Inverse Modelling, Sensitivity and Monte Carlo Analysis in R Using Package FME

Karline Soetaert
Netherlands Institute of Ecology

Thomas Petzoldt
Technische Universität Dresden

Abstract

Mathematical simulation models are commonly applied to analyze experimental or environmental data and eventually to acquire predictive capabilities. Typically these models depend on poorly defined, unmeasurable parameters that need to be given a value. Fitting a model to data, so-called inverse modelling, is often the sole way of finding reasonable values for these parameters. There are many challenges involved in inverse model applications, e.g., the existence of non-identifiable parameters, the estimation of parameter uncertainties and the quantification of the implications of these uncertainties on model predictions.

The R package **FME** is a modeling package designed to confront a mathematical model with data. It includes algorithms for sensitivity and Monte Carlo analysis, parameter identifiability, model fitting and provides a Markov-chain based method to estimate parameter confidence intervals. Although its main focus is on mathematical systems that consist of differential equations, **FME** can deal with other types of models. In this paper, **FME** is applied to a model describing the dynamics of the HIV virus.

Note: The original version of this vignette has been published as **?** in the Journal of Statistical Software, http://www.jstatsoft.org/v33/i03. Please refer to the original publication when citing this work.

Keywords: simulation models, differential equations, fitting, sensitivity, Monte Carlo, identifiability, R.

1. Introduction

Mathematical models are used to study complex dynamic systems in many research fields, such as the biological, chemical, physical sciences, in medicine or pharmacy, economy and so on. Based on (mass) conservation principles, these models often consist of differential equations, which formalize the exchange of material, individuals, energy or other quantities between model compartments (the state variables). As these models frequently describe exchanges in time, they are often referred to as 'dynamic' models, where time is the independent variable.

Several methods to solve differential equations have recently been implemented in the software R (**?**). They are included in package **deSolve** (**?**) which contains functions to integrate initial value problems of ordinary and partial differential equations, of delay differential equations, and differential algebraic equations, (among them most of the **ODEPACK** solvers, **?**), in package **ddesolve** (**?**) which provides a solver for delay differential equations; package **bvpSolve** (**?**) which solves boundary value problems and package **rootSolve** (**?**), offering functions

to estimate a system's steady-state (i.e., time-invariant) condition and to perform stability analysis.

In addition, several utility packages have been created to help in the modelling process. For instance, package **simecol** (**?**) provides a complete environment for solving and running dynamic models, **GillespieSSA** (**?**) implements the Gillespie Stochastic Simulation Algorithm, **ReacTran** (**?**) includes functions that describe (physical) transport in one, two or three dimensions. The R package **AquaEnv** (**?**) offers building blocks for pH and carbonate chemistry modelling, package **nlmeODE** (**?**) includes pharmacokinetics models. Because of these efforts, R is emerging more and more as a powerful environment for dynamic simulations (**??**).

Quantitative mathematical models depend on constant parameters, many of which are poorly known and cannot be measured. Thus, one essential step in the modelling process is model calibration, during which these parameters are estimated by fitting the model to data. This application of a model is also known as 'inverse' modelling, in contrast to 'forward' model applications, in which the model is used for forecasting or hypothesis testing. As the model equations are generally nonlinear, parameter estimation constitutes a non-linear optimization problem, where the objective is to find parameter values that minimise a measure of badness of fit, usually a least squares function, or a weighted sum of squared residuals. R contains both local and global search algorithms that are suitable for nonlinear optimization, in its base package (**?**) or in dedicated packages (**?**).

Apart from finding the *global* minimum, there exist many other challenges in inverse modelling. Many models comprise non-identifiable parameters which cannot be unambiguously determined with sufficient precision (**?**). Such non-identifiability is manifested by functionally related parameters, such that the effect of altering one parameter can be, at least partly, undone by altering some other parameter(s). This type of overparametrization is common for complex models and especially in ecological modelling nearly unavoidable (**?**). In order for the data fitting algorithms to converge, and for the parameters to be estimated with reasonable precision, the parameter set must be *identifiable*.

In addition, it is not only important to locate the best parameter values, but also to provide an estimate of the parameter *uncertainty*, and to quantify the effects of that uncertainty on other, unobserved, variables. The latter is necessary to evaluate the robustness of model-based predictions in the light of uncertain parameters. In addition, modelers do not necessarily want good estimates of the parameters; sometimes derived quantities are the object of interest.

Finally, although the methods from R's packages are efficient in solving a variety of differential equations, the computing time for solving these models is significantly larger than for a typical statistical application. Therefore, it becomes important to keep the number of runs to a minimum. This is especially necessary for Markov chain Monte Carlo (MCMC) methods, which generally require to run the model in the order of thousands of times for it to converge. One approach is to emulate the output of complex model codes and use this as input for formal Bayesian methods (**?**). For computationally expensive simulations that are run online, however, the MCMC functions already present in R are not the most efficient ones; other methods specifically aiming at dynamic models may be more suited (**?**).

FME is a package designed for inverse modelling, sensitivity and Monte Carlo analysis. It implements part of the functions from a Fortran simulation environment **FEMME** (**?**). It contains functions to

1. perform local and global sensitivity analysis (**??**), and Monte Carlo analysis,

2. estimate parameter identifiability using the method described in **?**,

3. fit a model to data, by providing a consistent interface to R's existing optimization methods; it also includes an implementation of the pseudo-random search method (**?**),

4. run a Markov chain Monte Carlo, to estimate parameter uncertainties. The DRAM method (Delayed Rejection Adaptive Metropolis) (**?**), which is well suited for use with dynamic models is implemented.

Most of the functions have suitable methods for printing and visualization.

In this paper, the potential of **FME** for inverse modelling is demonstrated by means of a simple 3-compartment dynamic model from the biomedical sciences that describes the dynamics of the HIV virus, responsible for the acquired immunodeficiency syndrome (AIDS). This model is chosen because it is relatively simple and its algebraic identifiability properties have been investigated by **?** and **?**. Also, the study of viral infection is of considerable interest in aquatic sciences, where viruses are deemed important factors in biogeochemical cycles, and causing death in a variety of organisms (**?**).

Similarly as in **?** the algorithms from **FME** are tested on simulated data to which random noise is added. Parameter estimation is done in several steps. First, the parameters to which the model is sensitive are identified and selected. Then an identifiability analysis allows to evaluate which set of model parameters can be estimated based on available observations. After fitting these parameters to the data, their uncertainty given the data is assessed using an MCMC method. Finally, by means of a sensitivity analysis the consequences of the uncertain parameters on the unobserved (latent) variables is calculated.

Although **FME** is used here with a dynamic compartment model, it can work with any type of model that calculates a response as a function of input parameters. **FME** is available from the Comprehensive R Archive Network at `http://CRAN.R-project.org/package=FME`.

2. The test model

The example models the dynamics of the HIV virus in human blood cells (Figure **??**)

The model describes three components, comprising the number of uninfected (T) and infected (I) CD4+ T lymphocytes, and the number of free virions (V). It consists of three differential equations:

$$\frac{dT}{dt} = \lambda - \rho T - \beta TV \tag{1}$$

$$\frac{dI}{dt} = \beta TV - \delta I \tag{2}$$

$$\frac{dV}{dt} = n\delta I - cV - \beta TV \tag{3}$$

with initial conditions (numbers at $t = 0$):

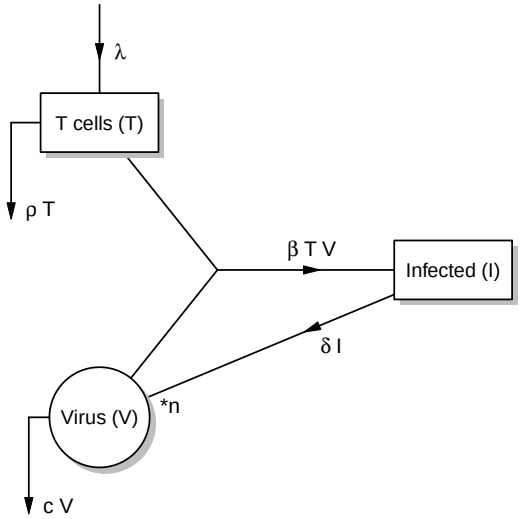

Figure 1: Schematic representation of the HIV test model.

$$T(0) = T_0$$
$$I(0) = I_0$$
$$V(0) = V_0$$

These equations express the rate of change of the components ($\frac{d.}{dt}$) as a sum of the sources minus the sinks. Uninfected cells are created from sources within the body (e.g., the thymus) at rate λ, they die off at a constant rate ρ, and become infected. The latter process is proportional to the product of the number of uninfected cells and the number of virions, by a parameter (β). δ is the death rate of infected cells, and n the number of virions that are released during lysis of one infected cell (the burst size); c is the rate at which virions disappear.

In practical cases, the parameters from this model are estimated based on clinical data obtained from individual patients. As it is more costly to measure the number of infected cells, I, (**?**), this compartment is often not monitored. The occurrence of unobserved variables is very common in mathematical models.

Here we assume that both the viral load and the number of healthy CD4+ T cells have been measured; the CD4+ T cells at 4 days intervals, the viral load at a higher frequency.

Without measurements of I, its initial condition, I_0, is not available. Therefore, it is estimated using equation (3) as:

$$I_0 = \frac{V_0' + cV_0}{n\delta}$$

where V_0' is the first derivative of the number of virions, estimated at the initial time. This can

be evaluated e.g., by fitting a spline through the initial points (**?**) or by simple differencing of the first observed data points.

2.1. Implementation in R

In R, this model is implemented as a function (`HIV_R`) that takes as input the parameter values (`pars`) and the initial conditions V_0, V_0', T_0, here called `V_0, dV_0` and `T_0` and that returns the model solution at selected time points.

Two versions of the model are given.

The first, `HIV_R` consists only of R code. The function contains the derivative function `derivs`, required by the integration routine (see help of **deSolve**). It calculates the rate of change of the three state variables (`dT, dI, dV`) and an output variable, the logarithm of the number of virions (`logV`). Viral counts are often represented logarithmically (**?**). After initialising the state variables (`y`), and specifying the output times (`times`), the model is integrated using **deSolve** function `ode` and the output returned as a `data.frame`.

```
R> HIV_R <- function (pars, V_0 = 50000, dV_0 = -200750, T_0 = 100) {
+
+    derivs <- function(time, y, pars) {
+      with (as.list(c(pars, y)), {
+        dT <- lam - rho * T - bet * T * V
+        dI <- bet * T * V - delt * I
+        dV <- n * delt * I - c * V - bet * T * V
+
+        return(list(c(dT, dI, dV), logV = log(V)))
+      })
+    }
+
+    # initial conditions
+    I_0   <- with(as.list(pars), (dV_0 + c * V_0) / (n * delt))
+    y     <- c(T = T_0, I = I_0, V = V_0)
+
+    times <- c(seq(0, 0.8, 0.1), seq(2, 60, 2))
+    out   <- ode(y = y, parms = pars, times = times, func = derivs)
+
+    as.data.frame(out)
+  }
```

In the second version of the model (`HIV`), the derivative function has been replaced by a subroutine written in Fortran, and presented to R as a DLL (a dynamic link library `FME.dll` on Windows respectively a shared library `FME.so` on other operating systems). This DLL contains two subroutines: `derivshiv` estimates the derivatives, and `inithiv` initialises the model. How to write model code in compiled languages is explained in vignette ("compiledCode") (**?**) in package **deSolve**. The Fortran code required for this second implementation can be found in the appendix; the DLL is part of the **FME** package.

```
R> HIV <- function (pars, V_0 = 50000, dV_0 = -200750, T_0 = 100) {
```

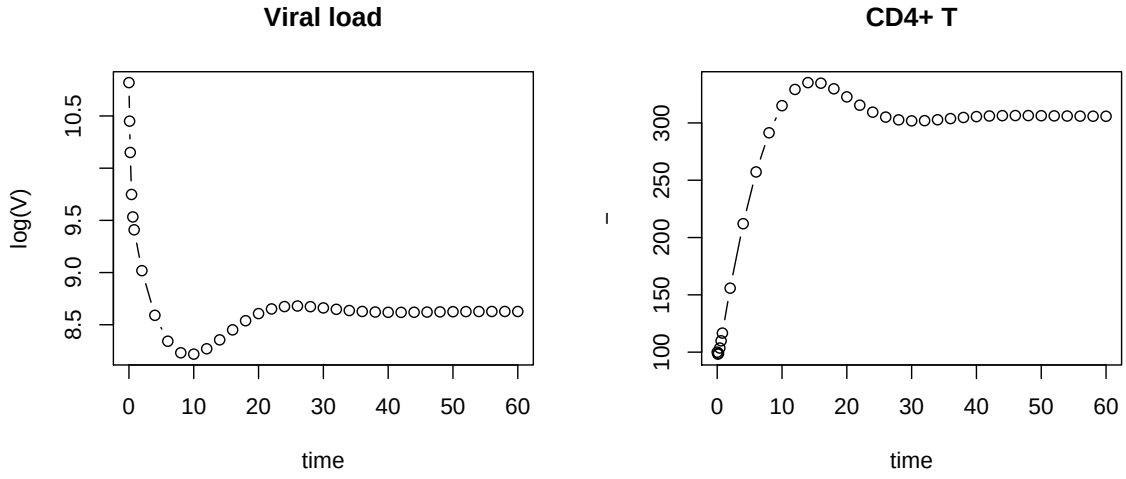

Figure 2: Viral load and number of uninfected T cells as a function of time.

```
+
+    I_0 <- with(as.list(pars), (dV_0 + c * V_0) / (n * delt))
+    y <- c(T = T_0, I = I_0, V = V_0)
+
+    times <- c(0, 0.1, 0.2, 0.4, 0.6, 0.8, seq(2, 60, by = 2))
+    out <- ode(y = y, parms = pars, times = times, func = "derivshiv",
+      initfunc = "inithiv", nout = 1, outnames = "logV", dllname = "FME")
+
+    as.data.frame(out)
+  }
```

After assigning values to the parameters and running the model, output is plotted (Figure **??**). It takes about 20 times longer to run the pure-R version, compared to the compiled version. As this will be significant when running the model multiple times, in what follows, the fast version (`HIV`) will be used.

```
R> pars <- c(bet = 0.00002, rho = 0.15, delt = 0.55, c = 5.5, lam = 80, n = 900)
R> out <- HIV(pars = pars)

R> par(mfrow = c(1, 2))
R> plot(out$time, out$logV, main = "Viral load", ylab = "log(V)",
+    xlab = "time", type = "b")
R> plot(out$time, out$T, main = "CD4+ T", ylab = "-", xlab = "time", type = "b")
R> par(mfrow = c(1, 1))
```

2.2. Observed data

The **FME** algorithms will be tested on simulated data. Such synthetic experiments are often

used to study parameter identifiability or to test fitting routines.

They involve the following steps: first "data" are generated by applying the model with known parameter values. To this output, a normally distributed error, with mean 0 and known standard deviation is added. Here the standard deviation is 0.45 for log (viral load) and 4.5 for the T cell counts (**?**). The data are in a matrix containing the time, the variable value and the standard deviation.

The virions have been counted at high frequency:

```
R> DataLogV <- cbind(time = out$time,
+              logV = out$logV + rnorm(sd = 0.45, n = length(out$logV)),
+              sd = 0.45)
```

The T cells are recorded at 4-days intervals; `[ii]` selects the model output that corresponds to these sampling times.

```
R> ii    <- which (out$time %in% seq(0, 56, by = 4))
R> DataT <- cbind(time = out$time[ii],
+                 T = out$T[ii] + rnorm(sd = 4.5, n = length(ii)),
+                 sd = 4.5)
R> head(DataT)
```

```
     time         T  sd
[1,]    0  97.54154 4.5
[2,]    4 206.91401 4.5
[3,]    8 292.08836 4.5
[4,]   12 336.39283 4.5
[5,]   16 332.48466 4.5
[6,]   20 320.31275 4.5
```

2.3. The model cost function

The model-data residuals and model cost are central to the parameter identifiability, model calibration and MCMC analysis.

Function `modCost` estimates weighted residuals of the model output versus the data and calculates sums of squared residuals, in an object of class `modCost`.

For any observed data point, `k`, of observed variable `l`, the *weighted and scaled residuals* are estimated as:

$$res_{k,l} = \frac{Mod_{k,l} - Obs_{k,l}}{error_{k,l} \cdot n_l}$$

where $Mod_{k,l}$ and $Obs_{k,l}$ are the modeled, respectively observed value.

$error_{k,l}$ is a weighing factor that makes the term non-dimensional; it can be chosen to be equal to the mean of all measurements, the overall standard deviation, or chosen to be a

different measurement error for each data point [1]. Weighing is important if different model variables have different units and magnitudes.

Some variables are measured at much higher resolution than others. In order to prevent the abundant data set to dominate the analysis, the residuals can also be scaled relative to the number of data points n_l for each variable `l`; by default n_l is 1.

Sums of these residuals per observed variable (the "variable" cost) and the total sum of squares (the "model" cost) are also estimated in function `modCost`.

For the HIV model example, the residuals and costs are estimated in a function (`HIVcost`) that takes as input the values of the parameters to be tested/fitted. The model cost is calculated in three steps. First, the model output, given the current parameter values is produced (`out`); then the residuals with the $\log(V)$ data, in matrix `DataLogV`, is estimated (`cost`); argument `err = "sd"` specifies the columnname with the weighting factors. Finally the cost is updated with the T cell observations in matrix `DataT`. Updating is done by passing the previously estimated cost (`cost = cost`) to function `modCost`.

```
R> HIVcost <- function (pars) {
+    out <- HIV(pars)
+    cost <- modCost(model = out, obs = DataLogV, err = "sd")
+    return(modCost(model = out, obs = DataT, err = "sd", cost = cost))
+  }
```

The sum of squared residuals is printed, and the residuals of model and data plotted, showing the random noise (Figure **??**).

```
R> HIVcost(pars)$model
```

```
[1] 47.55381
```

```
R> plot(HIVcost(pars), xlab="time")
```

3. Local sensitivity analysis

Not all parameters can be finetuned on a certain data set. Some parameters have little effect on the model outcome, while other parameters are so closely related that they cannot be fitted simultaneously.

Function `sensFun` estimates the sensitivity of the model output to the parameter values in a set of so-called sensitivity functions (**??**). When applied in conjunction with observed data, `sensFun` estimates, for each datapoint, the derivative of the corresponding modeled value with respect to the selected parameters. A schema of what these sensitivity functions represent can be found in Figure **??**.

[1] `modCost` assumes the measurement errors to be normally distributed and independent. If there exist correlations in the errors between the measurements, then `modCost` should not be used in the fitting or MCMC application, but rather a function that takes in to account the data covariances.

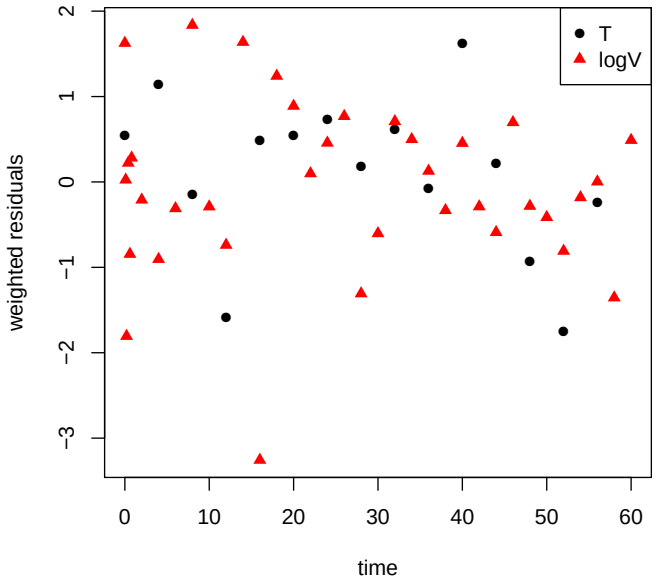

Figure 3: Residuals of model and pseudodata.

In **FME**, normalised, dimensionless sensitivities of model output to parameters are in a sensitivity matrix whose (i,j)th element $S_{i,j}$ contains:

$$\frac{\partial y_i}{\partial \Theta_j} \cdot \frac{w_{\Theta_j}}{w_{y_i}}$$

where y_i is an output variable, Θ_j is a parameter, and w_{y_i} is the scaling of variable y_i (usually equal to its value), w_{Θ_j} is the scaling of parameter Θ_j (usually equal to the parameter value).

These sensitivity functions can be collapsed into summary values. The higher the absolute sensitivity value, the more important the parameter, thus the magnitudes of the sensitivity summary values can be used to rank the importance of parameters on the output variables. As it makes no sense to finetune parameters that have little effect, this ranking serves to choose candidate parameters for model fitting.

In **FME**, sensitivity functions are estimated using function `sensFun` which takes as input the cost function (`HIVcost` that returns an instance of class `modCost`) and the parameter values.

```
R> Sfun <- sensFun(HIVcost, pars)
R> summary(Sfun)
```

```
       value   scale    L1    L2    Mean   Min  Max  N
bet  2.0e-05 2.0e-05 0.364 0.074 -0.1594 -1.28 0.30 51
rho  1.5e-01 1.5e-01 0.117 0.020 -0.1017 -0.34 0.16 51
delt 5.5e-01 5.5e-01 0.032 0.008  0.0014 -0.11 0.21 51
```

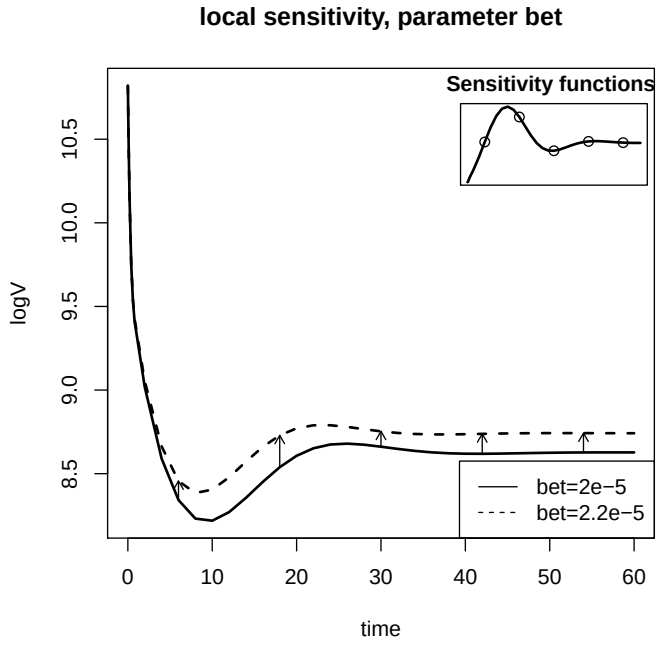

Figure 4: The sensitivity functions of `logV` to parameter bet, as a function of time (upper right) are the (weighted) differences of the perturbed output (at `bet = 2.2e-5`) with the nominal output (`bet = 2e-5`), main figure; the dots in the inset correspond to the arrows in the main figure.

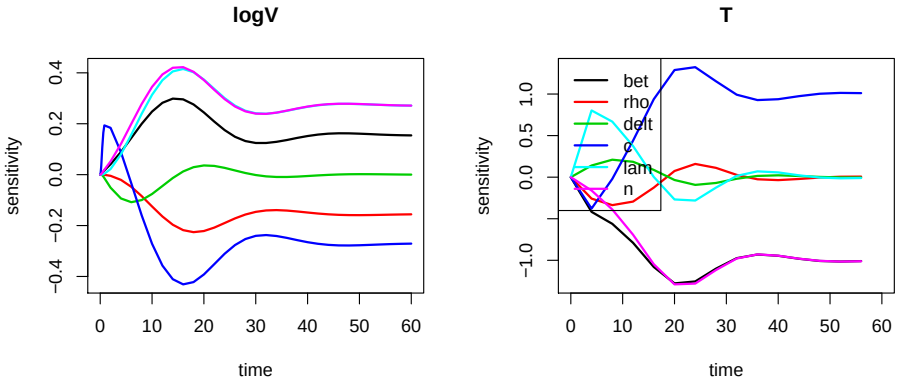

Figure 5: Sensitivity functions of model output to parameters.

```
c    5.5e+00 5.5e+00 0.414 0.076  0.0878 -0.43 1.32 51
lam  8.0e+01 8.0e+01 0.214 0.038  0.1863 -0.28 0.80 51
n    9.0e+02 9.0e+02 0.417 0.077 -0.0861 -1.29 0.42 51
```

Here L1 $= \sum |S_{ij}|/n$ and L2 $= \sqrt{\sum (S_{ij}^2)/n}$ are the L1 and L2 norm respectively.

Based on these summary statistics it is clear that parameter `delt` has the least effect on the output variables.

The sensitivities of the modelled viral and T cell counts to the parameter values change in time (see Figure **??**), thus it makes sense to visualise the sensitivity functions as they fluctuate. The plots are clearest if produced one per output variable (Figure **??**):

```
R> plot(Sfun, which = c("logV", "T"), xlab="time", lwd = 2)
```

As their corresponding sensitivity functions are always positive, parameters `bet, lam`, and `n` have a consistent positive effect on the number of free virions; higher values of `rho` consistently decrease `logV`. The initial positive effect on viral load when increasing viral loss (`c`) is caused by its impact on the calculated initial condition `I_0`.

There is strong similarity in several sensitivity functions for the output variable `logV`, indicating that the corresponding parameters have comparable effect on this output variable. If too similar, the joint estimation of these parameter combinations may not be possible on these observed data alone. The correlation between the sensitivity functions of `n` and `lam` is 1 (not shown), such that exactly the same output of `logV` will be generated by increasing `n`, if `lam` is decreased the appropriate amount. Similar findings were reported in **?**, based on an analytical analysis of parameter identifiability.

For output variable `T` the similarity between parameters `lam` and `n` and `bet` is also strong. Pairwise relationships are visualised with a `pairs` plot. Here we plot the sensitivity functions of both variables (Figure **??**) in one figure but with different colors; it is also instructive to select each variable separately (this can be done by means of the `which` argument – not shown).

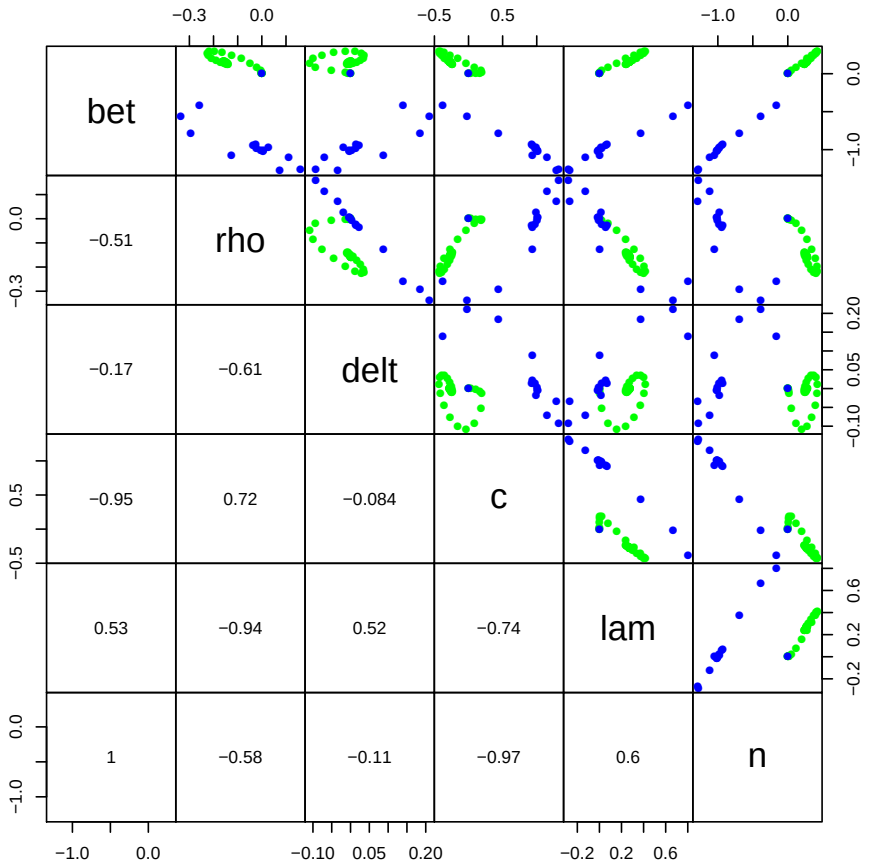

Figure 6: Pairwise plot of sensitivity functions.

```
R> pairs(Sfun, which = c("logV", "T"), col = c("blue", "green"))
```

The sensitivity functions for parameter pairs involving `bet`, `c` and `n`, and pair `rho` and `lam` are strongly correlated, with $r^2 > 0.85$.

It should be noted that none of the correlation coefficients is exactly 1 or -1, (the largest $|r|$ equals 0.995). Therefore, the model comprising both `logV` and `T` data is "algebraically" identifiable, as is indeed demonstrated by `?`. However, it is questionable whether the subtle differences produced in the output of some parameters will be sufficient to make them "practically" identifiable.

4. Multivariate parameter identifiability

The above `pairs` analysis investigated the identifiability of sets of *two* parameters. Function `collin` extends the analysis to all possible parameter combinations, by estimating the approximate linear dependence ("collinearity") of parameter sets. A parameter set is said to be identifiable, if all parameters within the set can be uniquely estimated based on (perfect)

measurements. Parameters that have large collinearity will not be identifiable [2].

The identifiability analysis included in **FME** was described in **?**. For any subset of columns of the sensitivity matrix, collinearity γ is defined as:

$$\gamma = \frac{1}{\sqrt{\min(\mathrm{EV}[\hat{S}^\top \hat{S}])}}$$

where

$$\hat{S}_{ij} = \frac{S_{ij}}{\sqrt{\sum_j S_{ij}^2}}$$

where $\hat{S}$ contains the columns of the sensitivity matrix that correspond to the parameters included in the set, EV estimates the eigenvalues. The collinearity index equals 1 if the columns are orthogonal, and the set is identifiable, it equals infinity if columns in the sensitivity matrix are linearly dependent.

A collinearity index γ means that a change in the results caused by a change in one parameter can be compensated by the fraction $1 - 1/\gamma$ by an appropriate change of the other parameters (**?**).

If the index exceeds a certain value, typically chosen to be 10–15, then the parameter set is poorly identifiable (**?**) (any change in one parameter can be undone for 90 respectively 93%).

The collinearity for all parameter combinations is estimated by function `collin`, taking the previously estimated sensitivity functions as argument.

```
R> ident <- collin(Sfun)
R> head(ident, n = 20)
```

```
   bet rho delt c lam n N collinearity
1    1   1    0 0   0 0 2          1.1
2    1   0    1 0   0 0 2          1.1
3    1   0    0 1   0 0 2          4.2
4    1   0    0 0   1 0 2          1.1
5    1   0    0 0   0 1 2          8.1
6    0   1    1 0   0 0 2          1.4
7    0   1    0 1   0 0 2          1.3
8    0   1    0 0   1 0 2          5.4
9    0   1    0 0   0 1 2          1.2
10   0   0    1 1   0 0 2          1.0
11   0   0    1 0   1 0 2          1.3
12   0   0    1 0   0 1 2          1.1
13   0   0    0 1   1 0 2          1.3
14   0   0    0 1   0 1 2          6.3
15   0   0    0 0   1 1 2          1.2
16   1   1    1 0   0 0 3          1.5
17   1   1    0 1   0 0 3          7.0
18   1   1    0 0   1 0 3          5.4
```

[2] The reverse need not be the case, as unidentifiable parameters may also be non-linearly related.

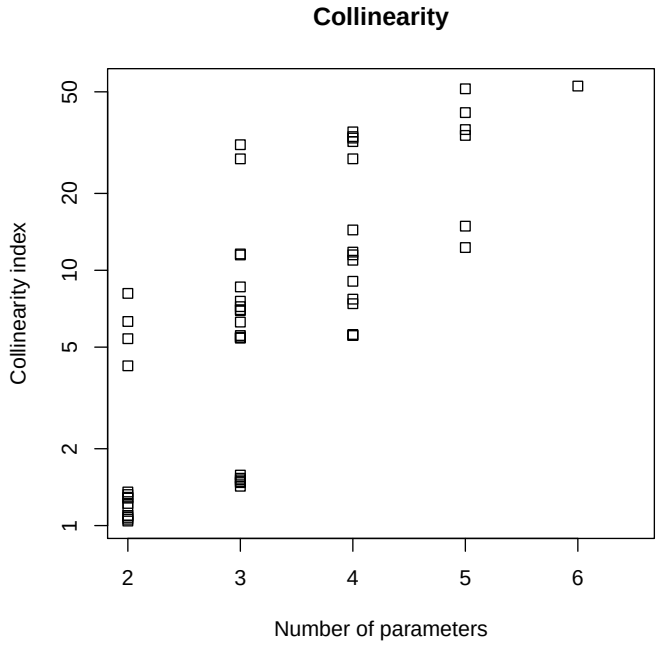

Figure 7: Collinearity plot.

```
19   1   1    0 0   0 1 3         27.3
20   1   0    1 1   0 0 3          6.3
```

In the output, `1` and `0` denotes that the parameter is included respectivity not included in the set; `N` is the number of parameters in the set.

The first 20 combinations show very large collinearity when parameters `bet`, `rho` and `n` are in the parameter set. Figure **??** shows how the collinearity index increases as more and more parameters are included in the set.

```
R> plot(ident, log = "y")
```

All parameters together have a collinearity which is too large for them to be fitted to the data. Thus, whereas **?** showed the full parameter set to be algebraically identifiable, in practical applications this may not be the case.

```
R> collin(Sfun, parset = c("bet", "rho", "delt", "c", "lam", "n"))
```

```
  bet rho delt c lam n N collinearity
1   1   1    1 1   1 1 6          53
```

One 5-parameter combination has collinearity lower than 15 (see Figure **??**), and is therefore possibly identifiable (according to **?**).

```
R> collin(Sfun, N = 5)

  bet rho delt c lam n N collinearity
1   1   1    1 1   1 0 5           12
2   1   1    1 1   0 1 5           51
3   1   1    1 0   1 1 5           41
4   1   1    0 1   1 1 5           34
5   1   0    1 1   1 1 5           36
6   0   1    1 1   1 1 5           15
```

We select parameters `bet, rho, delt, c,` and `lam`, the 5-parameter combination with the smallest collinearity for fitting. Note that this parameter set cannot be identified on either `logV` nor `T` data alone:

```
R> collin(Sfun, parset = c("bet", "rho", "delt", "c", "lam"), which = "logV")

  bet rho delt c lam n N collinearity
1   1   1    1 1   1 0 5           60

R> collin(Sfun, parset = c("bet", "rho", "delt", "c", "lam"), which = "T")

  bet rho delt c lam n N collinearity
1   1   1    1 1   1 0 5           60
```

5. Fitting the model to data

R has several built-in methods for nonlinear data fitting. Whereas the default optimization algorithms require one function value (the weighted sum of squares, a "model cost"), other algorithms such as the Levenberg-Marquardt method (**?**) need input of a vector of residuals.

To allow using both types of algorithms, a wrapper (`modFit`) is provided that has a consistent interface, taking as input argument either the vector of model-data residuals or an instance of class `modCost`.

Function `modFit` embraces the functions from `optim`, `nls`, and function `nlminb`, from R's base packages (**?**) and the Levenberg-Marquardt algorithm from package **minpack.lm** (**?**) In addition to the stochastic simulated annealing method as implemented in `optim`, another random-based method, the pseudo-random search algorithm of Price (**??**) is implemented in **FME**.

To fit the HIV model to the data, a new function is needed that takes as input the logarithm of all parameter values except `n` (which is given a fixed value `900`), and that returns the model cost.

The log transformation (1) ensures that the parameters remain positive during the fitting, and (2) deals with the fact that the parameter values are spread over six orders of magnitude (i.e., `bet = 2e-5`, `lam = 80`). Within the function `HIVcost2`, the parameters are backtransformed (`exp(lpars)`).

```
R> HIVcost2 <- function(lpars)
+    HIVcost(c(exp(lpars), n = 900))
```

After perturbing the parameters[3], the model is fitted to the data, and best-fit parameters and residual sum of squares shown.

```
R> Pars <- pars[1:5] * 2
R> Fit <- modFit(f = HIVcost2, p = log(Pars))
R> exp(coef(Fit))

         bet          rho         delt            c          lam
2.133316e-05 1.433836e-01 5.877661e-01 5.871155e+00 8.094194e+01

R> deviance(Fit)

[1] 44.62702
```

For comparison, the initial model output and the best-fit model are plotted against the data (Figure **??**).

```
R> ini   <- HIV(pars = c(Pars, n = 900))
R> final <- HIV(pars = c(exp(coef(Fit)), n = 900))

R> par(mfrow = c(1,2))
R> plot(DataLogV, xlab = "time", ylab = "logV", ylim = c(7, 11))
R> lines(ini$time, ini$logV, lty = 2)
R> lines(final$time, final$logV)
R> legend("topright", c("data", "initial", "fitted"),
+     lty = c(NA,2,1), pch = c(1, NA, NA))
R> plot(DataT, xlab = "time", ylab = "T")
R> lines(ini$time, ini$T, lty = 2)
R> lines(final$time, final$T)
R> par(mfrow = c(1, 1))
```

Approximate estimates of parameter uncertainty can be obtained by linearising the model around the best-fit parameters. If J is the numerical approximation of the Jacobian, then, based on linear theory, the parameter covariance is estimated as $(J^\top J)^{-1}S^2$ where S^2 is the sum of squared residuals of the best-fit. At the best fit, $(J^\top J) \approx 0.5H$, with H the Hessian (**?**). The Hessian is estimated in most of R's optimization functions.

The `summary` method of the `modFit` function estimates these approximate statistical properties (not shown).

6. MCMC

The previously applied identifiability analysis (Section **??**) gives insight into which parameters can be simultaneously estimated, given noise-free data and a perfect model (i.e., one that

[3]Perturbation is done mainly for the purpose of making the application more challenging.

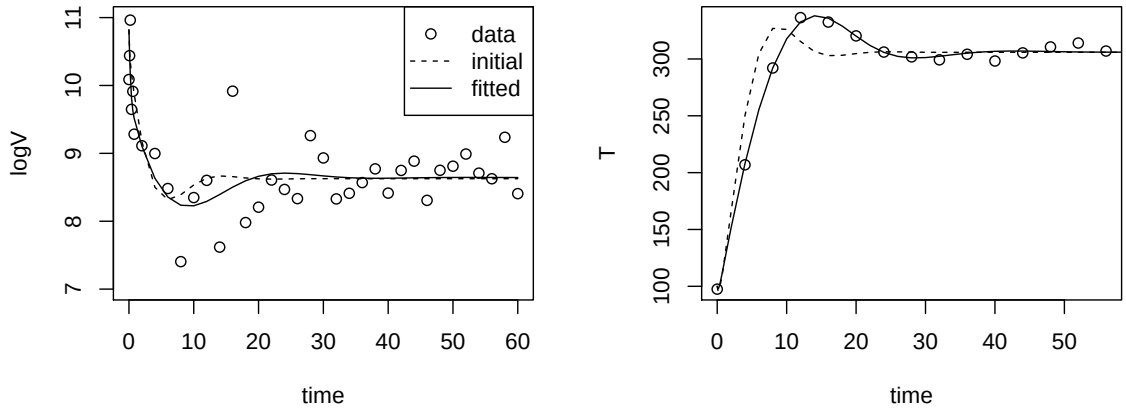

Figure 8: Best-fit and initial model run.

can fit the data perfectly). The model fitting (Section **??**) provided one "optimal" set of parameters, that produces the best fit to the measurements in the least squares sense.

However, even for perfectly identifiable parameter sets, the uncertainty may be very high or poor estimates may be obtained, if the data have too much noise. In practice, all measurements have error, thus it is important to quantify the effect of this on the parameter uncertainty.

Bayesian methods can be used to derive the data-dependent probability distribution of the parameters. Function `modMCMC` implements a Markov chain Monte Carlo (MCMC) method that uses the delayed rejection and adaptive Metropolis (DRAM) procedure (**??**). An MCMC method samples from probability distributions by constructing a Markov chain that has the desired distribution as its equilibrium distribution. Thus, rather than one parameter set, one obtains an ensemble of parameter values that represent the parameter distribution.

In the adaptive Metropolis method, the generation of new candidate parameter values is made more efficient by tuning the proposal distribution to the size and shape of the target distribution. This is realised by generating new parameters with a proposal covariance matrix that is estimated by the parameters generated thus far.

During delayed rejection, new parameter values are tried upon rejection by scaling the proposal covariance matrix. This provides a systematic remedy when the adaptation process has a slow start (**?**).

In the implementation in **FME**, it is assumed that the prior distribution for the parameters θ is either non-informative or gaussian.

If y, the measurements are defined as:

$$\begin{aligned} y &= f(x,\theta) + \xi \\ \xi &\sim N\left(0, \sigma^2\right) \end{aligned}$$

where $f(x,\theta)$ is the (nonlinear) model, x are the independent variables, θ the parameters, and

ξ is the additive, independent Gaussian error, with unknown variance σ^2. Then the posterior for the parameters will be estimated as (**?**):

$$p\left(\theta|y,\sigma^2\right) \propto \exp\left(-0.5 \cdot \left(\frac{SS(\theta)}{\sigma^2}\right)\right) \cdot p_{pri}(\theta)$$

where SS is the sum of squares function $SS(\theta) = \sum(y_i - f(x,\theta)_i)^2$, $p_{pri}(\theta)$ is the prior distribution of the parameters. For noninformative priors $p_{pri}(\theta)$ is constant for all values of θ (and can be ignored).

The error variance σ^2 is considered a nuisance parameter (**?**). A prior distribution needs to be specified and a posterior distribution is calculated by `modMCMC`. For the reciprocal of the error variance (σ^{-2}), a Gamma distribution is used as a prior:

$$p_{pri}\left(\sigma^{-2}\right) \sim \Gamma\left(\frac{n_0}{2}, \frac{n_0}{2}S_0^2\right)$$

At each MCMC step then, the reciprocal of the error variance is sampled from a gamma distribution (**?**):

$$p\left(\sigma^{-2}|(y,\theta)\right) \sim \Gamma\left(\frac{n_0+n}{2}, \frac{n_0 S_0^2 + SS(\theta)}{2}\right)$$

In function `modMCMC`, the corresponding input arguments for this Gamma distribution are `var0` $= S_0^2$ and `n0 = wvar0 * n`, and where `wvar0` or `n0` are input arguments to the function; `n` is the number of observations. Larger values of `wvar0` keep the sampled error variance closer to `var0`.

The MCMC method is now applied to the example model. In order to prevent long burn-in, the algorithm is started with the optimal parameter set (`Fit$par`) as returned from the fitting algorithm, while the prior error variance `var0` is chosen to be the mean of the unweighted squared residuals from the model fit (`Fit$var_ms_unweighted`); one for each observed variable (i.e., one for `logV`, one for `T`). The weight added to this prior is low (`wvar0 = 0.1`), such that this initial value is not so important.

The proposal distribution (used to generate new parameters) is updated every 50 iterations (`updatecov`). The initial proposal covariance (`jump`) is based on the approximated covariance matrix, as returned by the `summary` method of `modFit`, and scaled appropriately (**?**).

```
R> var0 <- Fit$var_ms_unweighted
R> cov0 <- summary(Fit)$cov.scaled * 2.4^2/5

R> MCMC <- modMCMC(f = HIVcost2, p = Fit$par, niter = 5000, jump = cov0,
+                  var0 = var0, wvar0 = 0.1, updatecov = 50)
```

Before plotting the results, the parameters in the chain are backtransformed; a `summary` is calculated. Alternatively, it is possible to calculate the summary via use of **coda**'s function summary (**?**); `summary(as.mcmc(MCMC$pars))` does this; this amongst other also gives a robust estimate of the parameter's standard error.

```
R> MCMC$pars <- exp(MCMC$pars)
R> summary(MCMC)
```

```
              bet        rho       delt         c
mean 2.161291e-05 0.14230799 0.58541968 5.9477168
sd   1.263971e-06 0.01541633 0.06289535 0.3581417
min  1.810476e-05 0.09675580 0.36926512 4.9442662
max  2.802711e-05 0.19763171 0.79659248 7.7508107
q025 2.072955e-05 0.13260306 0.54510112 5.7016033
q050 2.142912e-05 0.14244339 0.58476317 5.9057750
q075 2.221933e-05 0.15204020 0.62630211 6.1265227
          lam  sig.var_T sig.var_logV
mean 81.459503  21.801547   0.21264480
sd    4.403374  10.123970   0.05156229
min  68.302192   6.833226   0.09381529
max  99.015391 126.103886   0.52693385
q025 78.660030  15.212245   0.17644832
q050 81.302234  19.627388   0.20518809
q075 83.989885  25.794611   0.23964465
```

The error variances used to generate the perturbed data were $4.5^2 = 20.25$ and $0.45^2 = 0.2025$ for T and $\log(V)$ respectively and are to be compared with the mean in the columns labeled `sig.var` in the summary output. Due to the randomness involved, the used value is never exactly retrieved. The example was run 15 times, during which the variance sampled varied between 11–42 for T and 0.16–0.33 for $\log(V)$ respectively.

The MCMC chain is plotted, including the sampled error variances (`Full=TRUE`):

```
R> plot(MCMC, Full = TRUE)
```

The traces of the MCMC chain (grey line in Figure **??**) show that the chain has converged (there is no apparent drift). Note the error variances for each observed variable (last figure).

The pairs plot (Figure **??**) shows the strong relation between parameters `bet` and `c`. This plot visualises the pairwise relationship in the upper panel, the correlation coefficients in the lower panel, and the marginal distribution for each parameter, represented by a histogram, on the diagonal. To keep the size of the pairs plot reasonable, only 1000 parameters are plotted in the upper panels (`nsample = 1000`); but all samples are used to generate the histograms on the diagonal, and to estimate the pairwise correlations as shown in the lower panel. Note the large correlation between parameters `bet` and `c`.[4]

```
R> pairs(MCMC, nsample = 1000)
```

7. Model prediction

The effect of the parameter uncertainty on the model output can be estimated and visualised with function `sensRange`. This function takes as input the sample of the parameter probability

[4]Correlations were already large in the initial covariance matrix (argument jump). However, the results remain the same if a less good initial jump distribution is used.

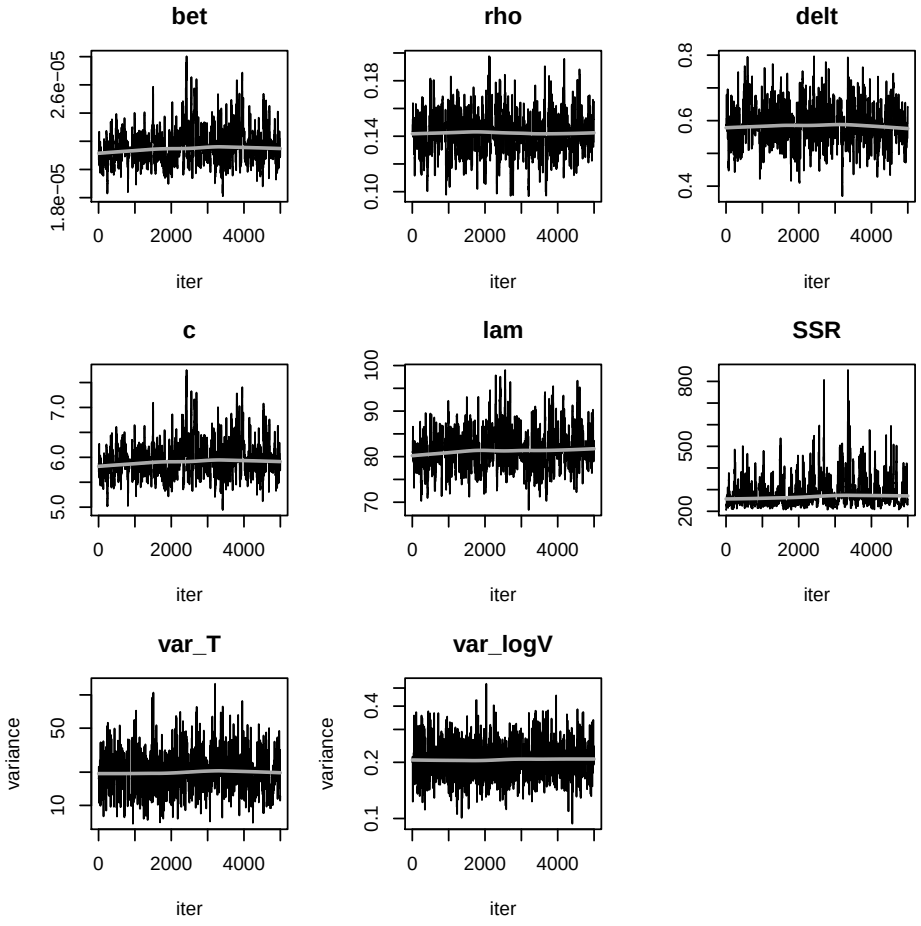

Figure 9: Results of the MCMC application.

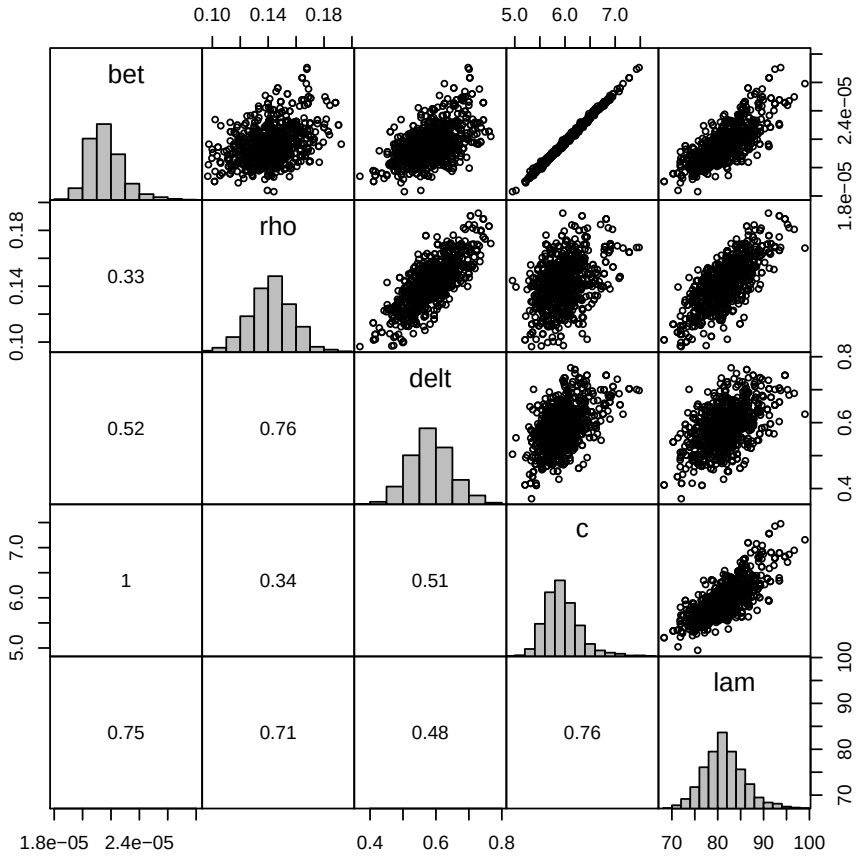

Figure 10: Pairs plot of the MCMC application.

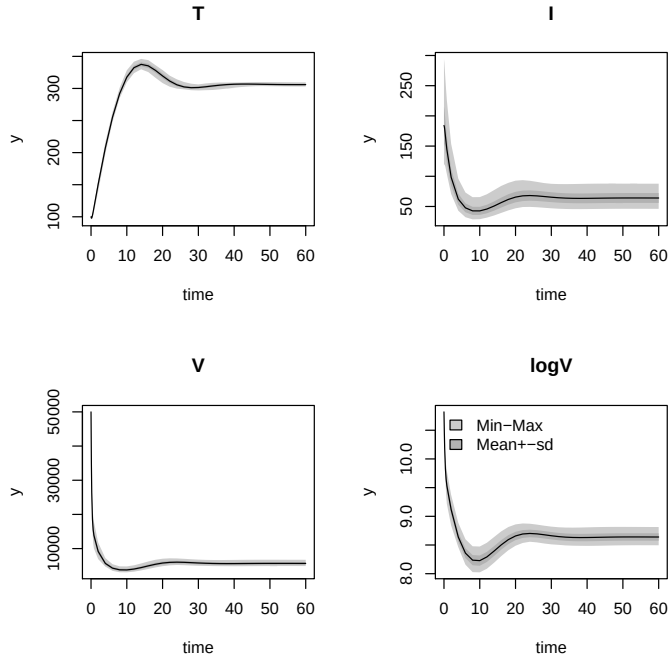

Figure 11: Sensitivity range based on parameter distribution as generated with the MCMC application.

density function as generated by `modMCMC`, and which is saved in `MCMC$par`. `sensRange` then executes the model 100 times, using a random draw of the parameters in the chain, and for each run output is saved.

The `summary` method estimates mean, standard deviation and quantiles based on these outputs, which can be visualised.

```
R> sR <- sensRange(func = HIV, parms = pars, parInput = MCMC$par)

R> plot(summary(sR), xlab = "time")
```

The figure (Figure **??**) shows amongst other the effect of parameter uncertainty on the unobserved variable `I`.

It should be noted that these ranges only represent the distribution of the model response as a function of the parameter values, generated by the MCMC. Another source of error is related to measurement noise as represented by the sampled values of the model variance. How this can be included is explained in **FME** vignette "`FMEother`" (`vignette("FMEother")`).

8. Monte Carlo applications

The sensitivity functions (Section **??**) explore the sensitivity of the model output at specific parameter values, i.e., they are *local* sensitivity measures. In contrast, *global* sensitivity

analyses determine the effect on model outcome as a function of an appropriate parameter probability density function.

The sensitivity ranges from the previous section (Figure **??**) were one example of a global sensitivity analysis, where the model outcome consisted of a time series. Function `modCRL` tests the effects on single output variables.

In the next application, all parameters are allowed to vary over 50% about their nominal value and the effect of that on the mean viral load is estimated. A function `crlfun` that takes as input the parameter values and outputs the mean viral load corresponding to these parameters is created first.

The correlation between mean virus load and the parameters is printed, and the relationships plotted (Figure **??**).

```
R> parRange <- cbind(min = 0.75 * pars, max = 1.25 * pars)
R> crlfun <- function (pars)  return(meanVirus = mean(HIV(pars)$V))
R> CRL <- modCRL(fun = crlfun, parRange = parRange, num = 500)
R> cor(CRL)[7, ]

        bet         rho        delt           c
 0.29735998 -0.29692142 -0.07594095 -0.35580057
        lam           n          X1
 0.50696004  0.53765543  1.00000000

R> plot(CRL, ylab = "number of virions", trace = TRUE)
```

9. Discussion

FME provides a comprehensive environment for the application of nonlinear models to data. Its functions can be used for identifying fine-tunable parameters, and for parameter fitting. They estimate parameter corelations and uncertainty, as well as the uncertainty in the model prediction curves which are due to uncertain parameters.

As the functions use numerical approximations, rather than rigorous analytical derivations of system properties, the **FME** functions can be readily applied in real cases, although its results are only approximations, the accuracy of which must be evaluated on a case-by-case basis.

Nevertheless, **FME**s parameter identifiability analysis, applied to the HIV model yielded similar conclusions as obtained in **?** and **?**, who, based on analytical derivations outlined the conditions under which the model was theoretically identifiable. However, in addition to this, it is shown here that this theoretical result may not necessarily imply practical identifiability, given the data uncertainties.

Whereas **FME** has been used here with a *dynamic* simulation model, its functions are more generally applicable. Four vignettes elucidate slightly different functionalities of the package. Vignette "`FMEdyna`" comprises a dynamic simulation example, similar as in the current paper, but putting more emphasis on sensitivity and Monte Carlo analysis, in lack of data. Vignette "`FMEsteady`" applies the functions to a mechanistic model describing oxygen dynamics in submersed sediments, and which is solved using one of **rootSolve**'s steady-state solvers. Vignette

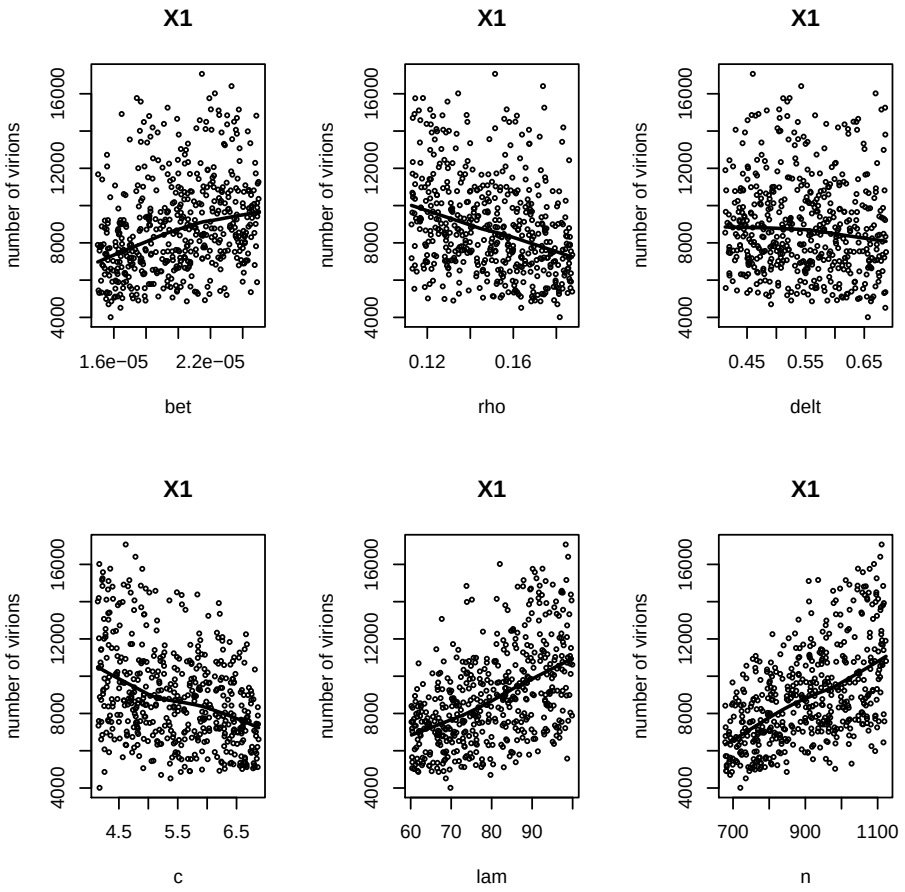

Figure 12: Global sensitivity; mean virus number (averaged over the simulation interval) as a function of the parameter values; parameters were generated according to a uniform distribution; solid line = lowess smoother.

"FMEother" develops a general nonlinear application, fitting a Monod function to experimental data. Finally "FMEmcmc" tests and demonstrates the functionality of the implemented Markov chain Monte Carlo method, e.g., using problems for which the analytical solution is known.

MCMC methods often suffer the curse of dimensionality, such that (1) efficient algorithms are needed to speed up convergence of the Markov chain, and (2) fast solution methods should keep the simulation time within acceptible limits. The first was achieved by implementing the delayed rejection and adaptive Metropolis algorithm (**?**), which has proven its worth in ecological applications (eg., **?**). The latter was ensured by using a Fortran implementation of the model. Bayesian approaches are implemented in R package **BACCO** (**?**) as well. However, the computation time in **BACCO** is reduced by emulating computationally expensive complex models with cheaper statistical estimates. In contrast, **FME** usually works with the original models directly and therefore can be used only with models of intermediate complexity, where one run is in the order of seconds, at most minutes. Just as a term of reference for the simulation time; the R code from this paper was excuted using `Sweave` (**?**). During the "weaving" process, the runs are executed, the graphs are created and written to file, and the LaTeX file written. So, one can use the time it takes to do that as an upper bound on the simulation time. It took about 70 seconds on an Intel Core (TM)2 Duo CPU T9300 2.5 GHz pentium PC with 3 GB of RAM to execute `Sweave`. With more than 5500 runs of the model performed here, this means that it takes less than 12 milliseconds to perform one run. These CPU times will almost certainly be beaten by methods fully implemented in low-level languages, but nevertheless, if one adds to the short simulation times the powerful post-processing capabilities of the R language, R emerges as a potent tool for mechanistic modelling.

Acknowledgments

The authors would like to thank Dick van Oevelen, Anna de Kluyver, Karel van den Meersche, Tom Cox and Pieter Provoost, students and post-docs who have tested the package.

The delayed rejection part of the DRAM MCMC method greatly benefited from the MATLAB implementation of this method by Marko Laine, for which Marko kindly gave permission of use. Marko Laine is also thanked for commenting on the R implementation.

We thank two anonymous reviewers for their constructive comments on this paper and the code.

A. The Fortran version of the model

The Fortran version of the model consists of two subroutines.

`inithiv` initialises the parameters of the model:

```
subroutine inithiv(odeparms)
external odeparms
double precision parms(6)
common /myparms/parms
 call odeparms(6, parms)
 return
```

Function	Description	S3 methods
`sensFun`	Sensitivity functions	`summary, plot, pairs, print.summary, plot.summary`
`sensRange`	Sensitivity ranges	`summary, plot, plot.summary`
`modCRL`	Monte Carlo (what-if)	`summary, plot, pairs, hist`
`modCost`	Model-data residuals, cost	`plot`
`modFit`	Fits a model to data	`summary, deviance, coef, residuals, df.residual, print, plot`
`modMCMC`	Markov chain Monte Carlo	`summary, plot, pairs, hist`
`collin`	Parameter collinearity	`print, plot`

Table 1: Summary of the main functions in package **FME**, with S3-methods.

```
    end
```

`derivshiv` estimates the rate of change.

```
    subroutine derivshiv (neq, t, y, ydot, yout, ip)
    double precision t, y, ydot, yout
    double precision bet,rho,delt,c,lam,N
    common /myparms/ bet,rho,delt,c,lam,N
    integer neq, ip(*)
    dimension y(3), ydot(3), yout(1)

    if(ip(1) < 1) call rexit("nout should be at least 1")
    ydot(1) = lam -rho*y(1) - bet*y(1)*y(3)
    ydot(2) = bet*y(1)*y(3) -delt*y(2)
    ydot(3) = N*delt*y(2) - c*y(3) - bet*y(1)*y(3)

    yout(1) = log(y(3))
    return
    end
```

Assuming that the file containing this code is called `fme.f`, then it can be compiled by writing, within R:

```
R> system("R CMD SHLIB fme.f")
```

which will create a DLL called `fme.dll`. This DLL needs to be loaded

```
R> dyn.load("fme.dll")
```

Note that to reproduce this example compiling and loading is not necessary, as the compiled version of the HIV model has been made part of the **FME** package.

Affiliation:

Karline Soetaert
Centre for Estuarine and Marine Ecology (CEME)
Netherlands Institute of Ecology (NIOO)
4401 NT Yerseke, The Netherlands
E-mail: k.soetaert@nioo.knaw.nl
URL: http://www.nioo.knaw.nl/users/ksoetaert/

Thomas Petzoldt
Institut für Hydrobiologie
Technische Universität Dresden
01062 Dresden, Germany
E-mail: thomas.petzoldt@tu-dresden.de
URL: http://tu-dresden.de/Members/thomas.petzoldt/