling process.

The steady-state relation between `TCA_cana` and `TCA_cell` can be derived analytically from the ODE, Figure 1A, by setting the right-hand side of $\frac{\mathrm{d}}{\mathrm{d}t}$`TCA_cana` to zero. It reads

$$\texttt{TCA\_cana} = \texttt{export\_cana * TCA\_cell / reflux} \tag{16}$$

This relation can be explicitly used in a parameter transformation to express `TCA_cana` in terms of other parameters. The dimension of the parameter space is thereby reduced by one. The following implementation shows how we would use the existing transformation function `p` to generate an alternative transformation function `pSS` which includes the steady-state condition and replaces `p` in the prediction function `g * x * p`.

```
R> pSS <- NULL
R> equations <- getEquations(p)
R> conditions <- names(equations)
R> for (n in conditions) {
+    equations[[n]]["TCA_cana"] <- "exp(export_cana)*exp(TCA_cell)/exp(reflux)"
+    pSS <- pSS + P(equations[[n]], condition = n)
+  }
```

We get all the information about the transformations from the `getEquations()` command. The equation for `TCA_cana` is substituted by our steady-state constraint. By the `"+"` operator, a new parameter transformation function `pSS` is iteratively constructed for all conditions.

Alternatively, we want to implement the steady-state constraint by an *implicit* parameter transformation, as opposed to the explicit transformation shown above. Let $\dot{x} = f(x, p)$ be our dynamic system. Then, under certain conditions, we find a function $g(p)$ such that $f\big(g(p), p\big) = 0$ for all p, i.e., $x_S = g(p)$ is a steady state of f. The parameter transformation we want to generate is the function $p \mapsto \big(p, g(p)\big)$. Here, the set of outer parameters is the set of the reaction rates p whereas the set of inner parameters contains these rates and the corresponding steady states as initial value parameters. The root of f must be determined numerically to which end `multiroot()` from the **rootSolve** package (Soetaert and Herman 2009) is used.

The following code is a reimplementation of the example above. The replacement of the buffer and the possibility for different reflux rates between the "standard" and "open" conditions are explicitly modeled by events at time zero.

```
R> reactions <- NULL
R> reactions <- addReaction(reactions, "TCA_buffer", "TCA_cell",
+    rate = "import*TCA_buffer", description = "Uptake")
R> reactions <- addReaction(reactions, "TCA_cell", "TCA_buffer",
+    rate = "export_sinus*TCA_cell", description = "Sinusoidal export")
R> reactions <- addReaction(reactions, "TCA_cell", "TCA_cana",
+    rate = "export_cana*TCA_cell", description = "Canalicular export")
R> reactions <- addReaction(reactions, "TCA_cana", "TCA_buffer",
+    rate = "(reflux*(1-switch) + reflux_open*switch)*TCA_cana",
+    description = "Reflux into the buffer")
R> reactions <- addReaction(reactions, "0", "switch", rate = "0",
+    description = "Create a switch")
```

The events for `TCA_buffer` and `switch` are generated as part of the model and, therefore, need to be provided to the `odemodel()` command together with the model.

```
R> events <- NULL
R> events <- addEvent(events, var = "TCA_buffer", time = 0, value = 0      )
R> events <- addEvent(events, var = "switch"    , time = 0, value = "OnOff")
R> mymodel <- odemodel(reactions, modelname = "bamodel2", events = events)
R> x <- Xs(mymodel)
```

Event times and values can both be either numeric or a character representing a parameter. These parameters are treated in the same way as all other parameters, meaning that they can be set in a condition-specific way or be estimated.

For the implicit parameter transformation we need the ODE which is obtained by the command `as.eqnvec()` from the reactions. The Jacobian of f is rank-deficient because the system has a conserved quantity $c =$ `TCA_buffer + TCA_cana + TCA_cell` which is the total TCA amount. Replacing one element of f by $c -$ `TCA_tot`, the rank of the Jacobian is completed, the condition for the local existence of the implicit function $g(p)$ is satisfied and the steady state is parameterized by p and the additional parameter `TCA_tot`.

```
R> f <- as.eqnvec(reactions)[c("TCA_buffer", "TCA_cana", "TCA_cell")]
R> f["TCA_cell"] <- "TCA_buffer + TCA_cana + TCA_cell - TCA_tot"
R> pSS <- P(f, method = "implicit", compile = TRUE, modelname = "pfn")
```

For the optimization, all outer parameters should still be log-parameters, implemented by an explicit parameter transformation. Outer parameters for the initial values are not necessary any more. They can be replaced by 0 since the initial values are computed by the implicit transformation. The switch parameter `OnOff` is defined in a condition-specific way: zero for the "standard" condition and one for the "open" condition. The final transformation will be a concatenation of the implicit and explicit transformations: `pSS*p`.

```
R> innerpars <- unique(c(getParameters(mymodel), getSymbols(observables),
+    getSymbols(f)))
R> trafo <- repar("x~x"     , x = innerpars)
R> trafo <- repar("x~0"     , x = reactions$states, trafo)
```

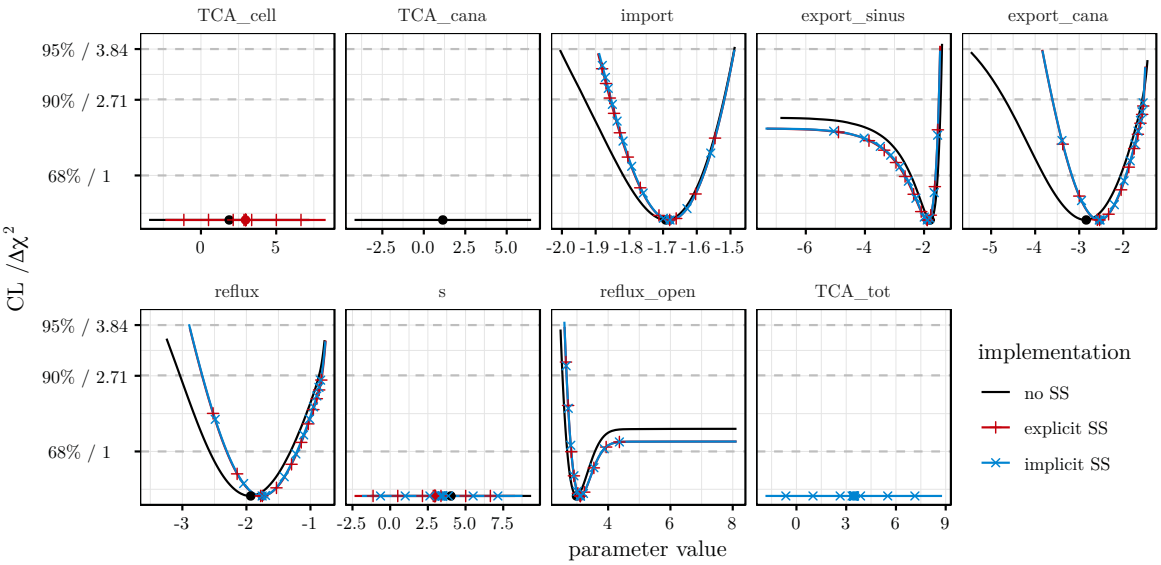

Figure 9: Parameter profiles for three different model implementations. The profile likelihood around the global optimum for the models without steady-state constraints, explicit steady-state constraints and implicit implementation of steady states is visualized by different colors. To illustrate that explicit (red) and implicit (blue) steady-state implementations yield the same result, the corresponding profiles are highlighted by red plus and blue cross signs, respectively.

```
R> trafo <- repar("x~exp(x)", x = setdiff(innerpars, "OnOff"), trafo)
R> p <- P(repar("OnOff~0", trafo), condition = "standard") +
+    P(repar("OnOff~1", trafo), condition = "open")
```

Although the observables have not changed compared to the set-up with purely explicit transformations, the observation function must be generated again because the ODEs have structurally changed and a new parameter `reflux_open` has appeared. Using the old observation function would result in a wrong propagation of parameter sensitivities.

```
R> g <- Y(observables, f = x, compile = TRUE, modelname = "obsfn2")
```

Finally, the objective function is defined. The prediction function is now a concatenation of four functions.

```
R> outerpars <- getParameters(p)
R> pouter <- structure(rep(-1, length(outerpars)), names = outerpars)
R> obj <- normL2(data, g * x * pSS * p) + constraintL2(pouter, sigma = 10)
```

The same simulated data set has been fitted by the fully explicit and the implicit/explicit model implementations. In both cases the global optimum is unique. Parameter profiles have been computed with both model implementations, shown in Figure 9. In addition, the original profiles without steady-state constraints are plotted. The optima found by all three approaches, no steady-state, analytic steady-state and numeric steady-state, are statistically

compatible. The two implementations using the steady-state information show exactly the same profiles. Since one formulation is parameterized by `TCA_cell` whereas the other is parameterized by `TCA_tot`, the plot highlights one of the fundamental properties of the profile likelihood: invariance under reparameterization. In comparison to the profiles without steady-state information, the new profiles are narrower, meaning that the parameters have smaller confidence intervals. This was to be expected because we have reduced the dimension of the parameter space by one.

4.8. Prediction uncertainty and validation profiles

Combining the steady-state constraint and two efflux experiments, one with closed canaliculi and the other with open canaliculi, we could fully identify the rate parameters `import`, `export_cana` and `reflux`. The amount parameter `TCA_tot` is fully coupled with the scaling parameter `s` such that both are structurally non-identifiable. The parameters `export_sinus` and `reflux_open` are practically non-identifiable since both parameters cannot be constrained to a finite interval with 95% confidence.

Next, we investigate the possibility to predict cellular amounts of TCA, `TCA_cell`, despite the non-identifiability of parameters. The amount of `TCA_cell` certainly depends on the total amount of TCA in the system. This total amount must be fixed in which case the parameters `TCA_cell`, `TCA_cana` and `s` become identifiable. The prediction uncertainty is assessed by a *prediction profile* which is computed based on a virtual data point for cellular TCA, measured at time point $t = 41$ in the "standard" condition. The **dMod** formulation reads as:

```
R> obj.validation <- datapointL2(name = "TCA_cell", time = 41, value = "d1",
+    sigma = 0.002, condition = "standard")
```

The uncertainty $\sigma = 0.002$ is set to a small value, i.e., below 1% of the prediction value. The `datapointL2()` command returns an objective function which evaluates the model prediction[2] and computes the least-squares function of the virtual datapoint, returning derivatives for the data-point parameter `d1`. Its value "d1" is yet to be determined. By fitting the objective function `obj` together with `obj.validation`, "d1" equals the value of `TCA_cell` at $t = 41$, as only then its contribution to the objective value is zero.

```
R> fixed <- c(TCA_tot = log(1))
R> myfit <- trust(obj + obj.validation,
+    parinit = c(d1 = 1, pouter[!names(pouter) %in% names(fixed)]),
+    fixed = fixed, rinit = 1, rmax = 10)
```

Using the derivative information provided by `datapointL2`, a prediction profile around `d1` is calculated.

```
R> profile_prediction <- profile(obj + obj.validation,
+    myfit$argument, "d1", limits = c(-5, 5), fixed = fixed)
```

The result is shown in Figure 10A. The interpretation of such a prediction profile is, that a measurement yielding a value for the cellular TCA level at time point $t = 41$ outside of the interval $[0.19, 0.21]$ does not conform to our model with 95% confidence.

[2]Several objective functions combined by the `"+"` operator share the same environment. Thus, the prediction computed by the first objective function can be evaluated by all other functions to come.

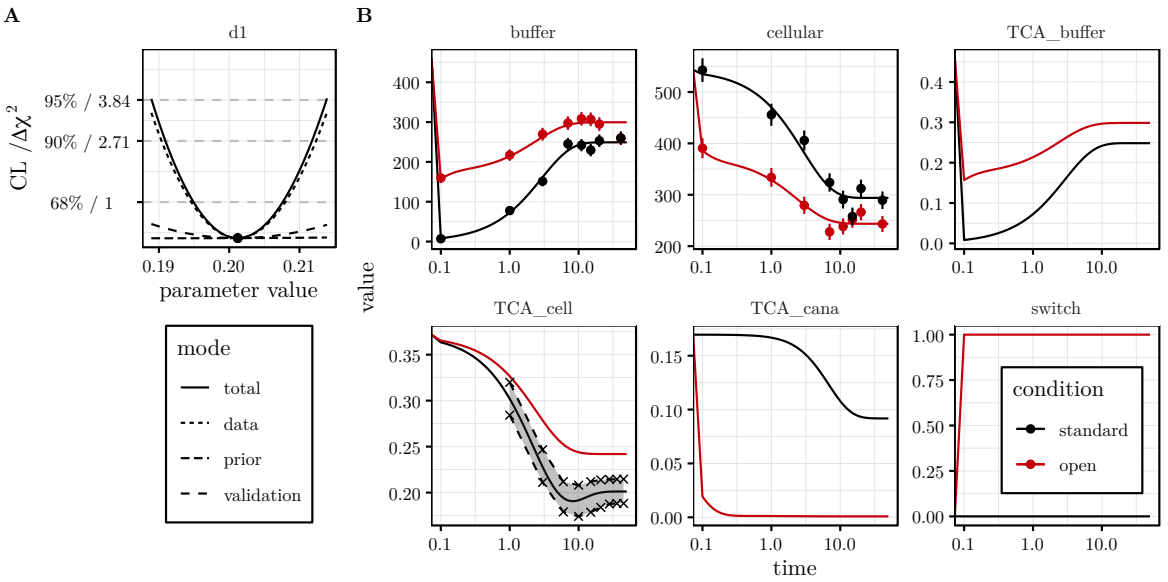

Figure 10: Validation profile and confidence bands for the model prediction. (A) The profile likelihood for the data point parameter `d1` describing `TCA_cell` at time point $t = 41$ is shown. (B) Computing the data parameter profile for different time points yields 95% confidence bands on the prediction of `TCA_cell`.

More precisely, changing the data-point parameter `d1`, the model is quickly forced to match the new data point. This is apparent from the least squares contribution entitled "validation" of the virtual data point, returned by `obj.validation`. The contribution is shown as dashed line in Figure 10A. It remains small at the expense of a larger deviation from original data points, indicated by the "data" contribution. Forcing `d1` and thereby the model prediction to deviate more than 0.01 from the original value, the profile exceeds the 95% confidence threshold providing a confidence interval for the prediction itself. By calculating prediction profiles for several time points, a confidence band for the course of `TCA_cell` is constructed as shown in Figure 10B. The 95% confidence band is closed towards small and large amounts.

In summary, we find that the prediction of cellular TCA amounts is highly precise despite the non-identifiability of the `export_sinus` and `reflux_open` parameters.

4.9. Speed comparison

Last but not least, the computational efficiency of **dMod** was tested. In comparison to other R packages offering functions to estimate parameters of ODE models from experimental data, see Table 1, **dMod** provides all observation-, prediction- and parameter transformation functions with derivatives. Those are based on symbolic expressions being translated into C code and being compiled alongside the model itself.

For the comparison, we have tested three different scenarios with **dMod**. The scenarios approximate the conditions found in different frameworks: (1) The default settings with derivatives and trust region optimization, (2) derivative-free optimization with the Nelder-Mead algorithm and, (3) optimization with the L-BFGS-B algorithm with numerically computed gradient. With either of these settings, the example model has been fitted 50 times and the

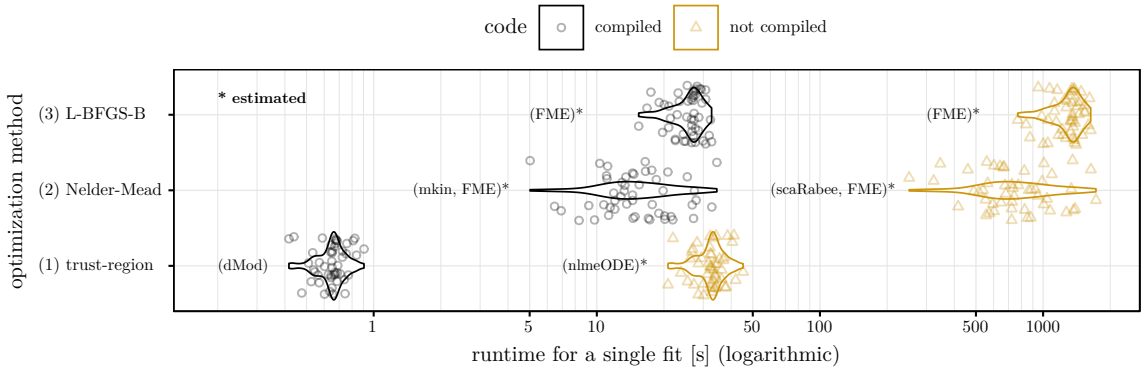

Figure 11: Comparison of the runtime for parameter estimation for different scenarios. The runtime values obtained from 50 fits per scenario with **dMod** are shown as black dots and violin lines. Runtime values for native R code (not compiled) were assumed be 50 times larger, shown as orange triangles and violin lines. Modeling frameworks have been assigned to either of the scenarios based on their characteristics.

runtime of each fit has been evaluated. The result of the runtime evaluation is shown in Figure 11 as black dots and violin lines. Since in **dMod** the ODE model and observation function are compiled, runtime values for the corresponding scenarios with native R code have been roughly estimated to exceed the runtime of compile code by a factor of 50, as indicated by Soetaert *et al.* (2010). The extrapolated runtimes are shown in orange. Finally, the available packages for parameter estimation in ODE models have been assigned to one or more scenarios depending on whether the framework supports compiled code or supports sensitivity equations. Except for the runtime values obtained with **dMod**, these assignments reflects a rough estimate, indicated by an asterisk.

The speed comparison shows that by a combination of compiled code and symbolically derived gradient and hessian, the runtime achieves its best value. The reason for the good performance is that the evaluation of sensitivity equations is only slower by a factor 2-3. On the other hand, the objective function needs to be called only once per iteration of the optimizer whereas the numeric evaluation of the gradient requires many function calls.

5. Extensions of dMod

Computer algebra and symbolic tools are not part of R's core functionality. In this section we illustrate two symbolic tools that are shiped with **dMod**, dealing with structural non-identifiability and steady-state constraints. They are implemented in Python and are interfaced via the **rPython** package (Bellosta 2015).

5.1. Lie-group symmetry detection

In Section 4.6, profile likelihood computation showed the existence of both practically and structurally non-identifiable parameters. While practical non-identifiability arises from insufficient information in the data, structural non-identifiabilty is connected to Lie-group sym-

metries, i.e., transformations of the states and parameters

$$\Psi : (x, \theta) \longmapsto (x^*, \theta^*)$$

that preserve the model prediction of the observables:

$$g(x^*, \theta^*) = g(x, \theta)\,.$$

Based on Merkt *et al.* (2015), the `symmetryDetection()` command outputs a list of available symmetry transformations. For example, the code

```
R> reactions <- NULL
R> reactions <- addReaction(reactions, "TCA_buffer", "TCA_cell",
+    rate = "import_baso*TCA_buffer")
R> reactions <- addReaction(reactions, "TCA_cell", "TCA_buffer",
+    rate = "export_sinus*TCA_cell")
R> reactions <- addReaction(reactions, "TCA_cell", "TCA_cana",
+    rate = "export_cana*TCA_cell")
R> reactions <- addReaction(reactions, "TCA_cana", "TCA_buffer",
+    rate = "reflux*TCA_cana")
R> observables <- eqnvec(buffer = "s*TCA_buffer",
+    cellular = "s*(TCA_cana + TCA_cell)")
R> symmetryDetection(as.eqnvec(reactions), observables)
```

returns the following output:

```
1 transformation(s) found:
variable   : infinitesimal : transformation
-----------------------------------------------------
#1: Type: scaling
TCA_buffer : -TCA_buffer   : TCA_buffer*exp(-epsilon)
TCA_cana   : -TCA_cana     : TCA_cana*exp(-epsilon)
TCA_cell   : -TCA_cell     : TCA_cell*exp(-epsilon)
s          : s             : s*exp(epsilon)
```

In agreement with the identifiability analysis by the profile-likelihood method, the parameters `s`, `TCA_buffer`, `TCA_cana` and `TCA_cell` are found to be non-identifiable due to a scaling symmetry. The corresponding scaling transformation, last column, leaves the observation invariant for any choice of `epsilon`. The parameter non-identifiability can be resolved choosing one representative from the orbit of the transformation. In our case, the scaling parameter could, for example, be fixed to 1.

5.2. Analytical steady-state constraints

While for the the present model, the steady-state could be explicitly calculated by hand, this might be much more challenging for models with a large number of states and parameters. For many of these models, the `steadyStates()` command outputs an analytical steady-state solution that can be incorporated in the model as an additional parameter transformation. Based on Rosenblatt *et al.* (2016), the steady-state constraint is solved for a combination

of state variables and kinetic parameters while positivity of the solution is ensured. As the paper states, the approach outperforms common methods of steady-state implementation with respect to reliability and performance of the optimization process. For our example, the code reads

```
R> steadyStates(reactions, file = "SS.Rds")
```

yielding the output

```
TCA_cana   = TCA_buffer*export_cana*import_baso/(reflux*(export_cana +
             export_sinus))
TCA_cell   = TCA_buffer*import_baso/(export_cana + export_sinus)
TCA_buffer = TCA_buffer
```

The solution is stored in an `.Rds` file. After loading the file by `readRDS()`, the equations can be used when defining the parameter transformation.

References

Azzalini A (1996). *Statistical Inference Based on the Likelihood*, volume 68. CRC Press.

Bellosta CJG (2015). ***rPython**: Package Allowing R to Call Python*. R package version 0.0-6, URL https://CRAN.R-project.org/package=rPython.

Bihorel S (2014). ***scaRabee**: Optimization Toolkit for Pharmacokinetic-Pharmacodynamic Models*. R package version 1.1-3, URL https://CRAN.R-project.org/package=scaRabee.

Brun R, Reichert P, Künsch HR (2001). "Practical Identifiability Analysis of Large Environmental Simulation Models." *Water Resources Research*, **37**(4), 1015–1030. doi:10.1029/2000wr900350.

Geyer CJ (2015). ***trust**: Trust Region Optimization*. R package version 0.1-7, URL https://CRAN.R-project.org/package=trust.

Hass H, Kreutz C, Timmer J, Kaschek D (2016). "Fast Integration-Based Prediction Bands for Ordinary Differential Equation Models." *Bioinformatics*, **32**(8), 1204–1210. doi:10.1093/bioinformatics/btv743.

Hooker G, Ramsay JO, Xiao L (2016). "**CollocInfer**: Collocation Inference in Differential Equation Models." *Journal of Statistical Software*, **75**(2), 1–52. doi:10.18637/jss.v075.i02.

Kaschek D (2019). ***cOde**: Automated C Code Generation for **deSolve**, **bvpSolve** and 'Sundials'*. R package version 1.0.0, URL https://CRAN.R-project.org/package=cOde.

Kaschek D, Sharanek A, Guillouzo A, Timmer J, Weaver RJ (2017). "A Dynamic Mathematical Model of Bile Acid Clearance in HepaRG Cells." *Toxicological Sciences*.

King AA, Ionides EL, Breto C, Ellner SP, Ferrari MJ, Kendall BE, Lavine M, Nguyen D, Reuman DC, Wearing H, Wood SN, Funk S, Johnson SG (2017). ***pomp**: Statistical Inference for Partially Observed Markov Processes*. R package version 1.14, URL https://CRAN.R-project.org/package=pomp.

Kreutz C, Raue A, Kaschek D, Timmer J (2013). "Profile Likelihood in Systems Biology." *FEBS Journal*, **280**(11), 2564–2571. doi:10.1111/febs.12276.

Kreutz C, Raue A, Timmer J (2012). "Likelihood Based Observability Analysis and Confidence Intervals for Predictions of Dynamic Models." *BMC Systems Biology*, **6**(1), 120. doi:10.1186/1752-0509-6-120.

Maiwald T, Hass H, Steiert B, Vanlier J, Engesser R, Raue A, Kipkeew F, Bock HH, Kaschek D, Kreutz C, Timmer J (2016). "Driving the Model to Its Limit: Profile Likelihood Based Model Reduction." *PloS ONE*, **11**(9), e0162366. doi:10.1371/journal.pone.0162366.

Merkt B, Timmer J, Kaschek D (2015). "Higher-Order Lie Symmetries in Identifiability and Predictability Analysis of Dynamic Models." *Physical Review E*, **92**(1), 012920. doi:10.1103/physreve.92.012920.

Murphy SA, Van der Vaart AW (2000). "On Profile Likelihood." *Journal of the American Statistical Association*, **95**(450), 449–465. doi:10.1080/01621459.2000.10474219.

Press WH, Teukolsky SA, Vetterling WT, Flannery BP (1996). *Numerical Recipes in C*, volume 2. Cambridge University Press Cambridge.

Ranke J, Lindenberger K, Lehmann R (2019). ***mkin**: Kinetic Evaluation of Chemical Degradation Data*. R package version 0.9.48.1, URL https://CRAN.R-project.org/package=mkin.

Raue A, Kreutz C, Maiwald T, Bachmann J, Schilling M, Klingmüller U, Timmer J (2009). "Structural and Practical Identifiability Analysis of Partially Observed Dynamical Models by Exploiting the Profile Likelihood." *Bioinformatics*, **25**(15), 1923–1929. doi:10.1093/bioinformatics/btp358.

Raue A, Kreutz C, Maiwald T, Klingmüller U, Timmer J (2011). "Addressing Parameter Identifiability by Model-Based Experimentation." *IET Systems Biology*, **5**(2), 120–130. doi:10.1049/iet-syb.2010.0061.

Raue A, Kreutz C, Theis FJ, Timmer J (2013a). "Joining Forces of Bayesian and Frequentist Methodology: A Study for Inference in the Presence of Non-Identifiability." *Philosophical Transactions of the Royal Society A*, **371**(1984), 20110544. doi:10.1098/rsta.2011.0544.

Raue A, Schilling M, Bachmann J, Matteson A, Schelker M, Kaschek D, Hug S, Kreutz C, Harms BD, Theis FJ, Klingmüller U, Timmer J (2013b). "Lessons Learned from Quantitative Dynamical Modeling in Systems Biology." *PloS ONE*, **8**(9), e74335. doi:10.1371/journal.pone.0074335.

R Core Team (2019). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Rosenblatt M, Timmer J, Kaschek D (2016). "Customized Steady-State Constraints for Parameter Estimation in Non-Linear Ordinary Differential Equation Models." *Frontiers in Cell and Developmental Biology*, **4**. doi:10.3389/fcell.2016.00041.

Sklyar O, Murdoch D, Smith M, Eddelbuettel D, François R, Soetaert K (2018). ***inline**: Functions to Inline C, C++, Fortran Function Calls from R*. R package version 0.3.15, URL https://CRAN.R-project.org/package=inline.

Soetaert K, Herman PMJ (2009). *A Practical Guide to Ecological Modelling: Using R as a Simulation Platform*. Springer-Verlag.

Soetaert K, Petzoldt T (2010). "Inverse Modelling, Sensitivity and Monte Carlo Analysis in R Using Package **FME**." *Journal of Statistical Software*, **33**(3), 1–28. doi:10.18637/jss.v033.i03.

Soetaert K, Petzoldt T, Setzer RW (2010). "Solving Differential Equations in R: Package **deSolve**." *Journal of Statistical Software*, **33**(9), 1–25. ISSN 1548-7660. doi:10.18637/jss.v033.i09.

Squire W, Trapp G (1998). "Using Complex Variables to Estimate Derivatives of Real Functions." *SIAM Review*, **40**(1), 110–112. doi:10.1137/s003614459631241x.

Tornoe CW (2012). ***nlmeODE**: Non-Linear Mixed-Effects Modelling in **nlme** Using Differential Equations*. R package version 1.1, URL https://CRAN.R-project.org/package=nlmeODE.

Venzon DJ, Moolgavkar SH (1988). "A Method for Computing Profile-Likelihood-Based Confidence Intervals." *Applied Statistics*, pp. 87–94. doi:10.2307/2347496.

Wright S, Nocedal J (1999). "Numerical Optimization." *Springer-Verlag*, **35**, 67–68. doi:10.1007/b98874.

Affiliation:

Daniel Kaschek
University of Freiburg
Institute of Physics
Hermann-Herder-Str. 3
79104 Freiburg, Germany
E-mail: daniel.kaschek@gmail.com

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
March 2019, Volume 88, Issue 10 *Submitted:* 2016-10-19
doi:10.18637/jss.v088.i10 *Accepted:* 2017-12-17