Software Documentation

GUTS: Software for the Calculation of the Likelihood Function of the GUTS Model

Version 0.4.1

Carlo Albert[*] Sören Vogel[**]

4 Mar 2015

GUTS is a software for the fast calculation of the logarithm of the likelihood of an empirical survival model. **GUTS** is currently available as an $\mathcal{R}$-package. This document describes the software as well as its typical usage.

Contents

Preliminaries

This document was created using “LaTeX” and “Sweave” (package Sweave, Leisch, 2002) with $\mathcal{R}$, version 3.1.2 (R Core Team, 2014). A function is written `function()`, a package is written Package, $\mathcal{R}$ input is marked *`R: input…`*, and $\mathcal{R}$ output is marked `output`. All $\mathcal{R}$ code is set in a framed box. Note that it may be required to install the latest version of $\mathcal{R}$ in order to run the software and the examples provided below (see section 5).

[*] EAWAG, 8600 Dübendorf, Switzerland, carlo.albert@eawag.ch

[**] soeren.vogel@posteo.ch

1 Theoretical Background

GUTS Jager, Albert, Preuss and Ashauer (2011) is a model for survival of organisms, exposed to any kind of quantifiable stress. The time-dependent stressor, $C(t)$, is assumed to cause a time-dependent damage, $D(t)$, which is described by the linear differential equation

$$\dot{D}(t) = k_r(C(t) - D(t))\,, \tag{1}$$

where k_r is called *recovery rate*. The damage is the same for all individuals. However, the individuals are assumed to have different thresholds, beyond which the damage increases their probability to die. Thus, the model combines two sources of stochasticity: On the one hand, death is considered a stochastic event, whose probability increases linearly with the damage, once it exceeds a certain threshold. That is, there is stochasticity at individual level. On the other hand, this threshold is assumed to vary stochastically over the population. Thus, there is stochasticity at population level.

The hazard, $h_z(t)$, of an individual with threshold z is determined by the formula

$$h_z(t) = k_k \max(D(t) - z, 0) + h_b\,, \tag{2}$$

where k_k is called *killing rate* and h_b is the *background mortality*. The hazard, in turn, determines the individual's probability to survive until time t, $S_z(t)$, via the linear differential equation

$$\dot{S}_z(t) = -h_z(t)S_z(t)\,. \tag{3}$$

Finally, each individual is assumed to draw its z from a distribution, $f_{\boldsymbol{\theta}}(z)$, on the positive real axis. Hence, the parameter vector of the model reads as

$$\boldsymbol{\theta} = (h_b, k_r, k_k, \dots)\,, \tag{4}$$

where the additional arguments are supposed to determine the distribution $f_{\boldsymbol{\theta}}(z)$.

Combining equations (2) and (3), we find that the probability for an arbitrarily chosen member of the population to survive until time t is given by the formula

$$S_{\boldsymbol{\theta}}(t) = \int \exp\left(-k_k \int_0^t \max(D(\tau) - z, 0)d\tau - h_b t\right) f_{\boldsymbol{\theta}}(z)dz\,. \tag{5}$$

Let $\mathbf{y} = (y_0, y_1, \dots, y_n)$ denote a time series of survivors, counted at times $(t_0 = 0, t_1, \dots, t_n)$, and set $y_{n+1} = 0$. Then, the logarithm of the likelihood, $f(\mathbf{y}|\boldsymbol{\theta})$, of the model output $\mathbf{y}$ given the parameters $\boldsymbol{\theta}$ is, up to $\boldsymbol{\theta}$-independent terms, given by the formula

$$\ln f(\mathbf{y}|\boldsymbol{\theta}) = \sum_{i=1}^{n+1} (y_{i-1} - y_i) \ln(S_{\boldsymbol{\theta},i-1} - S_{\boldsymbol{\theta},i})\,, \tag{6}$$

where we have set

$$S_{\boldsymbol{\theta},i} = S_{\boldsymbol{\theta}}(t_i)\,, \quad S_{\boldsymbol{\theta},n+1} = 0\,. \tag{7}$$

2 The Algorithm

The calculation of the log-likelihood requires two numerical integrations (see eq. (5)), and has, therefore, two large numbers, N and M. The following algorithm is of the order $\mathcal{O}(N) + \mathcal{O}(M)$. It is based on the approximation

$$\begin{align}
S_i &= \int \exp\left[-k_k \int_0^{t_i} \max(0, D(\tau) - z)d\tau - h_b t_i\right] f_{\boldsymbol{\theta}}(z)dz \tag{8}\\
&\approx \frac{1}{N}\sum_{j=1}^{N} \exp\left[-k_k\Delta\tau \sum_{D_l > z_j} (D_l - z_j) - h_b t_i\right] \tag{9}\\
&= \frac{1}{N} e^{-h_b t_i}\Big(e^{-k_k\Delta\tau(e_N - z_N f_N)} + e^{-k_k\Delta\tau(e_N + e_{N-1} - z_{N-1}(f_N + f_{N-1}))} + \dots \tag{10}\\
&\quad + e^{-k_l\Delta\tau(e_N + \dots + e_1 - z_1(f_N + \dots + f_1))}\Big)\,, \tag{11}
\end{align}$$

for an ordered sample $z_1 < \cdots < z_N$ from $f_{\boldsymbol{\theta}}(z)$, and with $D_l = D(\tau_l)$ on a grid $\tau_0 < \cdots < \tau_{M-1}$. The inner sum in the second line extends over all D_l, for which $\tau_l < t_i$, and we have set $\Delta\tau = t_n/M$. Furthermore,

$$e_j = \sum_{z_j < D_l < z_{j+1}} D_l\,, \tag{12}$$

and

$$f_j = \sharp\{D_l | z_j < D_l < z_{j+1}\}\,, \tag{13}$$

for $1 \le j \le N$ (Set $z_{N+1} = \infty$).

The corresponding algorithm for the calculation of (6) reads as follows:

1. Draw N thresholds from $f_{\boldsymbol{\theta}}(z)$ and order them $z_1 < \cdots < z_N$.

2. Refine the grid $t_0 < \cdots < t_n$ to a fine grid $\tau_0 < \cdots < \tau_{M-1}$.

3. Set $i = 0$.

4. Solve eq. (1), for $t_i \le \tau_l \le t_{i+1}$, using equation

$$\begin{aligned} D_l = D(\tau_l) = D(s_k)e^{-k_r(\tau_l - s_k)} + C_k\left(1 - e^{-k_r(\tau_l - s_k)}\right)& \\ + \frac{C_{k+1} - C_k}{s_{k+1} - s_k}\left(\tau_l - s_k - k_r^{-1} + k_r^{-1}e^{-k_r(\tau_l - s_k)}\right)&\,, \end{aligned} \tag{14}$$

   for $s_k \le \tau_l \le s_{k+1}$.

5. Update eq. (12) and eq. (13), for $1 \le j \le N$. (This can be done in time $\mathcal{O}(1)$, for each D_l.)

6. Calculate S_i using the recursion:

$$\begin{align}
F_j &= F_{j+1} + f_j\,, \tag{15}\\
E_j &= E_{j+1} + e_j\,, \tag{16}\\
S_{i,j} &= S_{i,j+1} + \exp\left(-k_k\Delta\tau(E_j - F_j z_j)\right)\,, \tag{17}
\end{align}$$

   for $j = N-1, \dots, 1$ and with $S_{i,N} = \exp(-k_k\Delta\tau(E_N - F_N z_N))$ and $F_N = f_N$, $E_N = e_N$. Then,

$$S_i = \frac{1}{N}e^{-h_b t_i} S_{i,1}\,. \tag{18}$$

7. Increment i and go to step 4.

8. Calculate the log-likelihood function according to equation (6).

A speed-up is achieved via importance sampling. That is, instead of sampling from $f_{\boldsymbol{\theta}}(z)$ we sample from a different distribution, $g_{\boldsymbol{\theta}}(z)$, and correct with weights. That is, we use

$$\begin{aligned} S_i &= \int \exp\left[-k_k \int_0^{t_i} \max(0, D(\tau) - z) d\tau - h_b t_i\right] f_{\boldsymbol{\theta}}(z) dz && (19) \\ &= \int \exp\left[-k_k \int_0^{t_i} \max(0, D(\tau) - z) d\tau - h_b t_i + \ln(f_{\boldsymbol{\theta}}(z)/g_{\boldsymbol{\theta}}(z))\right] g_{\boldsymbol{\theta}}(z) dz\,. && (20) \end{aligned}$$

The associated algorithm is then the same as above, except that we generate an ordered sample from $g_{\boldsymbol{\theta}}(z)$ and replace expression $\exp(-k_k \Delta\tau(E_j - F_j z_j))$ by $\exp(-k_k \Delta\tau(E_j - F_j z_j) + \ln(f_{\boldsymbol{\theta}}(z_j)/g_{\boldsymbol{\theta}}(z_j)))$. If $f_{\boldsymbol{\theta}}(z)$ is the lognormal distribution, we recommend using a log-uniform distribution covering the highest probability region of $f_{\boldsymbol{\theta}}(z)$. More precisely, we set $z_j = \exp(x_j)$, where $\{x_j\}$ is an ordered sample from the uniform distribution

$$\mathcal{U}[\mu - 3\sigma, \mu + 3\sigma]\,, \tag{21}$$

where

$$\mu = \ln(m) - \frac{1}{2}\sigma^2\,, \quad \sigma^2 = \ln\left(1 + \frac{s^2}{m^2}\right)\,, \tag{22}$$

with m and s^2, respectively, the mean and variance of the lognormal distribution. The weights become, up to an irrelevant $\boldsymbol{\theta}$-independent term,

$$\ln(f_{\boldsymbol{\theta}}(z_j)/g_{\boldsymbol{\theta}}(z_j)) = -\frac{1}{2}\frac{(\mu - \ln(z_j))^2}{\sigma^2}\,. \tag{23}$$

3 The C++ Class

The **GUTS** class allows to store the time series of exterior concentrations of the stressor, $\mathbf{C} = (C(s_0), \dots, C(s_m))$, the data, i.e., the time series of survivors, $\mathbf{y} = (y(t_0), \dots, y(t_n))$, parameter values, $\boldsymbol{\theta} = (h_b, k_r, k_k, \dots)$, of the model and the distribution, $f_{\boldsymbol{\theta}}(z)$, from which the thresholds of the model are sampled. Furthermore, it provides a method to generate a sample, to calculate damages, the survival probabilities, and the logarithm of the likelihood (see section 2).

Below follows a brief description. Refer to the source code to check for more details.

3.1 Fields of the C++ class

The **GUTS** C++ class has no public fields. Modifications of an existing **GUTS** object must therefore be made using setter methods. However, protected fields represent the attributes of an object and can be accessed using getter methods (see section 3.2). Of particular interest are the following attributes (due to programming conventions C++ field names may differ from the mathematical notations above):

- `Title`: the title of a **GUTS** experiment. Currently unused
- `C`: vector of (exterior) concentrations $(C_0, C_1, \dots, C_m)$.
- `Ct`: vector of time points of concentrations $(0 = s_0 < s_1 < \dots < s_m)$.
- `y`: vector of survivors $(y_0, y_1, \dots, y_n)$.
- `yt`: vector of time points of survivors $(0 = t_0 < t_1 < \dots < t_n \le s_m)$.
- `par`: parameter vector $(\boldsymbol{\theta} = (h_b, k_r, k_k, \dots))$ with the following parameters:
  1. background mortality rate (h_b)
  2. recovery rate (k_r)
  3. killing rate (k_k)

The additional arguments ($par_4 \dots$) determine parameters of the distribution from which thresholds are sampled. Currently, only the *lognormal* distribution is implemented, and the additional parameters are its *mean* and *standard deviation*. Note that this differs from the implementation in the $\mathcal{R}$ function `rlnorm()` where the parameters denote mean and standard deviation of the *corresponding normal* distribution. If attribute `dist` (see below) is "empirical", parameters for the distribution are ignored.

- `M`: number of grid points on the time axis for the numerical integration (numerical exactness). Defaults to 10000.
- `dist`: name of the distribution to sample from (currently implemented "lognormal", or "empirical"). Defaults to "lognormal".
- `N`: number of threshold samples (numerical exactness). Defaults to 10000.
- `z`: the actual sample of size `N`, either generated from `dist` with parameters from `par`, or provided as an ascendingly ordered positive numeric vector.
- `D`: vector of damages generated during the calculation of the survival probabilities.
- `S`: vector of survival probabilities.
- `LL`: the loglikelihood.

Note that in the source code each attribute is prefixed with an `m` indicating that this is a member variable set by the corresponding method.

3.2 Methods of the C++ Class

The C++ Class has setter methods, getter methods, and methods to compute values and vectors in a **GUTS** object.

The setter methods set up a complete **GUTS** experiment.

- `setTitle(string Title)`: set the title of a **GUTS** experiment to `Title`. Currently unused.
- `setConcentrations(vector<double> C, vector<double> Ct)`: set the vectors of concentrations (`C`) and concentration time points (`Ct`). `Ct` must start at 0. `C` and `Ct` must have the same length.
- `void setSurvivors(vector<int> y, vector<double> yt)`: set the vectors of survivors (`y`) and survivor time points (`yt`). `yt` must start at 0. `y` and `yt` must have the same length.
- `setParameters(vector<double> par)`: set the vector of parameters. See 3.1 for details.
- `setTimeGridPoints(int M)`: set the number of grid point on the time-axis. See 3.1 for details.
- `setDistribution(string dist)`: set the distribution to `dist`. See 3.1 for details.
- `setSampleLength(int N)`: set the sample length to `N`. See 3.1 for details.
- `setSample(vector<double> z)`: do not sample, but use the provided sample `z` instead. Using this method will bypass the sampling procedure. However, the vector is checked for consistency, and a sorted copy is created and assigned to `z`.

Each getter method is pasted using the prefix "get" plus the variable's name, e.g. `getC` for getting the vector of concentrations. Note, that in contrast to combined setters (e.g., `setConcentrations()`) getters are not combined, i.e., there is one getter each for access to the concentrations and the concentration time points. Additional getters are:

- `getD()`: returns `vector<double> D`, the vector of damages.
- `getS()`: returns `vector<double> S`, the vector of survival probabilities
- `getLL()`: returns `double LL`, the loglikelihood
- `getErrors`: returns `vector<bool> Errors`, a vector of booleans indicating errors. An error at the position exists if the element at the position is `true`.
- `getErrorMessages()`: returns `vector<string> ErrorMessages`, a vector of strings expressing what error occurred at the position.

The method `showObject()` prints formatted content of a **GUTS** object to the console output.

In addition to merely set- and get-methods, a **GUTS** object provides methods for calculating/computing values or vectors.

- `calcSample()` calculates a sample from parameters and the value of the field `N`.
- `calcSurvivalProbabilities()` calculates the survival probabilities. This method is overloaded and available in three variants:
  1. without an argument: use the values present in a **GUTS** object for the calculation
  2. with argument `vector<double> St`: `St` donate the survivor time points used for the calculation. A vector of the same length is created and filled with 0. Both vectors are supplied to the method `setSurvivors(...)`. Note that this will change the vector of survivors!
  3. with argument `int Stlength`: A vector of integers starting at zero and of length `Stlength` is created, as well as a corresponding vector of survivors filled with 0. Bot vectors are supplied to the method `setSurvivors()`. Note that this will change the vector of survivors!
- `calcLoglikelihood()` calculates the loglikelihood of a properly set up **GUTS** object. The method sets `LL`.

Two more protected methods are available in **GUTS**:

- `doCalcSampleLognormal` calculates a sample from the lognormal distribution.
- `doCalcSurvivalProbabilities` is the work horse for the calculation of the survival probabilities for either of the `calcSurvivalProbabilities`-version.

However, these are protected methods of the class and may not be called directly.

4 Implementation in $\mathcal{R}$

GUTS is exposed to $\mathcal{R}$ through the deployment of Rcpp (Eddelbuettel and Francois, 2011). **GUTS** is contained in a module (`modguts`) and can be used in $\mathcal{R}$ via the S4 reference class `Rcpp_GUTS`. Except the `Title`-method, all public setter and calculation methods are exposed to $\mathcal{R}$ and can be used on $\mathcal{R}$-objects with the appropriate signature. Except the getter of `Errors` and `Title`, all getters are exposed to $\mathcal{R}$ as *fields* of an $\mathcal{R}$-object of class `Rcpp_GUTS`. The show method (`showObject`) was rewritten in $\mathcal{R}$ to account for special formatting capabilities of $\mathcal{R}$ compared to console out.

In addition to the methods of the C++ class, the $\mathcal{R}$-implementation has two S3 generic *functions*, `print()` and `logLik()`, whith the latter being an alternative to invoke the calculation of the loglikelihood of a **GUTS**-object. For testing purposes the package also includes a small real-world data set, diazinon. For more information about Rcpp, `print()` and `logLik()` see the appropriate help files shipped with these $\mathcal{R}$-packages.

```
R: library('Rcpp')
R: help('Rcpp')
R: help('print')
R: help('logLik')
```

4.1 Dataset Diazinon

The $\mathcal{R}$-package also contains a small data set for use with **GUTS**. `diazinon` is a list with 13 slots:

- `Description`: a short line of description
- `C1`–`C3`: 3 vectors of concentrations of diazinon
- `Ct1`–`Ct3`: 3 time vectors corresponding to the concentrations vectors
- `y1`–`y3`: 3 vectors of survivors
- `yt1`–`yt3`: 3 time vectors corresponding to the survivors vectors

```
R: data(diazinon)
R: diazinon

$Description
[1] "3 experiments with survivors exposed to concentrations of diazinon"

$C1
 [1] 102.65  97.59   0.00   0.00 103.88  98.19   0.00   0.00   0.00   0.00

$C2
[1] 100.78 106.32   0.00   0.00 103.56  95.82   0.00   0.00   0.00

$C3
[1] 100.60  94.61   0.00   0.00 100.58  96.51   0.00   9.85

$Ct1
 [1]  0.00  1.02  1.03  2.99  3.01  4.01  4.02 11.01 18.01 22.01

$Ct2
[1]  0.00  1.02  1.03  8.00  8.01  9.00  9.01 15.00 22.01

$Ct3
[1]  0.00  1.02  1.03 16.00 16.01 17.00 17.01 22.01

$y1
 [1] 70 66 61 55 31 31 29 26 24 22 21 19 17 14 14 13 11 11 10  9  8  8  8

$y2
 [1] 70 65 59 56 54 50 47 46 46 40 23 22 22 21 18 17 17 13 13 13 11 11 11

$y3
 [1] 70 65 59 55 53 51 48 46 46 46 44 41 40 40 40 39 38 36 33 28 24 23 19

$yt1
 [1]  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22

$yt2
 [1]  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22

$yt3
 [1]  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
```

4.2 Caveats of the $\mathcal{R}$ Implementation

1. $\mathcal{R}$ objects of reference classes are *references* to objects!

   Each $\mathcal{R}$-object created with `object <- GUTS$new()` is a *reference* to a C++-object. Therefore, copying the $\mathcal{R}$-object would not result in copying the C++-object, but in duplicating the reference:

```
R: x <- new("Rcpp_GUTS")           # a new Rcpp_GUTS object
R: x$setTimeGridPoints(M = 10000)  # set some value
R: y <- x                          # copy x to y
R: y$setTimeGridPoints(M = 500)    # set some value to y
R: x$M                             # access the value on x

[1] 500
```

   Because the *reference* was copied, we now have access to *one and the same* C++-object via two $\mathcal{R}$-objects. Hence, any change on one such reference will change the underlying C++-object and will appear *in all other* references!

   A method to copy an entire **GUTS**-C++-object is currently not implemented.

2. Serialisation is currently not available.

5 Usage of the $\mathcal{R}$ Package

Using **GUTS** may require the user to install the latest version of $\mathcal{R}$. Refer to section "preliminaries" (page 1) to check, which version was used for the creation of this manual. In addition to the package **GUTS**, users must install the package Rcpp. A convenient way to install **GUTS** in $\mathcal{R}$ is provided through the $\mathcal{R}$-command:

`install.packages( 'GUTS', dependencies=TRUE )`

For the integration of **GUTS** into an MCMC application, users may be required to additionally install MHadaptive (Chivers, 2012). Professional plotting control can be achieved using the functions in package ggplot2 (Wickham, 2009). To plot multiple figures on one page users should install the package grid.

5.1 Basic Usage

The basic usage of **GUTS** is as follows:

```
R: library("GUTS")
R: go1 <- new("Rcpp_GUTS")
R: go1

GUTS object with the following attributes:
==========================================

Vector of concentrations (C, 0 elements)
Vector of concentration time points (Ct, 0 elements)
Vector of survivors (y, 0 elements)
Vector of survivor time points (yt, 0 elements)
Parameters (par, 0 elements)
Time grid points (M) : 10000
Distribution (dist)  : lognormal
```

```
Sample length (N)    : 10000
Sample vector (z, 0 elements)
Vector of sample weights (zw, 0 elements)
Vector of damages (D, 0 elements)
Vector of survival probabilities (S, 0 elements)
Loglikelihood (LL): NaN
Messages/warnings:
[1] "C not set up"                         "Ct not set up"
[3] "y not set up"                         "yt not set up"
[5] "par not set up"                       "z not available"
[7] "Survival probabilities not calculated" "Loglikelihood not calculated"

R: go1$LL

[1] NaN

R: logLik(go1)

'log Lik.' NA (df=NA)
```

In this example, a “factory-fresh” **GUTS** object (go1) is created. For the creation we can use the function new("Rcpp_GUTS") *or* the method new(), where the method must be invoked with the *creator object* GUTS. The object contains default or non-sense values, and the loglikelihood (accessed either by the field LL or using the function logLik()) delivers NA.

Having a **GUTS** object, users may fill it with more sensible data. One example can be found in the manual page of **GUTS** in $\mathcal{R}$, and is reproduced here. We use data from the data set diazinon. The data set is included in the package (see section 4.1).

```
R: data("diazinon")
R: go1$setConcentrations(C = diazinon$C1, Ct = diazinon$Ct1)
R: go1$setSurvivors(y = diazinon$y1, yt = diazinon$yt1)
R: go1$setParameters(par = c(0.05084761, 0.12641525, 1.61840054, 19.09911, 6.495246))
R: go1$setTimeGridPoints(M = 10000)
R: go1$setDistribution(dist = "lognormal")
R: go1$setSampleLength(N = 10000)
R: go1$calcLoglikelihood()
R: go1$LL

[1] -244.3179

R: logLik(go1)

'log Lik.' -245.3884 (df=23)

R: go1

GUTS object with the following attributes:
==========================================

Vector of concentrations (C, 10 elements):
    102.65, 97.59, 0, 0, 103.88, 98.19, 0, 0, 0, 0
Vector of concentration time points (Ct, 10 elements):
    0, 1.02, 1.03, 2.99, 3.01, 4.01, 4.02, 11.01, 18.01, 22.01
Vector of survivors (y, 23 elements):
    70, 66, 61, 55, 31, 31, 29, 26, 24, 22, 21, 19, 17, 14, 14, 13, 11, 11, 10, 9, 8, 8, 8
Vector of survivor time points (yt, 23 elements):
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22
Parameters (par, 5 elements):
    0.0508476, 0.1264152, 1.6184005, 19.09911, 6.495246
Time grid points (M) : 10000
```

```
Distribution (dist)  : lognormal
Sample length (N)    : 10000
Sample vector (z, 10000 elements):
  Min=6.702867, Max=48.76261, Mean=21.32126, SD=11.82601
Vector of sample weights (zw, 10000 elements):
  Min=-4.499328, Max=-2.134366e-09, Mean=-1.515948, SD=1.341898
Vector of damages (D, 10000 elements):
  Min=0, Max=20.45299, Mean=8.556884, SD=5.016242
Vector of survival probabilities (S, 23 elements):
    0.4119788, 0.3832566, 0.3470654, 0.3296463, 0.1899154, 0.1418366, 0.1346959, 0.1280181,
    0.1216714, 0.1156394, 0.1099064, 0.1044576, 0.0992789, 0.094357, 0.0896791, 0.0852332,
    0.0810076, 0.0769915, 0.0731746, 0.0695468, 0.0660989, 0.062822, 0.0597075
Loglikelihood (LL): -245.3884
Messages/warnings: none
```

Compared to the "factory-fresh" **GUTS** object above, all errors disappeared, because the object was set up properly. In addition, the invocation of the calculation of the loglikelihood (method `calcLoglikelihood()`) also caused the computation of a new sample, a vector of damages, a vector of survival probabilities, and the loglikelihood. The call to the S3 function `logLik()` re-computed a sample, and hence the resulting likelihood is slightly different from the previous one. Note, that each time the user either calls the method `calcLoglikelihood()` or the function `logLik()`, the object's loglikelihood is re-calculated. The recommended way of calculating and accessing the loglikelihood is:

```
R: go1$calcLoglikelihood()   # calculate it
R: go1$LL                    # get it

[1] -245.6743
```

5.2 Using GUTS in MCMC Routines

As a real-world example we perform a Bayesian parameter inference (with uniform priors) using the survival data of *Gammarus pulex* exposed to *Diazinon* Ashauer, Hintermeister, Caravatti, Kretschmann and Escher (2010). The data is contained in the data set `diazinon`, which is contained in the **GUTS** package. In these experiments three different exposure patterns (treatments) have been applied. Since we want to use all the data for the parameter inference, we represent the three exposure patterns by three instances of the GUTS class and use the sum of the three loglikelihoods. For the computation of the MCMC we use the package `adaptMCMC` by Andreas Scheidegger (andreas.scheidegger@eawag.ch).

First, we set up three different **GUTS** objects. Note that all data is contained in the data set `diazinon`.

```
R: # new objects
R: tmtA <- new("Rcpp_GUTS")
R: tmtB <- new("Rcpp_GUTS")
R: tmtC <- new("Rcpp_GUTS")
R: # concentrations and concentration time points
R: tmtA$setConcentrations(C = diazinon$C1, Ct = diazinon$Ct1)
R: tmtB$setConcentrations(C = diazinon$C2, Ct = diazinon$Ct2)
R: tmtC$setConcentrations(C = diazinon$C3, Ct = diazinon$Ct3)
R: # survivors and survivor time points
R: tmtA$setSurvivors(y = diazinon$y1, yt = diazinon$yt1)
R: tmtB$setSurvivors(y = diazinon$y2, yt = diazinon$yt2)
R: tmtC$setSurvivors(y = diazinon$y3, yt = diazinon$yt3)
R: # distribution
R: tmtA$setDistribution(dist = "lognormal")
```

```
R: tmtB$setDistribution(dist = "lognormal")
R: tmtC$setDistribution(dist = "lognormal")
R: # numercial exactness
R: tmtA$setTimeGridPoints(M = 10000)
R: tmtB$setTimeGridPoints(M = 10000)
R: tmtC$setTimeGridPoints(M = 10000)
R: tmtA$setSampleLength(N = 10000)
R: tmtB$setSampleLength(N = 10000)
R: tmtC$setSampleLength(N = 10000)
```

We now define a set of starting parameters, and create a function that updates the parameters on either of the three objects and delivers the loglikelihood:

```
R: par.start <- c(0.1, 0.3, 1, 1, 1)
R: loglikeli <- function(par) {
     if ( any(par < 0) ) {
       # Parameters must not be negative
       return(-Inf)
     } else {
       # The loglikelihood of the 3 treatments
       # is just the sum of the individual ones
       tmtA$setParameters(par)
       tmtB$setParameters(par)
       tmtC$setParameters(par)
       tmtA$calcLoglikelihood()
       tmtB$calcLoglikelihood()
       tmtC$calcLoglikelihood()
       out <- tmtA$LL + tmtB$LL + tmtC$LL
       return(out)
     }
   }
```

Note that if a particular error occurs (e.g., a sample mean smaller than 0), then this functions returns NA. Next, we set the jump standard deviations for the MCMC chain:

```
R: par.names <- c("h.b", "k.r", "k.k", "mean", "sd")
R: sigma <- diag(par.start/10)^2
```

Calculate the adaptive Markov chain using the package adaptMCMC:

```
R: library("adaptMCMC")
R: res.mcmc <- MCMC(p = loglikeli, n = 50000, init = par.start, scale = sigma,
       adapt = TRUE, acc.rate = 0.4)
```

Our result can be inspected in various ways, one is to look at the structure:

```
R: str(res.mcmc)

List of 7
 $ samples            : num [1:50000, 1:5] 0.1 0.0916 0.0916 0.0916 0.0874 ...
 $ log.p              : num [1:50000] -7471 -2876 -2876 -2876 -2560 ...
 $ cov.jump           : num [1:5, 1:5] 2.72e-05 8.35e-06 8.98e-04 1.55e-03 -1.04e-03 ...
 $ n.sample           : num 50000
 $ acceptance.rate    : num 0.417
 $ adaption           : logi TRUE
 $ sampling.parameters:List of 3
  ..$ sample.density:function (par)
  .. ..- attr(*, "srcref")=Class 'srcref'  atomic [1:8] 470 14 486 1 14 1 3 19
  .. .. .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcfile' <environment: 0x7ff3dd2ddd90>
  ..$ acc.rate      : num 0.4
  ..$ gamma         : num 0.5
```

5.3 Plotting the Outcomes of the MCMC Inference

Excellent plotting facilities are available through the routines in the package ggplot2 (Wickham, 2009). ggplot2 offers great features for generating object-oriented plots. To plot the trace of the parameter `mean` one may use the code below. Note that due to file size issues we reduce data to the each 50th iteration. The result is displayed in figure 1):

```
R: library("ggplot2")
R: # prepare a data frame, and select select each 10th observation
R: k <- nrow(res.mcmc$samples)
R: i <- seq(1, k, 50)
R: df <- data.frame(i, res.mcmc$samples[i, ])
R: colnames(df) <- c("iter", par.names)
R: str(df)

'data.frame':  1000 obs. of  6 variables:
 $ iter: num  1 51 101 151 201 251 301 351 401 451 ...
 $ h.b : num  0.1 0.0621 0.0613 0.0516 0.0483 ...
 $ k.r : num  0.3 0.00718 0.00593 0.00616 0.00664 ...
 $ k.k : num  1 1.33 1.34 1.36 1.35 ...
 $ mean: num  1 1.57 1.58 1.58 1.58 ...
 $ sd  : num  1 1.31 1.29 1.35 1.38 ...

R: # create the plot object
R: ggp.mean <- ggplot(df, aes(x=iter, y=mean)) +
     geom_line() +
     xlab('iteration') +
     ylab('Value of mean')
```

```
R: ggp.mean
```

In the following we create two plots using ggplot2, the trace plot of the five parameters (figure 2) and the density plots (figure 3). Both statistics should appear on one page each, hence we create a little helper function to set up the page (requires the package grid!). Note, that if one wants only one plot appear on one page (pdf), this helper function and the application of `multiplot()` is not necessary.

```
R: multiplot <- function(..., plotlist = NULL, cols) {
       require(grid)
       plots <- c(list(...), plotlist)
       numPlots = length(plots)
       plotCols = cols
       plotRows = ceiling(numPlots/plotCols)
       grid.newpage()
       pushViewport(viewport(layout = grid.layout(plotRows, plotCols)))
```

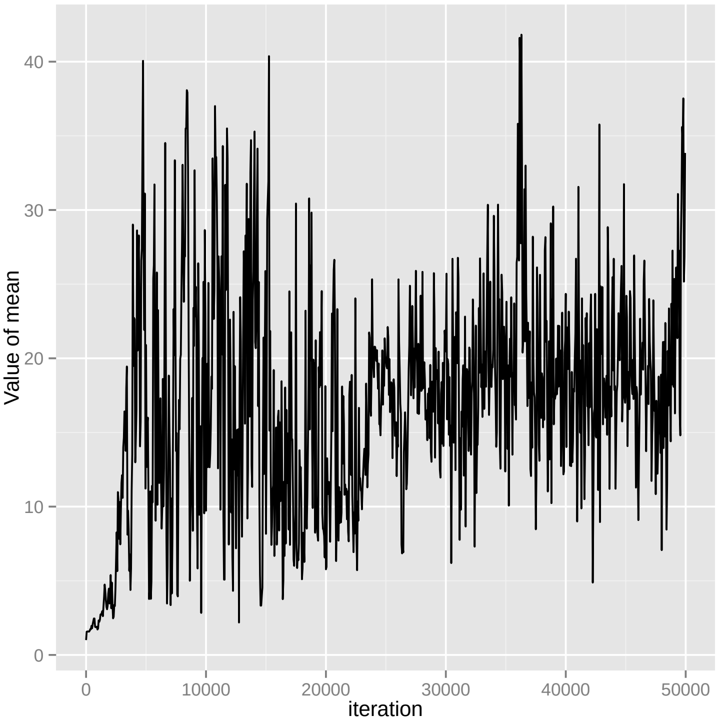

Figure 1: Plot of the trace of the parameter `mean` using package ggplot2.

```
        vplayout <- function(x, y) {
            viewport(layout.pos.row = x, layout.pos.col = y)
        }
        for (i in 1:numPlots) {
            curRow = ceiling(i/plotCols)
            curCol = (i - 1)%%plotCols + 1
            print(plots[[i]], vp = vplayout(curRow, curCol))
        }
    }
```

We now prepare the data frame holding the parameters as well as a column signing the loop in the MCMC. ggplot2 creates objects of plots. Printing these objects results in the final plot. This differs from the standard plot routines in $\mathcal{R}$, where in most of the cases the plot function plots itself. Note that due to plot size issues we reduce data to each 50th iteration.

```
R: # prepare a data frame
R: k <- nrow(res.mcmc$samples)
R: i <- seq(1, k, 50)
R: plot.data <- data.frame(i, res.mcmc$samples[i, ])
R: colnames(plot.data) <- c("iter", par.names)
R: str(plot.data)

'data.frame':  1000 obs. of  6 variables:
 $ iter: num  1 51 101 151 201 251 301 351 401 451 ...
 $ h.b : num  0.1 0.0621 0.0613 0.0516 0.0483 ...
 $ k.r : num  0.3 0.00718 0.00593 0.00616 0.00664 ...
 $ k.k : num  1 1.33 1.34 1.36 1.35 ...
 $ mean: num  1 1.57 1.58 1.58 1.58 ...
 $ sd  : num  1 1.31 1.29 1.35 1.38 ...

R: # five plots for five parameters
R: plot.h.b <- ggplot(plot.data, aes(x=iter, y=h.b)) +
     geom_line() +
     xlab('iter') +
     ylab('Value of h.b')
R: plot.k.r <- ggplot(plot.data, aes(x=iter, y=k.r)) +
     geom_line() +
     xlab('iter') +
     ylab('Value of k.r')
R: plot.k.k <- ggplot(plot.data, aes(x=iter, y=k.k)) +
     geom_line() +
     xlab('iter') +
     ylab('Value of k.k')
R: plot.mean <- ggplot(plot.data, aes(x=iter, y=mean)) +
     geom_line() +
     xlab('iter') +
     ylab('Value of mean')
R: plot.sd <- ggplot(plot.data, aes(x=iter, y=sd)) +
     geom_line() +
     xlab('iter') +
     ylab('Value of sd')
```

Now we use the `multiplot()` function defined above, and plot each ggplot object in the appropriate position.

```
R: multiplot(plot.h.b, plot.k.r, plot.k.k, plot.mean, plot.sd, cols=2)
```

The traces shown in figure 2 suffer from both, burn-in and adaption. Therefore, we cut off the first 20,000 sample points and then produce the density plots.

```
R: # create plot data frame
R: dat <- res.mcmc$samples
R: i <- 20001
R: k <- nrow(dat)
R: plot.data <- data.frame( i:k, dat[i:k, ] )
R: colnames(plot.data) <- c("iter", par.names)
R: str(plot.data)

'data.frame':  30000 obs. of  6 variables:
 $ iter: int  20001 20002 20003 20004 20005 20006 20007 20008 20009 20010 ...
 $ h.b : num  0.0485 0.0485 0.0528 0.054 0.054 ...
 $ k.r : num  0.0223 0.0223 0.0232 0.0258 0.0258 ...
 $ k.k : num  2.41 2.41 2.13 2.38 2.38 ...
 $ mean: num  5.77 5.77 5.9 6.07 6.07 ...
 $ sd  : num  4.86 4.86 4.23 3.69 3.69 ...

R: # five plots for five parameters
R: plot.dens.h.b <- ggplot(plot.data) +
     geom_density(aes(x = h.b)) +
     xlab('h.b') +
     ylab('Density')
R: plot.dens.k.r <- ggplot(plot.data) +
     geom_density(aes(x = k.r)) +
     xlab('k.r') +
     ylab('Density')
```

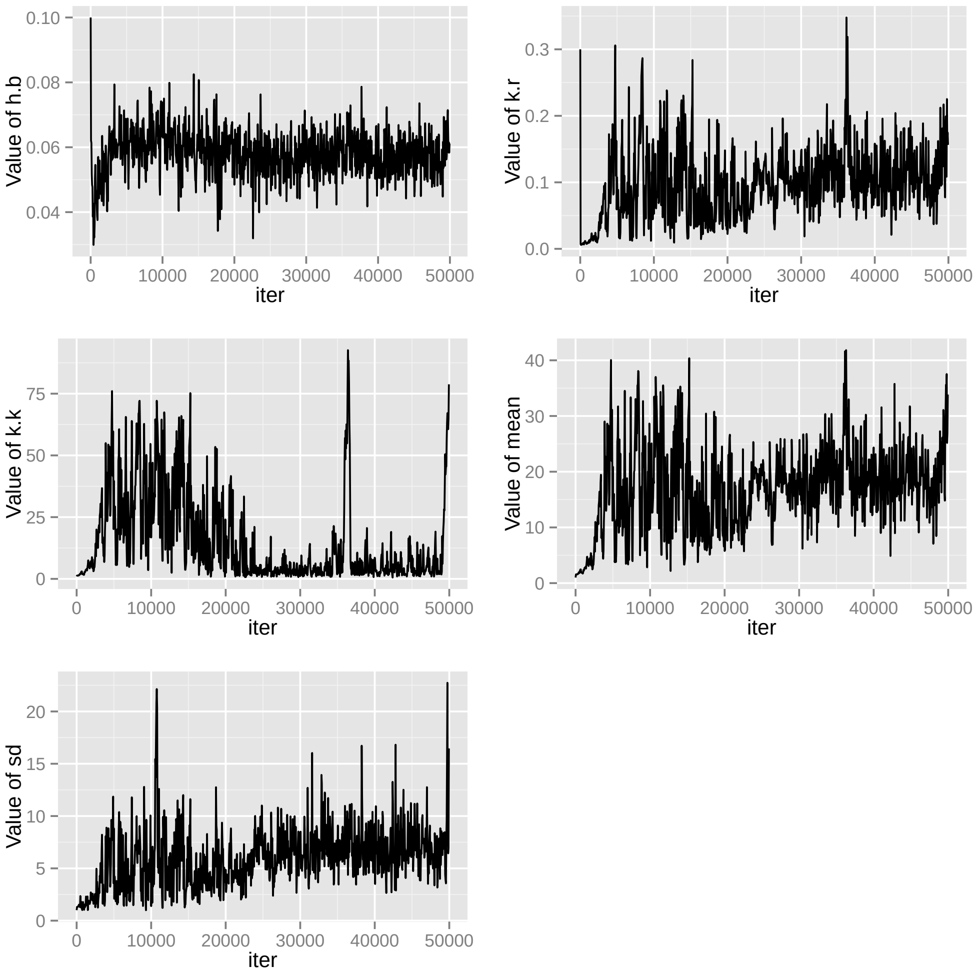

Figure 2: Plot of the trace of the parameters using package ggplot2.

```
R: plot.dens.k.k <- ggplot(plot.data) +
     geom_density(aes(x = k.k)) +
     xlab('k.k') +
     ylab('Density')
R: plot.dens.mean <- ggplot(plot.data) +
     geom_density(aes(x = mean)) +
     xlab('mean') +
     ylab('Density')
R: plot.dens.sd <- ggplot(plot.data) +
     geom_density(aes(x = sd)) +
     xlab('sd') +
     ylab('Density')
```

```
R: multiplot(plot.dens.h.b, plot.dens.k.r, plot.dens.k.k, plot.dens.mean, plot.dens.sd, cols=2)
```

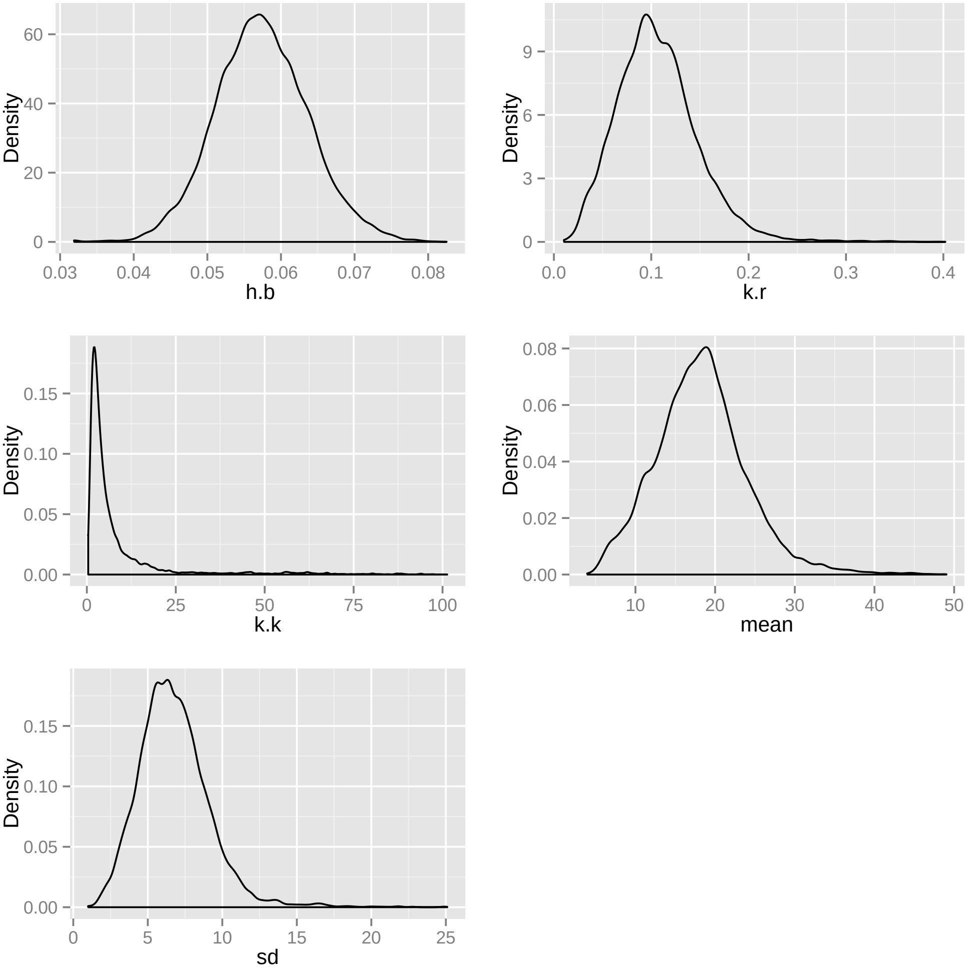

Figure 3: Plot of the density of the parameters using package ggplot2.

Next, we address the best fit. The result of the MCMC was saved in object `res.mcmc`. This object is a list with various attributes. Two attributes are of particular interest here: The vector of loglikelihoods saved during the MCMC (`log.p`), and the "vector" of parameters (saved as a matrix `samples`). Each of these two objects has a length equal to the iterations' number (here: $n = 50,000$). The "maximum probable parameters" can be found at the position of the maximum loglikelihood:

```
R: best.fit.pos <- which.max(res.mcmc$log.p)
R: best.fit.pars <- res.mcmc$samples[best.fit.pos, ]
```

We set the parameters of each of the three experimental **GUTS** objects the parameters of the best fit, and calculate the survival probabilities accordingly:

```
R: tmtA$setParameters(best.fit.pars)
R: tmtB$setParameters(best.fit.pars)
R: tmtC$setParameters(best.fit.pars)
```

```
R: tmtA$calcSurvivalProbabilities()
R: tmtB$calcSurvivalProbabilities()
R: tmtC$calcSurvivalProbabilities()
```

Now, we create three plots, one for each experiment. In each plot, we want to display the measured number of deaths ($y_{i-1} - y_i$) compared to the predicted number of deaths ($y_0(S_{i-1} - S_i)$), where mean and variance of the predicted number of deaths are calculated as:

$$\begin{aligned} mean &= y_0 p_i \\ var &= y_0 p_i (1 - p_i) \end{aligned}$$

with

$$p_i = S_{i-1} - S_i \,, \text{ for } i = 1, \ldots, n+1.$$

Figure 4 shows the results for the three experiments.

```
R: # prepare data frames
R: y.A <- c(tmtA$y, 0)
R: S.A <- c(tmtA$S, 0)
R: plot.data.A <- data.frame(
     "var" = rep( c("measured", "predicted"), each=length(tmtA$y) ),
     "yt"  = rep(tmtA$yt, 2),
     "y"   = c( -diff(y.A), (-diff(S.A) * tmtA$y[1]) ),
     "ye"  = c(
           rep(NA, length(tmtA$y)),
           sqrt( tmtA$y[1] * -diff(S.A) * (1 + diff(S.A)) )
         )
   )
R: y.B <- c(tmtB$y, 0)
R: S.B <- c(tmtB$S, 0)
R: plot.data.B <- data.frame(
     "var" = rep( c("measured", "predicted"), each=length(tmtB$y) ),
     "yt"  = rep(tmtB$yt, 2),
     "y"   = c( -diff(y.B), (-diff(S.B) * tmtB$y[1]) ),
     "ye"  = c(
           rep(NA, length(tmtB$y)),
           sqrt( tmtB$y[1] * -diff(S.B) * (1 + diff(S.B)) )
         )
   )
R: y.C <- c(tmtC$y, 0)
R: S.C <- c(tmtC$S, 0)
R: plot.data.C <- data.frame(
     "var" = rep( c("measured", "predicted"), each=length(tmtC$y) ),
     "yt"  = rep(tmtC$yt, 2),
     "y"   = c( -diff(y.C), (-diff(S.C) * tmtC$y[1]) ),
     "ye"  = c(
           rep(NA, length(tmtC$y)),
           sqrt( tmtC$y[1] * -diff(S.C) * (1 + diff(S.C)) )
         )
   )
R: # generate 3 plots using ggplot
R: plot.bf.A <- ggplot(plot.data.A, aes(x=yt, y=y, group=var, shape=var, colour=var)) +
     geom_point() +
     geom_errorbar( aes(ymin=y-ye, ymax=y+ye), width=.5 ) +
     xlab('survivor time points in experiment A') +
     ylab('deaths')
R: plot.bf.B <- ggplot(plot.data.B, aes(x=yt, y=y, group=var, shape=var, colour=var)) +
     geom_point() +
     geom_errorbar( aes(ymin=y-ye, ymax=y+ye), width=.5 ) +
     xlab('survivor time points in experiment B') +
     ylab('deaths')
R: plot.bf.C <- ggplot(plot.data.C, aes(x=yt, y=y, group=var, shape=var, colour=var)) +
     geom_point() +
     geom_errorbar( aes(ymin=y-ye, ymax=y+ye), width=.5 ) +
     xlab('survivor time points in experiment C') +
     ylab('deaths')
```

Again, we use the function `multiplot()` from above to merge all plots.

```
R: multiplot(plot.bf.A, plot.bf.B, plot.bf.C, cols=1)
```

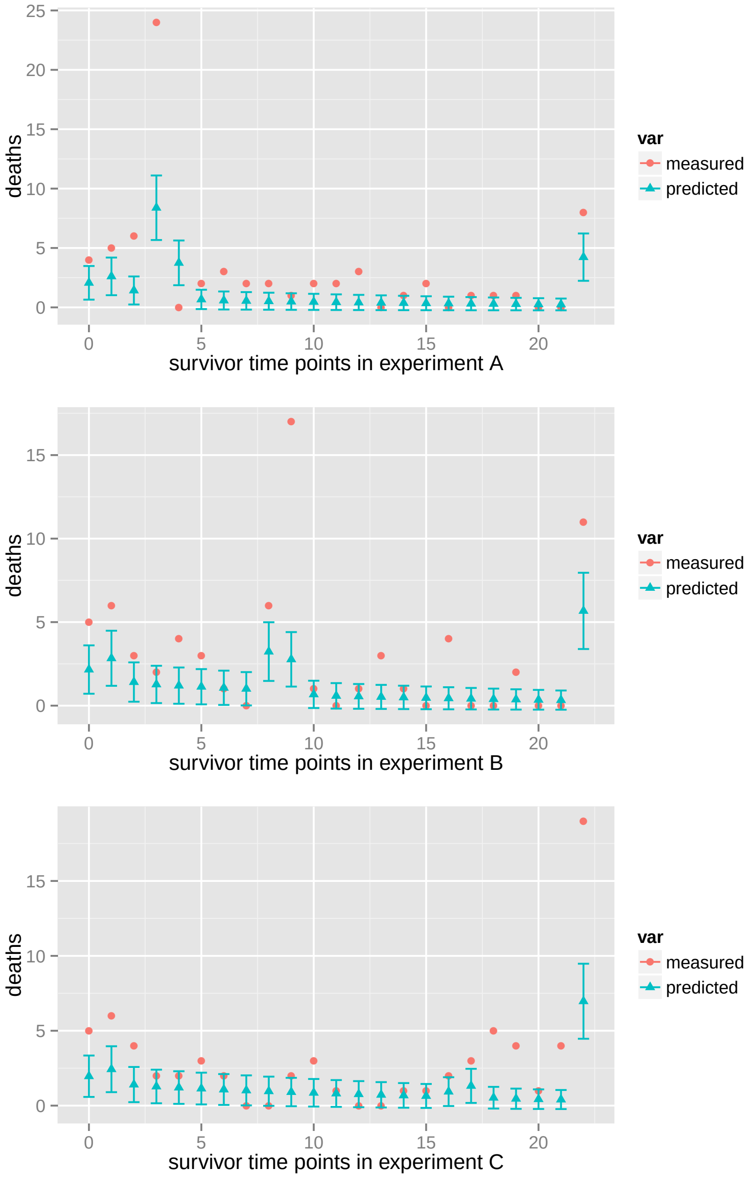

Figure 4: Plot of the best fit using package ggplot2.

References

Ashauer, R., Hintermeister, A., Caravatti, I., Kretschmann, A. & Escher, B. I. (2010). Toxicokinetic and toxicodynamic modeling explains carry-over toxicity from exposure to diazinon by slow organism recovery. *Environmental Science & Technology*, *44*(10), 3963–3971.

Chivers, C. (2012). *MHadaptive: general Markov Chain Monte Carlo for Bayesian inference using adaptive Metropolis-Hastings sampling*. R package version 1.1-8. Retrieved May 14, 2012, from http://CRAN.R-project.org/package=MHadaptive

Eddelbuettel, D. & Francois, R. (2011, April 13). Rcpp: seamless R and C++ integration. *Journal of Statistical Software*, *40*(8), 1–18. Retrieved March 21, 2011, from http://www.jstatsoft.org/v40/i08

Jager, T., Albert, C., Preuss, T. G. & Ashauer, R. (2011). General unified theory of survival – a toxicokinetic toxicodynamic framework for ecotoxicology. *Environmental Science & Technology*, *45*(7), 2529–2540. doi:10.1021/es103092a

Leisch, F. (2002). Sweave: dynamic generation of statistical reports using literate data analysis. In W. Härdle & B. Rönz (Eds.), *Compstat 2002—proceedings in computational statistics* (pp. 575–580). Heidelberg: Physica Verlag.

R Core Team. (2014, October 31). *R: a language and environment for statistical computing*. R Foundation for Statistical Computing. Vienna, Austria. Retrieved May 16, 2013, from http://www.r-project.org

Wickham, H. (2009). *ggplot2: elegant graphics for data analysis*. New York: Springer.