Design document for openGUTS

Tjalling Jager[*]

December 12, 2019

This documents is part of the openGUTS project, and can be downloaded from http://openguts.info/. The openGUTS project is made possible by funding from Cefic-LRI in project ECO39.2.

Contents

[*]DEBtox Research, De Bilt, The Netherlands. Email: tjalling@debtox.nl, http://www.debtox.nl/

1 Introduction and workflow

This documents provides the technical background for the openGUTS software. It explains design choices made, the concepts behind the algorithms for various stages of the workflow, the core mathematical equations implemented, and the role of the different functions. These elements are obviously shared between the standalone version of openGUTS and the Matlab version. However, the different Matlab functions have been translated into C++, which involved a number of modifications and some differences in functionality (which will be explained as much as possible in this document as well). In this document, the focus lies on the Matlab version. It is impossible to explain the workings of openGUTS in all its details in this document. However, the Matlab code is extensively commented to clarify what is done and why it is done. Note that this document deals with the technical design of openGUTS; other documents are available from the openGUTS website to help getting started with the software and to interpret the outputs. This document assumes knowledge about the GUTS concepts and symbol use, following the e-book [5].

The Matlab version was developed by DEBtox Research, taking elements of the well-tested BYOM package (`http://debtox.info/byom.html`) and placing them into a GUTS-only framework. The Matlab version served as prototype platform for developing and testing functionality and algorithms for the standalone software, and thereby acted as the blueprint for the C++ code produced in this project by WSC Scientific GmbH.[1] Clearly, the C++ code cannot be the same as the Matlab code, but it follows the Matlab version as closely as reasonably feasible, in terms of function/variable names and the functionality of the code blocks. This aids testing and future development. Note that both the Matlab version and the standalone software are distributed under under the terms of the GNU General Public License, version 3.

Please note that the plots and text output shown in this document are not from the last version of the software, and can deviate slightly from the latest version. This, however, does not affect the purpose of this document.

Relevant general design decisions:

- The workflow of openGUTS is based on the proposed workflow in the EFSA scientific opinion on TKTD models for use in risk assessment of pesticides [4]. However, (elements of this) workflow will be useful for many other purposes as well (including scientific research).

- The Matlab version and the standalone software only include the simplest cases of the model: GUTS-RED-SD and GUTS-RED-IT.

- The special cases for fast and slow kinetics are *not* implemented (see the GUTS e-book [5], Appendix C). Instead, a default range for k_d is used, which will basically

[1]Since the output of the Matlab version is identical to the standalone version (for all intents and purposes), and since the Matlab version has some additional options, users with a Matlab license may use this version to perform their calculations.

be the same for all data sets (see Section 4.3). The user will get a warning when k_d (or any other parameter) runs into the boundaries of its allowed range.

- The same input data format is used for the calibration and validation stages (for exposure profiles, it will be different). The Matlab version uses the same text file format as the standalone software, so input files can be used by both versions. However, the project files will *not* be compatible.

- All input data sets (including exposure profiles) will have the same internal representation.

- One function is used to plot the main series of figures (three rows of panels: exposure-time, damage-time, survival-time). This plot is used for inspecting input data (so that the user can check that the right data are being used, and the exposure scenario is correctly entered) and for plotting model output in calibration, validation and prediction.

The general workflow of the software is presented in Figure 1. Text files will be used as input (the C++ version also allows working with an input grid to create/edit these files). The Matlab version stores the results of a calibration a 'project file', which the remainder of the workflow will load when needed. Note the 'load calibration', which implies skipping the first sections (loading input data and the calibration itself), loading a previously saved project file, and plotting the results from that.

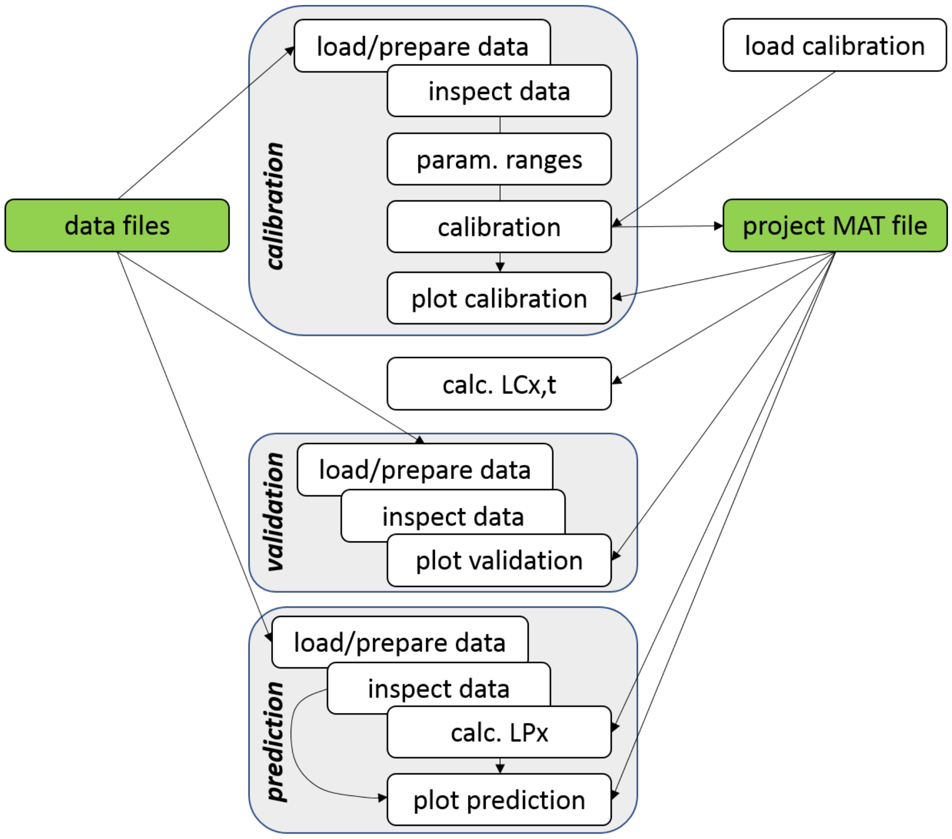


Figure 1: General workflow of the software (Matlab version). Blue fields indicate main parts of the workflow of the software: calibration, validation, prediction. The LC$x,t$ calculations are shown separately as it does not use data input. The direct arrow in the prediction stage, from 'inspect data' to 'plot prediction' could be used to plot model predictions for user-provided multiplication factors (this will not be part of the standalone software).

2 Directories, file types, and initial setup

2.1 Directories and file types

For the Matlab version, all files are located under a folder named `openGUTS`. The `engine` directory contains all of the functions needed for the openGUTS calculations. These files do not need to be changed by the user to perform an analysis. As distributed, the protoype has a single analysis folder, named `example_study`, which contains the basic script `openguts.m`. This basic script follows the general workflow of the software. This folder must be under the `openGUTS` folder, and will contain a number of sub-folders:

- `input_data` contains the input data files for survivors/exposure scenarios. Each file contains one data set (with a number of treatments) in a formatted manner. These are plain text files (tab delimited) that can be opened/modified/saved by Excel or NotePad.
- `input_profile` contains the input data files for exposure profiles used in predictions (LPx). Each file contains one data set (with a single treatment). These will thus be plain text files with two columns (tab delimited): time points and concentrations (and no headers or other information).
- `output_sample` will contain the binary `mat` files with the information from the parameter-space explorer such as the best-fitting parameter set and the sample from parameter space. Plotting and post-analyses (such as calculation of LC$x, t$ and LPx) will use these saved files. The saved files will have the name of the analysis (as defined in the script from which they were started), with text added to clarify the special case (`_SD` or `_IT`). This file has some of the functionality of the 'project file' of the standalone version, but does not include any model output other than results from the calibration (and is not readable).
- `output_report` will contain all of the output plots (by default as PDF) and a log file that collects all screen output. All saved files will have the name of the analysis, and a specification of what they are. The files are organised in sub-folders with the analysis name.

Differences with the C++ code.

- The C++ version does not use the same standard folders and organisation of input/output files. The software is installed in a standard location, and the user is prompted for a file name/location whenever there needs to be something loaded or saved (there is no automatic saving or loading).
- The input data files of the C++ version can also be used in the Matlab version (and *vice versa*).

- The project files will *not* be compatible. The C++ version will use a text file, include SD and IT in the same file, and will also include results from the post analyses (LC$x, t$, validation and predictions). It is only generated when the user asks for it to be saved. For the Matlab version, the project file is also used to communicate the calibration results to the rest of the workflow. The Matlab file is discussed in Section 4.2.
- In the Matlab version, input data have to be entered by text file; the C++ version also has the option of an input grid (to enter values directly, or copy-paste from Excel).

2.2 Initial things

Matlab search path. The function `pathdefine` is called from the script. It extracts the current directory name, finds out where `openGUTS` is in the directory name, and adds the `engine` directory to the search path for Matlab. It also creates the standard sub-folders in the present folder (if they do not exist already).

Initial setup. The script `initial_setup` starts by clearing any remaining variables (local and global) from the memory (to make sure nothing remains from previous runs), and next defines a number of helpful globals. This includes formatting for the plots (font sizes for axis labels etc., depending on screen resolution). It also prepares globals for the positions of the different parameters in the parameter matrix `pmat` (so we can address them by name in the code rather than by position). For example, m_w is in the second position, so `GLO.mw=2` and we can obtain the best value as `pmat(GLO.mw,1)`.

This function also defines names as strings for each parameter, which is used for plotting and printing. The `initial_setup` also defines a global table with χ^2 criteria, and settings for various functions (e.g., the x and t values for the calculation of LC$x, t$).

2.3 Built-in Matlab functions used

The Matlab version makes use of several (non-trivial) built-in Matlab functions:

- `cumtrapz` is a cumulative trapezoidal numerical integration.
- `interp1` does a linear interpolation on the profile likelihood.

The Matlab version initially made use of two more built-in Matlab functions, but these were replaced by Matlab versions of the same algorithm as used for the C++ version (to make the two versions as similar as possible):

- `fminsearch` is a Nelder-Mead simplex search algorithm. This is replaced by a similar algorithm in the function `nelmin`.[2]

[2]https://people.sc.fsu.edu/~jburkardt/m_src/asa047/asa047.html.

- `fzero` is a root-finding algorithm (finds where a function is zero by changing a single variable within a given range, or close to a given starting value). This is replaced by a similar algorithm in the function `zero`.[3]

Note that the alternative functions `nelmin` and `zero` have a different format of input and output than the built-in Matlab functions. This is corrected in the functions `setup_simplex` and `setup_rootfind`, which also have a switch to use the Matlab built-in functions instead (to allow for quick comparison between the two algorithms).

The `nelmin` is set up quite differently than the Matlab version. However, it is good to note that the simplex routine is only used as a small element for refinement in the complete optimisation algorithm. Since the overall algorithm also contains 'genetic' sampling aspects, profiling of the likelihood function, and checks for differences between profile and sample, the details of the simplex routine are of little relevance for the overall performance.

[3]`https://people.sc.fsu.edu/~jburkardt/m_src/brent/zero.m`.

3 Defining data sets and exposure scenarios

3.1 General

Relevant decisions:

- The openGUTS software only deals with the reduced cases of GUTS, which implies that we only have survival data and exposure concentrations (and no body residues).
- Multiple input data sets are allowed for simultaneous calibration. The Matlab version also allows multiple input data sets for validation and predictions.
- Missing observations are allowed in the data set. This is implemented by the internal reworking of the data: always splitting up the data matrix in separate treatments.
- All treatments need to be unique, and there must be one (and only one) exposure scenario for each treatment. The user can always enter replicates as two treatments, with a different identifier but the same exposure scenario.
- Each data set can have one control treatment (and no more than one), which needs to be specifically indicated by the user. In the Matlab version this is done by using the identifier named "Control" for that treatment. It is also possible to have no control in a data set.
- We only use linear interpolation for the time-varying exposure scenarios. However, it is also allowed to let the exposure concentration change instantly (by using two concentration values at the same time point).
- The input data sets for calibration and validation follow the same format. The input data for predictions (exposure profiles) will differ (and need to match FOCUS output). The input files are formatted (readable) text files.
- The same plotting routine with standard layout will be used in all stages of the analysis: panels for treatments, with first row exposure, second row damage, third row survival probability.
- Inspection plots can be made for each input data set, with confidence intervals (CIs) on the survival data. Inspection plots use the same standard plotting routine with three rows (even though some rows will be empty).
- For predictions, only one treatment is allowed per input data file. This means that each data set for prediction is one FOCUS profile (two columns: time and concentration). As identifier for the treatment, the filename of the input text file is used.
- Background hazard rate can be calculated from the controls (one value, using control survivors over all data sets entered). This can be done separately for calibration and validation (as a validation data set may have different control mortality).

- Concentration unit is a free text field in the data set (used only for printing/plotting). The software assumes that the time unit is fixed to days. The user has to make sure all input data are in the same units.

Differences with the C++ code.

- The standalone version has a restriction of only one input data set for validation, and also one for predictions (but multiple files are allowed for calibration and for batch-mode predictions).
- In the standalone version, the control treatment needs to be selected by the user in the GUI.

3.2 Types of data and definition

The input data for calibration and validation will follow a standard format as a tab-delimited text file. This file will contain a rectangular block of time-survivor data, and a rectangular block of time-exposure data. Both block will have headers (identifiers for each column), and there is a line for the concentration unit. Missing data are indicated by a character/string (e.g., `-` or `NaN`). The data set will need to obey to a certain format, so several checks will be needed. Initial checks:

- Identifiers for the treatments need to be unique. There cannot be more than 1 'Control' (though there may be none).
- All lines in the survival block (incl. the header line) need to have the same number of columns (the block must be rectangular).
- All lines in the exposure scenario block need to have the same number of columns (the block must be rectangular), and the same number of columns as the survival block.
- For all treatments in the survival data there must be a matching exposure scenario with the same identifier.

Secondary checks:

- All data sets used in the calibration must have the same concentration unit. Input data in the validation stage must have the same unit as in the calibration stage.
- The first time point in both the survival data block and the scenario block must be a zero.
- In both the survival data block and the scenario block, there can be no missing data in the first row ($t = 0$).
- For survival data, the time vector must be unique and increasing.

- For the exposure scenario block, the time vector can never decrease. A time point can occur twice (indicating an immediate change in concentration), but not more than twice.

- Survivor data must be whole positive numbers (positive natural). Scenario data need to be positive numbers (positive real). Time is entered in days (positive real).

- Within a column (treatment), the number of survivors should never increase between two subsequent steps (skipping any missing values).

For every treatment, there needs to be a corresponding definition of the exposure scenario. We decided on linear interpolation as the basis, as this is most flexible, and allows for a simple and consistent input data format. The internal representations allows the possibility to have instant changes in exposure (as in pulsed exposure). The user can enter two exposure concentrations at the same time point: the first indicates the end of the previous interval, and the second the start of the next interval. I will illustrate this in the next section with a few examples using the propiconazole case study from the ring test.

Loading data from file. For loading data from text files, a helper function is used: `load_data`. It receives one or more filenames from the main script, checks if the files exist, loads them, and returns a structured cell array with all of the input data sets (one in each cell). For the calibration/validation input, some more work is needed in Matlab to extract the information blocks from the formatted text file in the right way. This function also performs the initial checks from the list above. Missing observations need to be indicated by a character (or string of characters), and are then internally represented by `NaN` (not a number).

Translating input data to internal format. The function `prepare_data` performs a series of checks (secondary checks in list above) on the consistency of the input data matrices. Next, this function converts the output of `load_data` to the internal representation. The data set is split up into the separate treatments, and undergoes several recalculations (such as to survival probabilities and number of deaths). All information (for all data sets) is collected in a single structured object for a certain part of the workflow. This is explained in detail in Section 3.4. Note: in the Matlab version, the output of `load_data` is only used as input for `prepare_data`. Thus, they could be combined into one function. Since the functionality of `load_data` is Matlab-specific and irrelevant for the C++ version, I keep it separate.

Internally, there will be no explicit interpolation of exposure concentrations. Instead, an analytical solution for damage is used for each interval between two concentration entries. This analytical solution requires the start concentration for each interval and the linear slope by which it will change over time. The input concentration series is thus reworked to a series of concentrations and slopes.

As a last step in the function, `prepare_data` fits the background hazard rate over all data sets entered. It collects the controls (identifier = ‘Control’) into a separate data object `data_hb` with the same structure as all other data sets (therefore, it can simply be used in the same fitting procedure as used for calibration). It is fitted with a standard Nelder-Mead simplex routine (no need for genetic algorithms as this is a simple one-parameter optimisation).

The concentration unit in all data sets is included in a global: `GLO.concunit`, which is a text string used for plotting and reporting.

Differences with the C++ code.

- The Matlab version needs to read the input text files line-by-line and interpret them. The standalone version uses more powerful C++ functionality for that.
- The standalone version requires user to enter a minus sign (`-`) for missing data. Other non-numerical values are not accepted. Apart from that, the Matlab version uses the same input data files as the C++ version.
- The Matlab version only allows input data from text file. However,the standalone version also allows direct entry (or copy-paste) into an input grid in the GUI. The software is able to save data files entered into the input grid in the standard text-file format.
- The standalone version requires the user to select the control treatment from the available treatments in the GUI. Since the Matlab version does not have a GUI, the choice is made to select it based on the treatment name 'Control'.

3.3 Examples of data definition

The survival data set used here as example is for propiconazole in *Gammarus*. This set was also used as case study in the GUTS book [5], and is included in set of example files for the software as `propiconazole_constant.txt`. This is a set with (presumably) constant exposure concentrations. This set needs to be loaded into the GUTS software as a tab-delimited text file (I modified the tabs for readability below, so that the columns are lining up in the text):

```
Part of the GUTS ring test. Real data set for
propiconazole in Gammarus pulex from Nyman et al
(2012). Ecotoxicology 21, 1828-1840.
Survival time [d] Control T1 T2 T3 T4 T5 T6 T7
    0                20 20 20 20 21 20 20 20
    1                19 20 19 19 21 17 11 11
    2                19 20 19 19 20  6  4  1
    3                19 20 19 18 16  2  0  0
    4                19 19 17 16 16  1  0  0
Concentration unit: uM
Concentration time [d] Control T1    T2     T3    T4     T5     T6    T7
    0                0      8.05 11.906 13.8 17.872 24.186 28.93 35.924
```

The first lines are explanatory text that are ignored in the software. The function `load_data` looks for the trigger text `Survival time` or `survival time` to start reading the survivor matrix (so avoid this wording in the description of the data sets!). The survivor matrix contains a header row with the identifiers for the treatments. Directly below that, the survivor numbers with time in the first column (in days). Next, the function `load_data` looks for the trigger text `Concentration` or `concentration` to start reading the concentration unit. Next, the function `load_data` looks for the trigger text `Concentration time` or `concentration time` to start reading the exposure matrix. This matrix also has a header row with the names for the treatments, with the exposure scenarios below that (again, time in days in the first column).

Regarding the identifiers for the treatments, controls should be specifically marked with `Control` or `control` (other names for treatments are free to choose). Note that in this example, the exposure scenario only has a single row, which implies constant exposure. Also note that the exposure scenario does not need a final time point; it is extrapolated (with slope zero) to the length of the survival data set internally. The result is plotted in Figure 2 in the standard format (the missing row in the plot is for damage predictions, which we don't have for inspection). Note that the CIs on the survival data are the Wilson score interval (the CI on a bionomial proportion, see Section 8).

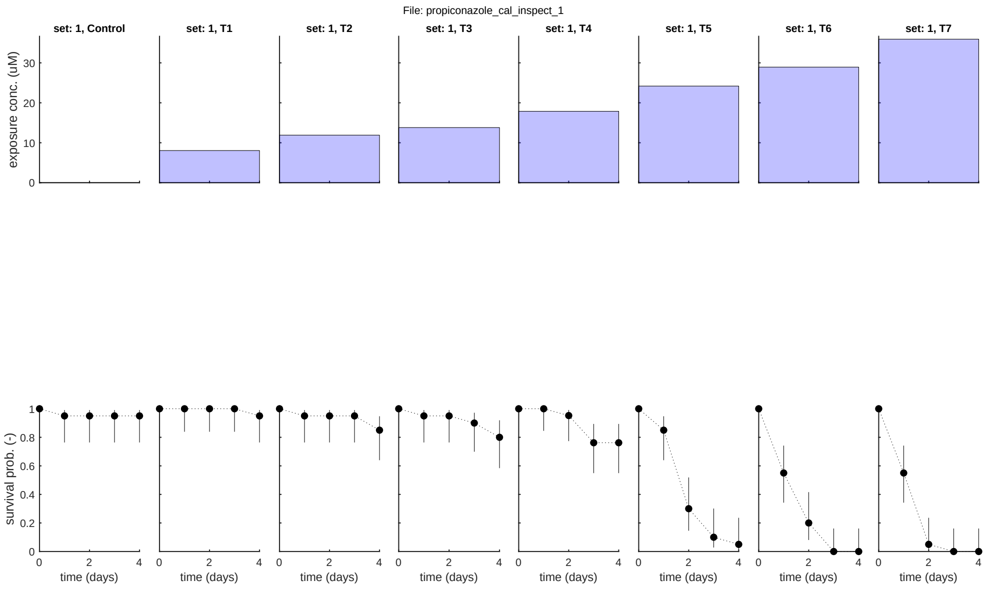


Figure 2: Inspection plot for the data set: constant exposure.

If we like to use a renewal scenario, with renewal at day 2, we could enter the data matrix as follows (only lower block of the matrix has changed):

```
Survival time [d] Control T1 T2 T3 T4 T5 T6 T7
         0           20   20 20 20 21 20 20 20
         1           19   20 19 19 21 17 11 11
```

```
        2            19   20 19 19 20  6  4  1
        3            19   20 19 18 16  2  0  0
        4            19   19 17 16 16  1  0  0
Concentration unit: uM
Concentration time [d] Control T1    T2    T3    T4      T5     T6    T7
        0                  0   8.05 11.906 13.8  17.872 24.186 28.93 35.924
        2                  0   6.04  8.93  10.35 13.4   18.14  21.7  26.94
        2                  0   8.05 11.906 13.8  17.872 24.186 28.93 35.924
        4                  0   6.04  8.93  10.35 13.4   18.14  21.7  26.94
```

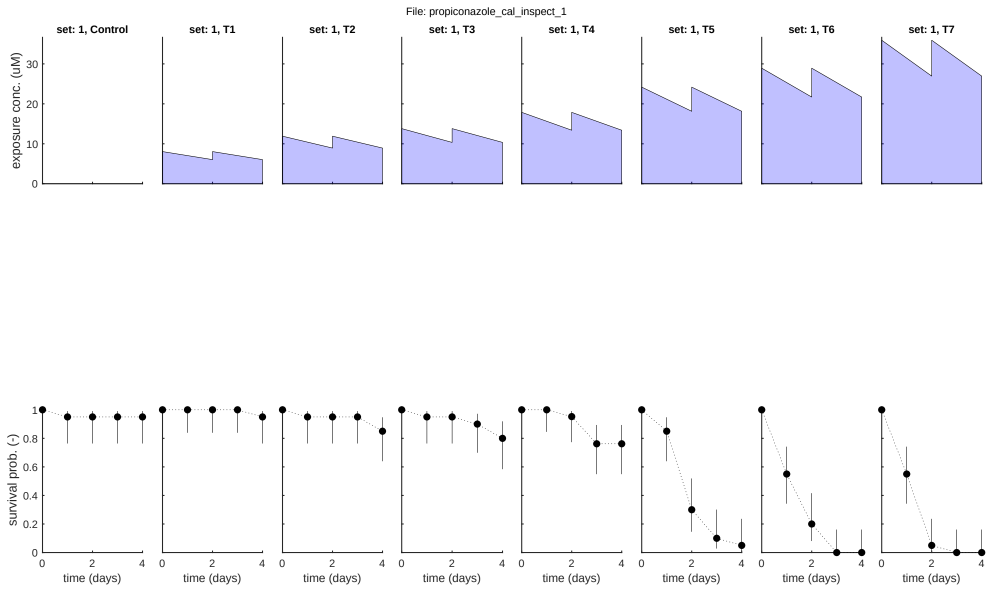


Figure 3: Inspection plot for the data set.

This scenario is plotted in Figure 3. Note that there is a linear change in between renewals (users can enter more time points to more closely mimic exponential decay). Also note that the renewal time is entered twice; it is not needed to make a time point at $t = 1.99$ and another at $t = 2$ (although it is allowed).

The next example is the pulsed data set for the propiconazole case. This data set is also used in the GUTS ring test [5], and is included in the examples files for the software as `propiconazole_pulsed_linear`. The exposure scenario is defined for linear interpolation, so with a rather awkward time vector. Here, the exposure scenario is defined with unique time points. Note that this is not needed in openGUTS; an alternative definition with instant changes is demonstrated in the file `propiconazole_pulsed_renewals` (which is preferable). Data entered (result is shown in Fig. 4):

```
Survival time [d] Control close pulses wide pulses constant
       0             60         70           70          70
       1             59         50           57          70
       2            58          49          53          69
       3            58          49          52          69
       4            57          45          50          68
       5            57          45          46          66
       6            56          45          45          65
       7            56          42          44          64
       8            56          38          40          60
       9            55          37          38          55
       10           54          36          37          54
Concentration unit:    uM
Concentration time [d] Control close pulses wide pulses constant
       0                  0       30.56      28.98      4.93
       0.96               0       27.93      27.66      4.69
       1                  0        0          0         4.69
       1.96               0        0.26       0.27      4.58
       2.96               0        0.21       0.26      4.58
       3                  0       27.69       0.26      4.58
       3.96               0       26.49       0.26      4.54
       4                  0        0          0.26      4.54
       4.96               0        0.18       0.25      4.58
       4.97               0        0.18       0.25      4.71
       5.96               0        0.18       0.03      4.71
       6.96               0        0.14       0         4.6
       7                  0        0.14      26.98      4.6
       7.96               0        0.18      26.28      4.59
       8                  0        0.18       0         4.59
       9                  0        0          0.12      4.46
       9.96               0        0          0.12      4.51
```

Another, wilder, example with missing data in there, is available in the example file `propiconazole_weird` (result is shown in Figure 5):

```
Survival time [d] Control T1 T2 T3  T4  T5 T6  T7
       0               20 20 20 20  21  20 20  20
       1               19 20 19 19  21  17 11   -
       2               19 20 19 19   -   6  4   1
       3               19 20 19 18  16   -  0   0
       4               19 19 17 16  16   1  0   0
Concentration unit:    uM
Concentration time [d] Control T1   T2   T3   T4    T5    T6   T7
       0                 0     8.1 11.9 13.8  17.9  24.2 28.9 35.9
       1.2               0    10.1 14.9    -  22.3   -   36.2 44.9
       1.2               0     0    0      -   0     -    0    0
       2.2               0     0    0      -   0     -    0    0
```

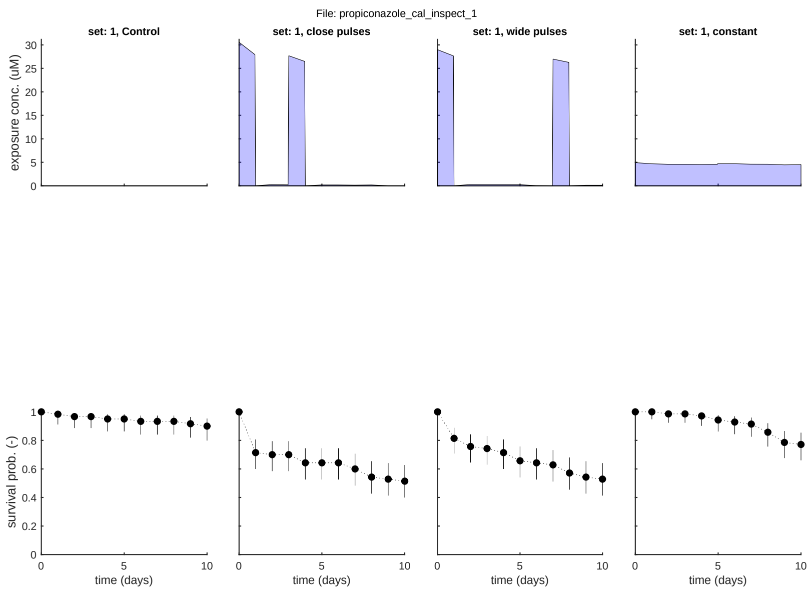


Figure 4: Inspection plot for the data set: pulsed treatments from the ring test.

```
   2.2              0    5    7.4    -  11.2   -    18.1 22.5
   3.2              0   15.1 22.3    -  33.5  35    54.2 67.4
   4                0    8.1 11.9    -     -  25       - 22.5
```

This file was used for testing the extremes of the input-format definition. Note the missing data points in the exposure-scenario list, which are steps that are ignored. If a missing point occurs at the end, or the exposure scenario time points do not cover the time vector for the survival data, an extrapolation is made by assuming that the concentration remains constant. I don't want to extrapolate with a slope other than zero beyond the data provided as it may lead to weird (or even negative) exposure concentrations. Note that in T5 there are several missing points, followed by some concentrations again. In that case, the concentration is simply assumed to change linearly from the last point before the missing ones to the first point after (the entries with `-` are just ignored completely, leading to interpolation). Also note the missing points in the survival data (these points are missing in the lower row of the figure). For such complex exposure scenarios, the inspection plots are helpful to provide visual feedback.

Exposure profiles, such as FOCUS scenarios, are entered in a different way. The text file will be a simple two-column affair without headers or text. Internally, it is placed in the same structured format, and a treatment identifier is added (the filename minus the extension). As an example, the first part of the FOCUS profile used in the ring test:

```
0     0
0.042 0.00003
0.083 0.00006
0.125 0.00009
```

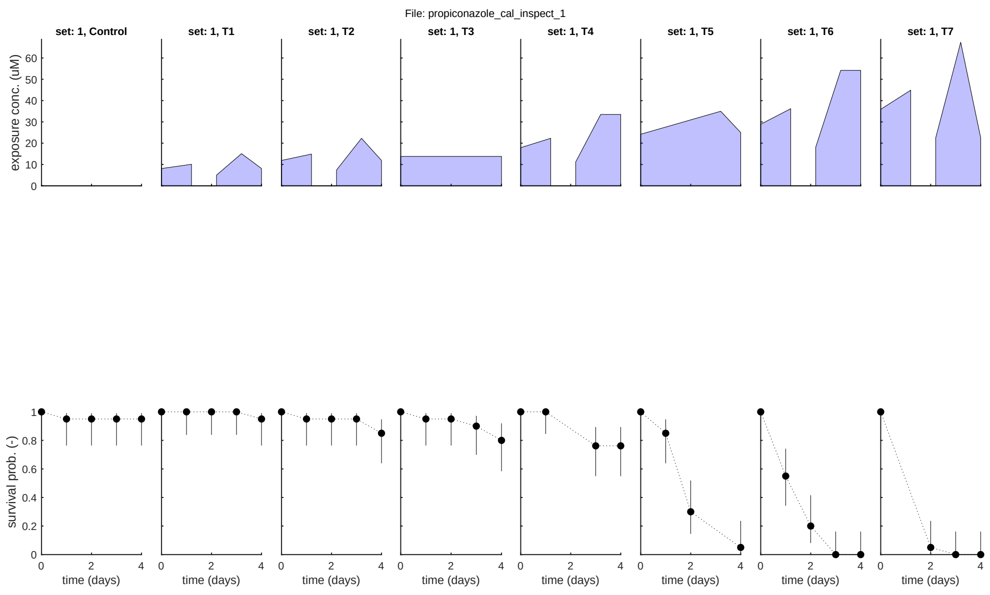


Figure 5: Inspection plot for the data set.

```
0.167 0.00012
0.208 0.00015
0.25  0.00018
0.292 0.00021
0.333 0.00024
0.375 0.00027
...   ...
```

3.4 Internal representation.

Data sets such as entered by the user should be saveable and loadable in the standard format. Tab-delimited plain text is efficient as that can easily be produced/edited by Excel and word processors. For calibration, multiple data sets can be defined/loaded.

Translation to internal representation. The function `prepare_data` translates the series of input matrices to the internal representation. Internally, each series of data sets that is entered is represented by a single object. There is thus one object for the calibration data sets entered, one object for the validation data, and one object for the exposure scenario used for predictions. The object for calibration is made global (`DATAcal`); this is practical as it needs to be passed on many times during the optimisation. For validation and prediction, the object is not made global, but is passed on with the function calls.

The objective of `prepare_data` is to calculate as many things as possible from the data as we can, already at the start. This implies splitting up the survivor and exposure scenario matrix into separate treatments (which also allows easy removal of any missing data points entered). In this way, the subsequent code can be simplified, as much of the required information is already available in the object.

Once for the object (referred to as `data_all` in `prepare_data`):

- `data_all.hb_hat` = fitted background hazard rate for all controls together
- `data_all.nr_sets` = number of data sets
- `data_all.nr_treat` = vector with number of treatments in each data set

For each data set:

- `data_all.time` = entire time vector for the survival data in this data set (as entered in the input matrix)
- `data_all.c_twa` = vector with time-weighted average concentrations for each treatment (used for initial parameter ranges)
- `data_all.c_max` = vector with maximum exposure concentration for each treatment (used for initial parameter ranges, and for scaling of plots)

For each treatment:

- `data_all.surv` = matrix with time, survivors, deaths, survival probability, Wilson score interval (min. and max.); missing points removed
- `data_all.scen` = optimised events-time matrix with time, concentration and slope; missing points removed and time matched to survival data
- `data_all.name` = name of this treatment (text string)

The survivor data set for propiconazole (first example with constant exposure) is thus broken up into individual treatments and returned in the variable `DATAcal` with the following fields:

```
DATAcal =

  struct with fields:

     nr_sets: 1
      hb_hat: 0.0131
        name: {'Control'  'T1'  'T2'  'T3'  'T4'  'T5'  'T6'  'T7'}
        time: {[5x1 double]}
    nr_treat: 8
        surv: {1x8 cell}
        scen: {1x8 cell}
       c_twa: {[0 8.0500 11.9060 13.8000 17.8720 24.1860 28.9300 35.9240]}
       c_max: {[0 8.0500 11.9060 13.8000 17.8720 24.1860 28.9300 35.9240]}
```

Zooming in on the field `surv` for the 5th cell (treatment T4), we can take a closer look at the survival matrix for this treatment:

```
DATAcal.surv{1,5}

ans =

         0          21           0           1           0           0
         1          21           1           1     0.84536           1
         2          20           4     0.95238      0.7733     0.99154
         3          16           0      0.7619     0.54908     0.89372
         4          16          16      0.7619     0.54908     0.89372
```

The first row is the time vector for this treatment (when the data set contains missing points for this treatment, this time vector will be shorter than the total vector). The second row is the number of survivors, third row the number deaths in each interval starting at the time in the first column (and for the last time point covering the survivors at the end, that will die after the test). The fourth row is the survival probability from the data, with in the fifth and sixth column their CI (Wilson score interval on proportions).

Looking at the exposure definition derived from the input data, the exposure scenario is translated into a series of concentrations and slopes:

```
DATAcal.scen{1,5}

ans =

         0      17.872           0
         4      17.872           0
```

This is for the simple case of constant exposure, and therefore, all slopes (last column) are zero. Note that a final time point is added at $t = 4$, which marks the end of the survival data set. The exposure scenario will always be recalculated to cover the same duration as the survival data.

The exposure scenario for T5 in the weird exposure scenario looks like this (missing points are removed and slopes calculated):

```
DATAcal.scen{1,6}

ans =

         0        24.2       3.375
       3.2          35       -12.5
         4          25           0
```

Also take a look at T6, which included multiple concentrations at the same time point:

```
DATAcal.scen{1,7}

ans =

         0   28.9000    6.0833
    1.2000         0         0
    2.2000   18.1000   36.1000
    3.2000   54.2000         0
    4.0000   54.2000         0
```

The double time points have been removed, and their information is included in the slopes. At $t = 0$, the concentration starts at 28.9 and continues with a slope of 6.0833 until $t = 1.2$. At that time, the concentration will switch to zero with a slope of zero. The end of interval 1 does not correspond to the start of interval 2, but that is irrelevant for the calculations (the calculated scaled damage over each interval *will* match at the switch points).

If we have two data sets, the object becomes more complex. This is illustrated by entering both the data for constant and time varying exposure for calibration, which produces:

```
DATAcal =

  struct with fields:

     nr_sets: 2
      hb_hat: 0.0109
        name: {2x8 cell}
        time: {[5x1 double]  [11x1 double]}
    nr_treat: [8 4]
        surv: {2x8 cell}
        scen: {2x8 cell}
       c_twa: {[1x8 double]  [1x4 double]}
       c_max: {[1x8 double]  [1x4 double]}
```

The field `surv` is now a 2×8 cell array. The first row will contain the 8 treatments for the constant exposure. For the second row, only the first 4 elements will be filled, as the second data set only has 4 treatments. The best estimate for the background hazard rate is now somewhat different from the previous value as it is estimated on the controls from both data sets combined.

4 Definition of parameter matrix and project file

Relevant design decisions:

- Parameter ranges and settings for the parameters will be generated automatically by the software, based on the input data provided for calibration. No user interaction is required to perform a calibration.
- Expert users can override default parameter ranges and settings (at their own risk; some settings will lead to poor performance or even failure of the calibration).[4]
- There is a user-allowed switch to fix background hazard rate h_b to a value fitted on the control data (combining all data sets at this stage of the analysis; separate for calibration and validation). Note that the fitted background hazard is already part of the data object.
- Since we always calibrate SD and IT, there will be two parameter matrices to worry about (and that have to be shown on screen). These will be (slightly) different matrices.
- Calibration results will always be saved in a project file (Matlab version only). This file will contain the necessary information to skip the calibration stage. This allows users to continue with an analysis without recalibration, to send an analysis to someone else for checking, and it allows simplification in the code.

Selecting how to use the background hazard. The user has to decide whether to fit the background hazard with the other parameters, or to fix it on a value that is estimated from the controls of all data sets, at this stage of the analysis. There are two global switches in the main script for how to deal with the background hazard rate: `GLO.fix_hb_cal` for the calibration data and `GLO.fix_hb_val` for the validation data. Note that `prepare_data` will always fit the background hazard rate on the controls, so that h_b is available from the data object whenever needed.

Number of parameters fitted. There are five model parameters in the GUTS-RED models altogether, but never more than four are fitted. It is also possible to fit less parameters (at least one in the Matlab version, but at least two in the standalone version). The code is set up in such a way that more parameters *can* be fitted. However, it is unclear whether the optimisation algorithm is still robust enough for more than four fitted parameters.[5]

[4] In the standalone version, changing the default settings will produce a warning in the output report. For the Matlab version, this is hardly useful as users can always modify the code in any way they like.

[5] This has not been properly tested. However, the algorithm suffers from the 'curse of dimensionality', so extension to more free parameters is not trivial.

4.1 Parameter matrix

We have, in total, five model parameters although a maximum of four will be fitted to a data set. The parameter definition is in the form of a 5 × 5 matrix:

`pmat =`

value	fit y/n	min	max	norm/log
k_d	1/0	min	max	1/0
m_w	1/0	min	max	1/0
h_b	1/0	min	max	1/0
b_w	1/0	min	max	1/0
F_s	1/0	min	max	1/0

The first column is used to pass parameter values to model calculations (during calibration and during calculation of model curves for plotting). Each parameter has a column for fitting (1) or fixing (0) the parameter in the optimisation. For SD, the fraction spread of the threshold distribution will automatically be fixed ($F_s = 1$), and for IT the killing rate ($b_w = \infty$). For these two parameters, it should not be possible to fit them if they are automatically fixed (i.e., for SD, F_s should never be fitted, and for IT, b_w should never be fitted). Additionally, users may want to fix other parameters to values of their liking.

Note that I prefer to use F_s as measure of spread in the distribution, rather than β, as F_s will have a more friendly parameter range (β goes to infinity when the distribution is very narrow, whereas F_s goes to 1). At this moment, β is shown in the output as a little extra information (helps in comparing output to other implementations).

Each parameter has a min-max range that is filled automatically. The method that I designed for scanning parameter space in multiple dimensions (Section 5) is aided by narrowing down the volume of space that needs to be searched. For GUTS, large parts of parameter space will yield nonsensical fits to the data, so these parts can be excluded. The range within which we can expect (or identify) a parameter depends on the design of the toxicity test. The definition of reasonable parameter ranges for GUTS analyses is presented in detail in Section 4.3.

The final column decides whether the parameter will be fitted on $\log_{10}$ scale or on normal scale. Log-scale fitting is more efficient when parameters span a large range, and when differences between very small values are meaningful (e.g., for k_d).

The values and ranges can be changed by the user, with certain constraints:

- All values must be positive, and are entered on normal scale.
- All upper bounds must be lower than infinity.
- All lower bounds must be higher than zero (to avoid having to make new bounds when a parameter is fitted on log scale).
- The upper bound must be higher than the lower bound (for fixed parameters, the ranges are set to the same value as the fixed parameter value).

- The lower bound for F_s must not be lower than 1.[6]
- For IT, b_w must be fixed to infinity, for SD, F_s must be fixed to 1.

A second parameter matrix will be filled by `calc_parspace`, which is called `pmat_print` as it contains the information to be printed on screen. Format is as similar to `pmat` with 5 rows, but now with 8 columns:

- Column 1 is the best-fitting value for each parameter. These are also placed in `pmat`, so that is some duplication.
- Column 2 and 3 contain the lower and upper edge of the 95% CI for each parameter.
- Column 4 and 5 contain the lower and upper edge of the joint 95% CI for each parameter. These are generated in `calc_parspace`, but are not used at the moment.
- Column 6 and 7 contain flags for the lower and upper boundary of the CI for each parameter. A 1 marks that the CI is running into a min-max boundary.
- Column 8 contains a flag of 1 when the CI of the parameter is actually a broken set (broken sets will be flagged, not calculated/reported; only the outer edges of the CI are given).

Deriving search ranges. The matrices `pmat` for SD and IT are generated automatically in the function `startgrid`, depending on the calibration data set in the global `DATAcal`, on the settings for the special case (SD or IT), and on the choice for dealing with the background hazard rate. Afterwards, (expert) users can override the automatic choices with their own settings. However, the Matlab version does not have any specific code for that (and no GUI), and it does not perform a check on the constraints on `pmat` as outlined above.

In `startgrid`, rules are used to calculate search ranges. There is some dependency between the rules for different parameters. At this moment: the minimum value for k_d is used to calculate the lower bound of m_w and the upper bound of b_w. This is not made interactive; we assume that the user should know what he/she is doing when changing the default ranges (after all, it is an 'expert setting').

Before calibration, the value column in `pmat` is only used for fixed parameters. For the others, no starting value is used (only the range). The first column is used internally to pass parameter values to be tried to the model calculations (it will finally be filled with the maximum-likelihood estimate for the fitted parameters).

Important note. The default settings of `pmat` have been extensively tested for robustness and efficiency. However, if users modify this matrix, it is impossible to guarantee that all possible settings will lead to a robust optimisation. In fact, we can be certain that

[6]Since $F_s = 1$ will lead to a 'division by zero' in IT calculations, the code makes sure that this parameter is at minimum $1 + 10^{-6}$. The C++ version always uses a lower range that is minimally 1.05.

some settings will lead to poor performance or even complete failure of the optimisation algorithm. However, we can say that the user is on his/her own when meddling with the defaults.

4.2 Calibration project file

After calibration, a project file is saved with all the necessary information about the calibration. This file is used internally for all subsequent calculations, and can also be used to continue with an analysis at a later time without needing to redo a calibration.

Project file contents. The project file for the Matlab version, saved by `calc_parspace`, will contain the following elements (separate files are made for SD and IT):

- `pmat` = parameter matrix used, with best values in first column.
- `coll_all` = the entire sample from parameter space (usually ten to thirty thousand rows with five or less columns).
- `pmat_print` = a matrix with best values of the parameters, confidence intervals, and warning flags.
- `coll_prof_pruned` = a limited cell array with the results from profiling of the likelihood (contains only the information needed for plotting).
- `DATAcal` = the data object used for calibration.
- `concunit` = concentration unit.
- `fix_hb_cal` = switch of what to do with background hazard.

This file is saved as a binary `mat` file. It is used internally to calculate model curves, model predictions, and CIs. This is the *only* way in which the calibration results are communicated to the rest of the code!

This file is saved by `calc_parspace`. To keep file size in check, `coll_all` only contains columns for parameters that were fitted (for fixed parameters, the whole column for this variable would have the same value). The information to reconstruct the entire matrix is provided in `pmat` by the settings of the fit/fix switch for each parameter, and the fixed values in the first column. Loading this file, and reconstructing the full `coll_all` matrix, is dealt with in `load_sample`. When this function is called with a flag that we are restarting an analysis from a saved calibration, it will call the function to produce the plot of parameter space for inspection of the sample.

Differences with the C++ code.

- The project file for the standalone version will be a readable text file, containing both the SD and IT results, and including various outputs. This file will not be compatible between the two platforms.
- The project file for the standalone version will only be generated on user request. In the Matlab version it is always saved and used to communicate calibration results to the rest of the workflow.

4.3 Deriving starting ranges for each parameter

The parameter ranges for the starting values are based on general experience and heuristics. It is impossible to define exact ranges that work in every situation, but we can make educated guesses based on the model, the data set, and common principles. In some cases, the resulting heuristic may seem rather arbitrary, but they are tweaked to provide good results in common cases. Cases may occur where these do not work properly, although these data sets probably have a number of quality issues that require further scrutiny anyway. These rules still need to be checked more thoroughly with time-varying exposure (especially block pulses). However, the software will allow (expert) users to override the default ranges.

Fitting on normal scale or log scale. For several of the GUTS parameters, it will be better to define parameter space on log scale. This is efficient when a parameter spans a large range, and when very small values are meaningful. For example, the dominant rate constant (k_d) will be (theoretically) within zero and infinity, and the difference between 0.01 and 0.02 d^{-1} is just as relevant for the fit (and the value of the other model parameters) as between 10 and 20 d^{-1}. We thus need the detail on the small values, and hence, a log scale for this parameter is advisable. Similarly, the killing rate (b_w) for SD cases spans a large range and small values are meaningful.

The threshold (m_w) requires some more thought. Are very small values of the threshold meaningful? That depends on the value of k_d. Small values of k_d imply low scaled damage levels, and hence small values of the threshold are meaningful. In fact, when k_d goes to zero in the model optimisation for a specific data set, so will m_w, and, for SD, b_w will go towards infinity (slow kinetics). However, for larger values of k_d, we do not need much detail on low values of m_w, and we may need more detail on larger values (e.g., in case of 'single-dose runaway'), and m_w may even be truly zero. The algorithm therefore starts with a normal scale for m_w, but regularly checks for signs of slow kinetics (when k_d is relatively low and there is a positive correlation between m_w and k_d). If there are, the analysis restarts completely with m_w on log-scale.

What about the background hazard rate (h_b)? For this parameter, very low values are meaningless, and simply imply that there is no background mortality to speak of. The difference between $h_b = 0.001$ and $h_b = 0.002$ d^{-1} is generally trivial and has no effect on the estimates for the other model parameters. A normal scale is thus most suitable for this parameter.

What about the factor spread for the threshold distribution (F_s)? The lowest values for this parameter is 1 (which means no differences between the individuals). Simulations show that changing the value from 1-2 has a large effect on the model curves, whereas the difference between 10 and 20 is less pronounced. Therefore, a log scale will be more efficient.

Background hazard rate, h_b. In GUTS, we take a a constant background hazard rate h_b. Survival probability depends on the hazard rate as follows:

$$\begin{aligned} S_b &= \exp(-h_b t) && (1) \\ \ln S_b &= -h_b t && (2) \\ h_b &= -\ln S_b / t && (3) \end{aligned}$$

The data object already includes an estimate for h_b, using the observed survivors in the control. If this parameter is fitted along with the exposure data, a default range is fine (note that we assume the time vector is in days):

$$h_b = [10^{-6}, 0.07] \qquad (4)$$

The 0.07 d^{-1} represents a substantial upper bound for the background hazard rate; it implies that at the end of a 4-day toxicity test, the survival probability in the control is 0.75. In other words, when we start with 20 animals in the control, we expect 15 to survive at the end of the 4-day test. This seems like a very high level of control mortality. However, for some data sets, the confidence regions stretches out to high values of h_b (which generally implies that h_b is better fixed to the background mortality).

The lower bound of 10^{-6} is set to a non-zero value for several reasons. Firstly, if the user decides to fit this parameter on a log scale, a non-zero bound is needed. Furthermore, an absolute zero probability of deaths seems impossible for any practical experimental test. And, finally, zero gives problems in the likelihood function (Eq. 45), where the logarithm of the death probability is taken. The value of 10^{-6} is low enough to be indistinguishable from zero for all intents and purposes.

Dominant rate constant, k_d. The true dominant rate constant may be anywhere between zero and infinity. However, the design of the toxicity test places limits to the values that we can still identify from the data (for some data sets, k_d is 'practically non-identifiable'). The k_d is linked to the fraction of steady state that is achieved for the scaled damage D_w at a certain time point t:

$$\begin{aligned} D_w(t) &= D_w(\infty)\,(1-\exp(-k_d t)) && (5) \\ F_{ss} = D_w(t)/D_w(\infty) &= 1-\exp(-k_d t) && (6) \\ \exp(-k_d t) &= 1-F_{ss} && (7) \\ k_d t &= -\ln(1-F_{ss}) && (8) \\ k_d &= -\ln(1-F_{ss})/t && (9) \end{aligned}$$

We could use the test design to set ranges within which the parameter can still be identified. However, in our predictions, we like to include the uncertainty in k_d: very high and (especially) very low values may not be identifiable from the data, but may still be very relevant for the model predictions. Since we decided to skip the special cases for $k_d = 0$ and $k_d = \infty$, it is better to set a fixed relevant range for all compounds. The upper boundary for k_d is the value that leads to 95% of steady state in half an hour (this relates to the hourly resolution of FOCUS profiles). For the lower boundary, we select a value that leads to 95% of steady state in 5 years (which seems long enough not to bother about anything even slower). This leads to the following range:

$$k_d = \left[\frac{\ln(20)}{5 \times 365}, \frac{\ln(20)}{0.5/24}\right] \tag{10}$$

When k_d runs into these ranges, we have a situation of fast or slow kinetics. Especially slow kinetics would be relevant as it signals (almost) irreversible effects. Nevertheless, the range that we set here seems wide enough for most standard tests and for most model predictions.

There may, however, be toxicity tests with observations after very short or very long exposure times. In those cases, it makes sense to extend the range on the basis of the actual time vector in the toxicity test. There are two reasons to do so. The first is that it can provide more accurate parameter estimates. The second is that accurate CIs require knowledge of the highest value of the likelihood achieved (the CIs are based on a likelihood ratio test, so relative to the best value). For tests with observations after very short or very long exposure times, the best value for the likelihood may increase if we allow k_d to go beyond the fixed bounds, as there is more information in this specific (non-standard) data set.

As a reasonable minimum for k_d, we take the situation where only 5% of steady state is reached by the end of the test. As a reasonable maximum, we take the situation where 99% of steady state is achieved before a one-tenth of the first observation time point after the start of the test ($t(2)$, as zero will be the first point).[7]

$$k_d = \left[\frac{-\ln(1-0.05)}{t(\mathrm{end})}, \frac{-\ln(1-0.99)}{0.1 \times t(2)}\right] \tag{11}$$

The value to use would be the lowest of the two lower bounds and the highest of the two upper bounds. The alternative bounds of Eq. 11 only come into play when the first observation is within the first 7.5 hours, or the total test duration is more than 310 days. In standard test situations, we would therefore only see the fixed test-independent boundaries of Eq. 10.

Threshold for effects, m_w. The threshold for effect will be somewhere between zero and infinite. However, the upper limit for identification will depend on the maximum

[7]This assumes that exposure starts at $t = 0$. If exposure starts later, and observations shortly after the start of the pulse are included in the test design, manually decreasing the lower range of k_d can be considered.

exposure concentration used in the test. For SD, the maximum exposure concentration in the test will truly be the upper limit: higher values are of course possible, but then there would have been no mortality at all in the test (and we would never be able to estimate the threshold from the data set). For IT, higher values are possible as the threshold is a distribution with m_w as median value. However, much higher values than the maximum test concentration would be unlikely (that would mean that the effect level in the highest treatment would be less than 50% at the end of the test).

We can thus depart from the following ranges, depending on whether we are using SD or IT:

$$m_w = [0, 0.99 \times C_w(\text{max})] \qquad \text{when using SD} \tag{12}$$

$$m_w = [0, 2 \times C_w(\text{max})] \qquad \text{when using IT} \tag{13}$$

The upper boundary will generally be way too large (as there will only be effect in the highest exposure treatment). However, when a data set with pulse exposure is used, a threshold close to the maximum pulse height is not unrealistic.

If we fit m_w on log-scale (when slow kinetics is indicated), we need to have a non-zero lower bound. To make things simple, we'll also use this lower bound when m_w is fitted on normal scale (it will be low enough not to impact the fits, and can always be lowered by the user). A reasonable lower bound is more difficult to establish than the upper one. It relates to the damage levels established during the test, which are given by:

$$D_w(t) = C_w \left(1 - \exp(-k_d t)\right) \tag{14}$$

As a reasonable lower bound, we could use the damage level that is achieved within four hours, at the lowest exposure treatment (above zero), at the lowest allowed dominant rate constant. This leads to the following range:

$$m_w = \left[C_w(\text{min}) \left(1 - \exp(-k_d(\text{min})(4/24))\right), a \times C_w(\text{max})\right] \tag{15}$$

Where a is 0.99 or 2, depending on whether SD or IT is used (see equations above).

Killing rate, b_w. Very low values of the killing rate are needed when the scaled damage levels above the threshold are very high. In such cases, even low b_w values can lead to substantial mortality. To find a reasonable minimum, we can thus focus on instantaneous steady state (k_d is very high) and a threshold of zero ($m_w = 0$). In this situation, $D_w = C_w$, and the hazard rate due to the chemical reduces to (ignoring the background hazard):

$$h_z = b_w C_w \tag{16}$$

Since we now have a constant hazard rate again, integration is simple:

$$S_z = \exp\left(-b_w C_w t\right) \tag{17}$$

For every exposure concentration C_w, we can now link survival to b_w. What is the lowest reasonable value for b_w? We can take that as the (pretty extreme) situation where, at the end of the test, in the highest exposure concentration, the survival probability (due to the chemical) at the end of the test is still as high as 0.90:

$$\begin{aligned} 0.90 &= \exp\left(-b_w C_w(\max) t(\text{end})\right) && (18) \\ -\ln 0.90 &= b_w C_w(\max) t(\text{end}) && (19) \\ b_w &= \frac{-\ln 0.90}{C_w(\max) t(\text{end})} && (20) \end{aligned}$$

The highest reasonable value for b_w is more complex to establish. When b_w is very high, the survival probability drops rapidly when the damage level exceeds the threshold. The highest values of b_w will be relevant in situations of 'slow kinetics': that is when scaled damage levels are very low, and a high b_w would be needed to create an effect. The rate at which the survival probability drops depends on two factors: how rapidly does the damage level increase, and how big is the b_w. However, as long as the damage level is below the threshold, there will be no effect at all.

What kind of value for the hazard rate do we consider 'very high'? A hazard rate times a short time interval is the probability to die (p) in that interval (given that you were alive at the start of it): $p = h\Delta t$ (it has to be a very short interval; we are approximating a continuous function with a discrete step here). We can say that a death probability of 0.95 in some short interval is very fast, and hence:

$$\begin{aligned} h_z \Delta t &= 0.95 && (21) \\ h_z &= \frac{1}{\Delta t} \times 0.95 && (22) \end{aligned}$$

We can thus say something about the maximum value for the hazard rate that we have to consider. The hazard rate is, of course, linked to the damage level and the threshold. However, when the damage level is below the threshold, the hazard rate due to the chemical is zero. We are thus interested in the moment at which the threshold is exceeded: the small increase over the next short time period should not lead to hazard levels that exceed our maximum. We are thus interested in how the scaled damage level increases over a short interval. To start guessing, we can look at the derivative of the damage equation:

$$\begin{aligned} D_w(t) &= C_w\left(1 - \exp(-k_d t)\right) && (23) \\ D_w(t) &= C_w - C_w \exp(-k_d t) && (24) \\ \frac{dD_w}{dt} &= k_d C_w \exp(-k_d t) && (25) \end{aligned}$$

For a (short) time interval interval Δt, we can thus estimate the change in scaled damage by:

$$\Delta D_w = k_d C_w \exp(-k_d t)\Delta t \tag{26}$$

The increase above the threshold will lead to mortality. When we start exactly at the threshold ($D_w = m_w$), an increases in the damage level by ΔD_w will produce a hazard rate:

$$h_z = b_w \Delta D_w \tag{27}$$
$$b_w = h_z / \Delta D_w \tag{28}$$

Next, we can fill in the equations for h_z and ΔD_w:

$$b_w = \frac{\frac{1}{\Delta t} \times 0.95}{k_d C_w \exp(-k_d t)\Delta t} \tag{29}$$
$$= \frac{0.95}{k_d C_w \exp(-k_d t)\Delta t^2} \tag{30}$$

For k_d, we take the minimum value as derived earlier. For C_w, we can take the highest tested concentration. For t, we can take the point half-way into the test, although the exact time is not too important (as the derivative dD_w/dt will be rather constant over the entire test duration at the minimum value of k_d). For Δt, we can take 1/24 of a day (one hour). This leads to:

$$b_w = \frac{24^2 \times 0.95}{k_d(\text{min}) C_w(\text{max}) \exp(-k_d(\text{min}) t(\text{half}))} \tag{31}$$

This is not exactly rocket science, but will always produce a b_w that leads to very steep survival curves, even at very low k_d values, and also for the lower exposure levels. Basing these values on the maximum C_w is probably more practical than using the lowest, and also works fine when using pulse exposures (where you generally don't have a range of concentration levels but only relatively high exposures that cause plenty of effect on short exposure).

The resulting range is then as follows, but note that the upper boundary depends on the unit of t (which should be days):

$$b_w = \left[\frac{-\ln 0.90}{C_w(\text{max}) t(\text{end})}, \frac{24^2 \times 0.95}{k_d(\text{min}) C_w(\text{max}) \exp(-k_d(\text{min}) t(\text{half}))} \right] \tag{32}$$

Spread of the threshold distribution, F_s. Theoretically, the lowest value of the factor spread of the threshold distribution is simply 1: this implies that there is no distribution (or only a Dirac-delta distribution). However, a value of exactly one causes some problems in the calculations: it yields a 'division by zero' in Eq. 38, and can cause some unrealistic behaviour in the optimisation routine. Therefore, a value of 1.05 is proposed as general lower value, which still implies extreme similarity between the individuals.[8]

The upper value is more difficult to set, and could depend on the test cohort (cloned animals from a laboratory population are likely more similar than field-collected animals). Nevertheless, as a reasonable upper bound, we can set 20 (I have only come across one data set yet that required a - much - higher value to fit properly). This implies a factor of 400 between the 0.025 and 0.975 quantile of the threshold distribution.

$$F_s = [1.05, 20] \tag{33}$$

Correlations between parameter ranges? The ranges specified in this section will be applied as independent ranges. They thus specify a hyper rectangle in parameter space. Obviously, a substantial volume of this hyper rectangle will yield nonsensical survival patterns. It would therefore be more efficient to *a priori* exclude certain parts of this space, i.e., by imparting some kind of correlation structure on the parameter ranges. At this moment, such refinements are not considered as they are difficult to generalise for all possible data sets entered. We therefore felt that it was more important to be robust than to be fast. Furthermore, the increase in calculation speed is probably not big enough since we are only talking about a limited speed increase in the first round of the optimisation algorithm.

[8]If the possibility for IT with extremely similar individuals needs to be explored, it is always possible to use SD with b_w fixed to a very high value.

5 Parameter-space explorer

Relevant general decisions:

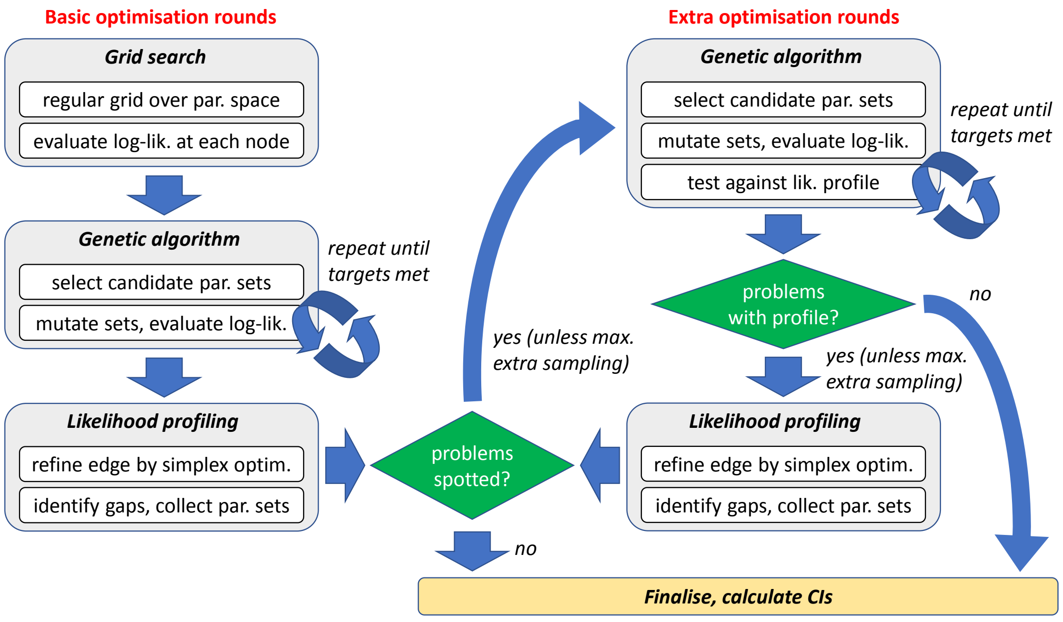
- We use frequentist inference, not Bayesian. This decision was made as Bayesian inference is more difficult to robustly include into a software without user interaction (see also Section 2.3.6 of the GUTS e-book [5]).
- We do not implement the special cases for fast and slow kinetics. The software will flag when the parameter CI runs into a boundary, which will require some knowledge of the user to interpret the output (see also the separate document on interpretation of model output).
- We use a dedicated algorithm, the parameter-space explorer, which is able to operate without starting values or user interaction. This algorithm is optimised for GUTS analyses, and combines grid search, a genetic algorithm, and likelihood profiling.
- The algorithm must be robust without interference from the user. This means that it should always find the global optimum, and always get a reasonable coverage of the parameter space that is used for CIs.
- Some of the settings for the algorithm are set in `initial_setup`, but it is not recommended to change them as they interact. However, the user can modify `pmat` and thereby the parameter ranges, fit/fix parameters, and change the log/normal setting.

The algorithm for exploring parameter space needs to do several things:

1. Find the best-fitting parameter set (the set with the highest resulting log-likelihood value) in a robust manner (insensitive to local minima).
2. Find CIs on estimated model parameters.
3. Generate a sample from parameter space that is within a certain threshold value from the highest log-likelihood. This sample is propagated to calculate CIs on model predictions.
4. Generate a sample from the joint-confidence region. This is only used to provide extra robustness, and to allow for a visual inspection of the shape of parameter space.

5.1 The algorithm

The framework to tackle the requirements listed above is based on sampling from parameter space. The general approach for frequentist sampling (and how to use it for statistical inference) is explained in Appendix D of the GUTS e-book [5]. The algorithm in openGUTS combines grid search, a genetic algorithm, and likelihood profiling, to provide extreme robustness. The outline of the entire algorithm is shown in Figure 6. The general idea is to try a large population of candidate parameter sets, and select the most promising sets



Basic optimisation rounds
Grid search
regular grid over par. space
evaluate log-lik. at each node
Genetic algorithm
select candidate par. sets
mutate sets, evaluate log-lik.
repeat until targets met
Likelihood profiling
refine edge by simplex optim.
identify gaps, collect par. sets
problems spotted?
yes (unless max. extra sampling)
no
Extra optimisation rounds
Genetic algorithm
select candidate par. sets
mutate sets, evaluate log-lik.
test against lik. profile
repeat until targets met
problems with profile?
no
yes (unless max. extra sampling)
Likelihood profiling
refine edge by simplex optim.
identify gaps, collect par. sets
Finalise, calculate CIs

Figure 6: General flowchart for how the optimisaton algorithm operates. Well-behaved data sets will only require basic optimisation (left part of the scheme); the extra optimisation (right part of the scheme) is a fail-safe to take care of more problematic data sets. The genetic-algorithm step will be repeated several times until its 'targets' are met (minimum number of sets in inner and outer rim, and for the extra rounds also a check against the profile likelihood). At several points in the process, there is a check against a maximum number of iterations to avoid getting stuck.

for a next round, where each set will be propagated with random mutations to a new set of candidates. The tried sets from the previous rounds do not need to be discarded: the only thing that counts is whether their likelihood is within a certain distance from the best-fitting set. However, as the best-fitting set is also subject to change with consecutive rounds, the confidence region (CR) will change as well (and a set that was acceptable in round i may not be anymore in round $i+1$).

In principle, we can keep all of the sets that are tried: their likelihood value is the only thing that counts, and that does not change as the optimisation progresses. However, in view of memory use (and the size of the file to be saved), I prune the set every round, and remove all sets that are too far outside of the 95% joint CR as established in that round. For propagating the uncertainty due to model predictions, we only need the sets within a smaller joint CR (cut off at $0.5 \times \chi^2_{df=1,\alpha=0.05}$). Nevertheless, for robustness (to make sure that local minima are not missed, and to still have a sample when the best fit improves) it is essential to let the algorithm generate a sample of the 95% joint CR, from which we can take a sub-sample with smaller coverage later on.

In general, the algorithm operates following these steps:

1. Create a regular grid in parameter space, covering the ranges of the (log-transformed)

parameter values (ranges were established by `startgrid` according to the rules in Section 4.3). The grid thus has the same dimensions as the number of fitted parameters.

2. Evaluate all parameter sets from the grid: each parameter set now gets a minloglik (minus log of the likelihood). This is not an optimisation, only a single evaluation. All parameter sets with their corresponding minloglik are stored in the matrix `coll_all`.
3. Find the best fit (lowest minloglik) so far, and select the parameter sets whose minlogliks are within a certain distance (defined in `chicrit_rnd`) from the best fit. If this results in less than a minimum number (`n_ok`) of sets, use the `n_ok` best ones instead. These parameter sets are collected in the matrix `coll_ok`.
4. At this point, the same series of steps is being repeated for subsequent rounds i. For each parameter set in the matrix `coll_ok`, create `n_tr_i` new parameter sets using a random mutation of the parameter values. For each parameter, a random number is added or subtracted between zero and a fraction `f_d_i` of the initial parameter-grid spacing (`f_d_i` will decrease with each round, such that the sample can contract).
5. The mutated parameter sets from the previous step are evaluated against the data set (again, no optimisation), yielding a minloglik, which is stored with the parameter set in the matrix `coll_all`.
6. Find the best fit (lowest minloglik) so far, and perform a quick and dirty simplex optimisation to improve upon the optimum achieved so far. Add this optimised parameter set to the matrix `coll_all`.
7. Check for signs of whether a restart (back to step 1) is needed (when slow kinetics is indicated). This check is only performed under certain conditions (when the threshold is fitted on normal scale and the dominant rate constant on log-scale).
8. See how many parameter sets are within the joint CR (minloglik within a specific criterion, depending on the number of free parameters) and the single-parameter region (minloglik within $0.5 \times \chi^2_{df=1,\alpha=0.05}$). If it is enough (at least `n_conf`, or until we reached a maximum number of rounds), jump to step 10.
9. Prepare for a new round. Based on the results so far, select an optimal set of parameters to continue with, and place them in `coll_ok`. In general, we will continue with promising new sets generated in the last round only (to avoid resampling from the same points over and over again). However, if the number of sets is too small, sets from all previous rounds will be included as well. In special situations, a more specific `coll_ok` is defined. When we have sufficient points in the total cloud, but not enough in the inner rim (the single-parameter region), only sets around the inner rim are taken. Check how many new mutations will be tried in the next round, and if it is very high, reduce `n_tr_i`. If few good values are found, increase `n_tr_i`. Return back to step 4.

10. Run a regular simplex optimisation, starting from the best set established so far in `coll_all`, to make sure we are in the optimum.

11. Refine the edges of the cloud for a single-parameter with profiling. The parameter's range (as found so far) is divided into a large number of slices (50). Two rough simplex optimisations are performed: one starting from the best-fitting set in this slice, and the best one from the previous slice (if available). The best of these is subsequently used in a normal simplex optimisation. This process goes from left to right (low to high values of the parameter). When it reaches the end of the sample cloud, it goes back to the left, optimising based on the previous point in the profile (this helps to catch cases where profiling misses a branch in parameter space). When the edges of the resulting profile are not well above the cut-off criterion (half of $\chi^2_{df=1,\alpha=0.05}$), the profiles are extended outwards until they are (or until they hit a boundary). This yields a very robust profile likelihood. If a better optimum is located, run a regular simplex optimisation to improve it.

12. The profiling collects the points where the profile (the optimised log-likelihood in a slice) is substantially better than the best point in that slice (this indicates an area where sampling has been poor). Furthermore, it collects the points where the profile extends beyond the range of sample points. If the profiling has located a much better optimum, this shifts the sample, and there may not be enough points in the inner rim anymore. If one of these situations happen, an additional series of sampling rounds is initiated, using the collected points as candidate sets. If no problems are identified, jump to step 16.

13. Mutate candidate parameter sets randomly and calculate the associated minloglik (again, no optimisation). Test the new total sample in relation to the profile likelihood. Collect points where the profile is much better than the sample, and flag when there are sample points that are below the profile line. These are the candidates for further sampling.

14. Decide whether to continue sampling. If there are insufficient sets in the inner rim, go back to step 13. If the previous step finds points where the profile is much better than the sample, use them in another round of sampling, go back to step 13.

15. When there are sample points that are (non-trivially) below the profile line, do another profiling step. If a better optimum is located, run a regular simplex optimisation to improve it. If this results in flagged points where the profile is much lower than the sample, use these points as candidate sets in another sampling round, go back to step 13.

16. Calculate CIs from the profiles. Check whether any CIs are open on one side (mark them). Check whether CIs run into bounds. Also check whether the CI is a broken set. The sets from the profile are added to the total matrix `coll_all` as well. This helps in rather extreme cases where the mutation algorithm had difficulties to sample in a relevant region of parameter space.

17. Save the calibration results to a `mat` file.

The values for `chicrit_rnd_i`, `n_tr_i` and `f_d_i` will change with each round of the analysis. The cut-off criterion for continuation of sets (`chicrit_i`) will decrease with each round to end up at the final value (half the χ^2 criterion, 95% with *df* the number of free parameters). The number of new tries per parameter set `n_tr_i` decreases as well: initially, we need many tries to get a robust coverage of parameter space, but when we already have a large number of sets within the joint CR, we can use less tries in subsequent rounds (otherwise, we end up with much more values than needed in `n_conf`). The maximum mutation distance `f_d_i` will also decrease with subsequent rounds, starting at 1 (so the maximum mutation is the same as the grid spacing used). All these settings have been tweaked to obtain a good balance between robustness and speed of contraction of the sampling cloud (for a wide range of relevant examples).

In step 8, the algorithm will also check how many sets there are within a narrower CR ($0.5 \times \chi^2_{df=1,\alpha=0.05}$), the inner rim. The edges of this cloud mark the single-parameter CIs. We need to have sufficient sets within this inner cloud, as it is used to calculate the CIs on the single parameters, and because it is used to calculate intervals on model predictions. If there are sufficient accepted sets in the joint region, but insufficient sets in the inner region, the next round will continue with mutating `coll_ok`, comprising parameter sets from the inner region.

Step 9 is a crucial one, and involves a number of criteria to select the optimal parameter sets and settings for the next round. In principle, only the new values tried in a round, that are within a certain criterion for this round, are propagated to the next round, to avoid mutating the same sets of parameters over and over in consecutive rounds. However, this set may be very large or very small, which requires some modifications of the set and/or of the number of new tries `n_tr_i` in the next round. The aim is to end up with an accepted parameter sample that is not too much more than the minimum size specified in `n_conf`, and that provides good coverage of the relevant part of parameter space, for all possible data sets.

Step 13-15 are a safety measure. For friendly data sets, they will not generally be triggered (and if they do, the refinement will not be substantial). However, there are data sets where the core sampling routine fails to sample a particular part of parameter space, or even (in very extreme cases) misses the global optimum. Profiling will indicate these issues, and the iterations between profiling and targeted sampling are effective to still obtain good coverage of the sample. Downside is increased calculation time and often a much larger sample in total. For application in ERA, robustness is more important than speed.

Optimisation logistics. The main script calls the function `calc_optim`, which handles some of the logistics. This function is called with a `pmat` matrix, setting for SD or IT, and a flag. We can call this function with an empty matrix for `pmat` and flag set to 2 to signal the function to simply read a saved sample from parameter space, and plot the parameter-space plot. It will open the project file with the name derived from the current setting of the global `GLO.basenm`. The function `calc_optim` calls `calc_parspace` to do the optimisation, and also arranges for a proper restart when `calc_parspace` indicates slow kinetics. Furthermore, `calc_optim` displays results to screen. For displaying the parameter matrices (and things calculated from them), it calls a separate function `disp_pmat`.

If `calc_optim` is used to display/plot a saved analysis, it calls `load_sample` to load the correct `.mat` file and return the relevant contents. To keep file size low, `calc_parspace` only saves the columns of `coll_all` for the parameters that were fitted (for the fixed parameters, an entire column would be filled with the same value). To simplify the code a bit, `load_sample` reconstructs `coll_all` such that the samples are complete again with the entire parameter vector.

Optimisation core function. The parameter-space exploration itself is conducted in the function `calc_parspace`. This is separated from `calc_optim` mainly because the parameter-space explorer restarts when it indicates slow kinetics (k_d running into low values and correlated with m_w), which requires m_w to be evaluated on log-scale. The parameter-space exploration goes through several rounds until the sample answers to a set of criteria. This is judged by the number of sets within the joint CR, the number of sets within the single-parameter CR (the inner rim), and the deviations between the profile likelihood and the sample. If needed, several rounds of refinement are performed, and finally, CIs on parameters calculated. So far, this procedure seems to be extremely robust. A number of other functions are called from `calc_parspace`, and from `calc_parspace` only. The reason to place this code into separate functions was because they are (or might be) called more than once, and to increase transparency of the code.

Optimisation helper functions. The function `rand_mutations` randomly mutates a number of candidate parameter sets, and calculates the associated likelihood. The function `calc_proflik` does the profiling from a provided sample, and the function `test_proflik` tests for differences between the profile likelihood and the sample. More information on profiling can be found in Section 2.3.5 and Appendix D of the e-book [5].

Plotting parameter space. For each round of the algorithm, the results will be plotted by `plot_grid`. The sample will be shown, projected in all binary combinations of dimensions (i.e., for each combination of two parameters). When the algorithm is finished, the sample will also be shown in a different way: per parameter, parameter value on the x-axis and relative likelihood on the y-axis. This is comparable to a profile-likelihood plot (the profile will be the edge of the plotting symbols). Since we refined the single-parameter CIs with profiling, these profiles will also be plotted, as red lines. These red lines should be very close to the lower edge of the sample.

Using the sample from parameter space. Functions that need a sample from parameter space for propagating errors call `load_sample`, which returns the 'outer hull'. This means: all the parameter sets that are within a certain likelihood band around the $0.5 \times \chi^2_{df=1,\alpha=0.05}$ criterion. At this moment, the thresholds are set at the 92 and 96% confidence level (so $\alpha = 0.04 - 0.08$). This gives a band of reasonable width with the most interesting parameter sets to propagate to the model predictions. These cut-offs are shown in the plots of parameter space as dotted horizontal lines.

Differences with the C++ version

- The Matlab version has an option to update the parameter-space plot throughout the optimisation. For testing and checking the algorithm, this is very helpful, but this is skipped for the C++ version (showing only the final plot is usually sufficient).
- The standalone version will not automatically save the sample from parameter space; that is only done on user request.

6 The actual GUTS model

Relevant decisions:

- Only implement reduced cases GUTS-RED-SD and GUTS-RED-IT.
- Accommodate time-varying exposure as piecewise-linear sections.
- Use analytical solutions as much as possible, and in any case for scaled damage.

6.1 Analytical solutions for damage

We only need to consider the two simplest reduced cases GUTS-RED-SD and GUTS-RED-IT. With some careful considerations, it was possible to avoid the use of ODE solvers (with all the associated troubles, especially for sudden changes in forcing). The ODE for scaled damage is a very simple first-order equation:

$$\frac{dD_w}{dt} = k_d(C_w - D_w) \quad \text{with } D_w(0) = 0 \tag{34}$$

For constant C_w, this equation can readily be solved:

$$D_w = C_w\left(1 - e^{-k_d t}\right) \tag{35}$$

However, closed-form solutions also exist for cases where C_w changes exponentially or linearly over time, including these situations where $D_w(0)$ is not zero. This opens up the possibility to divide the exposure profile into a series of episodes where the concentration remains constant or changes linearly. We can solve D_w analytically in an episode, and use the D_w at the end of the episode as the starting value for the next one. This implies that we can derive $D_w(t)$ analytically by applying closed-form solutions sequentially. For time-varying exposure, we still need to calculate the survival probability for SD with a numerical integration though. The end result is very fast, also for complicated exposure profiles such as those resulting from FOCUS calculations. Furthermore, it is not affected by sudden jumps in the exposure concentrations, and the precision and accuracy for $D_w(t)$ are high (although we still have rounding errors due to machine precision, which is an error that also propagates over the time series).

For the software, we only consider the case where the exposure concentration changes linearly over time with slope b (derived by Bob Kooi using Maple):

$$\frac{d}{dt}D_w = k_d(C_{w0} + bt - D_w) \quad \text{with } D_w(0) = D_{w0} \tag{36}$$

$$D_w = bt + C_{w0} - \frac{b}{k_d} + \exp^{-k_d t}\left(D_{w0} - C_{w0} + \frac{b}{k_d}\right) \tag{37}$$

Logistics for calculating model output. The function `calc_model` provides the framework for calculating the model output (survival and scaled damage) for a given time vector, parameter matrix, and exposure scenario (one scenario at a time). The time vector for the calculations requires some explanation. When calibrating, `calc_model` receives the time points where there are observations; when plotting, it receives a longer time vector (to make visually smooth model curves). Under constant exposure, this is sufficient as for both SD and IT analytical solutions can be applied for the survival probability. For time-varying exposure, more detail in the time vector is needed to obtain correct calculations:

- For IT, survival is linked to the highest level of scaled damage that was achieved up to that point. We thus need to make sure that we calculated this maximum, and it may occur at a time point where we don't have observations on survival. However, it *will* occur at a time point that is in our scenario definition `ev_ic` (this is only true for the reduced models where damage dynamics and TK are combined!). Therefore, we need to add the time points from the exposure scenario to the requested time vector entering `calc_model`.
- For SD, under time-varying exposure, the hazard rate is numerically integrated over time. This is an approximation that requires plenty of detail in the time vector (though it will generally still outperform the use of an ODE solver). Therefore, I add a regularly spaced time vector to the time vector entered in `calc_model`. The number of steps per day is now a global setting that is used in `calc_model` and in `calc_lpx`. Default is set at steps at every 15 minutes in `initial_setup`; this will suffice for most ecotoxicological applications (though it may be insufficient when used with time-varying concentrations that vary wildly within an hour).

In these cases, time points are added. However, after calculation of the survival probability, the added time points are removed again (only the requested time points are returned as output).

Calculating scaled damage. The function `calc_damage_linear` calculates the scaled damage. This function is only called from `calc_model`, but I keep it separate in case we, at some point, also need to implement exponential decay or an ODE solver. The function `calc_damage_linear` takes the matrix with scenario events for one treatment (one cell from the data object; for calibration from `DATAcal.scen`), runs through all the events, looks for the elements of the time vector that fall within the current event interval, and calculates the corresponding D_w analytically. The function `calc_damage_linear` returns a vector with damage levels D_w to `calc_model`.

For IT, the damage vector is treated such that damage only increases over time (to avoid dead animals to spring back to live). The damage vector is thus translated into a vector D_{wm} that is the moving maximum of D_w over time.

6.2 Survival probability

For IT, calculating survival probability from D_w is easy: we only need to make sure that $D_w(t)$ is never decreasing over time, and find the maximum damage level up to that time point ($D_{wm}(t)$). From that maximum, we can directly calculate survival probability due to chemical exposure from the log-logistic distribution:[9]

$$S_c = \frac{1}{1 + (D_{wm}/m_w)^\beta} \quad \text{with } \beta = \frac{\ln 39}{\ln F_s} \tag{38}$$

For SD, the hazard rate can be calculated from $D_w(t)$:

$$h_c = b_w \max(0, D_w - m_w) \tag{39}$$

However, turning this into a survival probability requires an integration over time:

$$S_c = \exp\left(-\int_0^t h_c(\tau) d\tau\right) \tag{40}$$

In general, I will use a numerical integration. However, when the exposure concentration is constant, an analytical integration is used. This has the benefit of increased speed (especially because the numerical integration requires a fine time scale to work accurately) but also of high precision. When the external concentration exceeds the threshold referenced to water ($C_w > m_w$), there will be an effect after some exposure time t_c. This time can then be found from:

$$t_c = -\frac{1}{k_d} \ln\left(1 - \frac{m_w}{C_w}\right) \tag{41}$$

Below the threshold, this time is infinite. The hazard rate due to chemical effects is now given by:

$$h_c = b_w \max(0, C_w \left(1 - e^{-k_d t}\right) - m_w) \tag{42}$$

For the survival probability, we need to take minus the exponent of the integrated hazard rate over time. In this case, that can be done analytically, which leads to [3]:

$$S_c = \exp\left(\frac{b_w}{k_d} C_w (e^{-k_d t_c} - e^{-k_d t}) - b_w (C_w - m_w)(t - t_c)\right) \tag{43}$$

This equation is used for the situations where there is an effect: $C_w > m_w$ and $t > t_c$, otherwise, $S_c = 1$.

For all cases, when S_c is established, it needs to be multiplied with the background survival probability. As we are usually dealing with short-term experimental data, background mortality is not due to ageing, and can usually be well represented by a constant hazard rate (representing handling effects and other random causes of death):

$$S = S_c \times \exp(-h_b t) \tag{44}$$

[9]In this document, I use a slightly different presentation than in the e-book, which requires slightly different symbols. I use S_c for the survival probability due to chemical stress, which requires multiplication with the background probability to yield the overall survival probability S. This is done to provide a closer link to how the model is implemented in the software.

Calculating survival over time. To turn the damage vector into survival probabilities, `calc_model` calls one of three functions:

- For IT, the function `calc_surv_it` is used. It is called with a vector that is the moving maximum of D_w over time (calculated in `calc_model`).
- For SD with constant exposure, the function `calc_surv_sd_c` is called, which contains the analytical solution for survival.
- For SD with time-varying exposure, the function `calc_surv_sd_t` is called, which does a numerical integration of the hazard rate over time (using the built-in Matlab function `cumtrapz`, using the trapezium rule).

For non-constant exposure, we need to make sure that the numerical integration is accurate enough. Hence, as explained in the previous section, `calc_model` increases the time vector where needed.

The calculation of survival is split up in three separate functions as these same functions will also be used for the calculation of the LCx and the LPx, in different ways (e.g, LCx calculations can be done analytically for IT, but require numerical root finding for SD). Therefore, their inputs and outputs are not the same.

7 Statistics

7.1 Calculating the likelihood for a fit

The standard likelihood function for GUTS is used, based on the multinomial distribution. This is the appropriate distribution for the data, as the data are for a single discrete (irreversible) event in the individual's life. The log-likelihood function for a parameter set θ and a data set X (with observations x_i) is (see [5] for details):

$$\ell(\theta|X) = \sum_i x_i \ln p_i \tag{45}$$

This equation uses the (unconditional) death probability for each interval (p_i) and the observed number of deaths in each interval (x_i). From this equation, it is clear that it does not matter whether we pool replicates or treat them separately ($a \times c + b \times c = (a+b) \times c$). Exception is when we know that the exposure pattern in two replicates was not exactly the same, in which situation we need to construct a slightly different exposure scenario for both replicates (and hence give them a different scenario identifier).

The unconditional death probabilities can be easily calculated from the GUTS predictions on survival probability at the start of each interval S_i, and the observed deaths follow from the observed number of survivors at the start of each interval y_i:

$$p_i = S_i(C_w, \theta) - S_{i+1}(C_w, \theta) \tag{46}$$

$$x_i = y_i - y_{i+1} \tag{47}$$

Note that the last interval to consider is from the end of the experimental test to infinity, as the probabilities over time need to sum to one. For the last interval $S_{i+1} = 0$ and $y_{i+1} = 0$. Therefore, we have one more interval than observation times in the test; in a two-day *Daphnia* test, we thus have three intervals.

The above derivation considered a single treatment (with several observations over time). However, this can easily be extended to a set of survival observations Y in multiple treatments C_{wj}; assuming that the treatments are independent (which should generally be the case), we can sum the log-likelihoods:

$$\ell(\theta|Y) = \sum_i \sum_j x_{ij} \ln p_{ij}(C_{wj}, \theta) \tag{48}$$

Calculating the likelihood. The observed numbers of deaths are already part of the data object `DATAcal`, for each data set and each treatment. The function `transfer` takes the subset of parameters that need to be fitted in `pfit`, and the entire parameter matrix `pmat`, to calculate the minus log-likelihood. This format is needed to ensure compatibility with the function that does the simplex optimisations. In `transfer`, the fitted parameters are included in the correct position in `pmat` and transferred to `calc_model` to calculate the corresponding model curves for each exposure scenario. The model function `calc_model` is called with the actual time vector for the survival data in this treatment, so the output of `calc_model` can directly be compared to the observed deaths (after translating the modelled survival probabilities into death probabilities). The log-likelihood value is calculated according to the multinomial likelihood for survival data, summed over all treatments and all data sets, and the total minus-log-likelihood is returned as output.

7.2 Goodness-of-fit measures

How can you judge whether a fit is 'good enough'? The GUTS e-book [5] discusses this issue in some detail, as it is far from trivial in the case of survival modelling. The EFSA opinion [4] provides several measures that are based on comparing observed and predicted survival probabilities or numbers of survivors. Relevant design decisions:

- Produce predicted-observed plots for survival probability and number of deaths per interval. Plotting survival probability is a better option than plotting numbers of survivors, since we do not include sampling error (but still want to include the CI on the predicted survival).
- Calculate a model efficiency (NSE, which is the same as the r^2 for regression models). I do this based on the survival probability. This criterion is calculated once for a calibration.
- Calculate a normalised root-mean-square error (NRMSE), as proposed in the EFSA opinion, based on survivor numbers. This criterion is calculated once for a calibration.
- Calculate a survival probability prediction error (SPPE), as proposed in the EFSA opinion, based on survival probability. This criterion needs to be calculated for each treatment in each data set.
- We skip the posterior-predictive check (PPC) proposed by EFSA. The PPC is a concept from Bayesian statistics, but our predicted-observed plots can be used for such a check in a qualitative manner.

The measures proposed in the EFSA opinion [4] include the normalised root-mean-square error (NRMSE), using the observed and predicted number of survivors over all time points i and treatments j, expressed as a percentage:

$$NRMSE = 100\% \times \frac{1}{\bar{y}}\sqrt{\frac{1}{n}\sum_i \sum_j (y_{ij} - y_{i0}S_{ij})^2} \tag{49}$$

Where $\bar{y}$ is the mean number of survivors. It can also be calculated using survival probabilities, which will give the same value when all treatments have the same initial number of individuals. For this metric, lower is better (a perfect fit would have a NRMSE of 0). The opinion states that the NRMSE is not expected to exceed 0.5 (or 50%). Even though it is not explicitly stated in the opinion, the results at $t = 0$ should be excluded as the observations are always in perfect correspondence with the model.

The NRMSE criterion is closely related to the model efficiency (NSE) or r^2 suggested by [5] (shown as a comparison of survival probabilities):

$$NSE = 1 - \frac{\sum_i \sum_j (y_{ij}/y_{i0} - S_{ij})^2}{\sum_i \sum_j (y_{ij}/y_{i0} - \bar{y}/y_{i0})^2} \tag{50}$$

Both metrics use the sum of squared residuals, but scale it in a different way.

Another criterion, the survival probability prediction error (SPPE), compares the observed and predicted survival probability at the end of the test, expressed as a percentage:

$$SPPE_j = 100\% \left(\frac{y_j(\text{end})}{y_j(0)} - S_j(\text{end}) \right) \tag{51}$$

This metric has a sign, and there is a value for each treatment. Again, a provisional cut-off at 50% was suggested.

A third goodness-of-fit approach that was proposed in the opinion is to make a posterior-predictive check (PPC), regularly used in a Bayesian context. In general, the PPC implies the simulation of replicated data sets under the assumption that the model is correct, and using the posterior distribution of the parameters. Subsequently, these simulated data are compared to the real observations to see if they are consistent. In the EFSA opinion, the simulated results are survivor numbers (including sampling error). Since we do not include sampling error, we cannot do the comparison in the same way. Therefore, we have to skip this measure for the software.

It is important to note that all these measures are problematic for discrete data, and even more so for survival data. Therefore, they have to be used in a qualitative manner.

Calculating goodness-of-fit measures. These goodness-of-fit measures are calculated in the main plotting function `plot_main`. In that function, model curves are calculated, and the observed and predicted values are plotted. That is exactly what we need for these goodness-of-fit measures, so this is the logical place to do it. While producing the main plots for all data sets and all treatments, the function collects the predicted values for survival probability so that they can later be used for predicted-observed plots and the various metrics.

8 Plotting for calibration, validation and prediction

Relevant design decisions:

- The main plot will have three rows of panels: top row exposure scenario vs. time, middle row damage vs. time, bottom row survival probability versus time.
- The same plot will be made for the three stages in the workflow that involve data: calibration, validation and prediction. It will also be used to make inspection plots of the data (to check the data set entered/loaded before calculations). The inspection plots will lack the middle row for damage (see examples in Section 3.3).
- For predictions, we don't show CIs on the model fits by default (this can be done as an option in the Matlab version, but is very time-consuming for FOCUS profiles, and likely not so interesting). However, CIs on the LPx are calculated.
- Error bars will be plotted on observed survival frequencies, which is the Wilson score interval (basically the CI of the predicted proportion).
- For calibration and validation, produce predicted-observed plots for survival probability and number of deaths per interval. Plotting survival probability is a better option than plotting numbers of survivors, since we do not include sampling error (but still want to include the CI on the predicted survival).

GUTS results are generally plotted as observed and predicted survival probability over time. Generally, all treatments are plotted in the same graph. This is less useful for time-varying exposures (where lines may cross) and when plotting CIs on the model curves as well (which easily becomes a mess). Therefore, we use a multi-panel plot.

The Wilson score interval is a nice way to express the uncertainty in the data. It is the CI on the binomial proportion: observed survivors over initial number of animals ($\hat{p}$). Disadvantage is that it looks at each observation in isolation (rather than as a survival series over time). However, this metric is often used for the multinomial distribution as well as the alternatives perform poorly for small data sets. The Wilson interval is calculated as:[10]

$$\frac{\hat{p}+\frac{z^2}{2n}}{1+\frac{z^2}{n}} \pm \frac{z}{1+\frac{z^2}{n}}\sqrt{\frac{\hat{p}(1-\hat{p})}{n}+\frac{z^2}{4n^2}} \tag{52}$$

Where z is the critical value from the standard normal distribution (1.96) and n is the initial number of individuals in that treatment. The nice thing about this interval is that it becomes tighter with increasing number of individuals tested. For small data sets (not uncommon for vertebrates), it becomes clear that the data are also uncertain and that a misfit is thus not necessarily a sign that the model is performing poorly.

[10]See https://en.wikipedia.org/wiki/Binomial_proportion_confidence_interval.

Plotting the main plot. The Wilson interval is already part of the data object (both when calibrating and when validating). The plotting routine is `plot_main`, which calls `calc_conf` with a long time vector and an exposure scenario (one treatment at a time). The `calc_conf` calls `load_sample` to load the saved information from the project file. This contains the optimised parameter set in `pmat` and a relevant part of the sample from parameter space (the sets that need to be propagated to model predictions). Next, `calc_conf` calls `calc_model` to calculate damage level and survival probability for the time vector and the scenario. It does that for the best set (in `pmat`) and also for all elements of the sample (the sub-set that needs to be propagated). For the sample, only the minimum and maximum model output at each time point is stored.

Note that the length of the time vector for plotting depends on the data set. A default setting is included in `initial_setup` as the global `GLO.t_plot`. This is a vector with the minimum number of points (100) and the maximum number of points per day (24, which implies one point every hour). It turns out that 100 points is not enough to capture damage dynamics for peaky FOCUS profiles and fast kinetics (peaks in damage were not shown, though the calculations were correct). Plotting on an hourly basis seems to suffice.

The `plot_main` function calls `make_fig` to create an empty figure window with a default size based on the number of panels to be plotted.

8.1 Validation: plotting predictions for new scenarios

The calibrated model can be used to make predictions for other experimental tests, with other exposure scenarios. This is exactly the same thing as for calibration, apart from the fact that we don't fit the parameters again but use the ones from the project file. The model will thus go through the same steps: selecting a data set, loading it, preparing it into a data object, and plotting it (calculating the model output with CIs). Relevant design decisions:

- Use the same format for the data set and same plotting routine as for calibration. This implies that we can always switch a calibration and validation data set, or calibrate on both data sets simultaneously.
- Validation always has data, which will be prepared in exactly the same way as the calibration data.
- Allow the background hazard rate h_b to be fixed on the controls of the validation data set.
- Allow multiple data sets to be entered for validation (Matlab version only).

Performing a validation. Validation applies the same routines as for making a calibration plot, using `plot_main`, `load_sample` and `calc_conf`. The only difference is that the data object is not made global (as it is only passed on to the `plot_main`, and does not need to go through `transfer`).

Differences with the C++ version. The standalone versions allows only one data set for validation.

9 Post analysis: LCx and LPx

After calibration, the parameterised model (and the parameter-space sample) can be used to make predictions (with CIs). Relevant design decisions:

- Only calculate specific values for x in LC$x, t$: 10, 20 and 50% (can be modified in `initial_setup`).
- Only calculate specific values for t in LC$x, t$: 1, 2, 3, 4, 7, 14, 21, 28, 42, 50, 100 days (can be modified in `initial_setup`).
- Only plot LC10 and LC50 versus time (can be modified in `initial_setup`). Only plot time points that are calculated, so no smoothing. User can select final time of the plot.
- LPx depends on data (exposure profile). Each data set can contain one exposure profile only (two columns: time and concentration). An inspection plot can be made (standard plot, but exposure profile only as there is no survival data).
- Only calculate specific values for x in LPx: 10 and 50% (this can also be modified in `initial_setup`).
- Create a plot for survival (at the end of the exposure profile) versus multiplication factor (MF), with CI (by default). At this moment, a plot is created that runs from 99.9% survival to 5% (with the idea that low effects are probably more relevant; this can be modified in `initial_setup`). The plot is made on log-scale for MF.
- For regular calculations, LPx is calculated with a CI by default (this can be time consuming, so calculation without CI is an option). Furthermore, a standard plot is made with exposure, damage and survival at the LPx (two columns: one for LP10 and one for LP50). These plots are made without CIs on model curves (the Matlab version allows calculations with CIs as an option).
- Batch calculations can be made for LPx, automatically running through a number of text files with profiles. Resulting LPx values are ranked (based on LP50). Plots are made with the exposure profile and survival versus time (no CIs, though the Matlab version allows CI calculation as an option, including the possibility to use a sub-sample for speed). Graphical output to file only, and using the standard plotting format. A table with sorted LPx values across all exposure profiles is printed on screen.

9.1 Calculation of LCx

The first prediction is for the LC$x, t$: the concentration that, when applied as constant exposure, is expected to lead to x% mortality after constant exposure for a duration t. The LC$x, t$ does not rely on data; it follows from a set of parameters (and the sample to propagate the errors to a CI).

For IT, it is straightforward to calculate the concentration at which there is $x\%$ effect after exposure duration t. The survival probability is determined by the maximum level of scaled damage over time (D_{wm}):[11]

$$S = \frac{1}{1 + (D_{wm}/m_w)^{\beta}} \tag{53}$$

Note that since we set background mortality to zero for LCx calculations, that $S = S_c$.

The maximum damage level D_{wm} can be calculated under constant exposure as:

$$D_{wm} = C_w \left(1 - e^{-k_d t}\right) \tag{54}$$

We can fill in the equation for D_{wm} in the equation for S. The LC$x, t$ is now the C_w where $S = 1 - x/100$. Rearranging leads to the final equation:

$$LC_{x,t} = \frac{m_w}{1 - e^{-k_d t}} \left(\frac{x/100}{1 - x/100}\right)^{1/\beta} \tag{55}$$

For SD, we cannot obtain a simple expression for LC$x, t$. However, it is possible to use the analytical solution (in `calc_surv_sd_c`) for S since we deal (per definition) with the situation of constant exposure. We still need root finding to find the C_w where $S = 1 - x/100$.

Calculating the LC$x, t$. The function `calc_lcx` does the calculation for LC$x, t$. It calls `load_sample` to retrieve the parameter matrix and the sample from the saved project file. For IT, the function will use an analytical calculation of LC$x, t$. For SD, it applies root finding (either the built-in function `fzero` or the alternative `zero` in the `engine`) to try a range of constant exposure concentrations to see if they yield $x\%$ effect after time t. Note that background hazard h_b is ignored by this function: the LC$x, t$ is defined as the concentration that yields $x\%$ mortality due to chemcial stress alone (and thus relative to the control).

For SD, zero finding is needed. The function `calc_lcx` calls the function `calc_surv_sd_c` (that is also used for calibration and plotting) for that purpose. That function is set up to return a criterion that will be zero when a certain effect percentage is reached. This allows to use root finding from `calc_lcx`. To narrow the range that the root-finding function needs to search (increasing speed), the LC$x, t$ values for SD are calculated in sequence: they will always decrease in time, and decrease with decreasing x, and they will always be higher than (or equal to) m_w. A similar strategy is followed for each element in the sample. At each time point and effect level, only the minimum and maximum will be collected (which serve as the CIs).

After the calculations, a plot is produced for LCx versus time, with CIs on the curve.

[11] The possibility to calculate LC$x, t$ analytically, for the reduced GUTS models, was brought to my attention by the paper of Baudrot and Charles [2].

9.2 Calculation of LPx

The software also calculates LPx values as defined in the EFSA opinion [4]: the factor by which an entire exposure concentration profile (e.g., FOCUS output) needs to be multiplied to yield $x\%$ effect at the end of the profile [1].

For IT, we can use an analytical solution for the LPx. The reason is that the damage level is proportional to the LPx: increasing the entire profile by a factor of ten leads to an increase in D_w by a factor of ten. Survival probability is only determined by the maximum level of damage over the exposure profile D_{wm}. Therefore, in an analogous manner to the derivation for LC$x, t$, we can obtain the equation:

$$LP_x = \frac{m_w}{D_{wm}} \left(\frac{x/100}{1 - x/100} \right)^{1/\beta} \tag{56}$$

We still have to calculate D_{wm} from the profile for each parameter set that we evaluate, which requires substantial calculation time (sequential application of one-compartment kinetics over a large amount of short, linear, exposure episodes). For SD, no analytical solutions can be used as exposure is time-varying, D_w is time-varying, and we need to integrate the hazard rate over time. The vector for D_w is calculated once for each parameter set, multiplied with a candidate LPx, h_z is calculated from that in `calc_surv_sd_t` and numerically integrated. Note that I multiply D_w by a factor rather than the exposure profile. This is allowed as the ODE for D_w is linear in C_w, and saves a lot of time as we don't need to calculate new D_w values across a profile for each multiplication factor that is tried. It is important to note for modifications to the software that this shortcut will fail for more complex (non-linear) TK/damage modules.

Calculating LPx. The function `calc_lpx` does the calculation for LPx. This calculation requires a specific exposure scenario, defined as a linear forcing series (time in days in column 1, and exposure concentration in column 2, which will be linearly 'interpolated'). These data are also prepared by `prepare_data` into the same format as calibration and validation data sets (see Section 3). The LPx calculation never contains survival data, but dummy survival data are added in `prepare_data` such that the same routine can be used.

The exposure scenario is turned into a three-column scenario, adding the slope in each interval, and removing intervals where the slope stays the same, to increase speed. As with the LCx, the background hazard h_b is ignored in this function: the LPx is defined as the the factor that yields $x\%$ mortality due to chemical stress alone (and thus relative to the control).

The function `calc_lpx` will calculate LP10 and LP50 values for one exposure scenario at a time (although it is prepared to work with multiple scenarios, as separate data sets). Furthermore, we need to calculate a range of multiplication factors and corresponding survival probabilities (at the end of the profile). This will make a sort of dose-response plot.

The LPx calculation has a lot of similarities with the LCx calculation: an analytical solution can be used for IT, and SD requires root finding to find LPx values. However, we are always dealing with time-varying exposure, which complicates matters, and always requires running through `calc_model`. Fortunately, we can make use of a short-cut: if we multiply an entire profile by a factor of x, we also multiply all scaled damage levels D_w by x. Therefore, we only need to calculate D_w levels once for each parameter set, for a multiplication factor of 1. From this vector, the LPx can be calculated directly (IT in `calc_surv_it`) or found with root finding (SD using `calc_surv_sd_t`). For SD, we can try a new LPx value by multiplying D_w with a multiplication factor to try, calculate the corresponding survival probability, and see if it matches $1 - x/100$.

For SD, it is efficient to first find the rough location of the multiplication factors (MF) that will give us the range of survival probabilities needed for the survival-versus-MF plot (by default 0.05-0.999). An algorithm will increase and/or decrease the MF by a factor of 10 until we know the range in which the survival probabilities are. This allows for more efficient application of the root-finding function (it works much faster if it has a limited range to search).

The same procedures are followed when calculating confidence intervals on the LPx and multiplication factors. For each parameter set from the sample, a D_w is obtained. This is used to calculate survival probabilities for each MF in the survival-MF plot; only the highest and lowest are collected for making confidence intervals. Furthermore, the LP10 and LP50 are calculated, and the highest and lowest collected.

Results are printed on screen and a plot of survival-MF is made in this function. After that, the LPx values are returned to the main script, and the main plotting function (`plot_main`) is called to plot the scenarios, damage over time, and survival over time for LP10 and LP50. Note that LPx will be calculated with CIs, but no CIs will be calculated (by default) on the plots for damage and survival over time.

9.3 Batch estimation of LPx

This facility is included to allow for a rapid screening of large numbers of exposure profiles in an automated manner.

Batch calculation. The function `calc_batch_lpx` does the batch calculation for LPx. It is called with a series of file names (for the exposure scenarios) from the Matlab file-open GUI. It runs through the files, loads them, prepares a standard data structure for them, and calculates LPx. It uses the standard functions for these purposes that are also used for other parts of the software (`load_data`, `prepare_data`, `calc_lpx`). For the plotting, it also uses the standard routine (`plot_main`), with a different flag that tells `make_fig` to create an invisible figure window (it will be saved but not shown on screen). A final table with the sorted scenarios will go to screen (and to an output report as well). Note that both the plots and the table with LPx values will not show CIs; this is done to increase calculation speed. The Matlab version has (with version 0.8) the possibility to batch-calculate LPx with a CI, and to use a sub-sample from parameter space to speed up the calculations (at the expense of underestimating the CIs somewhat).

References

[1] R. Ashauer, P. Thorbek, J. S. Warinton, J. R. Wheeler, and S. Maund. A method to predict and understand fish survival under dynamic chemical stress using standard ecotoxicity data. *Environmental Toxicology and Chemistry*, 32(4):954–965, 2013.

[2] V. Baudrot and S. Charles. Recommendations to address uncertainties in environmental risk assessment using toxicokinetics-toxicodynamics models. *bioRxiv*, page 356469, 2018.

[3] J. J. M. Bedaux and S. A. L. M. Kooijman. Statistical analysis of bioassays based on hazard modelling. *Environmental and Ecological Statistics*, 1:303–314, 1994.

[4] EFSA. Scientific opinion on the state of the art of toxicokinetic/toxicodynamic (TKTD) effect models for regulatory risk assessment of pesticides for aquatic organisms. *EFSA journal*, 16(8):5377, 2018.

[5] T. Jager and R. Ashauer. *Modelling survival under chemical stress. A comprehensive guide to the GUTS framework*. Toxicodynamics Ltd., York, UK. Available from Leanpub, `https://leanpub.com/guts_book`, Version 2.0, 8 December 2018, 2018.