# mrgsolve user guide

Kyle Baron

June 27, 2024

# Table of contents

# Introduction

Welcome to the mrgsolve user guide. The user guide is the main documentation source for how mrgsolve works and how to best use mrgsolve in your modeling and simulation project. As with most of the mrgsolve documentation, this is a work in progress. I am currently working to transition this to more of a reference resource, rather than demonstration. So key content in the user guide includes Chapter 2 on model specification, Chapter 1 on model components and Chapter 7 on the simulation sequence. Installation is a big topic but we defer to the wiki page for installation help since requirements tend to change frequently with new R releases. The other content is hopefully helpful as well. I'm leaving it all in place for now, but will gradually transition the "how-to" and demo type content over to the vignettes repository or the gallery repository (see below).

Please feel free to ask questions about anything mrgsolve-related on the issue tracker on the main github repo: https://github.com/metrumresearchgroup/mrgsolve/issues.

## Overview

If you are accessing the user guide via html, you should be able to see a table of contents on the left. That has a lot of detail in there so I'm going to give you high-level overview of what is here (the high-level should be pretty clear in the pdfbook).

- Chapter 1 tells you about what is in the mrgsolve model object (like parameters, compartments, C++ functions and the like)
- Chapter 2 tells you about model specification. There are two main sections to this chapter: Section 2.2) lists the different model code blocks and Section 2.3) lists the different variables that you can get (or set) in the different code blocks
- Chapter 3 shows you how to format input data sets
- Chapter 4 shows you how to create and work with simple data set objects called `event` objects
- Chapter 5 shows how to work with model matrices
- Chapter 6 tells you about the simulated output object and how you can work with it
- Chapter 7 dives into the simulation sequence, showing you the steps that mrgsolve takes to work through a problem
- Chapter 8 gives the details on how mrgsolve advances the system to a pharmacokinetic steady state

- Chapter 9 talks about different plugins you can use for more advanced modeling
- Chapter 10 talks about modeled events (discontinuities in the simulation that are triggered from within the model rather than from the data set)
- Chapter 11 is a collection of applied examples on different topics
- Chapter 12 are brief questions and answers providing some bits of information that don't obviously fit into another chapter
- Chapter 13 provides links to some Wiki pages that can help with installation

## PDF Version

This book is also available in pdf format here.

## Other Resources

- Main resource page: mrgsolve.github.io
- Vignettes: mrgsolve.github.io/vignettes
- R documentation: mrgsolve.github.io/docs
- Gallery: github.com/mrgsolve/gallery

This book was assembled on Thu Jun 27 15:09:28 2024 with mrgsolve version 1.4.2.

# 1 Model components

This chapter details the different components of a model in `mrgsolve`. Each component listed here is maintained within a "model object". This is an updatable S4 object in `R` that contains all of the basic information required to properly configure and simulate from the model.

## 1.1 Parameter list

The parameter list is an updatable set of name-value pairs. Referencing the name of an item in the parameter list will substitute the current value associated with that name. While the name "parameter" may have a certain connotation in the modeling world, in `mrgsolve` a "parameter" could be any category of numeric data: covariates (e.g. `WT`, `AGE`, `SEX`), flags, other numeric data that we commonly call "parameter" (e.g. `CL` or `VC`).

The parameter list is declared in the code block `$PARAM`. While there may be multiple `$PARAM` blocks in a model, these are condensed to a single parameter list stored in the model object. The names and numbers of all parameters in the model must be declared at the time that the model is compiled. Also, a default value for each parameter must be declared at model compile time, but the value of each parameter may be updated in one of several ways.

The parameters in a model object can be queried or updated with the `param()` function.

See also: Section 2.2.4, `?param` in the `R` help system after loading `mrgsolve`.

### 1.1.1 Central role of parameters in planning simulations

The data items in the parameter list are more than just values associated with a name. When an name is added to the parameter list, that name becomes a key word that `mrgsolve` will start to recognize in input data sets or when manipulating the model object.

For example, when you want to include a covariate in the model, say weight (`WT`), you'll include a column in the data set called `WT` that will indicate the weight of this or that patient. It is crucial that you also list `WT` in `$PARAM` with some default value. It helps if that value is sensible too. When `mrgsolve` receives the data set prior to simulating, the `WT` column is matched up with the `WT` parameter name. As `mrgsolve` works its way through the input data set (from person to person or from time to time), the value of `WT` is updated so that the symbol `WT` in `$MAIN` or `$ODE` or `$TABLE` always points to the value of `WT`. If the `WT` name is not in the

parameter list, it won't matter if it is in the data set or not. Only listing a name in `$PARAM` gets it "into the game".

Understanding the parameter update mechanism is very important for planning complicated simulations with `mrgsolve`. Please see the information in Section 3.1 and in Section 11.3.

## 1.2 Compartment list

Like the parameter list, the compartment list is a series of name-value pairs. The compartment list defines the number, names, and initial values of each compartment in the model. The names, numbers, and order of the compartment in a model is established at the time of model compile and changes to the compartment list require re-compilation of the model.

Compartments are declared in one of two code blocks: `$INIT` and `$CMT`. Nominal initial values must be supplied for each compartment. The main difference between `$INIT` and `$CMT` is that `$CMT` assumes a default initial value of 0 for each compartment; thus only compartment names are entered. When using `$INIT`, both names and values must be explicitly stated for each compartment.

The initial values for each compartment can be queried with the `init()` function. There are several different ways to set the initial conditions in a model; Section 11.2 illustrates several of these.

See also: Section 11.2 and `?init` in the `R` help system after loading `mrgsolve`.

## 1.3 Simulation time grid

The `mrgsolve` model object stores the parameters for the series of time points to be output for a a simulation. This is the default output time grid that will be used if not over-ridden by another mechanism.

The elements of the simulation time grid are: `start`, `end`, `delta` and `add`. `start`, `end`, `delta` are passed to `seq()` as `from`, `to`, and `by`, respectively. `add` is any arbitrary vector of additional times to simulate.

The simulation time grid in a model object may be queried with the `stime()` function or by printing the model object to the `R` console.

See also Section 3.2 for discussion of the simulation time grid and input data sets and Section 1.3.1 and Section 11.4 for using time grid objects .

### 1.3.1 `tgrid` objects

A `tgrid` object has `start`, `end`, `delta` and `add` attributes. This object is independent of the model object. `tgrid` objects may be created and combined to create complex sampling designs.

See Section 11.4 for examples and usage.

## 1.4 Solver settings

`mrgsolve` uses the `DLSODA` solver like the one from `ODEPACK`. Several of the settings for that solver are stored in the model object and passed to the solver when the problem is started. Settings include: `atol`, `rtol`, `maxsteps`, `hmax`, `hmin`, `ixpr`, `mxhnil`.

Solver settings can be changed through the `update()` method for your mrgsolve model object. For example, to change `rtol` to `1e-5`, write

```
mod <- update(mod, rtol = 1e-5)
```

where `mod` is your mrgsolve model object.

### 1.4.1 `atol`

Absolute tolerance parameter. Adjust this value lower when you see state variables (compartments) that are becoming very small and possibly turning negative. For example:

```
mod <- modlib("viral1", end = 144)

out <- mrgsim_e(mod, ev(amt = 1000)) %>% filter(V < 0)

out
```

```
. # A tibble: 12 x 8
.       ID  time expos        T         I         V  logV logChange
.    <dbl> <dbl> <dbl>    <dbl>     <dbl>     <dbl> <dbl>     <dbl>
.  1     1  95    1000 5187630. -1.79e-11 -1.16e-13   NaN       NaN
.  2     1  97.5  1000 5298189. -1.92e-10 -1.26e-12   NaN       NaN
.  3     1 100    1000 5407830. -2.44e-10 -1.60e-12   NaN       NaN
.  4     1 102.   1000 5516561. -2.75e-10 -1.80e-12   NaN       NaN
.  5     1 105    1000 5624390. -1.78e-10 -1.17e-12   NaN       NaN
.  6     1 108.   1000 5731324. -8.56e-11 -5.60e-13   NaN       NaN
```

```
.  7     1 110    1000 5837370. -1.24e-11 -8.12e-14   NaN       NaN
.  8     1 122.   1000 6354547. -1.67e-11 -1.10e-13   NaN       NaN
.  9     1 125    1000 6455421. -1.91e-11 -1.25e-13   NaN       NaN
. 10     1 128.   1000 6555459. -2.11e-11 -1.38e-13   NaN       NaN
. 11     1 130    1000 6654666. -1.23e-11 -8.07e-14   NaN       NaN
. 12     1 132.   1000 6753050. -2.75e-12 -1.82e-14   NaN       NaN
```

Adjusting `atol` to `1E-20` or `1E-30` will prevent this.

```
mrgsim_e(mod, ev(amt = 1000), atol = 1E-20)  %>% filter(time %in% out$time)
```

```
. # A tibble: 12 x 8
.       ID  time expos        T        I        V  logV logChange
.    <dbl> <dbl> <dbl>    <dbl>    <dbl>    <dbl> <dbl>     <dbl>
.  1     1  95    1000 5187630. 1.33e-11 8.72e-14 -13.1     -18.7
.  2     1  97.5  1000 5298189. 5.13e-12 3.36e-14 -13.5     -19.2
.  3     1 100    1000 5407830. 1.98e-12 1.30e-14 -13.9     -19.6
.  4     1 102.   1000 5516561. 7.66e-13 5.01e-15 -14.3     -20.0
.  5     1 105    1000 5624390. 2.96e-13 1.94e-15 -14.7     -20.4
.  6     1 108.   1000 5731324. 1.15e-13 7.51e-16 -15.1     -20.8
.  7     1 110    1000 5837370. 4.44e-14 2.91e-16 -15.5     -21.2
.  8     1 122.   1000 6354547. 3.94e-16 2.58e-18 -17.6     -23.3
.  9     1 125    1000 6455422. 1.54e-16 1.01e-18 -18.0     -23.7
. 10     1 128.   1000 6555459. 5.99e-17 3.92e-19 -18.4     -24.1
. 11     1 130    1000 6654666. 2.34e-17 1.53e-19 -18.8     -24.5
. 12     1 132.   1000 6753050. 9.14e-18 5.98e-20 -19.2     -24.9
```

### 1.4.2 `rtol`

Relative tolerance parameter. Adjust this value lower when you want more precision around the calculation of state variables as the system advances.

### 1.4.3 `maxsteps`

This is the maximum number of steps the solver will take when advancing from one time to the next. If the solver can't make it in `maxsteps` it will stop and give an error message like this:

```
DLSODA-  At current T (=R1), MXSTEP (=I1) steps
      taken on this call before reaching TOUT
In above message, I =
[1] 2000
In above message, R =
[1] 0.0004049985
DLSODA-  ISTATE (=I1) illegal.
In above message, I =
[1] -1
DLSODA-  Run aborted.. apparent infinite loop.
Error in (function (x, data, idata = null_idata, carry.out = character(0),  :
  error from XERRWD
```

You might see this when you have to integrate along time between records in a data set. There isn't necessarily a problem, but the solver might have to advance over many doses to get to the next record and it only has a limited number of steps it can take between those records before it stops with this error.

When you see this, increase `maxsteps` to 50000 or larger.

But keep in mind that sometimes the solver can't make it to the next record because there are issues with the model. It might take thousands of steps to make it 24 hours down the road. In that case, go back to the model code and look for problems in how it is coded.

### 1.4.4 `hmax`

The **maximum** step size. By default, the solver will take steps of different sizes based on what is happening in the simulation. Setting `hmax` tells the solver not to take a step larger than that value. So in a model where `time` is in hours, reducing `hmax` to `0.1` will prevent the solver from taking a step larger than `0.1` hours as it tries to advance to the next time. The will slow down the simulation a bit. But sometimes helpful when the solver starts taking large steps. We don't recommend using this routinely; for most applications, it should be reserved for troubleshooting situations. If your model doesn't give the results that you want without setting `hmax`, we'd recommend a new setup where this isn't needed.

### 1.4.5 `hmin`

The **minimum** step size. Only set this if you know what you're doing.

### 1.4.6 `ixpr`

A flag to enable printing messages to the R console when the solver switches between non-stiff and stiff solving modes. Rarely used.

### 1.4.7 `mxhnil`

The maximum number of messages printed when the model is solving. If you have a lot of messages, keep working on your model code.

## 1.5 Functions

There are four `C++` functions that `mrgsolve` creates and manages: `PREAMBLE`, `MAIN`, `ODE`, `TABLE`. Each function is created from an entire code block in the model specification file. The user is responsible for writing correct `C++` code in each of these blocks. mrgsolve will parse these blocks and augment this code with the necessary elements to create the `C++` function.

These functions may be specified in any order in the model specification file, but there is a **specific calling order** for these functions. Recognizing and understanding this calling order will help understand how the different pieces of the model specification fit together.

Just prior to starting the problem, mrgsolve calls `$PREAMBLE`. Then, during advance from time `T1` to `T2`, first `$MAIN` is called, then `$ODE` is called repeatedly as the solver finds the values of state variables at `T2`, and, once the solution is found, `$TABLE` is called to calculate derived quantities at `T2` and to specify variables that should be included in the model output. So, it is helpful to write model specification files in the order:

1. `$PREAMBLE` - called **only once** just prior to processing the first record of the data set
2. `$MAIN` - **before** advancing the system
3. `$ODE` - the system **advances** to `T2`
4. `$TABLE` - **after** advancing the system

But the order in which they are coded will not affect model compilation or the simulation result.

### 1.5.1 The `$PREAMBLE` function

The `PREAMBLE` function gets called only once, just prior to processing the first record of the data set. This function is composed of `C++` code and is used to initialize variables and get them set up prior to starting on the problem.

See Section 2.2.14 for details.

### 1.5.2 The `$MAIN` function

The `MAIN` function gets called at least once before the the solver advances from the current time (`T1`) to the next time (`T2`). In the `MAIN` function, the user may:

- Set initial conditions for any compartment
- Derive new variables to be used in the model
- Write covariate models
- Add between-subject variability to quantities to structural model parameters (e.g. `CL` or `VC`).

In addition to getting called once per record, the `MAIN` function may be called several times prior to starting the simulation run. The `MAIN` function is also called whenever the user queries the compartment list.

mrgsolve allows you access compartment amounts in the `MAIN` function. But it is important to remember the calling order of these model functions (Section 1.5): because `MAIN` is called *before* the system advances, the compartment amounts inside this function will reflect the pre-advance values. This is in contrast to accessing compartment amounts inside the `TABLE` function, which will reflect the values *after* the system advances. While there are some use cases where it is useful to check the pre-advance compartment amounts in `MAIN`, most applications should interact with compartment amounts after the system advances in `TABLE`.

The `MAIN` block contains code that is equivalent to the code you'd write in NONMEM's `$PK` block. You can use either `$MAIN` or `$PK` in your mrgsolve model.

See Section 2.2.8 and Section 2.2.9 for details.

### 1.5.3 The `$ODE` function

The `ODE` function is where the user writes the model differential equations. Any derived quantity that depends on a state variable and is used to advanced the system must be calculated inside `$ODE`. But, this function is called repeatedly during the simulation run, so any calculation that **can** be moved out of `$ODE` (for example: to `$MAIN`) should be.

The `ODE` block contains code that is equivalent to the code you'd write in NONMEM's `$DES` block. You can use either `$ODE` or `$DES` in your mrgsolve model.

See Section 2.2.10 and Section 2.2.11 for details.

### 1.5.4 The `$TABLE` function

The `TABLE` function is called **after** the solver advances in time. The purpose of `TABLE` is to allow the user to interact with the values of the state variables after advancing, potentially derive new variables, and to insert different outputs into the table of simulated results.

The `TABLE` block contains code that is equivalent to the code you'd write in NONMEM's `$ERROR` block. You can use either `$TABLE` or `$ERROR` in your mrgsolve model.

See Section 2.2.12 and Section 2.2.13 for details.

## 1.6 Random effect variances

The `mrgsolve` model object keeps track of a arbitrary number of block matrices that are used to simulate variates from multivariate normal distributions. Users can specify `OMEGA` matrices for simulating between-subject random effects (one draw per individual) or `SIGMA` matrices for simulating within-subject random effects (one draw per observation).

See Chapter 5 for complete details on how to work with `OMEGA` and `SIGMA` in your mrgsolve model.

### 1.6.1 `OMEGA`

The matrices are specified in `$OMEGA` blocks in the model specification file.

`OMEGA` may be queried or updated with the `omat()` function.

### 1.6.2 `SIGMA`

The matrices are specified in `$SIGMA` blocks in the model specification file.

`SIGMA` may be queried or updated by the `smat()` function.

# 2 Model specification

This chapter details the mrgsolve model specification format.

## 2.1 How / where to write a model

There are two ways to write your model:

1. Code in a **separate file** and source into your R script
2. Code **inline** as a character string already in your R script

We recommend method `1` (separate file) for any non-trivial modeling work. Method `2` is handy for quickly coding a model and you'll also see us using that approach frequently when demonstrating how to use mrgsolve.

### 2.1.1 Separate file

For most applications, you will want to put your model code in a standalone file. This file can have any extension, but there is some special behavior when you use the `.cpp` extension. We'll show you both here.

#### 2.1.1.1 Using `.cpp` file extension

Open a text editor and type the model code into a file with name that has the format `<model-name>.cpp`. This filename format identifies a "name" for your model (`<model-name>`, the "stem" of the file name). Note: this whole file will be read and parsed, so everything in it must be valid mrgsolve model specification elements.

Use the `mread()` function to read and parse this file. For the model called `mymodel` saved in `mymodel.cpp` (in the current working directory), issue the command:

```
mod <- mread("mymodel")
```

`mread()` returns a model object from which you can simulate.

#### 2.1.1.2 Using any file extension

You can use any file extension to save your mrgsolve model. Rather than using `.cpp`, you might use `.mod`. When you are using an extension other than `.cpp`, pass in the entire file name, both stem and extension. If you have the model code saved in `mymodel.mod` then you'd call

```
mod <- mread("mymodel.mod")
```

to load that model.

#### 2.1.1.3 The model project directory

The second argument to `mread()` is `project`. This is the directory where mrgsolve will look for your model file (passed as the first argument to `mread()`).

If your `myproject.mod` file is located in the `models` directory, then you could load that file with

```
mod <- mread("mymodel.mod", project = "models")
```

Alternatively, you can just past the entire path to the model file

```
mod <- mread("models/mymodel.mod")
```

### 2.1.2 Inline / `code`

Often it is more convenient to write a model right in your `R` script. The model might look something like this:

```
code <- '
$PARAM CL = 1, VC = 20
$PKMODEL ncmt=1
'
```

Here, we created a character vector of length 1 and saved it to the `R` object called `code`. The name of this object is irrelevant. But `code` will be passed into mrgsolve as the model definition. When mrgsolve gets a model like this along with a "name" for the model, mrgsolve will write the code to a file called `<model-name>.cpp` and read it right back in as if you had typed the code into this file (Section 2.1.1).

To parse and load this model, use the `mcode()` command:

```
mod <- mcode("mymodel", code)
```

`mcode()` is a convenience wrapper for `mread()`. `mcode` writes the code to `mymodel.cpp` in `tempdir()`, reads it back in, compiles and loads.

The `mcode` call is equivalent to:

```
mod <- mread("mymodel", tempdir(), code)
```

For help, see `?mread` , `?mcode` in the `R` help system after loading mrgsolve.

### 2.1.3 Comments

You can comment out code in the model specification file with two front slashes `//`

```
$MAIN

double a = 1.234;
// double b = 5.678;
```

This comment sequence is what is used in `C++` code, but you can use it *anywhere* in the mrgsolve model specification file.

## 2.2 Code blocks

### 2.2.1 About code blocks

#### Block identifier

Different types of code are organized in the model specification file and separated by block identifiers. There are two ways to formulate block identifiers that can be used in mrgsolve. In the first type, a dollar-sign is placed at the start of the block name

```
$BLOCKNAME
<block-code>
```

For example, a block of parameters would be

```
$PARAM
CL = 1
```

The second way to write this is with brackets

```
[ BLOCKNAME ]
<block-code>
```

There is no functional difference between the dollar-sign notation and the brackets. When model specification code is saved into a file with a `.cpp` extension, the code editor may make certain assumptions about formatting or styling the code. Using brackets will most-likely work better with the editor in that case.

Block identifiers are case-insensitive so all of these also work

```
$param
CL = 1
```

```
[ param ]
CL = 1
```

Users are free to include block code on the same line as the block identifier, but must include a space after the identifier. For example, the parser will recognize `$PARAM CL = 1` but not `$PARAMCL=1` as parameters.

**Block syntax** Different blocks may require different syntax. For example, code written in `$PARAM` will be parsed by the `R` parser and will generally need to adhere to `R` syntax requirements. On the other hand, code in `$MAIN`, `$ODE`, and `$TABLE` will be used to make functions in `C++` and therefore will need to be valid `C++` code, including terminal `;` on each line.

**Block options** Options may be specified on some code blocks that signal how the code is to be parsed or used in the simulation.

Block options can be boolean in which case you indicate the option name with nothing following. For example, to indicate `block = TRUE` for an OMEGA matrix, write

```
$OMEGA @block
1 2 3
```

Here `@block` indicates `block = TRUE`. Starting with mrgsolve 1.0.0, block options can be negated by writing `!` between `@` and the option name. To request a non-block (diagonal) OMEGA matrix

```
$OMEGA @!block
1 2 3
```

But note well that diagonal is the default configuration for OMEGA, so it is never necessary to write this.

Other non-boolean block options state the name and then a value to be assigned to that name. For example, if we wanted to assign labels to the ETAs from an OMEGA block, we'd write

```
$OMEGA @labels ECL EVC
1 2
```

This essentially sets labels equal to the vector `c("ECL", "EVC")`.

We note two specific boolean options (`@object` and `@as_object`) that have similar function across multiple blocks.

### 2.2.2 Programmatic or bulk initialization

The following describes syntax for initializing blocks as R objects using R code. This can be helpful, say, when you need to initialize a 50x50 OMEGA matrix or a series of systematically named parameters. We will describe the `@object` and `@as_object` options that are available on select blocks.

The `@object` option lets you name an object defined in `$ENV` to use to instantiate the block data. For example, to specify a series of parameters using the `@object` option, you'd write

```
$ENV
params <- list(CL = 1, V = 20, KA = 1.2)

$PARAM @object params
```

This tells mrgsolve that there is an object called `params` that it is in the `$ENV` environment and mrgsolve will use that to define the names and values. This is a trivial example to show how a simple series of parameters could be defined. However, the intended use for this functionality is to allow efficient creation of a large, systematically-named series of parameters in a large model.

The `@as_object` option is a boolean option that tells mrgsolve that the block code will actually return the object (rather than asking mrgsolve to look in `$ENV` for the object). The equivalent specification for the block above would be:

```
$PARAM @as_object
list(CL = 1, V = 20, KA = 1.2)
```

The following blocks contain both the `@object` and `@as_object` options:

- `$PARAM`
- `$INPUT`
- `$THETA`
- `$CMT`
- `$INT`
- `$OMEGA`
- `$SIGMA`

Please see the specific block documentation for more details on the specific type of object that should be returned when this syntax is invoked.

### 2.2.3 `$PROB`

**Syntax**: text
**Multiple allowed**: yes
**Options**: `@annotated`, `@covariates`

Use this block to make notes about the model. There are no restrictions on the text that gets entered here. mrgsolve does not routinely process the text in any way, except when rendering the model as a document. Frequently, we write the text here in markdown format so that it will render nicely in the model document. But this is completely optional.

See the annotated model in Section 11.1 for an example.

### 2.2.4 `$PARAM`

**Syntax**: R
**Multiple allowed**: yes
**Options**: `@annotated`, `@covariates`, `@tag`, `@input`, `@object`, `@as_object`

Define the parameter list in the current model. Parameters are names associated with values that can be used throughout the model. A value must be given for every parameter name. Names (and numbers) of parameters must be set at the time the model is compiled, but parameter values may be updated without re-compiling the model.

The `@covariates` option allows you to tag a collection of parameters as "covariates". It does not change the functionality of the model or simulation workflow in any way, but allows you to get that list of covariate names out of the model object.

The `@input` option adds a tag to the parameters in the block so that input data sets will be checked for these parameters when `check_data_names()` is called. Use the `@tag` option and supply a user-defined tag that can also be used for checking data sets and parameters.

Example:

```
[ PARAM ] CL = 1, VC = 20, KA = 1.2
KM = 25, VMAX = 400, FLAG = 1, WT = 80
SEX = 0, N = sqrt(25)
```

Note that the "values" in the parameter list will get evaluated by the R interpreter and the evaluation will happen inside the environment which is defined by the `$ENV` block. For example

```
$ENV MWT = 1.2

$PARAM
WT = 80 * 2.2
MASS = 600 * MWT
```

Annotated example:

```
[ PARAM ] @annotated
CL :   1 : Clearance (L/hr)
VC :  20 : Volume of distribution (L)
KA:  1.2 : Absorption rate constant (1/hr)
```

See the `popex` model for an example of using `@covariates`

```
.
. Model file: popex.cpp
.
. $PARAM
. TVKA = 0.5, TVCL = 1, TVV = 24
.
. $PARAM
. @covariates
. WT = 70
```

then

```
mod <- modlib("popex")
```

```
as.list(mod)$covariates
```

```
. [1] "WT"
```

Notes:

- Multiple blocks are allowed
- Values are evaluated by the `R` interpreter

The `@object` and `@as_object` options should name or return a named list of parameters.

See also: Section 2.2.5, Section 2.2.23 and Section 2.2.6.

See `?param` in the `R` help system after loading mrgsolve.

### 2.2.5 `$INPUT`

**Syntax**: R
**Multiple allowed**: yes
**Options**: `@annotated`, `@covariates`, `@tag`, `@object`, `@as_object`

This block operates just like $PARAM (Section 2.2.4), in that it is a way to introduce parameters in to the model. Additionally, the `@input` tag is added to the model parameters and input data sets will be checked for these parameters when `check_data_names()` is called.

See also: Section 2.2.4, Section 2.2.23, Section 2.2.6.

### 2.2.6 `$FIXED`

**Syntax**: R
**Multiple allowed**: yes
**Options**: `@annotated`, `@object`, `@as_object`

Like `$PARAM`, `$FIXED` is used to specify `name=value` pairs. Unlike `$PARAM`, however, the values associated with names in `$FIXED` are not able to be updated.

By default, names in `$FIXED` are associated with their value through a `C++` preprocessor `#define` statement.

Usually, `$FIXED` is only used when there are a very large number of parameters ($> 100$ or 200). When some of these parameters never need to be updated, you can move them to a `$FIXED` block to get a modest gain in efficiency of the simulation.

Items in $FIXED will not be shown when parameters are queried.

Example:

```
[ PARAM ] CL = 2, VC = 20

[ FIXED ]
g = 9.8
```

Annotated example:

```
$FIXED @annotated
g : 9.8 : Acceleration due to gravity (m/s^2)
```

See also: Section 2.2.4 and Section 2.2.23.

Notes:

- Multiple blocks are allowed
- Values are evaluated by the R interpreter

### 2.2.7 $CMT and $INIT

**Syntax**: text
**Multiple allowed**: yes
**Options**: @annotated, @object, @as_object

Declare the names of all compartments in the model.

- For $CMT give the names of compartments; initial values are assumed to be 0
- For $INIT give the name and initial value for all compartments

Note that both $CMT and $INIT declare compartments, so any compartment name should get declared in either $CMT or $INIT, but never both.

Examples:

```
[ CMT ] GUT CENT RESPONSE
```

```
[ INIT ] GUT  = 0, CENT = 0, RESPONSE = 25
```

Annotated examples:

```
[ CMT ] @annotated
GUT      : Dosing compartment (mg)
CENT     : Central PK compartment (mg)
RESPONSE : Response
```

```
$INIT @annotated
GUT      :   0 : Dosing compartment (mg)
CENT     :   0 : Central PK compartment (mg)
RESPONSE :  25 : Response
```

The `@object` and `@as_object` options should name or return a named list of compartments and initial values when used in `$INIT` and a character vector of compartment names when used in `$CMT`.

See `?init` in the `R` help system after loading mrgsolve.

### 2.2.8 `$MAIN`

**Syntax**: C++
**Multiple allowed**: no

This code block has two main purposes:

- Derive new algebraic relationships between parameters, random, effects and other derived variables
- Set the initial conditions for model compartments

For users who are familiar with NONMEM, `$MAIN` is similar to `$PK`.

`$MAIN` is wrapped into a `C++` function and compiled / loaded by mrgsolve.
The `MAIN` function gets called just prior to advancing the system from the current time to the next time for each record in the data set. `$MAIN` also gets called several times before starting the problem (`NEWIND == 0`) and just prior to simulating each individual (`NEWIND == 1`). Finally, `$MAIN` gets called every time the model initial conditions are queried with `init()`.

New variables may be declared in `$MAIN`. See Section 2.5 for details.

Examples:

```
[ CMT ] CENT RESP

[ PARAM ] KIN = 100, KOUT = 2, CL = 1, VC = 20

[ MAIN ]
```

```
RESP_0 = KIN/KOUT;

double ke = CL/VC;
```

### 2.2.9 $PK

This is an alias for $MAIN.

### 2.2.10 $ODE

**Syntax**: C++
**Multiple allowed**: no
**Options**: @param

Use $ODE to define model differential equations. For all compartments assign the value of the differential equation to dxdt_CMT where CMT is the name of the compartment. The dxdt_ equation may be a function of model parameters (via $PARAM), the current value of any compartment (CMT) or any user-derived variable.

For example:

```
[ CMT ] GUT CENT

[ ODE ]
dxdt_GUT = -KA*GUT;
dxdt_CENT = KA*GUT - KE*CENT;
```

It is important to make sure that there is a dxdt_ expression defined for every compartment listed in $CMT or $INIT, even if it is dxdt_CMT = 0;

The $ODE function is called repeatedly during a simulation run. So it is wise to do as many calculations as possible outside of $ODE, usually in $MAIN. But remember that any calculation that depends on an amount in a compartment and helps determine the dxdt_ expression in a model must be written in $ODE.

New variables may be declared in $ODE. See Section 2.5 for details.

For example:

```
$CMT CENT RESP
$PARAM VC = 100, KE = 0.2, KOUT = 2, KIN = 100
$ODE
double CP = CENT/VC;
double INH = CP/(IMAX+CP)

dxdt_CENT = -KE*CENT;
dxdt_RESP =  KIN*(1 - INH) - RESP*KOUT;
```

If the model needs to refer to the current time, use the `SOLVERTIME` variable.

Notes:

- `$ODE` is written in `C++` syntax; every line must end in `;`
- There may be only one `$ODE` block in a model

### 2.2.11 `$DES`

This is an alias for `$ODE`.

### 2.2.12 `$TABLE`

**Syntax**: C++
**Multiple allowed**: no

Use `$TABLE` to interact with parameters, compartment values, and other user-defined variables **after** the system advances to the next time.

For example:

```
[ TABLE ]
double CP = CENT/VC;
```

**NOTE** mrgsolve formerly had a `table()` macro for inserting derived values into simulated output. This macro has been deprecated. The only way to insert derived values into the simulated output is via `$CAPTURE`.

**NOTE** When variables are marked for capture (see Section 2.2.16), the values of those variables are saved at the **end** of the `$TABLE` function. This process is carried out automatically by mrgsolve and therefore requires no user intervention.

### 2.2.13 $ERROR

This is an alias for $TABLE.

### 2.2.14 $PREAMBLE

**Syntax**: C++
**Multiple allowed**: no

This is the fourth C++ code block. It is called once in two different settings (Chapter 7):

1. Immediately prior to starting the simulation run
2. Immediately prior to calling $MAIN when calculating initial conditions

$PREAMBLE is a function that allows you to set up your C++ environment. It is only called one time during the simulation run (right at the start). The code in this block is typically used to configure or initialize C++ variables or data structures that were declared in $GLOBAL.

For example:

```
[ PLUGIN ] Rcpp

[ GLOBAL ]
namespace{
  Rcpp::NumericVector x;
}

[ PREAMBLE ]
x.push_back(1);
x.push_back(2);
x.push_back(3);

[ MAIN ]
<some code that uses x vector>
```

In this example, we want to use a numeric vector x and declare it in $GLOBAL so that we can use it anywhere else in the code (the declaration is also made in an unnamed namespace to ensure that the variable is local to the model file). Then, in $PREAMBLE, we put 3 numbers into the vector and we use x in $MAIN. Since $MAIN, $TABLE and (especially) $ODE are called repeatedly as the simulation run progresses, we put the initialization of x in $PREAMBLE to make sure the initialization of x only happens once.

Notes:

- $PREAMBLE is written in C++ syntax; every line must end in ;
- There may be only one $PREAMBLE block in a model
- Like $MAIN, $ODE and $TABLE, double, int and bool variables initialized in $PREAMBLE are actually initialized for global (within the model file)

See also: Chapter 7 and Section 2.2.22.

### 2.2.15 $PRED

**Syntax**: C++
**Multiple allowed**: no

Use $PRED to write a model without differential equations. In this block, write all algebraic expressions for derived parameters, the response, and any other derived output quantities.

For example:

```
[ PARAM ] TVEO = 100, AUC50 = 100, IMAX = 40, AUC = 0

[ PRED ]
double E0 = EVE0*exp(ETA(1));

double RESP = E0 - IMAX*AUC/(AUC50+AUC);
```

In this example, the entire model is written in the $PRED block. It is an error to include the following blocks when $PRED is being used: $MAIN, $TABLE, $PKMODEL, $ODE, $CMT, $INIT.

See Section 3.7 for additional information regarding data sets in use with $PRED block.

### 2.2.16 $CAPTURE

**Syntax**: text
**Multiple allowed**: yes
**Options**: @annotated, @etas

This is a block to identify variables that should be captured in the simulated output. The @etas option is new with version 1.0.8.

For example:

```
[ PARAM ] A = 1, B = 2

[ MAIN ]
double C = 3;
bool yes = true;

[ CAPTURE ] A B C yes
```

This construct will result in four additional columns in the simulated output with names `A`, `B`, `C`, and `yes`.

Users can also rename captured variables by providing a `newname = oldname` specification.

```
$PARAM WT = 70, THETA1 = 2.2

$MAIN
double CL = THETA1*pow(WT/70,0.75)*exp(ETA(1));

$OMEGA 1

$CAPTURE WEIGHT = WT TVCL = THETA2 CL  ETA(1)
```

In this example, the names of the captured data items will be `WEIGHT,TVCL,CL,ETA_1`.

Users can use the `capture` type to declare variables in `$MAIN` and `$TABLE`. `capture` types are really `double`s, but using that type will signal mrgsolve to automatically capture that value. For example:

```
$PARAM VC = 300

$CMT CENT

$TABLE
capture DV = (CENT/VC);
```

Since we used type `capture` for `DV`, `DV` will show up as a column in the simulated data.

Annotated example:

```
$MAIN
double CLi = TVCL*exp(ECL);
```

```
$TABLE
double DV = (CENT/VC)*exp(PROP);

$CAPTURE @annotated
CLi : Individual clearance (L/hr)
DV  : Plasma concentration (mcg/ml)
```

Use the `@etas` option to specify an expression that evaluates to integers which identify the `ETA` number(s) to capture. For example

```
$OMEGA 0.1 0.2 0.3

$CAPTURE @etas 1:LAST
CL DV
```

Will capture all three `ETA`s into the simulation outputs. NONMEM-style names will be used (e.g. `ETA1`). This is in contrast to the names that are generated when `ETA(1)` is captured; in this case, the parens are removed and replaced with `_` so the output name is `ETA_1`. mrgsolve will provide `LAST` (and `last`) which will evaluate to the maximum number of `ETA`s in the problem (in this case, 3). You can write any valid expression in `@etas`. When the expression is evaluated, it will be evaluated in the model environment created through the `$ENV` block (Section 2.2.27).

> **Tip**
>
> In addition to listing model variables for output in the `$CAPTURE` block, users can "dynamically" capture model variables through the `capture` argument to `mread()`. These may be model parameters (listed in `$PARAM`) or user-defined model variables.
> To get a listing of available user-defined variables, you can use
>
> ```
> mod <- house()
> as.list(mod)$cpp_variables
> ```

### 2.2.17 `$OMEGA`

**Syntax**: text
**Multiple allowed**: yes
**Options**: `@annotated`, `@block`, `@correlation`, `@labels`, `@name`, `@object`, `@as_object`

See `?modMATRIX` for more details about options for this block.

Use this block to enter variance/covariance matrices for subject-level random effects drawn from multivariate normal distribution. All random effects are assumed to have mean of 0. Off diagonal elements for block matrices are assumed to be correlation coefficients if the `@correlation` option is used (see below).

By default, a **diagonal** matrix is assumed. So:

```
$OMEGA
1 2 3
```

will generate a 3x3 omega matrix.

A **block** matrix may be entered by using `block = TRUE`. So:

```
$OMEGA @block
0.1 0.02 0.3
```

will generate a 2x2 matrix with covariance 0.02.

A 2x2 matrix where the off-diagonal element is a correlation, not a covariance can be specified like this:

```
$OMEGA @correlation
0.1 0.67 0.3
```

Here, the correlation is 0.67. mrgsolve will calculate the covariances and substitute these values. The matrix will be stored and used with these covariances, not the correlation.

A name can be assigned to each matrix:

```
$OMEGA @name PK @block
0.2 0.02 0.3

$OMEGA @name PD
0.1 0.2 0.3 0.5
```

to distinguish between multiple `$OMEGA` blocks and to facilitate updating later. The model in the preceding example will have two `$OMEGA` matrices: 2x2 and 4x4.

Labels can be assigned which function as aliases for the different `ETAs`

```
$OMEGA @block @labels ETA_CL ETA_V
0.1 0.05 0.2
```

The number of labels should match the number of rows (or columns) in the matrix.

Annotated example (diagonal matrix):

```
$OMEGA @annotated
ECL: 0.09 : ETA on clearance
EVC: 0.19 : ETA on volume
EKA: 0.45 : ETA on absorption rate constant
```

Annotated example (block matrix):

```
$OMEGA @annotated @block
ECL: 0.09 : ETA on clearance
EVC: 0.001 0.19 : ETA on volume
EKA: 0.001 0.001 0.45 : ETA on absorption rate constant
```

The `@object` and `@as_object` options should name or return a square numeric matrix. If `rownames` are included in the matrix, then they will be used to form labels for the realized ETAs. For example, we can initialize a very large `$OMEGA` matrix with

```
$OMEGA @as_object
n <- 20
lbl <- paste0("ETA_", LETTERS[1:n])
matrix(0, 20, 20, dimnames = list(lbl, lbl))
```

Note: this only initializes the matrix; you will (likely) need to update it with meaningful values after the model is loaded (Chapter 5).

### 2.2.18 `$SIGMA`

**Syntax**: text
**Multiple allowed**: yes
**Options**: `@annotated`, `@block`, `@correlation`, `@labels`, `@name`, `@object`, `@as_object`

See `?modMATRIX` for more details about options for this block.

Use this block to enter variance/covariance matrices for within-subject random effects drawn from multivariate normal distribution. All random effects are assumed to have mean of 0. Off diagonal elements for block matrices are assumed to be correlation coefficients if the `@correlation` option is used (see below).

The `@object` and `@as_object` options should name or return a square numeric matrix. If `rownames` are included in the matrix, then they will be used to form labels for the realized EPS values.

The `$SIGMA` block functions like the `$OMEGA` block. See `$OMEGA` for details.

### 2.2.19 `$SET`

**Syntax**: R
**Multiple allowed**: no

Use this code block to set different options for the simulation. Use a `name = value` format, where `value` is evaluated by the `R` interpreter.

Most of the options that can be entered in `$SET` are passed to `update`.

For example:

```
[ SET ] end = 240, delta = 0.5, req = "RESP"
```

Here, we set the simulation `end` time to 240, set the time difference between two adjacent time points to 0.25 time units, and request only the `RESP` compartment in the simulated output.

### 2.2.20 `$GLOBAL`

**Syntax**: C++
**Multiple allowed**: no

The `$GLOBAL` block is for writing `C++` code that is outside of `$MAIN`, `$ODE`, and `$TABLE`.

There are no artificial limit on what sort of `C++` code can go in `$GLOBAL`.
However there are two more-common uses:

1. Write `#define` preprocessor statements
2. Define global variables, usually variables other than `double`, `bool`, `int` (see Section 2.5)

**Preprocessor directives** Preprocessor `#define` directives are direct substitutions that the `C++` preprocessor makes prior to compiling your code.

For example:

```
[ GLOBAL ]
#define CP (CENT/VC)
```

When this preprocessor directive is included, everywhere the preprocessor finds a `CP` token it will substitute `(CENT/VC)`. Both `CENT` and `VC` must be defined and the ratio of `CENT` to `VC` will be calculated depending on whatever the current values are. Notice that we included parentheses around `(CENT/VC)`.
This makes sure the ratio between the two is taken first, before any other operations involving `CP`.

**Declaring global variables** Sometimes, you may wish to use global variables and have more control over how they get declared.

```
$GLOBAL
bool cure = false;
```

With this construct, the boolean variable `cure` is declared and defined right as the model is compiled.

**Declare in bulk**

If you have a large number of variables to declare, you can do that in bulk in `$GLOBAL`. For example, we know we will need a long list of `double` precision variable for covariate modeling, we can declare them all at once like this:

```
[ global ]
double TVCL, TVV2, TVQ = 0, TVV3 = 0;

[ main ]

TVCL = THETA1 * pow(WT / 70.0, 0.75);
TVV2 = THETA2 * WT / 70.0;
TVQ  = THETA3 * pow(WT / 70.0, 0.75);
TVV3 = THETA4 * WT / 70.0;
```

This isn't a terribly long list, but let's pretend it is to illustrate how to do this. You can also declare `int`, `bool` and other types like this. I've initialized the last two (`TVVQ` and `TVV3`) to illustrate how to do this. It's good practice to do that but not necessary as long as everything gets initialized to **something** before they are used.

### 2.2.21 `$PKMODEL`

**Syntax**: R
**Multiple allowed**: no

This code block implements a one- or two-compartment PK model where the system is calculated by algebraic equations, not ODEs. mrgsolve handles the calculations and an error is generated if both `$PKMODEL` and `$ODE` blocks are included in the same model specification file.

This is an options-only block. The user must specify the number of compartments (1 or 2) to use in the model as well as whether or not to include a depot dosing compartment. See `?PKMODEL` for more details about this block, including specific requirements for symbols that must be defined in the model specification file.

The `$CMT` or `$INIT` block must also be included with an appropriate number of compartments. Compartment names, however, may be determined by the user.

Example:

```
[ CMT ] GUT CENT PERIPH

[ PKMODEL ] ncmt=2, depot=TRUE
```

As of version `0.8.2`, we can alternatively specify the compartments right in the `$PKMODEL` block:

```
$PKMODEL cmt="GUT CENT PERIPH", depot = TRUE
```

Specifying three compartments with `depot=TRUE` implies `ncmt=2`. Notice that a separate `$CMT` block is not appropriate when `cmt` is specified in `$PKMODEL`.

### 2.2.22 `$PLUGIN`

**Syntax**: text
**Multiple allowed**: no

Plugins are a way to add extensions to your mrgsolve model. Plugins can either link your model to external libraries (like `boost` or `Rcpp`) or they can open up access to additional functionality provided by mrgsolve itself.

Plugins are listed and discussed in more detail in Chapter 9.

**Usage**

To invoke a plugin, list the plugin name in the code block. For requesting `Rcpp` headers in your model, call

```
$PLUGIN Rcpp
```

### Available plugins

The following plugins make additional mrgsolve-specific functionality available

- `mrgx`: extra `C++` functions (see below)
- `tad`: track time after dose in your model
- `N_CMT`: get the number of a compartment

The following plugins give you a different model specification experience

- `autodec`: mrgsolve will find assignments in your model and automatically declare them as `double`s
- `nm-vars`: a NONMEM look and feel for compartmental models; use `F1`, `A(1)` and `DADT(1)` rather than `F_GUT`, `GUT` and `dxdt_GUT`

The following plugin lets you customize how your model is compiled

- `CXX11`: compile your model with `C++11` standard; this adds the compiler flag `-std=c++11`

The following plugins will let you link to external libraries

- `Rcpp`: include `Rcpp` headers int your model
- `BH`: include `boost` headers in your model
- `RcppArmadillo`: include `Armadillo` headers in your model

Note that `Rcpp`, `RcppArmadillo` and `BH` only allow you to link to those headers. To take advantage of that, you will need to know how to use `Rcpp`, `boost` etc. For the `BH` plugin, no headers are included for you; you must include the proper headers you want to use in `$GLOBAL`.

**mrgx** This is a general collection of functions that we made available.

Functions provided by `mrgx`:

- `T get<T>(std::string <pkgname>, std::string <objectname>)`
  - This gets an object of any Rcpp-representable type (`T`) from any package
- `T get<T>(std::string <objectname)`
  - This gets an object of any Rcpp-representable type (`T`) from `.GlobalEnv`
- `T get<T>(std::string <objectname>, databox& self)`
  - This gets an object of any Rcpp-representable type (`T`) from `$ENV`
- `double rnorm(double mean, double sd, double min, double max)`

  - Simulate one variate from a normal distribution that is between `min` and `max`

- `double rlognorm(double mean, double sd, double min, double max)`
  - Same as `mrgx::rnorm`, but the simulated value is passed to `exp` after simulating

- `Rcpp::Function mt_fun()`
  - Returns `mrgsolve::mt_fun`; this is usually used when declaring a `R` function in `$GLOBAL`
  - Example: `Rcpp::Function print = mrgx::mt_fun();`

**IMPORTANT** All of these functions are in the `mrgx` namespace. So, in order to call these functions you must include `mrgx::` namespace identifier to the front of the function name. For example, don't use `rnorm(50,20,40,140)`; use `mrgx::rnorm(50,20,40,140)`.

#### 2.2.22.1 Some examples

**Get a numeric vector from `$ENV`**

```
[ PLUGIN ] Rcpp mrgx

[ ENV ]
x <- c(1,2,3,4,5)

[ GLOBAL ]
Rcpp::NumericVector x;

[ PREAMBLE ]
x = mrgx::get<Rcpp::NumericVector>("x", self);
```

**Get the `print` function from `package:base`**

```
$PLUGIN Rcpp mrgx

$GLOBAL
Rcpp::Function print = mrgx::mt_fun();

$PREAMBLE
print = mrgx::get<Rcpp::Function>("base", "print");

$MAIN
print(self.rown);
```

Note that we declare the `print` in `$GLOBAL` and use the `mt_fun()` place holder.

**Simulate truncated normal variables** This simulates a weight that has mean 80, standard deviation 20 and is greater than 40 and less than 140.

```
$PLUGIN Rcpp mrgx

$MAIN
if(NEWIND <=1) {
  double WT = mrgx::rnorm(80,20,40,140);
}
```

See also: Section 2.2.14.

### 2.2.23 `$THETA`

**Syntax**: text
**Multiple allowed**: yes
**Options**: `@annotated`, `@name`, `@object`, `@as_object`

Use this code block as an efficient way to add to the parameter list where names are determined by a prefix and a number. By default, the prefix is `THETA` and the number sequentially numbers the input values.

For example:

```
[ THETA ]
0.1 0.2 0.3
```

is equivalent to

```
$PARAM THETA1 = 0.1, THETA2 = 0.2, THETA3 = 0.3
```

Annotated example:

```
$THETA @annotated
0.1 : Typical value of clearance (L/hr)
0.2 : Typical value of volume (L)
0.3 : Typical value of ka (1/hr)
```

To change the prefix, use `@name` option

```
$THETA @name theta
0.1 0.2 0.3
```

would be equivalent to

```
[ PARAM ] theta1 = 0.1, theta2 = 0.2, theta3 = 0.3
```

The `@object` and `@as_object` options should name or return an unnamed vector of parameter values (names are ignored).

See also: Section 2.2.4.

### 2.2.24 `$NMXML`

**Syntax**: R
**Multiple allowed**: yes

The `$NMXML` block lets you read and incorporate results from a NONMEM run into your mrgsolve model.

#### Syntax

If your model run is 1000 and the `.xml` file is located in `1000/1000.xml`, use this syntax

```
$NMXML
run = 1000
project = "../model/nonmem"
root = "cppfile"
```

If your model run is 1000 and the `.xml` file is located in some other location use the `path` argument to specify the *complete* path to the `.xml` file

```
$NMXML
path = "../<path>/<to>/<output>/<files>/1000.xml"
root = "cppfile"
```

This latter syntax may be needed when using PsN and output files are organized in a way different from the previous example.

### Details

From the NONMEM run, `THETA` will be imported into your parameter list (see Section 2.2.4 and Section 1.1), `OMEGA` will be captured as an `$OMEGA` block (Section 2.2.17) and `SIGMA` will be captured as a `$SIGMA` block (Section 2.2.18). Users may optionally omit any one of these from being imported.

`$NMXML` contains a `project` argument and a `run` argument. By default, the estimates are read from from the file `project/run/run.xml`. That is, it is assumed that there is a directory named `run` that is inside the `project` directory where `$NMXML` will find `run.xml`. Your NONMEM run directories may not be organized in a way that is compatible with this default. In that case, you will need to provide the `path` argument, which should be the path to the `run.xml` file, either as a full path or as a path relative to the current working directory.

Once the model object is obtained, the path to the `xml` file that formed the source for imported parameters can be retrieved by coercing the model object to list and looking for `nm_import`:

```
mod <- modlib("1005", compile = FALSE)

as.list(mod)$nm_import
```

For help on the arguments / options for `$NMXML`, please see the `?nmxml` help topic in your `R` session after loading the mrgsolve package.

### An example

There is a NONMEM run embedded in the mrgsolve package

```
path <- file.path(path.package("mrgsolve"),"nonmem")
list.files(path, recursive=TRUE)
```

```
.  [1] "1005/1005.cat"    "1005/1005.coi"    "1005/1005.cor"    "1005/1005.cov"
.  [5] "1005/1005.cpu"    "1005/1005.ctl"    "1005/1005.ext"    "1005/1005.grd"
.  [9] "1005/1005.lst"    "1005/1005.phi"    "1005/1005.shk"    "1005/1005.shm"
. [13] "1005/1005.tab"    "1005/1005.xml"    "1005/1005par.tab" "1005/INTER"
. [17] "2005/2005.ext"
```

We can create a mrgsolve control stream that will import `THETA`, `OMEGA` and `SIGMA` from that run using the `$NMXML` code block.

```
// inline/nmxml-1.cpp

$NMXML
run = 1005
project = path
root = "cppfile"

olabels = c("ECL", "EVC", "EKA")
slabels = c("PROP", "ADD")

$MAIN
double CL = THETA1*exp(ECL);
double V2 = THETA2*exp(EVC);
double KA = THETA3*exp(EKA);
double Q = THETA4;
double V3 = THETA5;

$PKMODEL ncmt=2, depot=TRUE

$CMT GUT CENT PERIPH

$TABLE
double CP = (CENT/V2)*(1+PROP) + ADD/5;

$CAPTURE CP

$SET delta=4, end=96
```

**NOTE**: in order to use this code, we need to install the `xml2` package.

```
mod <- mread("inline/nmxml-1.cpp", compile = FALSE)

mod
```

```
.
.
. ---------------  source: nmxml-1.cpp  ---------------
.
.   project: /Users/kyleb/git...-guide/inline
.   shared object: nmxml-1.cpp-so-a8213fe7e29f <not loaded>
.
```

```
.   time:          start: 0 end: 96 delta: 4
.                  add: <none>
.
.   compartments:  GUT CENT PERIPH [3]
.   parameters:    THETA1 THETA2 THETA3 THETA4 THETA5
.                  THETA6 THETA7 [7]
.   captures:      CP [1]
.   omega:         3x3
.   sigma:         2x2
.
.   solver:        atol: 1e-08 rtol: 1e-08 maxsteps: 20k
. -----------------------------------------------------
```

```
param(mod)
```

```
.
.  Model parameters (N=7):
.  name   value  . name   value
.  THETA1 9.51   | THETA5 113
.  THETA2 22.8   | THETA6 1.02
.  THETA3 0.0714 | THETA7 1.19
.  THETA4 3.47   | .      .
```

```
revar(mod)
```

```
. $omega
. $...
.              [,1]        [,2]        [,3]
. ECL:   0.21387884  0.12077020 -0.01162777
. EVC:   0.12077020  0.09451047 -0.03720637
. EKA:  -0.01162777 -0.03720637  0.04656315
.
.
. $sigma
. $...
.              [,1]      [,2]
. PROP:  0.04917071 0.0000000
. ADD:   0.00000000 0.2017688
```

**root argument**: please use the `root = "cppfile"` argument going forward.

As of mrgsolve `0.11.0`, we added an argument called `root` to `$NMXML` that tells mrgsolve the location where it should read the `xml` file from. The default behavior is the `"working"` directory. When this is the case, mrgsolve assumes that the the `xml` file can be found relative to the "working" directory. The only other value that `root` can take is `"cppfile"`. When `root = "cppfile"`, then mrgsolve will look for the `xml` file in a directory that is relative to **where the model source code file is located**. Please take a look at Section 2.2.25 for more discussion and examples.

Starting with version 1.0.8, when the NONMEM run is found using the `run` + `project` arguments, you can pass `run = "@cppstem"` and mrgsolve will look for the NONMEM run number that matches the stem of the current mrgsolve file. For example, if you are coding a model to match NONMEM run `101.ctl`, then call your mrgsolve model file `101.cpp` or `101.mod`; just match the file stem. Then write

**Listing 2.1** `101.cpp`

```
$NMXML
run = "@cppstem"
project = <nonmem-run-directory>
```

mrgsolve will check the stem of the mrgsolve model file and look for the NONMEM `.xml` file in

```
<nonmem-run-directory>/101/101.xml
```

Please see the `?nmxml` help topic for more information on arguments that can be passed to `$NMXML`.

See also: Section 2.2.25.

### 2.2.25 `$NMEXT`

**Syntax**: R
**Multiple allowed**: yes

Like `$NMXML`, `$NMEXT` allows the import of `$THETA`, `$OMEGA`, and `$SIGMA` from your NONMEM run into your mrgsolve model, but the estimates are read from the `.ext` file output. `$NMEXT` is able to import the NONMEM estimates much faster than `$NMXML` when loading from sampling based methods (mainly `METHOD=BAYES`).

### Syntax

If your model run is 1000 and the `.ext` file is located in `1000/1000.ext`, use this syntax

```
$NMEXT
run = 1000
project = "../model/nonmem"
root = "cppfile"
```

If your model run is 1000 and the `.ext` file is located in some other location use the `path` argument to specify the *complete* path to the `.ext` file

```
$NMEXT
path = "../<path>/<to>/<output>/<files>/1000.ext"
root = "cppfile"
```

This latter syntax may be needed when using PsN and output files are organized in a way different from the previous example.

## Details

Once the model object is obtained, the path to the `ext` file that formed the source for imported parameters can be retrieved by coercing the model object to list and looking for `nm_import`.

```
mod <- modlib("1005", compile = FALSE)

as.list(mod)$nm_import
```

See the `$NMXML` topic in Section 2.2.24 for an example of equivalent functionality.

**IMPORTANT**: while `$NMEXT` works very similarly to `$NMXML`, there is one key difference between the two: when using `$NMEXT`, you will always get one `$OMEGA` matrix and one `$SIGMA` matrix, regardless of the block structure used in the NONMEM control stream. When using `$NMXML`, you get the same block structure in the mrgsolve model as the NONMEM model. For example: in the NONMEM control stream, `$OMEGA` is one 2x2 matrix and one 3x3 matrix. Importing those estimates with `$NMEXT` will give you one 5x5 matrix, while importing the estimates with `$NMXML` will give you a list of one 2x2 and one 3x3 matrix.

**root argument**: please use the `root = "cppfile"` argument going forward.

As of mrgsolve `0.11.0`, we added an argument called `root` to `$NMEXT` that tells mrgsolve the location where it should read the `ext` file from. The default behavior is the `"working"`

directory. When this is the case, mrgsolve assumes that the the `ext` file can be found relative to the "working" directory. The only other value that `root` can take is `"cppfile"`. When `root = "cppfile"`, then mrgsolve will look for the `ext` file in a directory that is relative to **where the model source code file is located**.

We recommend that users start using the `root` argument and set it to "cppfile". This will eventually become the default. For example:

```
$NMEXT
run = 1005
project = "../../model/pk"
root = "cppfile"
```

This tells mrgsolve to find the nonmem run back two directories (`../../`) and then into `model --> pk --> 1005` relative to where the mrgsolve model file is located. This is in contrast to the previous expected behavior that the path should be relative to the current working directory.

We also note here that when users pass an absolute path, the relative path doesn't matter at all. Users are free to use pathing tools to generate the absolute path to the project. For example, the `here::here()` function can be used like this

```
$NMEXT
run = 1005
project = here::here("model/pk")
```

When used in the proper context, `here()` will generate an absolute path from your projects root directory to the nonmem project directory. Please refer to the `here::here()` documentation for proper use of this function.

Starting with version 1.0.8, when the NONMEM run is found using the `run` + `project` arguments, you can pass `run = "@cppstem"` and mrgsolve will look for the NONMEM run number that matches the stem of the current mrgsolve file. See details and example in Section 2.2.24.

See the `?nmext` R help topic for arguments that can be passed to `$NMEXT` block. Notably, the user can select the function to read in the ext file.By default, mrgsolve will try to load data.table and use the `fread` function. If data.table can't be loaded, then mrgsolve will use `utils::read.table`.

See also: Section 2.2.24.

### 2.2.26 $INCLUDE

**Syntax**: text
**Multiple allowed**: no

To include your own header file(s) in a model use $INCLUDE

```
$INCLUDE
mystuff.h
otherstuff.h
```

or

```
$INCLUDE
mystuff.h,  otherstuff.h
```

mrgsolve will insert proper #include preprocessor directives into the C++ code that gets compiled.

**Requirements**

- All header files listed in $INCLUDE are assumed (expected) to be under the project directory; don't use $INCLUDE for header files that are in any other location
- An error is generated if the header file does not exist
- An error is generated if any quotation marks are found in the file name (don't use quotes around the file name; mrgsolve will take care of that)
- A warning is issued if the header file does not end in .h
- When the header file is changed (MD5 checksum changes), the model will be forced to be rebuilt (recompiled) when mread() or mcode() (but not mread_cache() or mcode_cache()) is called; this feature is only available for header files listed in $INCLUDE (see below)
- Do not use $INCLUDE to include Rcpp, boost, RcppArmadillo or RcppEigen headers; use the appropriate $PLUGIN instead

For applications that don't fit into the requirements listed above, users can always include header files in the model in $GLOBAL like this:

```
$GLOBAL
#include "/Users/me/libs/mystuff.h"
```

But be careful when doing this: if there are changes to mystuff.h but not to any other part of the model specification, the model may not be fully compiled when calling mread. In this case, always use preclean=TRUE argument to mread to force the model to be built when calling mread.

### 2.2.27 $ENV

**Syntax**: R
**Multiple allowed**: no

This block is all R code (just as you would code in a stand-alone R script. The code is parsed and evaluated into a new environment when the model is compiled.

For example:

```
$ENV

Sigma <- cmat(1,0.6,2)

mu <- c(2,4)

cama <- function(mod) {
  mod %>%
    ev(amt=100, ii=12, addl=10) %>%
    mrgsim(obsonly=TRUE,end=120)
}
```

Objects inside $ENV can be utilized in different C++ functions (see Section 2.2.22) or other parts of the simulation process.

Objects inside $ENV will also be made available when parsing different parts of the model specification file. For example, we can initialize an $OMEGA block with the Sigma object in the $ENV block example above:

```
$OMEGA @object Sigma
```

Another example is providing constants or functions to simplify code in the $PARAM block

```
$ENV
rescale <- 1/1000

convert <- function(x) x * 5 / 166.2

$PARAM
p = 500 * rescale

q  = convert(2.51)
```

You can also make your model specification file depend on an R package; for example, if you want to use the `here` package to render a path to your NONMEM model

```
$ENV
if(!requireNamespace("here")) {
  stop("the `here` package is required to load this model.")
}

$NMXML
run = 101
project = here::here("model/nonmem")
```

You can access the model environment using the `env_get()` function:

```
env_get(mod)
```

or directly in the `envir` slot

```
mod@envir
```

See also `env_ls()` to list the names of variables in the model environment, `env_update()` to change the values of objects in the model environment, or `env_eval` to re-evaluate the model environment code.

## 2.3 Variables and Macros

This section describes some macros and internal variables that can be used in model specification files. In the following section, we adopt the convention that `CMT` stands for a compartment in the model.

**IMPORTANT NOTE**:

It should be clear from the usage examples which variables can be set by the user and which are to be read or checked. **All internal variables are pre-defined and pre-initialized by mrgsolve.** ***The user should never try to declare an internal variable; this will always result in an compile-time error***.

### 2.3.1 `ID`

The current subject identifier. `ID` is an alias for `self.id`.

### 2.3.2 TIME

Gives the time in the current data set record. This is usually only used in `$MAIN` or `$TABLE`. `TIME` is an alias for `self.time`. Contrast with `SOLVERTIME`.

### 2.3.3 SOLVERTIME

Gives the time of the current timestep taken by the solver. *This is can only be used in `$ODE`.* Contrast with `TIME`.

Starting with mrgsolve version 1.0.0, the variable `T` is provided as a synonym to `SOLVERTIME` inside `$ODE` when you invoke the `nm-vars` plugin (Section 9.2).

### 2.3.4 T

Only provided when the `nm-vars` plugin is invoked; see `SOLVERTIME`.

### 2.3.5 EVID

`EVID` is an event id indicator. mrgsolve recognized the following event IDs:

- 0 = an observation record
- 1 = a bolus or infusion dose
- 2 = other type event, with solver reset
- 3 = system reset
- 4 = system reset and dose
- 8 = replace

`EVID` is an alias for `self.evid`.

### 2.3.6 CMT

`CMT` is the current compartment number. In this case, `CMT` is used literally (not a stand-in for the name of a compartment). For example:

```
$CMT GUT CENT PERIPH

$MAIN

if(CMT==2) {
```

```
  // ....
}
```

In the example, GUT, CENT and PERIPH are the amounts in the respective compartments and CMT refers to the value of CMT in the data record / data set.

CMT is an alias for self.cmt.

### 2.3.7 AMT

AMT is the current value of dose amount.

AMT is an alias for self.amt.

### 2.3.8 NEWIND

NEWIND is a new individual indicator, taking the following values:

- 0 for the first event record of the data set
- 1 for the first event record of a subsequent individual
- 2 for subsequent event record for an individual

For example:

```
[ GLOBAL ]
int counter = 0;

[ MAIN ]
if(NEWIND <=1) {
  counter  = 0;
}
```

NEWIND is an alias for self.newind.

### 2.3.9 SS_ADVANCE

This is a bool data item (either true or false) which is always false unless mrgsolve is currently advancing the system to steady state (Chapter 8); then it will be true. This variable is only available in the $ODE (or $DES) block.

Use this variable to modify the calculation of your model differential equations while the system is advancing to steady state. One use case is when you have an accumulation compartment to calculate AUC but you want to halt accumulation while the system is working toward steady state

```
$ODE
dxdt_CENT = -(CL/V) * CENT;
dxdt_AUC = CENT/V;
if(SS_ADVANCE) dxdt_AUC = 0;
```

### 2.3.10 simeta()

The simeta() function can be used to re-simulate ETA values. For example,

```
$MAIN
simeta();
```

will re-simulate all ETA(n).

### 2.3.11 simeps()

simeps() works like simeta(), but all EPS(n) values are re-simulated rather than the ETA values.

### 2.3.12 self

self is an object that gets passed to your model function that contains data items and functions that can be called. It is a struct (see the source code here). A partial list of members are documented in the following sections.

self functions include

- self.tad()
- self.mtime()
- self.mevent()

- `self.stop()`
- `self.stop_id()`
- `self.stop_id_cf()`

`self` data members include

- `self.id` (`ID`)
- `self.amt` (`AMT`)
- `self.cmt` (`CMT`)
- `self.evid` (`EVID`)
- `self.time` (`TIME`)
- `self.newind` (`NEWIND`)
- `self.nid`
- `self.idn`
- `self.rown`
- `self.nrow`
- `self.envir` (see note below)

This information is provided for transparency and is not exhaustive. We provide an interface to these data items through pre-processor directives with simpler names (e.g. `EVID` will translate to `self.evid`) and users are encouraged to use the simpler, API name.

### 2.3.13 `self.time`

The current data set `TIME`.

### 2.3.14 `self.cmt`

The current compartment number regardless of whether it was given as `cmt` or `CMT` in the data set. There is no alias for `self.cmt`.

For example:

```
$TABLE

double DV = CENT/VC + EPS(1);

if(self.cmt==3) DV = RESPOSE + EPS(2);
```

### 2.3.15 `self.amt`

The current `amt` value regardless of whether it was given as `amt` or `AMT` in the data set. There is no alias for `self.amt`.

```
[ PREAMBLE ]
double last_dose = 0;

[ MAIN ]

if(EVID==1) {
  last_dose = self.amt;
}
```

### 2.3.16 `self.nid`

The number of IDs in the data set.

### 2.3.17 `self.idn`

The current ID number. Numbers start at 0 and increase by one to `self.nid-1`. So if you want to test for the last ID in the output data set, you would write:

```
[ table ]

if(self.idn == (self.nid-1)) {
  // do something ....
}
```

### 2.3.18 `self.nrow`

The total number of rows in the output data set.

### 2.3.19 `self.rown`

The current row number. Numbers start at 0 and increase by one to `self.rown-1`. So if you want to test for the last row in the output data set, you would write:

```
[ table ]

if(self.rown == (self.nrow-1)) {
  // do something ....
}
```

### 2.3.20 self.envir

This item is a null pointer that can be cast to an Rcpp::Environment only when the Rcpp plugin is invoked (see Section 9.6).

For example

```
[ plugin ] Rcpp mrgx

[ preamble ]
Rcpp::Environment env = mrgx::get_envir(self);
```

### 2.3.21 self.tad()

This is a function that calculates and returns the time after the most recent dose event record (any record with EVID equal to 1 or 4). self.tad() will return -1 when it is called before the first dose record within an individual (NEWIND <= 1).

This function should be called in $MAIN every time $MAIN is called.

For example:

```
$MAIN

double TAD = self.tad();
```

Do not make this calculation depend on any test or condition; it must be called every time $MAIN is called in order to see every dose.

### 2.3.22 `self.mtime(<time>)`

This is a function that creates a modeled even time. The only argument is a numeric time in the future for the current individual that indicates when a discontinuity should be included in the simulation.

When when `self.mtime()` is called, a new record is added to the current individual's record set with `EVID` set to 2. This means that when the system advances to this record (this time), the differential equation solver will reset and restart, creating the discontinuity. The function returns the time of this event so the user can work with it in subsequent code.

For example,

```
$PARAM change_point = 5.13;

$MAIN

double KA = 1.1;

double mt1 = self.mtime(change_point);

if(TIME >= mt1) KA = 2.1;
```

### 2.3.23 `self.mevent(<time>, <evid>)`

Related to `self.mtime()` (Section 2.3.22), except you can set a specific `EVID` for this intervention and you track the change time via `EVID` rather than the time. You'd use this if you want to anchor several events in the future and be sure you can distinguish between them. For example

```
[ main ]
self.mevent(change_point1, 33);
self.mevent(change_point2, 34);
```

Now, you can look for `EVID==33` to know you have hit `change_point1` or `EVID==34` to indicate that you have hit `change_point2`. Notice that `self.mevent()` doesn't return any value. You will know that you've hit the change time by checking `EVID`.

### 2.3.24 `self.stop()`

This `self` function is available to be called from `$PREAMBLE`, `$MAIN`, and `$TABLE`. When this function is called, the entire problem is stopped upon processing the next simulation record.

This might be called when something really bad happened and you just want to stop the simulation with an error.

### 2.3.25 `self.stop_id()`

This `self` function is available to be called from `$PREAMBLE`, `$MAIN`, and `$TABLE`. When this function is called, processing of the current individual is stopped and missing values (`NA_real`) are filled in for remaining compartment' and capture outputs.

This might be called when some condition is reached in the current individual that indicates either that the rest of the outputs are inconsequential or there was a problem with this particular individual.

See also `self.stop_id()` and `self.stop()`.

### 2.3.26 `self.stop_id_cf()`

This `self` function is available to be called from `$PREAMBLE`, `$MAIN`, and `$TABLE`. When this function is called, processing of the current individual is stopped and current values are carried forward (`cf`) for the remaining output records for that individual.

This might be called when some condition is reached in the current individual that indicates either that the rest of the outputs are inconsequential or there was a problem with this particular individual.

See also `self.stop_id_cf()` and `self.stop()`.

### 2.3.27 `THETA(n)`

`THETA(n)` is ubiquitous in NONMEM control streams, representing estimated fixed effect parameters. mrgsolve does not attach any special meaning to `THETA(n)` but starting with version 1.0.0 it will translate `THETA(n)` to `THETAn`. This is useful when importing NONMEM estimates using `$NMXML` or `$NMEXT`, where `THETA`s are imported as parameters with names `THETA1`, `THETA2`, `THETA3` and so on (see Section 2.2.24 and Section 2.2.25). `THETA(n)` is simply a NONMEM-oriented style for referring to `THETAn`.

### 2.3.28 ETA(n)

ETA(n) is the value of the subject-level variate drawn from the model OMEGA matrix. ETA(1) through ETA(25) have default values of zero so they may be used in a model even if appropriate OMEGA matrices have not been provided.

For example:

```
$OMEGA
1 2 3

$MAIN
double CL = TVCL*exp(ETA(1));
double VC = TVVC*exp(ETA(2));
double KA = TVKA*exp(ETA(3));
```

Here, we have a 3x3 OMEGA matrix. ETA(1), ETA(2), and ETA(3) will be populated with variates drawn from this matrix. ETA(4) through ETA(25) will be populated with zero.

### 2.3.29 EPS(n)

EPS(n) holds the current value of the observation-level random variates drawn from SIGMA. The basic setup is the same as detailed in ETA(n).

Example:

```
[ CMT ] CENT

[ PARAM ] CL=1, VC=20

[ SIGMA ]
25 0.0025

[ TABLE ]
double DV = (CENT/VC)*(1+EPS(2)) + EPS(1);
```

### 2.3.30 SIGMA(n)

Starting with version 1.0.8, users have read-only access to on-diagonal elements of $\Sigma$ through the SIGMA() macro. For example

```
$SIGMA 0.1 12

$TABLE
double STD = sqrt(SIGMA(1) + pow(F,2)*SIGMA(2));
```

### 2.3.31 table(<name>)

This macro has been deprecated. Users should **not** use code like this:

```
[ TABLE ]
table(CP) = CENT/VC;
```

But rather this:

```
$TABLE
double CP = CENT/VC;

$CAPTURE CP
```

See: Section 2.2.12 and also Section 2.2.16.

### 2.3.32 F_CMT

For the CMT compartment, sets the bioavailability fraction for that compartment.

Example:

```
$MAIN
F_CENT = 0.7;
```

### 2.3.33 ALAG_CMT

For the CMT compartment, sets the lag time for doses into that compartment.

Example:

```
$MAIN
ALAG_GUT = 0.25;
```

### 2.3.34 R_CMT

For the CMT compartment, sets the infusion rate for that compartment. The infusion rate is only set via R_CMT when rate in the data set or event object is set to -1.

Example:

```
$MAIN
R_CENT = 100;
```

### 2.3.35 Rn

Only available when the nm-vars plugin is invoked; see R_CMT.

### 2.3.36 D_CMT

For the CMT compartment, sets the infusion duration for that compartment.
The infusion duration is only set via D_CMT when rate in the data set or event object is set to -2.

Example:

```
$MAIN
D_CENT = 2;
```

### 2.3.37 NONMEM-like syntax

There is a NONMEM-like syntax that lets you write F1 rather than F_GUT, D2 rather than D_CENT as well as other variables that are commonly used in a NONMEM control stream. To make this syntax available, the nm-vars plugin must be invoked.

See Section 9.2 on the nm-vars plugin for details.

## 2.4 Reserved words

Reserved words cannot be used as names for parameters, compartments or other derived variables in the model. **Note** that some of these words are "reserved" for you to use in your data set.

```
ID
amt
cmt
ii
ss
evid
addl
rate
time
SOLVERTIME
table
ETA
EPS
AMT
CMT
ID
TIME
EVID
simeps
self
simeta
NEWIND
DONE
CFONSTOP
DXDTZERO
CFONSTOP
INITSOLV
_F
_R
_ALAG
SETINIT
report
_VARS_
VARS
SS_ADVANCE
AMT
CMT
II
SS
ADDL
RATE
THETA
```

```
pred_CL
pred_VC
pred_V
pred_V2
pred_KA
pred_Q
pred_VP
pred_V3
double
int
bool
capture
```

Other reserved words depend on the compartment names in your model. For example, if you have a compartment called `CENT` in the model, then the the following will be reserved

```
F_CENT
R_CENT
D_CENT
ALAG_CENT
N_CENT
```

## 2.5 Derive new variables

New `C++` variables may be derived in `$GLOBAL`, `$PREAMBLE` `$MAIN`, `$ODE` and `$TABLE`. Because these are `C++` variables, the type of variable being used must be declared. For the vast majority of applications, the `double` type is used (double-precision numeric value).

```
$MAIN

double CLi = TVCL*exp(ETA(1));
```

We want `CLi` to be a numeric value, so we use `double`. To initialize a `boolean` variable (true / false), write

```
$MAIN
bool cure = false;
```

### 2.5.1 Special handling for `double`, `int`, `bool`

When variables of the type `double`, `int`, and `bool` are declared and initialized in `$PREAMBLE`, `$MAIN`, `$ODE`, `$TABLE`, mrgsolve will detect those declarations, and modify the code so that the variables are actually declared once in `$GLOBAL` not in `$MAIN`, `$ODE`, or `$TABLE`. This is done so that variables declared in one code block (e.g. `$MAIN`) can be read and modified in another code block (e.g. `$TABLE`).

For example, in the following code:

```
$MAIN
double CLi = TVCL*exp(ETA(1));
```

a double-precision numeric variable is created (`CLi`) in the `$MAIN` block. When mrgsolve parses the model file, this code gets translated to

```
$GLOBAL
namespace {
  double CLi;
}

$MAIN
CLi = TVCL*exp(ETA(1));
```

That is, `CLi` is declared in `$GLOBAL` in an unnamed namespace so that variables like this are global variables within the model file only.

This way, we can still read the `CLi` variable in `$TABLE`:

```
$MAIN
double CLi = TVCL*exp(ETA(1));
double VCi = TVVC*exp(ETA(2));

$TABLE
double KEi = CLi/VCi;

$CAPTURE KEi
```

To declare a variable that is local to a particular code block:

```
$MAIN

localdouble CLi = TVCL*exp(ETA(1));
```

The `localdouble` type is still just a double-precision variable. The difference is that it is protected from this re-declaration process and the variable will be local to (in this case) the `$MAIN` block.

### 2.5.2 Initializing and resetting

mrgsolve takes a hands-off approach to variables created by the user; it will not initialize or reset variables from record to record or from individual to individual. Failing to properly initialize or reset user variables can have unintended consequences, especially when conditional logic is used.

Generally, a default value for all variables should be set prior to conditional assignment based on some test condition.

### 2.5.3 Using other types globally

As we noted in the previous section, `double`, `int`, and `bool` are processed in a special way so that they are by default global to the file. Many times we want to work with other variable types in a global manner. Whenever you want a data structure to be accessible across functions (e.g. `$MAIN`, `$TABLE`, etc.) they should be declared in `$GLOBAL`, optionally in an unnamed namespace.

For example:

```
[ GLOBAL ]
std::vector<double> myvec;
```

or

```
[ GLOBAL ]
namespace {
  std::vector<double> myvec;
}
```

In case that object needs some configuration prior to starting the problem, use `$PREAMBLE` to do that work

```
[ GLOBAL ]
std::vector<double> myvec;

[ PREAMBLE ]
myvec.assign(3,1);
```

## 2.6 Random number generation

Users can simulate random numbers inside a model file using functions that are similar to the functions you'd normally use in R (e.g. `rnorm()` and `runif()`). This functionality is provided by `Rcpp` and therefore requires using the `Rcpp` plugin (see Chapter 9 and Section 9.6).

`Rcpp` provides these functions inside the `R` namespace so you will have to prefix the function call with `R::`.

As an example, to make a draw from Uniform (0,1)

```
[ plugin ] Rcpp

[ error ]
double draw = R::runif(0,1);
```

Note that the `0` gets used as `min` and the `1` gets used as `max`;
we didn't pass `n` here and `draw` is a single number (not a vector like you'd get from (`runif(100, 0, 1)` on your R console). So in general, these functions work like their `R` counterparts, but without the `n` argument.

Another example showing how to draw from binomial distribution with probability 0.5

```
[ plugin ] Rcpp

[ error ]
double draw = R::rbinom(1, 0.5);
```

Here, the `1` is used as `size` (not `n`) and `0.5` is used as `prob`.

Other helpful functions could be `R::rnorm()` or `R::rlnorm()` but you can call any of the `r` functions as well as the corresponding `dpq` functions through this `R` namespace.

Documentation, including functions to call and arguments can be found in the Rcpp API docs

http://dirk.eddelbuettel.com/code/rcpp/html/namespaceR.html

## 2.7 Examples

The following sections show example model specification. The intention is to show how the different blocks, macros and variables can work together to make a functional model. Some models are given purely for illustrative purpose and may not be particularly useful in application.

### 2.7.1 Simple PK model

Notes:

- Basic PK parameters are declared in `$PARAM`; every parameter needs to be assigned a value
- Two compartments `GUT` and `CENT` are declared in `$CMT`; using `$CMT` assumes that both compartments start with 0 mass
- Because we declared `GUT` and `CENT` as compartments, we write `dxdt_` equations for both in `$ODE`
- In `$ODE`, we refer to parameters (`CL/VC/KA`) and the amounts in each compartment at any particular time (`GUT` and `CENT`)
- `$ODE` should be `C++` code; each line ends in `;`
- We derive a variable called `CP` in `$TABLE` that has type `capture`; mrgsolve will enter the `CP` name into the `$CAPTURE` block list

```
$PARAM CL = 1, VC = 30, KA = 1.3

$CMT GUT CENT

$ODE

dxdt_GUT = -KA*GUT;
dxdt_CENT = KA*GUT - (CL/VC)*CENT;

$TABLE
capture CP = CENT/VC;
```

This model can also be written without differential equations

```
[ PARAM ] CL = 1, VC = 30, KA = 1.3

[ PKMODEL ] cmt = "CMT GUT CENT", depot = TRUE

$TABLE
capture CP = CENT/VC;
```

### 2.7.2 PK/PD model

Notes:

- We use a preprocessor `#define` directive in `$GLOBAL`; everywhere in the model where a `CP` token is found, the expression `(CENT/VC)` … with parentheses … is inserted
- We write the initial value for the `RESP` compartment in `$MAIN` as a function of two parameters `KIN/KOUT`
- A new variable - `INH`- is declared and used in `$ODE`
- Since `CP` is defined as `CENT/VC`, we can "capture" that name/value in `$CAPTURE`
- Both `$MAIN` and `$ODE` are `C++` code blocks; don't forget to add the `;` at the end of each statement

```
$PARAM CL = 1, VC = 30, KA = 1.3
KIN = 100, KOUT = 2, IC50 = 2

$GLOBAL
#define CP (CENT/VC)

$CMT GUT CENT RESP

$MAIN
RESP_0 = KIN/KOUT;

$ODE

double INH = CP/(IC50+CP);

dxdt_GUT = -KA*GUT;
dxdt_CENT = KA*GUT - (CL/VC)*CENT;
dxdt_RESP = KIN*(1-INH) - KOUT*RESP;

$CAPTURE CP
```

### 2.7.3 Population PK model with covariates and IOV

Notes:

- Use `$SET` to set the simulation time grid from 0 to 240 by 0.1
- There are two `$OMEGA` matrices; we name them `IIV` and `IOV`
- The IIV "etas" are labeled as `ECL/EVC/EKA`; these are aliases to `ETA(1)/ETA(2)/ETA(3)`. The `IOV` matrix is unlabeled; we must refer to `ETA(4)/ETA(5)` for this
- Because `ETA(1)` and `ETA(2)` are labeled, we can "capture" them as `ECL` and `EVC`
- We added zeros for both `$OMEGA` matrices; all the etas will be zero until we populate those matrices (Section 5.3)

```
$PARAM TVCL = 1.3, TVVC=28, TVKA=0.6, WT=70, OCC=1

$SET delta=0.1, end=240

$CMT GUT CENT

$MAIN

double IOV = IOV1
if(OCC==2) IOV = IOV2;

double CLi = exp(log(TVCL) + 0.75*log(WT/70) + ECL + IOV);
double VCi = exp(log(TVVC) + EVC);
double KAi = exp(log(TVKA) + EKA);

$OMEGA @name IIV @labels ECL EVC EKA
0 0 0
$OMEGA @name IOV @labels IOV1 IOV2
0 0

$SIGMA 0

$ODE
dxdt_GUT = -KAi*GUT;
dxdt_CENT = KAi*GUT - (CLi/VCi)*CENT;

$TABLE
capture CP = CENT/VCi;

$CAPTURE IOV ECL EVC
```

# 3 Input data sets

Input data sets are used in mrgsolve to allow the user to specify interventions and input data items.

Please see the mrgsolve help topic `?exdatasets` for examples of all of the data sets discussed in this chapter. The example data sets are embedded in the mrgsolve package and may be used at any time.

## 3.1 Overview

Data sets are the primary mechanism for establishing the scope of your simulations in `mrsolve`, including individuals, interventions, observation times, and parameter values. For both `data_set` and `idata_set` (see below), you may include columns in the data sets that have the same names as the parameters in your model (Section 1.1, Section 2.2.4). mrgsolve can recognize these columns and update the parameter list as the simulation proceeds. This process is of key importance when planning and executing complex simulations and is further discussed in section Section 11.3.

## 3.2 Event data sets (`data`)

Event data sets are entered as `data.frame`, with one event per row. Events may be observations, doses, or other type events. In mrgsolve documentation, we refer to these data sets as `data` or `data_set` (after the function that is used to associate the data set with the model object prior to simulation).

Event data sets have several special column names that mrgsolve is always aware of:

- `ID` the subject id. This id does not need to be unique in the `data_set`: mrgsolve detects an new individual when the current value of `ID` is different from the immediate preceding value of `ID`. However, we always recommend using unique `ID`.
- `time` or `TIME`: states the time of the data record
- `evid` or `EVID`: the event id indicator. `evid` can take the values:
  - **0** = observation record

  - **1** = dosing event (bolus or infusion)
  - **2** = other type event, with solver stop and restart
  - **3** = system reset
  - **4** = reset and dose
  - **8** = replace the amount in the compartment with `amt`

- `amt` or `AMT`: the dose amount (if `evid==1`)
- `cmt` or `CMT`: the dosing compartment number. This may also be a character value naming the compartment name. The compartment number must be consistent with the number of compartments in the model for dosing records (`evid`==1).
  For observation records, a `cmt` value of 0 is acceptable. Use a negative compartment number with `evid` 2 to turn a compartment off.

- `rate` or `RATE`: if non-zero and `evid=1` or `evid=4`, implements a zero-order infusion of duration `F_CMT*amt/rate`, where `F_CMT` is the bioavailability fraction for the dosing compartment. Use `rate = -1` to model the infusion rate and `rate = -2` to model the infusion duration, both in `$MAIN` (see Section 2.2.8, Section 2.3.32, Section 2.3.34, Section 2.3.36).
- `ii` or `II`: inter-dose interval; `ii=24` means daily dosing when the model time unit is hours
- `addl` or `ADDL`: additional doses; a non-zero value in `addl` requires non-zero `ii` on the same record
- `ss` or `SS` steady state indicator; use 1 to implement steady-state dosing; 0 otherwise. mrgsolve also recognizes dosing records where `ss=2`. This allows combination of different steady state dosing regimens under linear kinetics (e.g. 10 mg QAM and 20 mg QPM daily to steady state).

The column names in the preceding list were written either as lower case form (e.g. `amt`) or upper case form (e.g. `AMT`). Either may be used, however the data set will be checked for consistency of usage. An error will be issued in case a mixture of lower and upper case names are found.

In addition to these special column names, mrgsolve will recognize columns in `data_set` that have the same name as items in the parameter list (see Section 2.2.4 and Section 1.1). When mrgsolve sees that the names match up, it will update the values of those matching names based on what it finds as it moves through the data set (see Section 11.3).

### 3.2.1 Two types of `data_set`

mrgsolve distinguishes between two types of data sets: data sets that have *at least one observation record* (`evid=0`) and data sets that have *no records with* `evid=0`.

- **Full data sets** have a mix of observations and dosing events (likely, but not required). When mrgsolve finds one record with `evid=0`, it assumes that **ALL** output observation times are to come from the data set. In this case the simulation output time grid discussed in Section 1.3 is ignored and only observations found in the data set appear in the simulated output. Use full data sets when you want a highly customized sampling schedule or you are working with a clinical data set.
- **Condensed data sets** have no records with `evid=0`. In this case, mrgsolve will fill the simulated output with observations at times specified by the output time grid (Section 1.3 and see Section 11.5 too). These are very convenient to use because there is less data assembly burden and output data sets can easily be created with very dense sampling scheme or highly customized sampling schemes with very little work. Use a condensed data set when you want a uniform set of sampling times for all subjects in the data set.

**Example of condensed data set**

```
data(extran1, package = "mrgsolve")
extran1
```

```
.   ID  amt cmt time addl ii rate evid
. 1  1 1000   1    0    3 24    0    1
. 2  2 1000   2    0    0  0   20    1
. 3  3 1000   1    0    0  0    0    1
. 4  3  500   1   24    0  0    0    1
. 5  3  500   1   48    0  0    0    1
. 6  3 1000   1   72    0  0    0    1
. 7  4 2000   2    0    2 48  100    1
. 8  5 1000   1    0    0  0    0    1
. 9  5 5000   1   24    0  0   60    1
```

See `?exdatasets` in the `R` help system after loading mrgsolve.

**Example of full data set**

```
data(exTheoph, package = "mrgsolve")

head(exTheoph)
```

```
.   ID   WT Dose time  conc cmt  amt evid
. 1  1 79.6 4.02 0.00  0.00   1 4.02    1
. 2  1 79.6 4.02 0.25  2.84   0 0.00    0
. 3  1 79.6 4.02 0.57  6.57   0 0.00    0
. 4  1 79.6 4.02 1.12 10.50   0 0.00    0
. 5  1 79.6 4.02 2.02  9.66   0 0.00    0
. 6  1 79.6 4.02 3.82  8.58   0 0.00    0
```

See `?exdatasets` in the `R` help system after loading mrgsolve.

**Augmenting observations in a clinical data set** Occasionally, we want to simulate from a clinical data set (with observation records as actually observed in a population of patients) but we also want to augment those observations with a regular sequence of times (for example, to make a smooth profile on a plot). In that case, you can set `obsaug = TRUE` when calling `mrgsim`.

For example:

```
mod <- house()

out <-
  mod %>%
    data_set(exTheoph, ID==1) %>%
    carry.out(a.u.g) %>%
    obsaug %>%
    mrgsim(end = 24, delta = 1)

out
```

```
. Model:  housemodel
. Dim:    36 x 8
. Time:   0 to 24.37
. ID:     1
.     ID time a.u.g    GUT  CENT  RESP      DV      CP
. 1:   1 0.00     1 0.0000 0.000 50.00 0.00000 0.00000
. 2:   1 0.00     0 4.0200 0.000 50.00 0.00000 0.00000
. 3:   1 0.25     0 2.9781 1.035 49.95 0.04552 0.04552
. 4:   1 0.57     0 2.0285 1.961 49.81 0.08624 0.08624
. 5:   1 1.00     1 1.2108 2.729 49.61 0.12001 0.12001
. 6:   1 1.12     0 1.0484 2.875 49.57 0.12643 0.12643
. 7:   1 2.00     1 0.3647 3.422 49.34 0.15048 0.15048
. 8:   1 2.02     0 0.3560 3.428 49.33 0.15072 0.15072
```

```
out %>% select(time) %>% unlist() %>% unname()
```

```
.  [1]  0.00  0.00  0.25  0.57  1.00  1.12  2.00  2.02  3.00  3.82  4.00  5.00
. [13]  5.10  6.00  7.00  7.03  8.00  9.00  9.05 10.00 11.00 12.00 12.12 13.00
. [25] 14.00 15.00 16.00 17.00 18.00 19.00 20.00 21.00 22.00 23.00 24.00 24.37
```

`obsaug` requests that the data set be augmented with observations from the simulation time grid. We can optionally request an indicator called `a.u.g` to appear in the output that takes value of 1 for augmented observations and 0 for observations from the data set.

### 3.2.2 Sorting requirements

The `ID`s in the data set can appear in any order. However, an error will be generated if `time` on any record is less that `time` on the previous record within any `ID`.

### 3.2.3 Creating data sets

The `expand.ev` function is provided by mrgsolve to help in creating data sets of a certain style. But any `R` code that produces a valid data set is fine to use.

### 3.2.4 Example

To create a data set of 3 people each receiving **250 mg every 8 hours for 12 total doses**:

```
data <- expand.ev(ID=1:3, amt=250, ii=8, addl=11)

data
```

```
.   ID time amt ii addl cmt evid
. 1  1    0 250  8   11   1    1
. 2  2    0 250  8   11   1    1
. 3  3    0 250  8   11   1    1
```

Notice that `expand.ev` assumes that `time` is 0 and `cmt` is 1. To dose as a 2-hour infusion into the second compartment use:

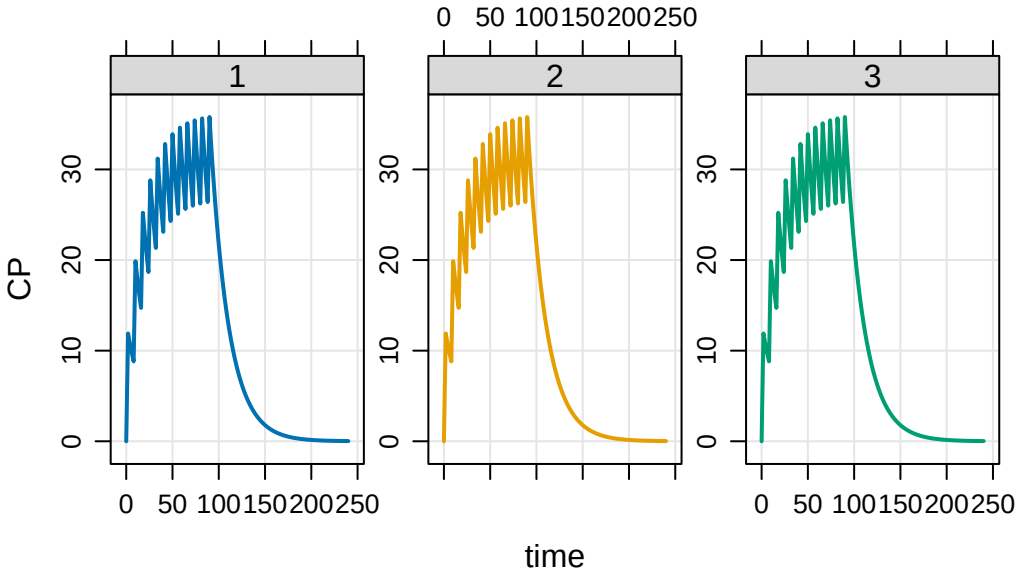
```
data <- expand.ev(ID=1:3, amt=250, rate=125, ii=8, addl=11, cmt=2)

data
```

```
.   ID time amt rate ii addl cmt evid
. 1  1    0 250  125  8   11   2    1
. 2  2    0 250  125  8   11   2    1
. 3  3    0 250  125  8   11   2    1
```

Use `data_set` to pass the data into the problem.

For example:

```
mod <- house()

mod %>%
  data_set(data) %>%
  mrgsim(end=240) %>%
  plot(CP~time|factor(ID))
```




## 3.3 Individual data sets (`idata`)

Individual data sets carry individual-level data. This individual data is used in several different ways:

- **Individual-level parameters**: Just prior to simulating any individual, mrgsolve checks the appropriate row in `idata` (if supplied) for any columns with parameter names. If parameter names are found, the parameter list is updated and that update remains in effect for the duration of that individual's data records.

- **Individual- or group-level designs**: Each individual or group of individual may be assigned a different sampling design. For example, individuals in arm 1 may need to be simulated for 4 weeks whereas individuals in arm 2 may need to be simulated for 8 weeks. `idata` may be used to identify one of several sampling designs for each individual or group of individuals.
- **Individual-level compartment initialization**: if a model has a compartment called `CMT` and mrgsolve finds a column in `idata` called `CMT_0`, the value of `CMT_0` will be used to initialize that compartment with, potentially a different value for each individual. Note that there are several other ways to initialize compartments detailed in Section 11.2.

`idata_set` are entered as `data.frame` with one unique `ID` per row. In mrgsolve documentation, we refer to individual data sets `idata` or `idata_set` to distinguish them from event data sets (see Section 3.2).

An `idata_set` looks like this:

```
data(exidata)

exidata
```

```
.    ID    CL    VC     KA  KOUT  IC50 FOO
. 1   1 1.050 47.80 0.8390 2.450 1.280   4
. 2   2 0.730 30.10 0.0684 2.510 1.840   6
. 3   3 2.820 23.80 0.1180 3.880 2.480   5
. 4   4 0.552 26.30 0.4950 1.180 0.977   2
. 5   5 0.483  4.36 0.1220 2.350 0.483  10
. 6   6 3.620 39.80 0.1260 1.890 4.240   1
. 7   7 0.395 12.10 0.0317 1.250 0.802   8
. 8   8 1.440 31.20 0.0931 4.030 1.310   7
. 9   9 2.570 18.20 0.0570 0.862 1.950   3
. 10 10 2.000  6.51 0.1540 3.220 0.699   9
```

Here we have an `idata_set` with 10 subjects, one subject per row. The `ID` column connects the data in each row to the data in a `data_set`, which also requires an `ID` column.

The `ID` column is the only required column name in `idata_set` and `ID` should always be a unique identifier for that row.

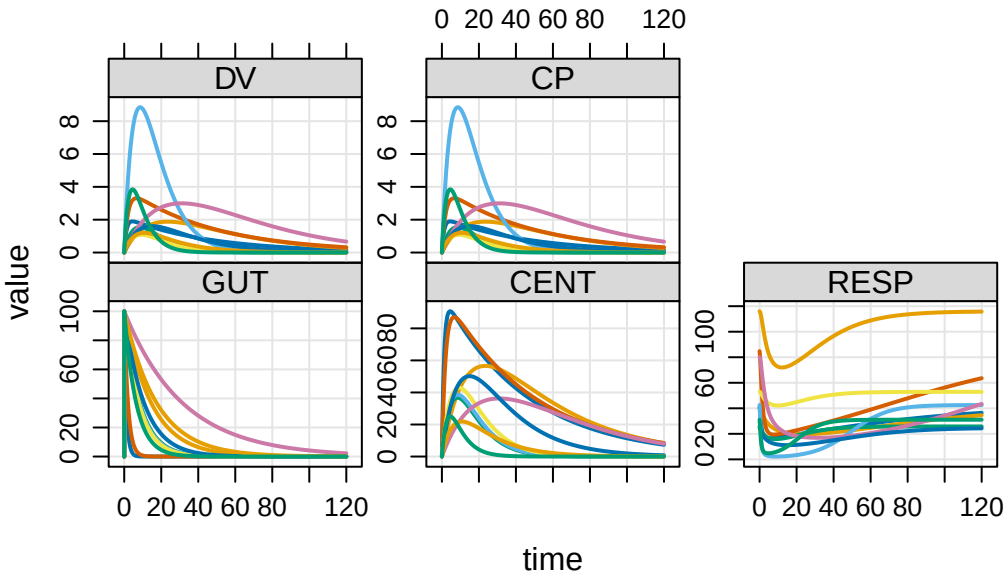
### 3.3.1 Use case

`idata_set` is usually helpful for implementing a batch of simulations when a `data_set` is not used. The batch may be as a sensitivity analysis or for population simulation. Usually, an events object is used with `idata`, but it is not required.

Use the `idata_set` function to pass the data set into the problem.

For example:

```
mod <- house()

mod %>%
  idata_set(exidata) %>%
  ev(amt = 100) %>%
  mrgsim() %>%
  plot()
```




Because there were 10 subjects in the `idata_set`, we get 10 profiles in the output. Each "individual" or "unit" received the same 100 mg dose. We would use a `data_set` to assign different doses to different individuals.

## 3.4 Numeric data only

The `data.frame` holding the `data_set` or `idata_set` may have any type of data in its columns. However, only numeric data can actually get passed into the simulation engine. mrgsolve will automatically look for non-numeric columns and drop them from the `data_set` or `idata_set` with a warning.

## 3.5 Missing values

If missing values (`NA`) are found in columns which match parameter names, a warning will be issued and the user should expect `NaN` in the simulated output if these parameters factor into the advance of the system.

Starting with version 1.0.8, missing values in the following columns of input `data_sets` will be silently converted to `0`

- `CMT`
- `AMT`
- `RATE`
- `EVID`
- `II`
- `ADDL`
- `SS`

The lower case versions of these names may also include `NA` and will get converted to `0`

- `cmt`
- `amt`
- `rate`
- `evid`
- `ii`
- `addl`
- `ss`

`idata_sets` are not checked for missing values in these columns.

## 3.6 Data set validation

A the time of simulation, mrgsolve will validate the input data set, removing non-numeric columns, checking for missing values in parameter columns, checking compartment numbers, etc.

Users can pre-validate the data set so that this does not need to happen at run time.

```
data(exTheoph)

head(exTheoph)
```

```
.   ID   WT Dose time  conc cmt  amt evid
. 1  1 79.6 4.02 0.00  0.00   1 4.02    1
. 2  1 79.6 4.02 0.25  2.84   0 0.00    0
. 3  1 79.6 4.02 0.57  6.57   0 0.00    0
. 4  1 79.6 4.02 1.12 10.50   0 0.00    0
. 5  1 79.6 4.02 2.02  9.66   0 0.00    0
. 6  1 79.6 4.02 3.82  8.58   0 0.00    0
```

```
mod <- modlib("pk1", compile = FALSE)

valid <- valid_data_set(exTheoph, mod)

str(valid)
```

```
.  'valid_data_set' num [1:132, 1:9] 1 1 1 1 1 1 1 1 1 1 ...
.  - attr(*, "dimnames")=List of 2
.   ..$ : NULL
.   ..$ : chr [1:9] "ID" "WT" "Dose" "time" ...
```

This can improve efficiency when performing a very large number of replicate simulations on the same data set, but is unlikely to provide a meaningful speed-up for a single simulation or a small number of simulations.

## 3.7 Data sets for use with `$PRED`

Because there are no compartments involved, there are relaxed data set requirements for models that utilize `$PRED`.

- `time` or `TIME` is not required as input; when this is not supplied, a `time` column will be included in output with value 0
- When `time` or `TIME` is supplied, it may be negative; but records must still be sorted by `time` or `TIME`
- If supplied, `cmt` or `CMT` must be zero
- An error is generated if `rate` or `RATE` is supplied
- An error is generated if `ss` or `SS` is supplied

# 4 Event objects

Event objects are similar to the data sets described in Chapter 3, but are simpler and easier to create. This is the fastest way to implement a basic intervention (like dosing) for a single "individual" into your model.

Event objects also offer an elegant way to compose complicated dosing regimens. Typically, the different parts of a regimen are composed as individual event objects and then combined to create a multi-faceted dose regiment.

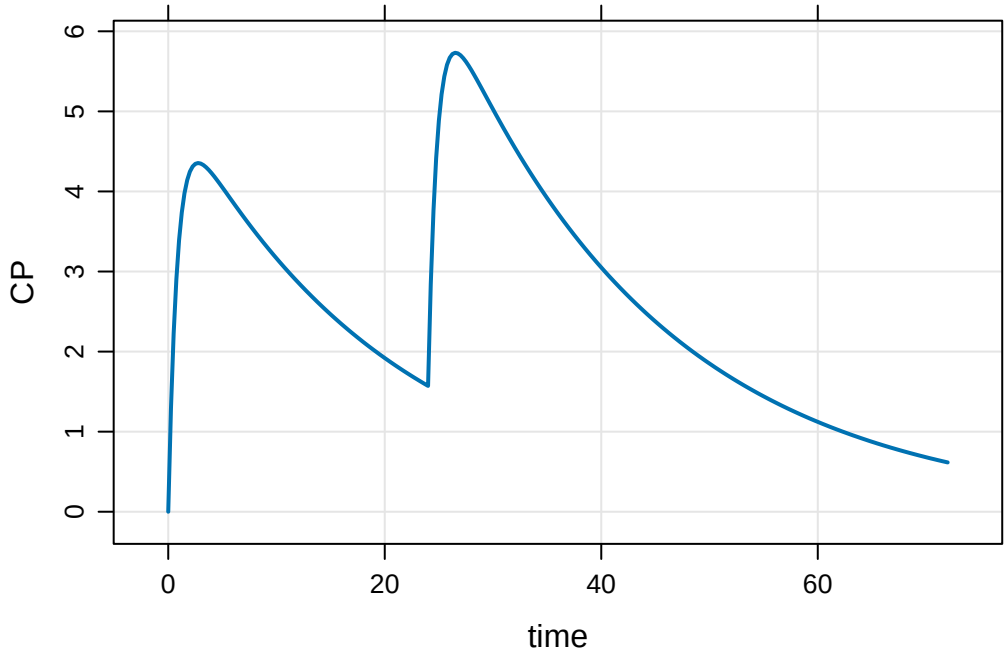
Finally, once an event object is created (either simple or complex), this object can be "expanded" into multiple individuals to create a population data set for simulation.

See details in the subsequent sections.

## 4.1 Usage

Event objects are frequently used in a pipeline to simulate a dosing regimen. For example

```
mod <- house(end = 72)

mod %>% ev(amt = 100, ii = 24, addl = 1) %>% mrgsim() %>% plot("CP")
```




This used the `ev()` constructor to make an event object for two 100 mg doses and this is passed into `mrgsim()` to implement this regimen.

Alternatively, we can create a standalone object containing the event information

```
regimen <- ev(amt = 100, ii = 24, addl = 1)
```

and pass that into the simulation pipeline

```
mod %>% ev(regimen) %>% mrgsim() %>% plot("CP")
```

If you are not using the pipe syntax, the following would be equivalent calls

```
mrgsim(mod, events = regimen) %>% plot("CP")
```

And there are `mrgsim()` variants that explicitly accept an event object

```
mrgsim_e(mod, regimen) %>% plot("CP")
```

More will be said about how to create and manipulate event objects in the following sections.

## 4.2 Construction

A new event object can be created with the `ev()` constructor. For a single, 100 mg dose it would be

```
e <- ev(amt = 100)
```

When you print the object to the R console we see the 100 mg dose along with the following defaults

- `time` set to 0
- `cmt` set to 1 (the first compartment)
- `evid` set to 1 (a bolus dose)

```
e
```

```
. Events:
.   time amt cmt evid
. 1    0 100   1    1
```

Of course, we can override any of these defaults or add additional items as needed. For a single 100 mg dose infused over 2 hours in compartment 2 one hour after the simulation starts

```
e <- ev(amt = 100, rate = 50, cmt = 2, time = 1)
```

To use this event object, we can pass it into `mrgsim()` under the `events` argument

```
mod <- house(delta = 1, end = 24)

mrgsim(mod, events = e)
```

```
. Model:  housemodel
. Dim:    26 x 7
. Time:   0 to 24
. ID:     1
.     ID time GUT  CENT  RESP    DV    CP
. 1:   1    0   0  0.00 50.00 0.000 0.000
. 2:   1    1   0  0.00 50.00 0.000 0.000
. 3:   1    1   0  0.00 50.00 0.000 0.000
. 4:   1    2   0 48.77 44.12 2.439 2.439
. 5:   1    3   0 95.16 36.98 4.758 4.758
```

```
. 6:   1    4   0 90.52 34.61 4.526 4.526
. 7:   1    5   0 86.11 34.75 4.305 4.305
. 8:   1    6   0 81.91 35.22 4.095 4.095
```

Event object inputs can be functions of previously defined inputs. For example

```
ev(amt = 100, rate = amt / 2)
```

```
. Events:
.   time amt rate cmt evid
. 1    0 100   50   1    1
```

See the `?ev()` help topic for more information on additional arguments when constructing event objects. Here, I'd like to specifically highlight a handful of options that can be helpful when constructing event objects.

### Infusion duration

Above, we created some infusion event objects by adding an infusion rate to the input. We can also indicate an infusion by adding an infusion time through the `tinf` argument

```
ev(amt = 100, tinf = 2)
```

```
. Events:
.   time amt rate cmt evid tinf
. 1    0 100   50   1    1    2
```

### ID

While the primary use case for event objects are for *single individuals*, we can code a series of IDs into the object too

```
ev(amt = 100, ID = 1:3)
```

```
. Events:
.   ID time amt cmt evid
. 1  1    0 100   1    1
. 2  2    0 100   1    1
. 3  3    0 100   1    1
```

Here, we asked for 3 IDs in the object. Once this is turned into a simulation data set (see below), we'll have a population data set from which to simulate.

**Additional data items**

We can also pass through arbitrary data columns through the event object. For example, we can pass through `WT`

```
ev(amt = 100, WT = 80)
```

```
. Events:
.   time amt cmt evid WT
. 1    0 100   1    1 80
```

## 4.3 Coerce to data set

As we noted, event objects are very similar to data sets and they are nothing but data sets under the hood. We can take the event objects we created above and coerce them to other objects.

**Using as_data_set**

```
as_data_set(e)
```

```
.   time amt rate cmt evid ID
. 1    1 100   50   2    1  1
```

This will ensure that there is an `ID` column in the output and it will be suitable to use for simulation.

**Using as.data.frame**

```
as.data.frame(e) %>% mutate(ID = 5)
```

```
.   time amt rate cmt evid ID
. 1    1 100   50   2    1  5
```

**Upper case names**

See Section 4.8 for a constructor for an event object that renders with upper case names.

## 4.4 Extract information

There is a `$` operator for event objects

```
e$amt
```

```
. [1] 100
```

## 4.5 Combining event objects

### 4.5.1 Concatenate

Two or more event objects can be concatenated using the `c` operator

```
e1 <- ev(amt = 100)
e2 <- ev(amt = 200, time = 24)

c(e1, e2)
```

```
. Events:
.   time amt cmt evid
. 1    0 100   1    1
. 2   24 200   1    1
```

This essentially "rbinds" the rows of the individual event objects and sorts the rows by `time`.

**NOTE**: the result of this manipulation is another event object.

### 4.5.2 Sequence

Event objects can also be combined to happen in a sequence. In the previous example, we wanted the 200 mg to happen at 24 hours and we had to code that fact into `time` accordingly.

By specifying a dosing interval (`ii`) we can ask mrgsolve to do that automatically by calling the `seq()` method.

```
e1 <- ev(amt = 100, ii = 24)
e2 <- ev(amt = 200, ii = 24)

seq(e1, e2)
```

```
. Events:
.   time amt ii addl cmt evid
. 1    0 100 24    0   1    1
. 2   24 200 24    0   1    1
```

This was a trivial example to get a simple result. We can try something more complicated to make the point

```
e3 <- ev(amt = 100, ii = 6,  addl = 28)
e4 <- ev(amt = 200, ii = 12, addl = 124)
e5 <- ev(amt = 400, ii = 24, addl = 3)

seq(e3, e4, e5)
```

```
. Events:
.   time amt ii addl cmt evid
. 1    0 100  6   28   1    1
. 2  174 200 12  124   1    1
. 3 1674 400 24    3   1    1
```

**NOTE**: when mrgsolve puts event objects into a sequence, it starts the next segment of the regimen one dosing interval after the previous regimen finished. Going back to the simple example

```
seq(e1, e2)
```

```
. Events:
.   time amt ii addl cmt evid
. 1    0 100 24    0   1    1
. 2   24 200 24    0   1    1
```

`e1` was just a single dose at time 0. mrgsolve will have `e2` start one dosing interval (24 hours) after the last (only) dose in `e1`. We can alter the amount of time between segments of the regimen by using the `wait` argument. For example, to push `e2` out by an additional 24 hours we'd use

```
seq(e1, wait = 24, e2)
```

```
. Events:
.   time amt ii addl cmt evid
. 1    0 100 24    0   1    1
. 2   48 200 24    0   1    1
```

We can also use a negative value for `wait` to make the next dose happen sooner

```
seq(e1, wait = -12, e2)
```

```
. Events:
.   time amt ii addl cmt evid
. 1    0 100 24    0   1    1
. 2   12 200 24    0   1    1
```

Finally, we should note that event objects can be used multiple times in a sequence

```
seq(e1, e2, wait = 7*24, e2, e1)
```

```
. Events:
.   time amt ii addl cmt evid
. 1    0 100 24    0   1    1
. 2   24 200 24    0   1    1
. 3  216 200 24    0   1    1
. 4  240 100 24    0   1    1
```

### 4.5.3 repeat

Like the `seq()` method for event objects, `ev_repeat` will put an event object into a sequence `n` times

```
ev_repeat(e1, n = 3)
```

```
.   time amt ii cmt evid addl
. 1    0 100 24   1    1    0
. 2   24 100 24   1    1    0
. 3   48 100 24   1    1    0
```

By default, this function returns a regular data frame. To return an event object instead call

```
ev_repeat(e1, n = 3, as.ev = TRUE)
```

You can put a waiting period too. To illustrate this, let's compose a more complicated regimen and repeat that

```
e1 <- ev(amt = 500, ii = 24)
e2 <- ev(amt = 250, ii = 24, addl = 5)
e3 <- ev_seq(e1, e2)

e3 %>% realize_addl()
```

```
. Events:
.   time amt ii addl cmt evid
. 1    0 500  0    0   1    1
. 2   24 250  0    0   1    1
. 3   48 250  0    0   1    1
. 4   72 250  0    0   1    1
. 5   96 250  0    0   1    1
. 6  120 250  0    0   1    1
. 7  144 250  0    0   1    1
```

In this regimen, we have daily dosing for 7 doses. The last dose is given at 144 hours. When putting this into a sequence, we'll wait one dosing interval and then the `wait` period and then start again

```
ev_repeat(e3, n = 3, wait = 7*24)
```

```
.   time amt ii addl cmt evid
. 1    0 500 24    0   1    1
. 2   24 250 24    5   1    1
. 3  336 500 24    0   1    1
. 4  360 250 24    5   1    1
. 5  672 500 24    0   1    1
. 6  696 250 24    5   1    1
```

### 4.5.4 Create a "data_set"

Use the `as_data_set()` function to combine multiple event objects into a single data set.

```
as_data_set(e1, e2)
```

```
.   ID time cmt evid amt ii addl
. 1  1    0   1    1 500 24    0
. 2  2    0   1    1 250 24    5
```

It's important to note that

1. The result is a regular old `data.frame()`; once you call `as_data_set()`, you exit the event object world
2. Each event object is given a different `ID`

Recall that we can create event objects with multiple `IDs`; `as_data_set()` is handy to use with this feature

```
as_data_set(
  ev(amt = 100, ID = 1:3),
  ev(amt = 200, ID = 1:3),
  ev(amt = 300, ID = 1:2)
)
```

```
.   ID time cmt evid amt
. 1  1    0   1    1 100
. 2  2    0   1    1 100
. 3  3    0   1    1 100
. 4  4    0   1    1 200
. 5  5    0   1    1 200
. 6  6    0   1    1 200
. 7  7    0   1    1 300
. 8  8    0   1    1 300
```

Notice that `as_data_set` has created unique IDs for the 3 subjects in the 100 mg group, the 3 subjects in the 200 mg group, and the 2 subjects in the 300 mg group.

We'll cover a function called `ev_rep()` below to "expand" an event object to multiple individuals

```
as_data_set(
  e1 %>% ev_rep(1:300),
  e2 %>% ev_rep(1:300)
)
```

## 4.6 Modifying an event object

### 4.6.1 Tidy-like manipulation

Event objects can be mutated

```
mutate(e, amt = 200)
```

```
. Events:
.   time amt rate cmt evid
. 1    1 200   50   2    1
```

Columns can be removed from event objects

```
ev(amt = 100, WT = 50, AGE = 12) %>% select(-WT)
```

```
. Events:
.   time amt cmt evid AGE
. 1    0 100   1    1  12
```

Rows can be removed from event objects

```
e <- c(ev(amt = 100), ev(amt = 200, time = 12), ev(amt = 300, time = 24))

filter(e, time <= 12)
```

```
. Events:
.   time amt cmt evid
. 1    0 100   1    1
. 2   12 200   1    1
```

### 4.6.2 realize_addl

"Additional" doses can be made explicit in an event object

```
ev(amt = 100, ii = 6, addl = 3) %>% realize_addl()
```

```
. Events:
.   time amt ii addl cmt evid
. 1    0 100  0    0   1    1
. 2    6 100  0    0   1    1
. 3   12 100  0    0   1    1
. 4   18 100  0    0   1    1
```

### 4.6.3 ev_rep

Event objects can be "expanded" into multiple IDs to create a population; use the `ev_rep()` function for this.

```
ev(amt = 100) %>% ev_rep(1:5)
```

```
.     ID time amt cmt evid
. 1    1    0 100   1    1
. 1.1  2    0 100   1    1
. 1.2  3    0 100   1    1
. 1.3  4    0 100   1    1
. 1.4  5    0 100   1    1
```

By default, `ev_rep()` returns a regular data frame. You can request that an event object is returned

```
ev(amt = 100) %>% ev_rep(1:5, as.ev = TRUE)
```

`ev_rep()` can work on an event object with any complexity.

## 4.7 Creative composition

mrgsolve has a couple of more creative ways to construct event objects.

### 4.7.1 ev_days

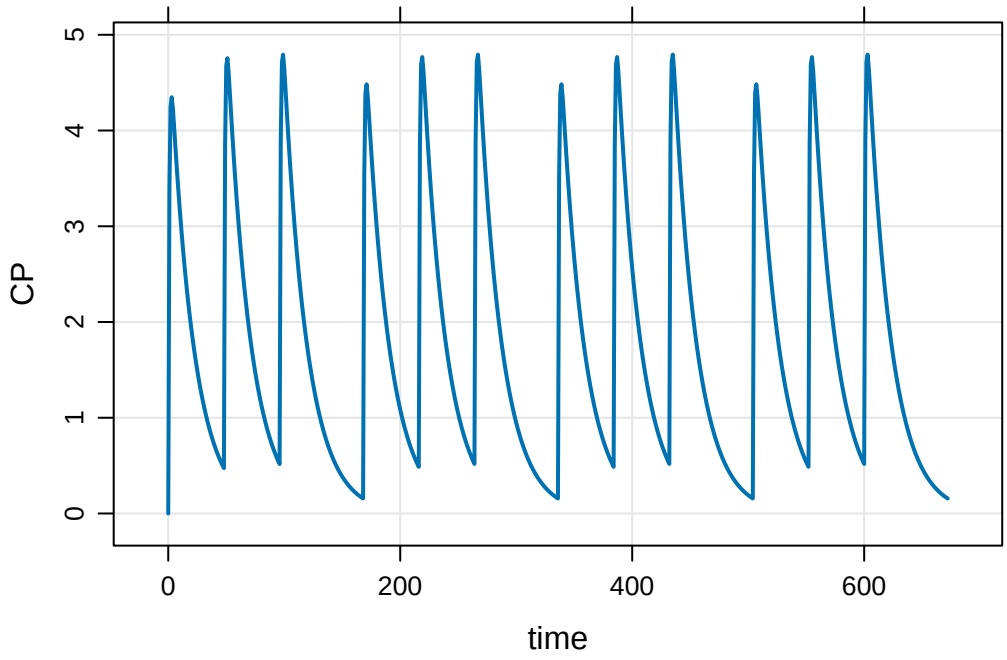
`ev_days()` will create dosing sequences when dosing are on certain days (of the week). For example, to dose only on Monday, Wednesday, and Friday for on month

```
e <- ev_days(ev(amt = 100), ii = 168, addl = 3, days = 'm,w,f')
e
```

```
.   time amt cmt evid  ii addl
. 1    0 100   1    1 168    3
. 2   48 100   1    1 168    3
. 3   96 100   1    1 168    3
```

We can see how this works by simulating the regimen

```
mrgsim_e(mod, e, end = 168*4) %>% plot("CP")
```




### 4.7.2 ev_rx

`ev_rx()` is a way to write a regimen out with notation similar to what you might see on a prescription. For example, 100 mg twice daily for 3 doses into compartment 2 would be

```
ev_rx("100 mg q12h x3 in 2")
```

```
. Events:
.   time amt ii addl cmt evid
. 1    0 100 12    2   2    1
```

To code an infusion

```
ev_rx("500 mg over 2 hours q 24 h x3 in 1")
```

```
. Events:
.   time amt rate ii addl cmt evid
. 1    0 500  250 24    2   1    1
```

See the `ev_rx()` documentation for more details and limitations.

## 4.8 Upper case names

You'll notice in the previous sections that most of the column names were rendered with lower case letters when we convert the event object to a data set like object:

```
as.data.frame(ev(amt = 100))
```

```
.   time amt cmt evid
. 1    0 100   1    1
```

And when this event object is use to simulate, you'll see these lower case names in the simulated output. The reasons for this are historical and this behavior is unlikely to change because it goes so far back into the history of mrgsolve.

Nevertheless, recent versions of mrgsolve have included similar constructor functions that will render column names in upper case which are commonly seen in analysis data sets.

The `evd()` constructor behaves just like `ev()`, but it will render upper case names when coerced to a data set or used for simulation.

```
e <- evd(amt = 100)

e
```

```
. Events Data:
.   time amt cmt evid
. 1    0 100   1    1
```

The `d` in `evd()` indicates that the event object will render with names like a Data set and you will see a reminder of the data set like nature of this object when it is printed.

When this object is rendered to a data frame, you will see the names rendered in upper case

```
as.data.frame(e)
```

```
.   TIME AMT CMT EVID
. 1    0 100   1    1
```

You can also coerce an event object created with `ev()` to one that behaves as if it were created via `evd()`

```
as.evd(ev(amt = 100))
```

```
. Events Data:
.   time amt cmt evid
. 1    0 100   1    1
```

It is important to note that the case of the column names aren't made upper case until the data frame is rendered. So, in the previous example, `time` and `amt` are in lower case because we have not rendered yet. If you want to work on this object before it is rendered, be sure to lower case names before rendering and upper case after rendering. For example we use `rate` in the the following example, not `RATE`:

```
evd(amt = 100) %>% mutate(rate = amt / 5)
```

```
. Events Data:
.   time amt rate cmt evid
. 1    0 100   20   1    1
```

You can coerce a traditional event object to a data like event object with

```
as.evd(ev(amt = 100))
```

```
. Events Data:
.   time amt cmt evid
. 1    0 100   1    1
```

And finally, there are two utility functions for changing the names of a data like or event object. To convert to upper case use `uctran()`

```
ev(amt = 100) %>% as_data_set() %>% uctran()
```

```
.   TIME AMT CMT EVID ID
. 1    0 100   1    1  1
```

To convert to lower case, use `lctran()`

```
evd(amt = 100) %>% as_data_set() %>% lctran()
```

```
.   time amt cmt evid ID
. 1    0 100   1    1  1
```

The utility functions also work on event objects. For example,

```
ev(amt = 100) %>% uctran()
```

```
. Events Data:
.   time amt cmt evid
. 1    0 100   1    1
```

# 5 Model Matrices

Model matrices include `$OMEGA` (for subject-level variability) and `$SIGMA` (for residual unexplained variability). These matrices are coded into the model file (see Section 2.2.17 and Section 2.2.18) and can be manipulated in different ways via the model object. Because, `$OMEGA` and `$SIGMA` matrices are handled using identical approach (only the names of the functions change), we will focus on working with `$OMEGA` in the following examples with references to the equivalent functions that can be used to work on `$SIGMA`. Also note that, for simplicity, we will not compile the examples presented in this chapter.

## 5.1 Basics

### 5.1.1 Simple matrix lists

We can look at the `popex` model in the internal library for a starting example to show how model matrices can be seen from the model object.

Once the model is loaded

```
mod <- modlib("popex", compile = FALSE)
```

We can print the model object to the console and see the matrix structure

```
mod
```

```
.
.
. ----------------  source: popex.cpp  ----------------
.
.   project: /Users/kyleb/ren...gsolve/models
.   shared object: popex-so-8a0113f33223 <not loaded>
.
.   time:          start: 0 end: 240 delta: 0.5
.                  add: <none>
.
```

```
.   compartments:  GUT CENT [2]
.   parameters:    TVKA TVCL TVV WT [4]
.   captures:      CL V ECL IPRED DV [5]
.   omega:         3x3
.   sigma:         1x1
.
.   solver:        atol: 1e-08 rtol: 1e-08 maxsteps: 20k
. ------------------------------------------------------
```

In this output, we see that `omega` is a 3 by 3 matrix and `sigma` is 1 by 1. We can view both matrices by calling `revar()` on the model object

```
revar(mod)
```

```
. $omega
. $...
.       [,1] [,2] [,3]
. ECL:   0.3  0.0  0.0
. EV:    0.0  0.1  0.0
. EKA:   0.0  0.0  0.5
.
.
. $sigma
. $...
.     [,1]
. 1:     0
```

This shows the 3x3 `$OMEGA` matrix with all off-diagonals set to zero and the 1x1 `$SIGMA` which is currently fixed to 0.

The `$OMEGA` matrix can be extracted with the `omat()` function

```
omat(mod)
```

```
. $...
.       [,1] [,2] [,3]
. ECL:   0.3  0.0  0.0
. EV:    0.0  0.1  0.0
. EKA:   0.0  0.0  0.5
```

Use the `smat()` function to extract the `$SIGMA` matrix. The result of these calls are `matlist` objects; for `$OMEGA` the class is `omegalist` (which inherits from `matlist`)

```
omat(mod) %>% class()
```

```
. [1] "omegalist"
. attr(,"package")
. [1] "mrgsolve"
```

and for `$SIGMA` it is `sigmalist`. These are lists of matrices. In this example, there is just one `$OMEGA` block in the code

```
blocks(mod, OMEGA)
```

```
.
. Model file: popex.cpp
.
. $OMEGA
. @labels ECL EV EKA
. 0.3 0.1 0.5
```

so the length of the `omegalist` object is also 1

```
om <- omat(mod)
length(om)
```

```
. [1] 1
```

Functions are provided to check the names

```
names(om)
```

```
. [1] "..."
```

and the labels

```
labels(om)
```

```
. [[1]]
. [1] "ECL" "EV"  "EKA"
```

as well as getting the dimensions or number of rows

```
dim(om)
```

```
. $...
. [1] 3 3
```

```
nrow(om)
```

```
. ...
.   3
```

The `omegalist` (`sigmalist`) object can be converted to a standard R list

```
as.list(om) %>% str()
```

```
. List of 1
.  $ ...: num [1:3, 1:3] 0.3 0 0 0 0.1 0 0 0 0.5
.   ..- attr(*, "dimnames")=List of 2
.   .. ..$ : chr [1:3] "ECL" "EV" "EKA"
.   .. ..$ : chr [1:3] "ECL" "EV" "EKA"
```

or it can be rendered as a matrix

```
as.matrix(om)
```

```
.      [,1] [,2] [,3]
. [1,]  0.3  0.0  0.0
. [2,]  0.0  0.1  0.0
. [3,]  0.0  0.0  0.5
```

### 5.1.2 Multiple matrix lists

Let's look at an example where there is a more complicated `$OMEGA` structure.

```
// inline/multiple-matrices.cpp

$OMEGA @name first @labels a b c
1 2 3

$OMEGA @name second @labels d e
4 5

$OMEGA @name third @labels f g h i
6 7 8 9
```

```
mod <- mread("inline/multiple-matrices.cpp", quiet = TRUE)
```

Each `$OMEGA` block codes a diagonal matrix; the interesting feature is that there are 3 different `$OMEGA` blocks.

Now, when we look at the model

```
mod
```

```
.
.
. ----------  source: multiple-matrices.cpp  ----------
.
.   project: /Users/kyleb/git...-guide/inline
.   shared object: multiple-matrices.cpp-so-8a0147071477
.
.   time:          start: 0 end: 24 delta: 1
.                  add: <none>
.
.   compartments:  <none>
.   parameters:    <none>
.   captures:       <none>
.   omega:         3x3,2x2,4x4
.   sigma:         0x0
.
.   solver:        atol: 1e-08 rtol: 1e-08 maxsteps: 20k
. ------------------------------------------------------
```

We see that `$OMEGA` is one 3x3 matrix, one 2x2 matrix and on 4x4 matrix, in that order. Calling `revar()` on this model object

```
revar(mod)
```

```
. $omega
. $first
.     [,1] [,2] [,3]
. a:     1    0    0
. b:     0    2    0
. c:     0    0    3
.
. $second
.     [,1] [,2]
. d:     4    0
. e:     0    5
.
. $third
.     [,1] [,2] [,3] [,4]
. f:     6    0    0    0
. g:     0    7    0    0
. h:     0    0    8    0
. i:     0    0    0    9
.
.
. $sigma
. No matrices found
```

we only see the `$OMEGA` matrices as the model was coded. Now the length of the `omegalist` object is 3

```
length(omat(mod))
```

```
. [1] 3
```

and the number of rows is 3 in the first matrix, 2 in the second, and 4 in the third

```
nrow(omat(mod))
```

```
.  first second  third
.      3      2      4
```

We can also check `dim()`

```
dim(omat(mod))
```

```
. $first
. [1] 3 3
.
. $second
. [1] 2 2
.
. $third
. [1] 4 4
```

The omega matrix can be converted from this segmented list into a single block matrix

```
as.matrix(omat(mod))
```

```
.       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
.  [1,]    1    0    0    0    0    0    0    0    0
.  [2,]    0    2    0    0    0    0    0    0    0
.  [3,]    0    0    3    0    0    0    0    0    0
.  [4,]    0    0    0    4    0    0    0    0    0
.  [5,]    0    0    0    0    5    0    0    0    0
.  [6,]    0    0    0    0    0    6    0    0    0
.  [7,]    0    0    0    0    0    0    7    0    0
.  [8,]    0    0    0    0    0    0    0    8    0
.  [9,]    0    0    0    0    0    0    0    0    9
```

The result is 9x9 with all off diagonals between the different list positions set to zero.

Otherwise, we might work with this object as a list

```
as.list(omat(mod))
```

```
. $first
.   a b c
. a 1 0 0
. b 0 2 0
. c 0 0 3
.
. $second
```

```
.   d e
. d 4 0
. e 0 5
.
. $third
.   f g h i
. f 6 0 0 0
. g 0 7 0 0
. h 0 0 8 0
. i 0 0 0 9
```

## 5.2 Collapsing matrices

The functionality described in this subsection is new with mrgsolve 1.0.0.

The structure of the matrices and the order in which they appear in the list matter when updating each matrix (see below). We saw how to create one big matrix out of all the smaller matrices using the `as.matrix()`. This section describes how to combine matrices within the confines of the `matlist` object.

mrgsolve provides functions to collapse (or combine) matrices in a `matlist` object. We can call `collapse_omega()`, passing in the model object

```
collapse_omega(mod) %>% revar()
```

```
. $omega
. $...
.     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
. a:     1    0    0    0    0    0    0    0    0
. b:     0    2    0    0    0    0    0    0    0
. c:     0    0    3    0    0    0    0    0    0
. d:     0    0    0    4    0    0    0    0    0
. e:     0    0    0    0    5    0    0    0    0
. f:     0    0    0    0    0    6    0    0    0
. g:     0    0    0    0    0    0    7    0    0
. h:     0    0    0    0    0    0    0    8    0
. i:     0    0    0    0    0    0    0    0    9
.
.
. $sigma
. No matrices found
```

and now we have an `omegalist` object inside a model object and the matrix is a single 9x9 `$OMEGA` matrix. The row names have been retained, but there is now no name for the matrix; this can be provided when collapsing

```
collapse_omega(mod, name = "only") %>% omat()
```

```
. $only
.     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
. a:     1    0    0    0    0    0    0    0    0
. b:     0    2    0    0    0    0    0    0    0
. c:     0    0    3    0    0    0    0    0    0
. d:     0    0    0    4    0    0    0    0    0
. e:     0    0    0    0    5    0    0    0    0
. f:     0    0    0    0    0    6    0    0    0
. g:     0    0    0    0    0    0    7    0    0
. h:     0    0    0    0    0    0    0    8    0
. i:     0    0    0    0    0    0    0    0    9
```

Suppose that we only want to combine the first two matrices, leaving the third matrix alone. In that case, call `collapse_omega()` with the range argument

```
collapse_omega(mod, range = c(1,2), name = "first_second") %>% omat()
```

```
. $first_second
.     [,1] [,2] [,3] [,4] [,5]
. a:     1    0    0    0    0
. b:     0    2    0    0    0
. c:     0    0    3    0    0
. d:     0    0    0    4    0
. e:     0    0    0    0    5
.
. $third
.     f g h i
. f:  6 0 0 0
. g:  0 7 0 0
. h:  0 0 8 0
. i:  0 0 0 9
```

Now the matlist topology has changed; there are still 9 (total) rows and columns, but the matlist object is length 2 with 5x5 in the first position (newly named `first_second`) an the old 4x4 matrix in the second position. Collapsing matrices is an irreversible process; at this

time there is no mechanism to cut matrices back into smaller chunks. But collapsing matrices can be very helpful when they need to be updated.

Use `collapse_sigma()` to collapse `$SIGMA` matrices if needed.

Also, the function `collapse_matrix()` can be called on a `omegalist` or `sigmalist` object to collapse

```
omat(mod) %>% collapse_matrix(range = c(2,NA))
```

```
. $first
.     a b c
. a:  1 0 0
. b:  0 2 0
. c:  0 0 3
.
. $...
.     [,1] [,2] [,3] [,4] [,5] [,6]
. d:     4    0    0    0    0    0
. e:     0    5    0    0    0    0
. f:     0    0    6    0    0    0
. g:     0    0    0    7    0    0
. h:     0    0    0    0    8    0
. i:     0    0    0    0    0    9
```

### 5.2.1 Collapse on load

You can also collapse `$OMEGA` or `$SIGMA` from within the model specification file so that the matrices are collapsed on load.

In the `$SET` set block (Section 2.2.19), pass either `collapse_omega` or `collapse_sigma` set to `TRUE`

```
$SET collapse_omega = TRUE
```

or

```
$SET collapse_sigma = TRUE
```

This will produce a model object with collapsed matrices when the model is loaded via `mread()`.

## 5.3 Updating $OMEGA and $SIGMA

Like the values of parameters in the parameter list, we may want to update the values in $OMEGA and $SIGMA matrices. We can do so without re-compiling the model.

### 5.3.1 Matrix helper functions

mrgsolve keeps $OMEGA and $SIGMA in block matrices (regardless of whether the off-diagonal elements are zeros or not). Recall that in the model specification file we can enter data for $OMEGA and $SIGMA as the lower triangle of the matrix (see Section 2.2.17). In R, we need to provide a matrix (as an R object). mrgsolve provides some convenience functions to help … allowing the user to enter lower diagonals instead of the full matrix.

dmat() for diagonal matrix

```
dmat(1,2,3)
```

```
.      [,1] [,2] [,3]
. [1,]    1    0    0
. [2,]    0    2    0
. [3,]    0    0    3
```

bmat() for block matrix

```
bmat(1,2,3)
```

```
.      [,1] [,2]
. [1,]    1    2
. [2,]    2    3
```

cmat() for a block matrix where the diagonal elements are variances and the off-diagonals are taken to be correlations, not covariances

```
cmat(0.1, 0.87,0.3)
```

```
.           [,1]      [,2]
. [1,] 0.1000000 0.1506884
. [2,] 0.1506884 0.3000000
```

`mrgsolve` will convert the correlations to covariances.

`mrgsolve` also provides `as_bmat()` and `as_dmat()` for converting other `R` objects to matrices or lists of matrices.

Consider this list with named elements holding the data for a matrix:

```
m <- list(OMEGA1.1 = 0.9, OMEGA2.1 = 0.3, OMEGA2.2 = 0.4)
```

These data could form either a 3x3 diagonal matrix or a 2x2 block matrix. But the names suggest a 2x2 form. `as_bmat()` can make the matrix like this

```
as_bmat(m, "OMEGA")
```

```
.      [,1] [,2]
. [1,]  0.9  0.3
. [2,]  0.3  0.4
```

The second argument is a regular expression that `mrgsolve` uses to find elements in the list to use for building the matrix.

Frequently, we have estimates in a data frame like this

```
data(exBoot)
head(exBoot)
```

```
.   run  THETA1 THETA2  THETA3 OMEGA11   OMEGA21 OMEGA22 OMEGA31  OMEGA32 OMEGA33
. 1   1 -0.7634  2.280  0.8472 0.12860  0.046130  0.2874 0.13820 -0.02164  0.3933
. 2   2 -0.4816  2.076  0.5355 0.12000  0.051000  0.2409 0.06754 -0.07759  0.3342
. 3   3 -0.5865  2.334 -0.4597 0.11460  0.097150  0.2130 0.16650  0.18100  0.4699
. 4   4 -0.6881  1.824  0.7736 0.14990  0.000003  0.2738 0.24700 -0.05466  0.5536
. 5   5  0.2909  1.519 -1.2440 0.07308  0.003842  0.2989 0.06475  0.05078  0.2500
. 6   6  0.1135  2.144 -1.0040 0.13390 -0.019270  0.1640 0.10740 -0.01170  0.3412
.    SIGMA11 SIGMA21 SIGMA22
. 1 0.002579       0  1.0300
. 2 0.002228       0  1.0050
. 3 0.002418       0  1.0890
. 4 0.002177       0  0.8684
. 5 0.001606       0  0.8996
. 6 0.002134       0  0.9744
```

We can use `as_bmat()` with this data frame to extract the `$OMEGA` matrices

```
omegas <- as_bmat(exBoot, "OMEGA")
length(omegas)
```

```
. [1] 100
```

```
dim(exBoot)
```

```
. [1] 100  13
```

```
omegas[[6]]
```

```
.           [,1]     [,2]    [,3]
. [1,]  0.13390 -0.01927  0.1074
. [2,] -0.01927  0.16400 -0.0117
. [3,]  0.10740 -0.01170  0.3412
```

```
omegas[[16]]
```

```
.          [,1]    [,2]   [,3]
. [1,] 0.08126 0.01252 0.1050
. [2,] 0.01252 0.16860 0.0149
. [3,] 0.10500 0.01490 0.4062
```

The result of calling `as_bmat` or `as_dmat` is a list of matrices, one for each row in the data frame.

Note in this example, we could have called

```
sigmas <- as_bmat(exBoot,"SIGMA")
```

to grab the `$SIGMA` matrices.

For help on these helper functions, see `?dmat`, `?bmat`, `?cmat`, `?as_bmat`, `?as_dmat` in the `R` help system after loading `mrgsolve`.

### 5.3.2 Fill a matrix with zeros

Sometimes we write a population model that includes random effects, but we would like to simulate from that same model without the random effects implemented. For example, we want to simulate some typical PK profiles from a population PK model that includes IIV on some parameters and / or RUV on the simulated outputs.

To do this, pass the model through the `zero_re()` function. By default, this will convert all `$OMEGA` and `$SIGMA` matrix elements to zeros. See the R help file (`?zero_re`) to see some options for selectively zeroing out only one or the other.

For example we have this population PK model

```
mod <- modlib("popex", compile = FALSE)

omat(mod)
```

```
. $...
.       [,1] [,2] [,3]
. ECL:   0.3  0.0  0.0
. EV:    0.0  0.1  0.0
. EKA:   0.0  0.0  0.5
```

We can turn that matrix to all zeros with

```
mod %>% zero_re() %>% omat()
```

```
. $...
.       [,1] [,2] [,3]
. ECL:     0    0    0
. EV:      0    0    0
. EKA:     0    0    0
```

And when we simulate right after that, all `ETA(n)` will be zero as well and you'll get your fixed-effects simulation (the following is for example only and is not evaluated)

```
mod %>%
  zero_re() %>%
  ev(amt = 100) %>%
  mrgsim() %>%
  plot()
```

### 5.3.3 Example: unnamed matrix

Here is a model with only a 3x3 `$OMEGA` matrix

```
// inline/matrix.cpp

$OMEGA
1 2 3
```

```
mod <- mread("inline/matrix.cpp", compile = FALSE, quiet = TRUE)
```

Let's check the values in the matrix using `omat()`

```
mod %>% omat
```

```
. $...
.      [,1] [,2] [,3]
. 1:      1    0    0
. 2:      0    2    0
. 3:      0    0    3
```

We also use `omat()` to update the values in the matrix

```
mod %>% omat(dmat(4,5,6)) %>% omat
```

```
. $...
.      [,1] [,2] [,3]
. 1:      4    0    0
. 2:      0    5    0
. 3:      0    0    6
```

To update `$OMEGA`, we must provide a matrix of the same dimension, in this case 3x3. An error is generated if we provide a matrix with the wrong dimension.

```
ans <- try(mod %>% omat(dmat(11,23)))
```

```
. Error : improper signature: omat
```

```
ans
```

```
. [1] "Error : improper signature: omat\n"
. attr(,"class")
. [1] "try-error"
. attr(,"condition")
. <simpleError: improper signature: omat>
```

### 5.3.4 Example: named matrices

When there are multiple `$OMEGA` matrices, it can be helpful to assign them names. Here, there are two matrices: one for interindividual variability (`IIV`) and one for interoccasion variability (`IOV`).

```
// inline/iov.cpp

$OMEGA @name IIV
1 2 3
$OMEGA @name IOV
4 5
```

```
mod <- mread("inline/iov.cpp", compile = FALSE, quiet = TRUE)

revar(mod)
```

```
. $omega
. $IIV
.     [,1] [,2] [,3]
. 1:     1    0    0
. 2:     0    2    0
. 3:     0    0    3
.
. $IOV
.     [,1] [,2]
. 4:     4    0
. 5:     0    5
.
.
. $sigma
. No matrices found
```

Now, we can update either `IIV` or `IOV` (or both) by name

```
mod %>%
  omat(IOV = dmat(11,12), IIV = dmat(13, 14, 15)) %>%
  omat()
```

```
. $IIV
.      [,1] [,2] [,3]
. 1:    13    0    0
. 2:     0   14    0
. 3:     0    0   15
.
. $IOV
.      [,1] [,2]
. 4:    11    0
. 5:     0   12
```

Again, an error is generated if we try to assign a 3x3 matrix to the `IOV` position

```
ans <- try(mod %>% omat(IIV = dmat(1, 2)))
```

```
. Error : improper dimension: omat
```

```
ans
```

```
. [1] "Error : improper dimension: omat\n"
. attr(,"class")
. [1] "try-error"
. attr(,"condition")
. <simpleError: improper dimension: omat>
```

### 5.3.5 Example: unnamed matrices

If we do write the model with unnamed matrices, we can still update them

```
// inline/multi.cpp

$OMEGA
1 2 3

$OMEGA
4 5
```

```
mod <- mread("inline/multi.cpp", compile = FALSE, quiet = TRUE)
```

In this case, the only way to update is to pass in a **list** of matrices, where (in this example) the first matrix is 3x3 and the second is 2x2

```
mod %>% omat(list(dmat(5, 6, 7), dmat(8, 9))) %>% omat()
```

```
. $...
.     [,1] [,2] [,3]
. 1:     5    0    0
. 2:     0    6    0
. 3:     0    0    7
.
. $...
.     [,1] [,2]
. 4:     8    0
. 5:     0    9
```

# 6 Simulated output

## 6.1 Output types

When `mrgsim()` is used to simulate from a model, it by default returns an object with class `mrgsims`. This is an S4 object containing a data.frame of simulated output and a handful of other pieces of data related to the simulation run that can be coerced to other types (like `data.frame` or `tibble`).

For simulations with large outputs or extremely brief simulations where efficiency is important, users can request the output be returned as a data frame. This is most efficient when the features provided by the `mrgsims` object are not needed. To do this, pass the `output` argument to `mrgsim()`

```
out <- mrgsim(mod, ..., output = "df")
```

or use `mrgsim_df()`

```
out <- mrgsim_df(mod, ....)
```

## 6.2 Methods for mrgsim output

`mrgsolve` provides several methods for working with `mrgsims` objects or coercing the simulation matrix into other `R` objects. Note the discussion in the following subsections all refer to working with `mrgsims` objects, not data.frame output.

### 6.2.1 Coercion methods

- `as_tibble()`: convert to `tibble`
- `as.data.frame()`: convert to `data.frame`
- `as.matrix()`: convert to `matrix`

### 6.2.2 Query methods

- `head()`: shows the first `n = 5` rows
- `tail()`: shows the last `n = 5` rows
- `names()`: shows the column names
- `dim()`: shows the number of rows and columns
- `summary()`: shows a numeric summary of all columns
- `$`: extracts a column

### 6.2.3 Graphical methods

There is a `plot()` methods for simulated output that is aware of independent and dependent variables from the simulation. If `out` is the simulated output (an `mrgsims` object)

```
plot(out)
```

Plot with a formula; the following example selects only the `CP` and `RESPONSE` outputs and plots them versus `time`

```
plot(out, CP + RESPONSE ~ time)
```

To select a large number of responses to plot, pass a character vector or comma-separated character data containing output columns to plot

```
plot(out, "CP, RESPONSE, WT, DOSE")
```

### 6.2.4 Methods for `dplyr` verbs

`mrgsolve` provides several S3 methods to make it possible to include `dplyr` verbs in your simulation pipeline.

For example

```
library(dplyr)
library(mrgsolve)

mod <- house()

mod %>%
  ev(amt=100) %>%
  mrgsim() %>%
  filter(time >= 10)
```

Here, `mrgsim()` returns an `mrgsims` object. When `dplyr` is also loaded, this object can be piped directly to `dplyr::filter()` or `dplyr::mutate()` etc.

It is important to note that when `mrgsims` output is piped to `dplyr` functionality, it is coerced to `tibble` (`data.frame`) and there is no way to get the data back to `mrgsims` object. Most of the time, this is desirable and there is no need to explicitly coerce to `tibble()` when calling `dplyr` verbs on simulated output.

Other `dplyr` functions that can be used with `mrgsims` objects

- `group_by()`
- `mutate()`
- `filter()`
- `summarise()`
- `select()`
- `slice()`
- `pull()`
- `distinct()`
- `slice()`

### 6.2.5 Modify methods

You can modify the underlying data in the `mrgsims` object and keep it as an `mrgsims` object.

- `filter_sims()`: calls `dplyr::filter()` to pick rows to keep or discard
- `select_sims()`: calls `dplyr::select()`; note that `ID` and `time` columns are always retained
- `mutate_sims()`: calls `dplyr::mutate()` to add or modify columns

## 6.3 Controlling output scope

### 6.3.1 Background

Limiting the volume of simulated data can have a major impact on simulation efficiency, memory footprint, and ease (or lack of ease) in reviewing and dealing with the output. For any large simulation or any simulation from a large model, the user should consider selecting what gets returned when the simulation is performed.

By default, `mrgsim()` returns a data.frame with the following

1. `ID`: regardless of whether you simulated a population or not
2. `time` / `TIME`: the independent variable
3. Simulated values for all model compartments

4. Simulated values for derived outputs listed in `$CAPTURE`

You will always get `ID` and `time` and the compartments and any captured items must be written into the model file. This defines the list of data items that **could (possibly)** get returned under items 3 and 4 above. Again: this must be written into the model file and is locked at the time the model is compiled.

However, `mrgsolve` allows the user to pick what is actually returned at run time. Because this is done at run time, different runs can return different data items. And (importantly) `mrgsim()` only allocates space in the output for data items that are requested. So, opting **out** of unneeded outputs will decrease memory consumption and increase efficiency.

### 6.3.2 Implementation

The mrgsolve model object tracks compartments and captures that are currently being requested. This can be queried using `outvars()`

```
mod <- house()

outvars(mod)
```

```
. $cmt
. [1] "GUT"  "CENT" "RESP"
.
. $capture
. [1] "DV" "CP"
```

Items are listed under `cmt` and `capture`. The user can update the model object with the names of columns that are being requested by passing `outvars` to `update()`

```
mod <- update(mod, outvars = "CP, RESP")

outvars(mod)
```

```
. $cmt
. [1] "RESP"
.
. $capture
. [1] "CP"
```

This will exclude anything that isn't named in the update. The `outvars` list can be reset by passing `(all)`

```
mod <- update(mod, outvars = "(all)")
```

Remember that `...` passed to `mrgsim()` are also passed to `update()` so it is possible to select outputs right in your `mrgsim()` call

```
out <- mrgsim(mod, outvars = "CP, RESP")
```

### 6.3.3 Copy items from data to simulated output

Users can also use `carry_out` and `recover` to copy items from the input data into the output. This is covered in a different chapter.

# 7 Simulation sequence

This section is intended to help the user understand the steps `mrgsolve` takes when working through a simulation problem. The focus is on the order in which `mrgsolve` calls different user-defined functions as well as when parameter updates and output writing happens during the simulation sequence.

## 7.1 Functions to call

The model specification results in the definition of four functions that `mrgsolve` calls during the simulation sequence. Naming them by their code block identifiers, the functions are

1. `$PREAMBLE`
2. `$MAIN`
3. `$ODE`
4. `$TABLE`

## 7.2 Problem initiation

Just prior to starting the problem (when `NEWIND` is equal to `0`), `mrgsolve` calls `$PREAMBLE`. This function is only called once during the simulation sequence. The goal of `$PREAMBLE` is to allow the user to work with different `C++` data structures to get them ready for the simulation run.

## 7.3 Subject initiation

After the `$PREAMBLE` call, `mrgsolve` simulates each `ID` in the data set, one after another. `mrgsolve` runs this sequence just prior to simulating a given `ID`

1. Copy any parameters that are found in the `idata_set` to the working parameter list

2. Copy any parameters that are found in the `data_set` to the working parameter list, with the copy being taken from the first actual data set row for that individual. If the first actual data set record in the data set is not the first record for the individual, `mrgsolve` still copies from the first data set record as long as the `fillbak` argument to `mrgsim` is `TRUE`.
3. Set initial estimates from the base initial estimate list
4. Copy initial estimates from `idata_set` if they are found there.
5. Call `$MAIN`
6. Start simulating the records for that individual

## 7.4 Sequence for a single record

`mrgsolve` executes this sequence while working from record to record for a given `ID`

1. If `nocb` (next observation carried backward) is `TRUE`, then parameters are copied from the current record if that is an actual data set record. Note that if `nocb` is `FALSE` then `locf` (first observation carried forward) is assumed to be `TRUE` (see below). This is the last parameters will be copied from any input data set prior to advancing the system (when `locf` is being used). Therefore, when parameter columns are found in both an `idata_set` and a `data_set`, it will be the value found in the `data_set` that will overwrite both the base list and any parameter value that was copied from an `idata_set`. It is not an error to have different parameter values in an `idata_set` and a `data-set`, but the value found in the `data_set` will be used when this happens. More on parameters and the parameter update sequence can be found in Section 11.3 and Section 1.1.
2. `$MAIN` is called
3. The system is advanced via `$ODE` or `$PKMODEL`, whichever one is invoked in the model specification file.
4. If the current record is a dosing record, the dose is implemented (e.g. bolus made or infusion started).
5. If the system is advancing according to `locf`, then parameters are copied from the current record if that is an actual data set record. This is in contrast to `nocb` advance (see above).
6. The `$TABLE` function is called
7. If the current record is marked for inclusion in the simulated output, results are written to the output matrix.
8. Continue to the next record in the individual.
9. Once the last record is processed in an individual, a new individual is started.

# 8 Steady state

## 8.1 Key information

Keep reading for all the details; I'm including this brief list of key items up front for your convenience.

- Use `ss_rtol` and `ss_atol` arguments to `mrgsim()` to control the local error estimate when `mrgsolve` is finding steady state
- Use `ss_n` to limit the number of doses that will be administered when advancing to steady state; if the number of doses exceeds `ss_n` then a warning is issued and `mrgsolve` moves on
- Use `ss_fixed = TRUE` to silence the warning when `ss_n` is exceeded; you are essentially saying dose up to `ss_n` and then give up and move on without warning
- Use `SS_ADVANCE` in `$ODE` to check if the system is currently being advanced
- Use `ss_cmt` in `[ set ]` (inside your model) to select the compartments to be considered when finding steady state; you might have better success / efficiency if you focus on key compartments (or exclude unhelpful compartments like a depot compartment)

## 8.2 Introduction

Within mrgsolve, the term "steady state" (`SS`) applies specifically to the pharmacokinetic dosing system and indicates that the rate of drug administration is equal to the rate of drug elimination. Steady state dosing can take the form of repeated intermittent doses (bolus or infusion, administered intermittently at a given dosing interval) or a continuous infusion administered to steady state.

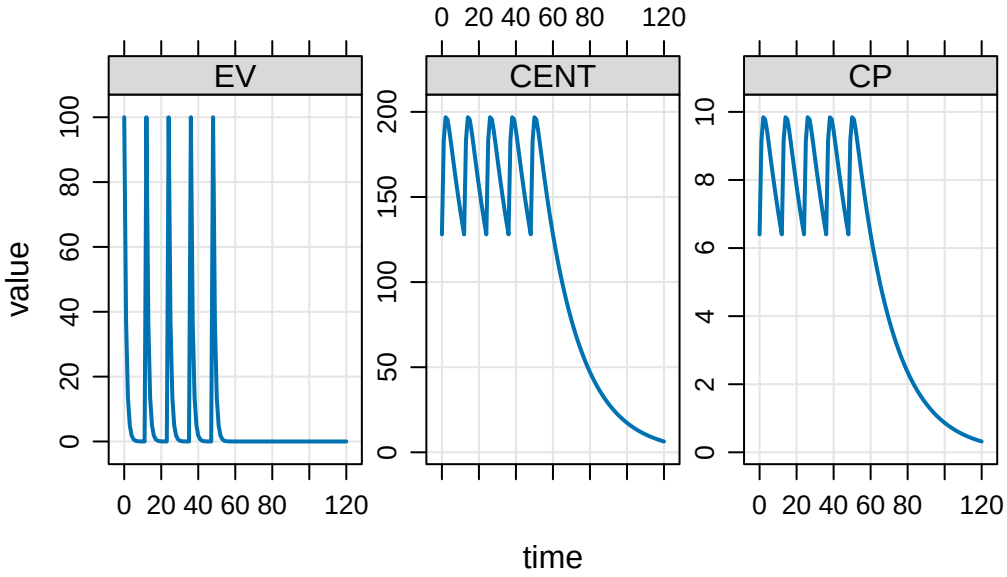
### 8.2.1 Intermittent doses

The user can direct mrgsolve to advance the system to steady state for intermittent dosing by including `ss=1` in an event object or input data set. For example:

```
mod <- modlib("pk1", end = 120)
```

```
. Loading model from cache.
```

```
dose <- ev(amt = 100, ii = 12, addl = 4, ss = 1)

mrgsim(mod, dose, recsort = 3) %>% plot()
```

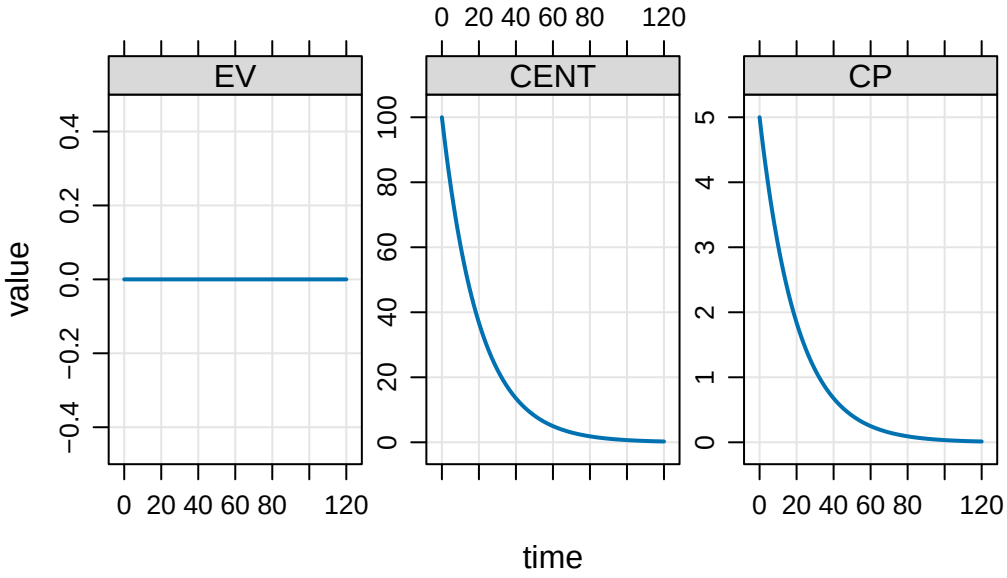



In this example, the `ss=1` flag tells mrgsolve to advance the system to steady state under a dosing regimen of 100 mg every 12 hours (and then give a total of 5 doses). When using the `ss=1` flag, the user is required to indicate the dosing interval (here every 12 hours) and additional doses are optional. Similar behavior can be achieved for intermittent infusions by setting the infusion `rate`.

### 8.2.2 Continuous infusion

A continuous infusion can be dosed to steady state by including the `ss=1` flag, a value for `rate` (any positive rate or -1 if the rate is being modeled), and setting the dose amount (`amt`/`AMT`) to zero:

```
infus <- ev(amt = 0, rate = 5, ss = 1, cmt = "CENT")

mrgsim(mod, infus, recsort = 3) %>% plot()
```




Because `CL` is equal to 1 in this model, we see that the continuous (never-ending) infusion was started at steady state with a value of 5.

## 8.3 Advance to SS

It's important to recognize that `SS` is related to the PK dosing system; it is finding the state of the system after an infinite number of doses have been administered under a certain regimen. And this is essentially how mrgsolve goes about finding steady state: when the `ss=1` flag is encountered, mrgsolve starts repeatedly administering doses and advancing the system to the next dose according to the inter-dose interval (`ii`). Once mrgsolve determines that the amounts in the system at any dose are the same as they were at the preceding dose, mrgsolve declares that `SS` has been achieved.

## 8.4 Control advance to SS

### 8.4.1 Tolerances for SS

mrgsolve uses a local error estimate to determine the degree to which concentrations are changing or not changing between doses on the way to `SS`. This is determined by a relative tolerance parameter (`ss_rtol`) and an absolute tolerance parameter (`ss_atol`). As of mrgsolve version 0.10.3, these tolerances are distinct from the tolerances used for solving the differential equations (`rtol` and `atol`, respectively). Note that when advancing to `SS` in an ODE model, `ss_rtol` (the relative tolerance for determining `SS`) must be larger (less precise) than `rtol`

(the relative tolerance used by the ODE solver). Once the difference between two trough concentrations is less than $A_{trough}$ * `ss_rtol` + `ss_atol`, then the system is said to be at steady state. By default, this calculation is done for every single compartment in the model and all compartments have to meet this criteria before the system is said to be at steady state. So, increasing `ss_rtol` (say from 1e-8 to 1e-3) will also allow us to call it "good" with respect to steady state sooner.

Both tolerances for steady state are stored in the model object and can be set with the update method. For example,

```
mod <- house()

mod <- update(mod, ss_rtol = 1e-5, ss_atol = 1e-8)
```

### 8.4.2 Max dose number

It was noted above that mrgsolve advances the system to steady state with a brute force approach: doses are repeatedly administered at a regular interval (for intermittent `SS`) until pre-dose concentrations are the same dose to dose. mrgsolve sets an upper limit (equal to 500) to the number of doses that will be administered before giving up on trying to find steady state. Once this maximum number of doses is exceeded, mrgsolve will issue a warning that steady state was not achieved and continue on with the problem. For example:

```
dose <- ev(amt = 100, ii = 12, ss = 1, VC = 800)

out <- mrgsim(house(), dose)
```

```
. Warning in (function (x, data, idata = no_idata_set(), carry_out = carry.out, : [steady_bo
.   ss_n: 500, ss_rtol: 1e-08, ss_atol: 1e-08
```

Here, mrgsolve administered 500 doses and the pre-dose concentrations were still not similar enough to declare the system to be at steady state.

The maximum dose number can be set with the `ss_n` argument to `mrgsim` (or one of the variants; see `?do_mrgsim` help topic). This number can be increased to prevent the warning:

```
out <- mrgsim(house(), dose, ss_n  = 1000)
```

Alternatively, the `ss_fixed` argument to mrgsim (see `?do_mrgsim`) can be set to `TRUE` to silence the warning. In this case, **up to** `ss_n` doses will be administered and if `SS` is not achieved with that many doses, the problem will continue with no warning.

### 8.4.3 Include / exclude compartments for SS

Sometimes it might be sufficient to only consider one compartment when determining `SS` (e.g. the central compartment in a PK model). Other times, it might be helpful to exclude a compartment when determining `SS` (e.g. a depot dosing compartment when concentrations can get very small toward the end of the dosing interval).

mrgsolve allows the user to identify compartments to include or exclude in determining `SS`. This is done through the `ss_cmt` option in `$SET`. To only consider the `CENT` compartment, write the following in the model file:

```
[ set ] ss_cmt = "CENT"
```

This says to **only** look at the `CENT` compartment when determining `SS`.

Alternatively, you can exclude certain compartments like this:

```
[ set ] ss_cmt = "-GUT,DEPOT"
```

This says to ignore the `GUT` and `DEPOT` compartments when determining `SS`.

As another example, you might want to exclude an accumulator compartment when calculating `SS`

```
[ set ] ss_cmt = "-AUC"

[ ode ]
dxdt_CENT = -kel * CENT;

dxdt_AUC = CENT/VC;
```

This is just a partial model snippet, but it shows how you might exclude the AUC compartment when determining `SS`.

### 8.4.4 SS_ADVANCE flag

mrgsolve also provides an `SS_ADVANCE` indicator that is passed into `$ODE` and evaluates to `true` when the system is being advanced to steady state. So a better way to exclude the accumulator compartment from being considered for `SS` calculation would be this:

```
[ ode ]

dxdt_AUC = CENT/VC;

if(SS_ADVANCE) dxdt_AUC = 0;
```

This code prevents the `AUC` compartment from changing during the advance to `SS` and the dose to dose difference in `AUC` will always be zero, effectively excluding this compartment from factoring into the `SS` determination. This should be the preferred approach to dealing with an `AUC` compartment.

# 9 Plugins

## 9.1 autodec

Available as of mrgsolve version 1.0.0.

When this plugin is invoked, mrgsolve will search your model code for assignments and automatically declare them as `double` precision numbers. The following blocks are searched

- `$PREAMBLE`
- `$MAIN` (or `$PK`)
- `$ODE` (or `$DES`)
- `$TABLE` (or `$ERROR`)
- `$PRED`

For example, the following code requires that `CL` gets assigned a type

```
$PARAM WT = 70, TVCL = 1.2

$PK
double CL = TVCL * pow(WT/70, 0.75);
```

This is the default mrgsolve behavior and has been since the beginning.

The `autodec` plugin lets you write the following

```
$PLUGIN autodec

$PARAM WT = 70, TVCL = 1.2

$PK
CL = TVCL * pow(WT/70, 0.75);
```

mrgsolve will find `CL = ...` and understand that this is a user initiated variable and will declare it as `double` for you. Don't worry about `WT = 70` in `$PARAM`; mrgsolve should already know about that won't try to declare it.

When you are using the `autodec` plugin, you can still declare variables as `double` or `int` or `bool`. mrgsolve already finds those variables and will understand to leave those declarations alone. Note that it may still very convenient to declare using the `capture` type those variables that you want captured into the output

```
$PLUGIN autodec

$ERROR
capture Y = IPRED * exp(EPS(1));
```

The `capture` typedef makes `Y` a `double`; we didn't need to declare it with `autodec` in play, but decided to declare with `capture` so that it is copied into the simulated output.

The `autodec` plugin is intended for more straightforward models where most / all variables are real valued. Because mrgsolve can handle any valid C++ code in these blocks, there is a possibility that the code could get much more complicated, including custom classes and methods. In this case, we recommend to bypass this feature and take control of declaring variables as you would in the default mode.

In case mrgsolve does try to declare (as `double`) a variable that shouldn't be handled that way, you can note this name in an environment variable inside your model called `MRGSOLVE_AUTODEC_SKIP`

```
$ENV MRGSOLVE_AUTODEC_SKIP = c("my_variable_1")
```

This can be a vector of variable names to NOT declare when `autodec` is invoked.

## 9.2 nm-vars

Available as of mrgsolve version 1.0.0.

The `nm-vars` plugin provides a more NONMEM-like set of macros to use when coding your compartmental model. Only a small subset of the NONMEM model syntax is replicated here.

### F, R, D, ALAG

- To set bioavailability for the nth compartment, use `Fn`
- To set the infusion rate for the nth compartment, use `Rn`
- To set the infusion duration for the nth compartment, use `Dn`
- To set the lag time for the nth compartment, use `ALAGn`

For example

```
$CMT GUT CENT GUT2

$PK
F1 = 0.87;    // equivalent to F_GUT  = 0.87;
R2 = 2.25;    // equivalent to R_CENT = 2.25;
ALAG3 = 0.25; // equivalent to ALAG_GUT2 = 0.25;
```

### A, A_0, DADT

- To refer to the amount in the nth compartment, use `A(n)`
- To refer to the initial amount in the nth compartment, use `A_0(n)`
- To refer to the differential equation for the nth compartment, use `DADT(n)`

For example

```
$CMT CMT1 CMT2

$PK
A_0(2) = 50;

$DES
DADT(1) = -KA * A(1);
DADT(2) =  KA * A(1) - KE * A(2);
```

### Math

Starting with version 1.0.1, macros are provided for several math functions

- `EXP(a)` gets mapped to `exp(a)`
- `LOG(a)` gets mapped to `log(a)`
- `SQRT(a)` gets mapped to `sqrt(a)`

These are purely for convenience, so that upper-case versions from NMTRAN don't require conversion to lower-case; this happens automatically via the C++ preprocessor.

### Other syntax

- Using `THETA(n)` in model code will resolve to `THETAn`; this feature is always available, even when `nm-vars` hasn't been invoked; we mention it here since it is a fundamental piece of the NONMEM syntax that mrgsolve has internalized
- Use `T` in `$DES` to refer to the current time in the odesolver rather than `SOLVERTIME`

### Reserved words with nm-vars is invoked

There are some additional reserved words when the `nm-vars` plugin is invoked

- `A`
- `A_0`
- `DADT`
- `T`

It is an error to use one of these symbols as the name of a parameter or compartment or to try to declare them as variables.

### mrgsolve syntax that is still required

There are a lot of differences remaining between mrgsolve and NONMEM syntax. We mention a few here to make the point

- mrgsolve continues to require `pow(base, exponent)` rather than `base**exponent`
- mrgsolve continues to require a semi-colon at the end of each statement (this is a C++ requirement)
- mrgsolve continues to require that user-defined variables are declared with a type, except when the `autodec` plugin (Section 9.1) is invoked

### An example

There is an example of this syntax (along with `autodec` features) in the internal model library

```
mod <- modlib("nm-like")
see(mod)
```

```
.
. Model file:  nm-like.cpp
. $PROB Model written with some nonmem-like syntax features
.
. $PLUGIN nm-vars autodec
.
. $PARAM
. THETA1 = 1, THETA2 = 21, THETA3 = 1.3, WT = 70, F1I = 0.5, D2I = 2
. KIN = 100, KOUT = 0.1, IC50 = 10, IMAX = 0.9
.
. $CMT @number 3
.
. $PK
. CL = THETA(1) * pow(WT/70, 0.75);
. V  = THETA(2);
```

```
. KA = THETA(3);
.
. F1 = F1I;
. D2 = D2I;
. A_0(3) = KIN / KOUT;
.
. $DES
. CP = A(2)/V;
. INH = IMAX*CP/(IC50 + CP);
.
. DADT(1) = -KA*A(1);
. DADT(2) =  KA*A(1) - (CL/V)*A(2);
. DADT(3) =  KIN * (1-INH) - KOUT * A(3);
.
. $ERROR
. CP = A(2)/V;
```

## 9.3 tad

**Purpose** Advanced calculation time after dose within your model. We call this "advanced" because it lets you track doses in multiple compartments. See the note below about a simpler way to calculate time after dose that should work fine if doses are only in a single compartment. This functionality is provided by mrgsolve.

**Usage**

First, tell `mrgsolve` that you want to use the `tad` plugin

```
$PLUGIN tad
```

The create `tadose` objects, one for each compartment where you want to track time after dose. One approach is to do this in `[ global ]`

```
[plugin] tad

[ global ]
mrg::tadose tad_cmt_1(1);
mrg::tadose tad_cmt_2(2);
```

Notice that we pass the compartment number that we want to track in each case and also that we refer to the `mrg::` namespace for the `tadose` class.

The `tadose` objects contain the following (public) members

- `cmt` the compartment to track
- `told` the time of last dose; defaults to `-1e9`
- `had_dose` indicates if a dose has already been given for the current individual
- `tad(self)` the function to call to calculate time after dose
  - the `self` object (Section 2.3.12) must be passed as the only argument
  - when the member function is called prior to the first administered dose, a value of `-1.0` is returned
- `reset()` resets the state of the object; be sure to reset prior to simulating a new individual

As an example, you can call the `reset()` method on one of the `tadose` objects

```
tad_cmt_1.reset();
```

You can find the source code for this object here.

A working example model that tracks doses in compartments `1` and `2` is provided here

```
[plugin] tad

[ global ]
mrg::tadose tad_cmt_1(1);
mrg::tadose tad_cmt_2(2);

[ pkmodel ] cmt = "GUT,CENT", depot = TRUE

[ param ] CL = 1, V = 20, KA = 1

[ main ]
capture tad1 = tad_cmt_1.tad(self);
capture tad2 = tad_cmt_2.tad(self);
```

#### Static approach

Another approach would be to make these static in `[ main ]` but this approach would only work if you only use these in `[ main ]`; the `[ global ]` approach is preferable since then you can access the object in any block (function).

### 9.3.1 Note

Note there is a simpler way to calculate time after dose when only dosing into a single compartment

```
[ main ]
double tad = self.tad();
```

The `self` object (Section 2.3.21) contains a `tad()` member which will track time after dose. Note that this needs to be called every record.

## 9.4 evtools

### Purpose

The `evtools` plugin is a set of functions and classes you can use to implement dosing regimens from *inside* your model. It first became available in mrgsolve 1.4.1. The most common use for this plugin is when you want to implement dynamic dosing simulations where the dose amount or the dosing interval is able to change based on how the system has advanced up to a certain point. For example, you might have a PKPD model for an oncology drug that includes a PK model for the drug as well as a dynamic model for platelets where a decline in platelets is driven by the drug concentration. In this case you might monitor platelets at different clinical visits and reduce the or hold dose or increase the dosing interval in response to Grade 3 or Grade 4 thrombocytopenia.

### Usage

Like all other plugins, you must invoke `evtools` in the `$PLUGIN` block

```
$PLUGIN evtools
```

### 9.4.1 Namespace

All the functionality made available by the `evtools` plugin is located in a namespace called `evt`. So you will need to prefix all functions and classes with `evt::`. For example, you can read about a function called `bolus` below; when you call that function, you need to refer to `evt::bolus`, locating that function in the `evt` namespace.

### 9.4.2 Event object type

Chapter 10 introduces a C++ event object called `mrg::evdata`. The `evt` namespace provides an easier to remember `typedef` for that object called `evt::ev`. So if a function returns an event object, you can use `evt::ev` for that type. For example

```
evt::ev dose = evt::bolus(100, 1);
```

is equivalent to

```
mrg::evdata dose evt::bolus(100, 1);
```

### 9.4.3 Simple administration of single doses now

The `evt` namespace allows you to easily administer single bolus or infusion doses. The functions are

- `void evt::bolus(self, <amt>, <cmt>)` where
  - `self` is the the self object, described in Section 2.3.12
  - `<amt>` is the dose amount (type `double`)
  - `<cmt>` is the dosing compartment (type `double`)
- `void evt::infuse(self, <amt>, <cmt>, <rate>)` where
  - `self` is the the self object, described in Section 2.3.12
  - `<amt>` is the dose amount (type `double`)
  - `<cmt>` is the dosing compartment (type `double`)
  - `<rate>` is the infusion rate (type `double`)

Note that `self` is passed as the first argument, there is no return value, and there is no `time` specified for the dose. All doses invoked this way are given `now`, as-is; so you should only call these functions when the model code has decided it is time to administer a dose.

*Important*: Because doses are given `now`, these functions should almost be called in `$TABLE` (i.e., `$ERROR`).

### 9.4.4 Customized doses, potentially given later

The `evt` namespace also provides variants of these functions which return the event object to you so you can modify some of the attributes, including potentially scheduling the dose in the future, prior to sending the object back to mrgsolve. These functions are

- `evt::ev evt::bolus(<amt>, <cmt>)` where
  - `<amt>` is the dose amount (type `double`)
  - `<cmt>` is the dosing compartment (type `double`)
- `evt::ev evt::infuse(<amt>, <cmt>, <rate>)` where

- `<amt>` is the dose amount (type `double`)
- `<cmt>` is the dosing compartment (type `double`)
- `<rate>` is the infusion rate (type `double`)

Note that we *don't* pass in the `self` object here; just the dose amount, compartment, and rate for infusions. These functions also return and event object (type `evt::ev`) that you can work with. See Section 10.4 for documentation of those attributes.

### 9.4.5 API for customizing doses

While Section 10.4 shows you some low-level ways to customize the event object, the `evt` namespace provides some API for making these changes.

**`evt::retime`**

The `evt::retime` function can be used to set the time attribute.

```
evt::ev dose = evt::bolus(100, 1);
evt::retime(dose, 24);
self.push(dose);
```

**Arguments**:

- an event object (`evt::ev`)
- the new dose time (`<double>`)

**Return**: `void` (or nothing)

When doses are retimed this way, the `now` attribute is forced to be `false`.

**`evt::now`**

Use `evt::now` to set the `now` attribute to `true`

```
evt::now(dose);
```

**Argument**:

- an event object (`evt::ev`)

**Return**: `void` (nothing)

**`evt::push`**

The `evt` namespace includes a `push` function to send an event object back to mrgsolve. For example

```
evt::ev dose = evt::bolus(100, 1);
evt::retime(dose, 24);
evt::push(self, dose);
```

This function will continue to be available in the `evt` namespace. But note that `self` has a `push()` method as of mrgsolve 1.4.1 to do the same thing

`evt::near()`

Use this function to test for equality between floating put numbers. For example, to test if `TIME` is (about) equal to `24.5`, you can call

```
if(evt::near(TIME, 24.5)) {
  // do something
}
```

This function is similar to the `dplyr::near()` function.

**Arguments**:

- a number to test (`double`)
- another number to test (`double`)
- optional argument `<eps>`, which is the tolerance for establishing equality between the two test numbers; `<eps>` defaults to `1e-8`

**Return**: `bool`

### 9.4.6 Class to implement a dosing regimen

The `evtools` namespace also includes a class for implementing "automatic" dosing in a regimen. The documentation presented here will be limited to a brief discussion of the constructor and member functions for this class. More is written in Chapter 10 about how you can use this class effectively.

- The constructor `evt::regimen::regimen()` does not take any arguments, but it dose call the `reset()` method.
- `void init(self)` initializes the object; the argument is the `self` object (see Section 2.3.12)
- `void reset()` resets the object to sensible defaults
  - dose compartment is set to 1
  - dose amount is set to 0
  - infusion rate is set to 0
  - dosing interval is set to 1e9

  - dosing duration is set to 1e9
  - other internal configuration

A series of setter functions let you set different attributes for the dosing regimen. All of the following functions return `void`. In the examples below, `object` refers to an object with class `evt::regimen`.

- `object.amt(<double>)` sets the dose amount
- `object.cmt(<int>)` sets the dosing compartment number
- `object.rate(<double>)` sets the infusion rate
- `object.ii(<double>)` sets the dosing interval
- `object.until(<double>)` sets the time of the last dose

Similarly, there are a set of getter functions to return these data members

- `double object.amt()` returns the dose amount
- `int object.cmt()` returns the dosing compartment number
- `double object.rate()` returns the infusion rate
- `double object.ii()` returns the dosing interval
- `double object.until()` return the time of the last dose

To start the dose regimen, call

```
object.execute()
```

This should almost always be called in `$TABLE` (i.e., `$ERROR`).

To force the simulation to stop at the time of the next dose with `EVID` set to 3333, use the `flagnext()` member function

```
object.flagnext();
```

This is usually set once at the start of the problem, either in `$PREAMBLE` or in `$MAIN` when `NEWIND <= 1`.

## 9.5 CXX11

**Purpose**

Compile your model file with `C++11` standard.

**Usage**

```
$PLUGIN CXX11
```

## 9.6 Rcpp

**Purpose**

Link to `Rcpp` headers into your model.

**Usage**

```
$PLUGIN Rcpp
```

Note that once your model is linked to `Rcpp`, you can start using that functionality immediately (without including `Rcpp.h`).

A very useful feature provided by `Rcpp` is that it exposes all of the `dpqr` functions that you normally use in R (e.g. `rnorm()` or `runif()`). So, if you want to simulate a number from Uniform (0,1) you can write

```
$PLUGIN Rcpp

$TABLE
double uni = R::runif(0,1);
```

Note that the arguments are the same as the R version (`?runif`) **except** there is no `n` argument; you always only get one draw.

Information about `Rcpp` can be found here: https://github.com/RcppCore/Rcpp

## 9.7 mrgx

Compile in extra C++ / Rcpp functions that can be helpful to you for more advanced model coding. The `mrgx` plugin is dependent on the `Rcpp` plugin.

The functions provided by `mrgx` are in a namespace of the same name, so to invoke these functions, you always prepend `mrgx::`.

### 9.7.1 Get the model environment

Note that your model object (`mod`) contains an R environment. For example

```
mrgsolve::house()@envir
```

```
. <environment: 0x10b5a9990>
```

The objects in this environment are created by a block called `$ENV` in your model code (see Section 2.2.27);

To access this environment in your model, call

```
Rcpp::Environment env = mrgx::get_envir(self);
```

## 9.8 Extract an object from the model environment

When you have an object created in `$ENV`

```
[ env ]
rand <- rnorm(100)
```

You can extract this object with

```
[ preamble ]
Rcpp::NumericVector draw = mrgx::get("rand", self);
```

## 9.9 RcppArmadillo

**Purpose**

Link to `RcppArmadillo` headers into your model.

**Usage**

```
$PLUGIN RcppArmadillo
```

Information about `armadillo` can be found here: http://arma.sourceforge.net/ Information about `RcppArmadillo` can be found here: https://github.com/RcppCore/RcppArmadillo

## 9.10 BH

### Purpose

Link to `boost` headers into your model.

### Usage

```
$PLUGIN BH
```

Note that once your model is linked to `BH` (`boost`), you will be able to include the `boost` header file that you need. You have to include the header file that contains the `boost` function you want to use.

Information about `boost` can be found here: https://boost.org. Information about `BH` can be found here: https://github.com/eddelbuettel/bh

# 10 Modeled events

Modeled events are interventions that you can introduce into your simulation from within your model. These aren't any different in substance to the dosing records (`EVID=1`) or other intervention type records (`EVID=2`) that you might include in your input data set when you know what they are before you run the simulation. Modeled events do the same thing (stop the simulation and execute some event at some time) but you don't need to know about them prior to running the simulation. These are similar to the `MTIME` functionality that you get in NONMEM but they have a very different syntax and there is more functionality provided.

Note that there is no way to get additional records in your simulated output. Regardless of the approach or level of complexity, you will not see modeled events as separate rows in your simulated output. These are always executed under the hood and the number of rows in the simulated output and their times will be the same regardless of what modeled events you set up as discussed here.

## 10.1 `evtools` plugin

`evtools` is a plugin that runs on top of the

## 10.2 Simple MTIME

Use this when you just want to introduce a non-dose discontinuity in your simulation at a specific time. For example, you want a parameter to change value at a specific time and you don't know about the values or times prior to simulating.

To schedule a discontinuity, call the `mtime()` member (Section 2.3.22) of the `self` object (Section 2.3.12). This is typically done in the `$MAIN` block.

```
[ main ]
double mt = self.mtime(14.12);

if(TIME >= mt) {
  // do something
}
```

Here, we have told `mrgsolve` to stop at `14.12` hours so we can do something. Notice that `self.mtime()` returns the value of the modeled even time so you can check it later.

We can also schedule an event to happen some amount of time in the future

```
[ main ]
if(NEWIND <= 1) {
  double mt = 1e9;
}

if(EVID==1) {
  mt = self.mtime(TIME + 8.2);
}

if(TIME >= mt) {
  // do something
}
```

## 10.3 MTIME with specific EVID

You can call `self.mevent()` and pass both `time` and `evid` and then check for when that `EVID` comes around again. For example

```
self.mevent(TIME + 8.2, 33);

if(EVID==33) {
  // do something
}
```

This is similar in functionality to `self.mevent()`.

## 10.4 Modeled doses

The previous examples showed you how to get the simulation to stop so you can do something in `$MAIN`. In this section, we show you how to schedule doses in a similar way. This will take some extra coding and will also serve to uncover how `self.mtime()` and `self.mevent()` work.

You can set up the following code in either `$MAIN` or `$TABLE`.

***Create an evdata object***

Once you know when you want the dose, create an `evdata` object.

```
mrg::evdata ev(14.2, 1);
```

This will create (construct) an object called `ev` with class `evdata`. The constructor takes two arguments:

1. the `TIME` the event should happen
2. the `EVID` for the event

This is the only available constructor for `evdata` objects. You can browse the source code for the `evdata` object here.

**Modify the evdata object** Once the object is created, you can modify the following public members

- `time`: the event time (`double`)
- `evid`: the event ID (`int`)
- `amt`: the dose amount (`double`)
- `cmt`: the compartment number (`int`)
- `rate`: the rate to infuse `amt` (`double`)
- `now`: should the dose be given immediately? (`bool`)
- `check_unique`: should the event log be checked for identical events? see Section 10.5 (`bool`)

If you are using this (lower-level) interface, chances are you will want to set at least `amt` and `cmt`. As an example, we will dose 100 mg into compartment 2 immediately (now)

```
ev.amt = 100;
ev.cmt = 2;
ev.now = true;
```

The other members are set in a similar way.

**Push the evdata object into the self object**

After the object has been created and modified, you have to attach this object to the `self` object in order to make it available to mrgsolve. Do this by calling `push_back()` on `self.mevector`

```
self.mevector.push_back(ev);
```

Starting with mrgsolve 1.4.1, you can also push the event object back via the `push` member

```
self.push(ev);
```

Again, this sequence should get called in either `$MAIN` or `$TABLE`. When that code block finishes running (for the current record), `mrgsolve` will find the event record and add that event to the simulation sequence.

### 10.4.1 `evtools` plugin

`evtools` is a plugin for executing modeled doses, running on top of the machinery here. The goal of the plugin is to create simplified workflows for executing doses or other discontinuities from inside your model. See Section 9.4 for details on how to use this plugin.

## 10.5 Event log - tracking duplicate events

The way modeled events are implemented in mrgsolve, it is possible to inadvertently ask to execute the same event a large number of times.

For example, when we have the following code

```
$PK
double mt = self.mtime(244, 33);
```

mrgsolve only needs to know once to stop the model at `time = 244` with `evid = 33`. Once that event is processed, we don't need more records piling up at `time = 244` with that evid.

To make sure we only get one record with `evid = 33` at `time = 244`, mrgsolve keeps a log of modeled events and checks that log whenever a new event comes back from the model code. If there is already a record with `time = 244` and `evid = 33`, it declines to schedule an additional / identical record in the record stack; only one is needed. Once mrgsolve starts working on a new individual, the event log is reset to allow events to get scheduled at any time for that individual.

Currently mrgsolve checks the following event attributes for uniqueness

- `time` - event time
- `evid` - event id
- `amt` - dose amount
- `cmt` - dose compartment

An exception to this check for uniqueness is when the `now` attribute is set on modeled events (Section 10.4); this most commonly happens with doses that the user wants to trigger immediately. Because these events happen "now", we assume that you really want this event to happen; that is, we assume that some code is in place that has checked to make sure it is the right `time` to execute this event. A similar argument could be made for *any* dosing event, but for the time being, doses that are scheduled for the future (even doses that we want to happen at the current time but are not marked with the `now` attribute).

I think this is the behavior we want *most of the time*. In case you do want multiple doses into the same compartment with the same amount at the same time, there is a `check_unique` attribute that you can set to `false`. This will bypass the check of the event log and execute that event without checking for duplicates. The `check_unique` attribute is only needed for events scheduled for the future (not `now`). Using the `check_unique` attribute to bypass the check of the event log could theoretically help simulation speed when a large number of events are added to the log. I don't expect a huge speed difference, but it might be worth a try.

# 11 Topics

## 11.1 Annotated model specification

Here is a complete annotated `mrgsolve` model. The goal was to get in several of the most common blocks that you might want to annotate. The different code blocks are rendered here separately for clarity in presentation; but users should include all relevant blocks in a single file (or R string).

```
$PROB

# Final PK model

- Author: Pmetrics Scientist
- Client: Pharmaco, Inc.
- Date: `r Sys.Date()`
- NONMEM Run: 12345
- Structure: one compartment, first order absorption
- Implementation: closed form solutions
- Error model: Additive + proportional
- Covariates:
  - WT on clearance
- SEX on volume
- Random effects on: `CL`, `V`, `KA`
```

```
[PARAM] @annotated
TVCL : 1.1   : Clearance (L/hr)
TVV  : 35.6  : Volume of distribution (L)
TVKA : 1.35  : Absorption rate constant (1/hr)
WT   : 70    : Weight (kg)
SEX  : 1     : Male = 0, Female 1
WTCL : 0.75  : Exponent weight on CL
SEXV : 0.878 : Volume female/Volume male
```

```
[MAIN]
double CL = TVCL*pow(WT/70,WTCL)*exp(ECL);
double V  = TVV *pow(SEXVC,SEX)*exp(EV);
double KA = TVKA*exp(EKA);
```

```
[OMEGA] @name OMGA @correlation @block @annotated
ECL : 1.23 : Random effect on CL
EV  : 0.67 0.4 : Random effect on V
EKA : 0.25 0.87 0.2 : Random effect on KA
```

```
[SIGMA] @name SGMA @annotated
PROP: 0.25 : Proportional residual error
ADD : 25   : Additive residual error
```

```
[CMT] @annotated
GUT  : Dosing compartment  (mg)
CENT : Central compartment (mg)
```

```
[PKMODEL] ncmt = 1, depot=TRUE
```

```
[TABLE]
capture IPRED = CENT/V;
double DV = IPRED*(1+PROP) + ADD;
```

```
[CAPTURE] @annotated
DV  : Concentration (mg/L)
ECL : Random effect on CL
CL  : Individual clearance (L/hr)
```

## 11.2 Set initial conditions

```
library(mrgsolve)
library(dplyr)
```

### 11.2.1 Summary

- `mrgsolve` keeps a base list of compartments and initial conditions that you can update **either** from `R` or from inside the model specification

- When you use `$CMT`, the value in that base list is assumed to be 0 for every compartment
- `mrgsolve` will by default use the values in that base list when starting the problem
- When only the base list is available, every individual will get the same initial condition
- You can **override** this base list by including code in `$MAIN` to set the initial condition
- Most often, you do this so that the initial is calculated as a function of a parameter
- For example, `$MAIN RESP_0 = KIN/KOUT;` when `KIN` and `KOUT` have some value in `$PARAM`
- This code in `$MAIN` overwrites the value in the base list for the current `ID`
- For typical PK/PD type models, we most frequently initialize in `$MAIN`
- This is equivalent to what you might do in your NONMEM model
- For larger systems models, we often just set the initial value via the base list

### 11.2.2 Make a model only to examine `init` behavior

Note: `IFLAG` is my invention only for this demo. The demo is always responsible for setting and interpreting the value (it is not reserved in any way and `mrgsolve` does not control the value).

For this demo

- Compartment `A` initial condition defaults to 0
- Compartment `A` initial condition will get set to `BASE` **only** if `IFLAG > 0`
- Compartment `A` always stays at the initial condition

```
code <- '
$PARAM BASE=100, IFLAG = 0

$CMT A

$MAIN

if(IFLAG > 0) A_0 = BASE;

$ODE dxdt_A = 0;
'
```

```
mod <- mcode("init",code)
```

#### Check the initial condition

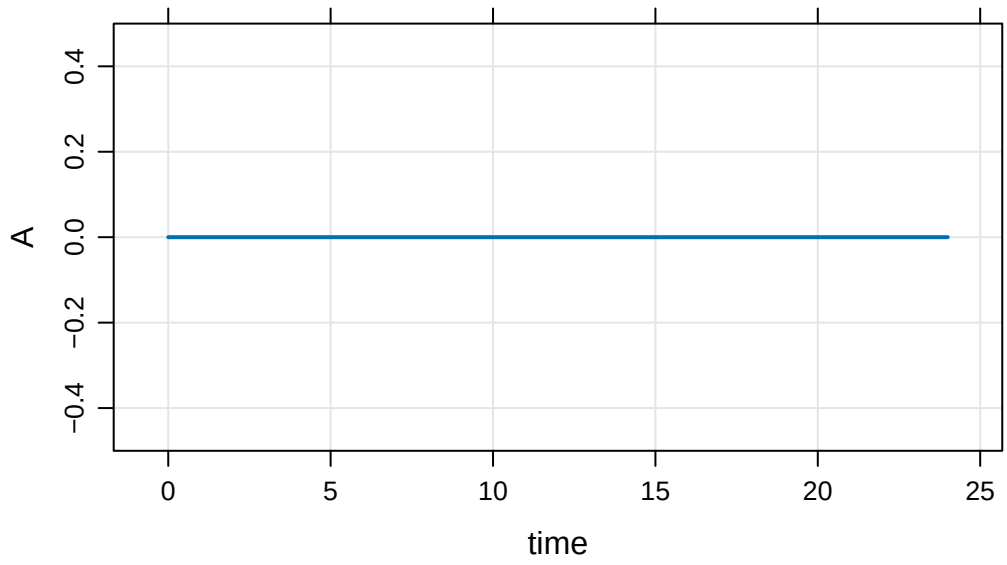
```
init(mod)
```

```
.
.  Model initial conditions (N=1):
.  name    value . name    value
.  A (1)   0     | . ...   .
```

Note:

- We used `$CMT` in the model spec; that implies that the base initial condition for `A` is set to 0
- In this chunk, the code in `$MAIN` doesn't get run because `IFLAG` is 0
- So, if we don't update something in `$MAIN` the initial condition is as we set it in the base list

```
mod %>% mrgsim() %>% plot()
```

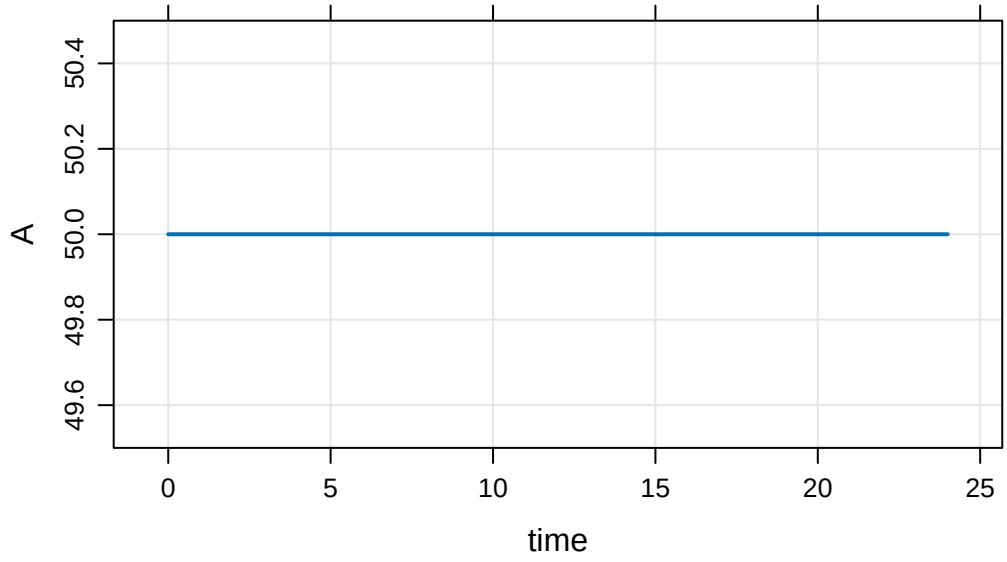



**Next, we update the base initial condition for `A` to 50**

Note:

- The code in `$MAIN` still doesn't get run because `IFLAG` is 0

```
mod %>% init(A = 50) %>% mrgsim() %>% plot()
```

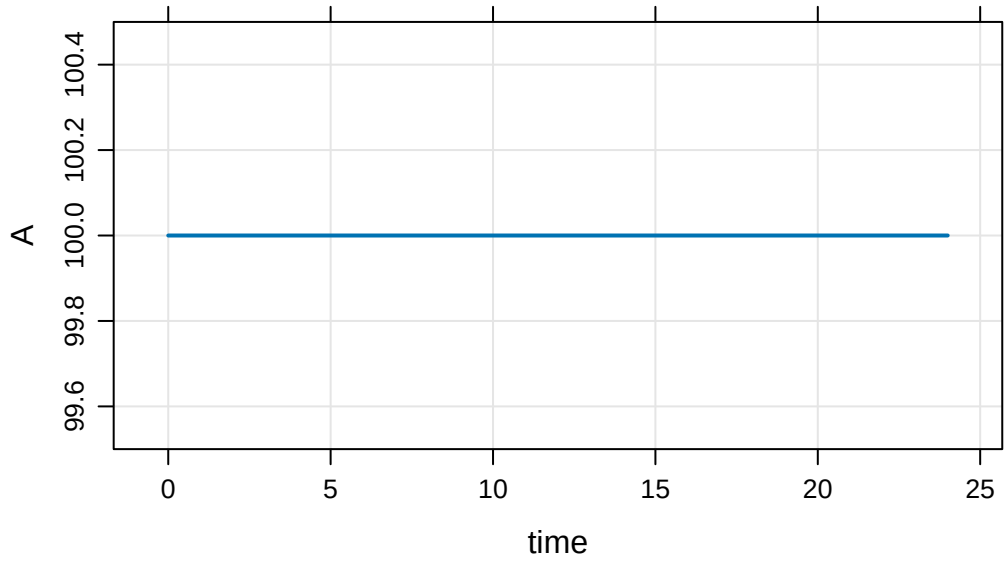



**Now, turn on `IFLAG`**

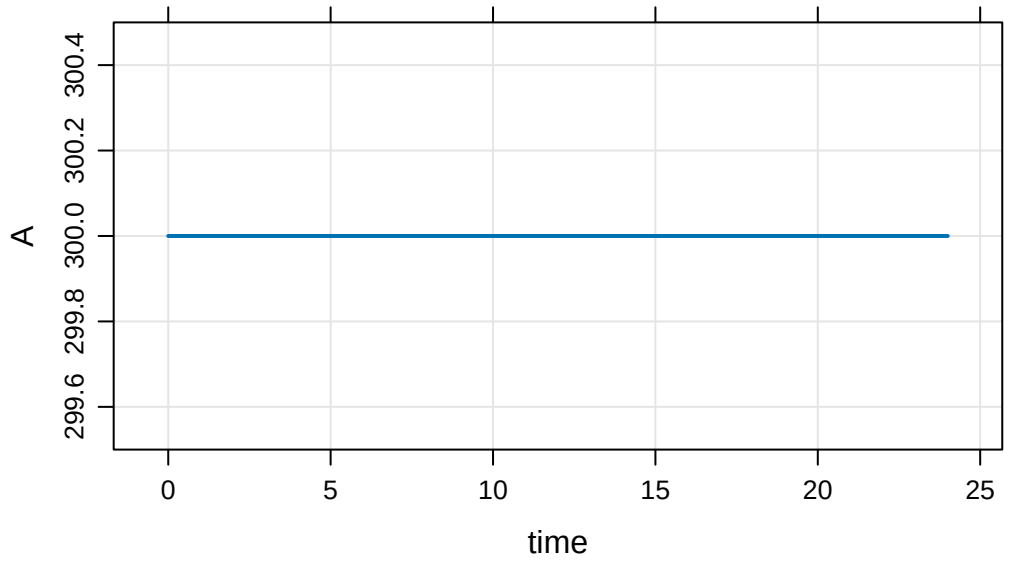
Note:

- Now, that code in `$MAIN` gets run
- `A_0` is set to the value of `BASE`

```
mod %>% param(IFLAG=1) %>% mrgsim() %>% plot()
```



```
mod %>% param(IFLAG=1, BASE=300) %>% mrgsim() %>% plot()
```




### 11.2.3 Example PK/PD model with initial condition

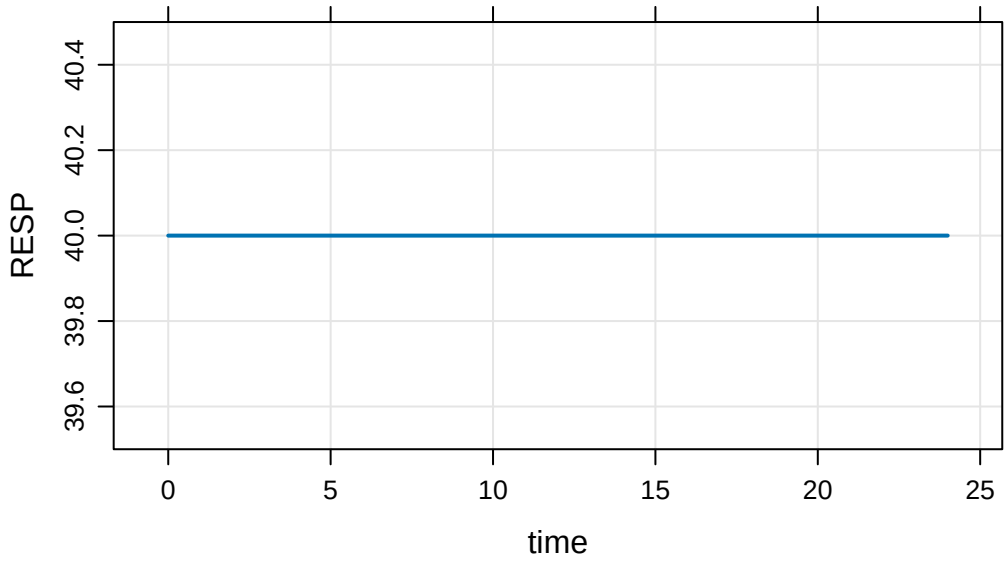
Just to be clear, there is no need to set any sort of flag to set the initial condition as seen here:

```
code <- '
$PARAM AUC=0, AUC50 = 75, KIN=200, KOUT=5

$CMT RESP

$MAIN
RESP_0 = KIN/KOUT;

$ODE

dxdt_RESP = KIN*(1-AUC/(AUC50+AUC)) - KOUT*RESP;

'
```

```
mod <- mcode("init2", code)
```

The initial condition is set to 40 per the values of `KIN` and `KOUT`

```
mod %>% mrgsim() %>% plot()
```




Even when we change `RESP_0` in `R`, the calculation in `$MAIN` gets the final say

```
mod %>% init(RESP=1E9) %>% mrgsim()
```

```
. Model:  init2
. Dim:    25 x 3
. Time:   0 to 24
. ID:     1
.     ID time RESP
. 1:   1    0   40
. 2:   1    1   40
. 3:   1    2   40
. 4:   1    3   40
. 5:   1    4   40
. 6:   1    5   40
. 7:   1    6   40
. 8:   1    7   40
```

### 11.2.4 Remember: calling `init` will let you check to see what is going on

- It's a good idea to get in the habit of doing this when things aren't clear
- `init` first takes the base initial condition list, then calls `$MAIN` and does any calculation you have in there; so the result is the calculated initials

```
init(mod)
```

```
.
.  Model initial conditions (N=1):
.  name       value . name    value
.  RESP (1)   40    | . ...   .
```

```
mod %>% param(KIN=100) %>% init()
```

```
.
.  Model initial conditions (N=1):
.  name       value . name    value
.  RESP (1)   20    | . ...   .
```

### 11.2.5 Set initial conditions via `idata`

Go back to house model

```
mod <- house()
```

```
init(mod)
```

```
.
.  Model initial conditions (N=3):
.  name       value . name       value
.  CENT (2)   0     | RESP (3)   50
.  GUT (1)    0     | . ...      .
```

Notes

- In `idata` (only), include a column with `CMT_0` (like you'd do in `$MAIN`).
- When each ID is simulated, the `idata` value will override the base initial list for that subject.
- But note that if `CMT_0` is set in `$MAIN`, that will override the `idata` update.

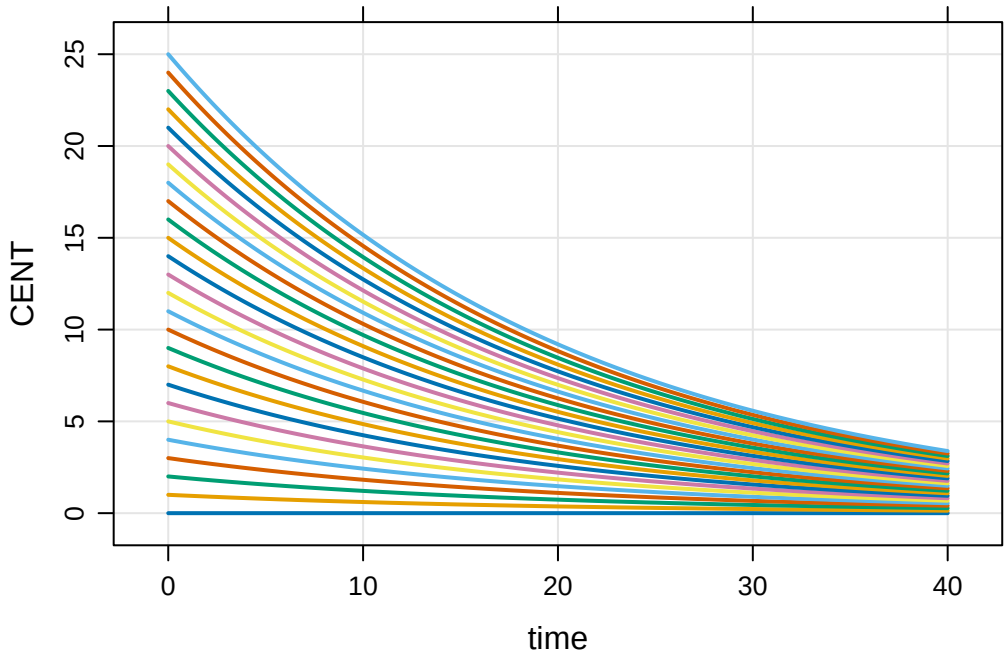
```
idata <- expand.idata(CENT_0 = seq(0,25,1))
```

```
idata %>% head()
```

```
.   ID CENT_0
. 1  1      0
. 2  2      1
. 3  3      2
. 4  4      3
. 5  5      4
. 6  6      5
```

```
out <-
  mod %>%
  idata_set(idata) %>%
  mrgsim(end=40)
```

```
plot(out, CENT~.)
```




## 11.3 Updating parameters

The parameter list was introduced in Section 1.1 and the `$PARAM` code block was shown in Section 2.2.4. Once a model is compiled, the names and number of parameters in a model is

fixed. However, the values of parameters can be changed: parameters may be updated either by the user (in `R`) or by `mrgsolve` (in the `C++` simulation engine, as the simulation proceeds).

- To update in `R`, use the `param()` function (see examples below)
- To have `mrgsolve` update the parameters, attach columns to your data set (either `data_set` or `idata_set`) with the same name as items in the parameter list

Both of these methods are discussed and illustrated in the following sections.

### 11.3.1 Parameter update hierarchy

As we noted above, new parameter values can come from three potential sources:

1. Modification of the (base) parameter list
2. A column in an `idata_set` that has the same name as a model parameter
3. A column in a `data_set` that has the same name as a model parameter

These sources for new parameter values are discussed below. We note here that the sources listed above are listed in the order of the parameter update *hierarchy*. So, the base parameter list provides the value by default. A parameter value coming from an `idata_set` will override the value in the base list. And a parameter value coming from a `data_set` will override the value coming from the base list or an `idata_set` (in case a parameter is listed in both the `idata_set` and the `data_set`). In other words, the hierarchy is:

1. base parameter list is the default
2. the `idata_set` overrides the base list
3. the `data_set` overrides the `idata_set` and the base list

The parameter update hierarchy is discussed in the following sections.

#### Base parameter set

- Every model has a base set of "parameters"
- These are named and set in `$PARAM`
- Parameters can only get into the parameter list in `$PARAM` (or `$THETA`)
- No changing the names or numbers of parameters once the model is compiled
- But, several ways to change the values

```
code <- '
$VCMT KYLE
$PARAM CL = 1.1, VC=23.1, KA=1.7, KM=10
$CAPTURE CL VC KA KM
'
mod <- mcode("tmp", code, warn=FALSE)
```

```
param(mod)
```

```
.
.  Model parameters (N=4):
.  name value . name value
.  CL   1.1   | KM   10
.  KA   1.7   | VC   23.1
```

### The base parameter set is the default

The base parameter set allows you to run the model without entering any other data; there are some default values in place.

### The parameters in the base list can be changed or updated in R

Use the `param()` function to both set and get:

```
mod <-  param(mod, CL=2.1)
```

```
param(mod)
```

```
.
.  Model parameters (N=4):
.  name value . name value
.  CL   2.1   | KM   10
.  KA   1.7   | VC   23.1
```

But whatever you've done in R, there is a base set (with values) to use. See Section 11.3.2 for a more detailed discussion of using `param()` to updated the base list.

### Parameters can also be updated during the simulation run

Parameters can be updated by putting columns in `idata` set or `data_set` that have the same name as one of the parameters in the parameter list. But there is no changing values in the base parameter set once the simulation starts.
That is, the following model specification will not compile:

```
$PARAM CL = 2

$MAIN CL = 3; // ERROR
```

You cannot over-write the value of a parameter in the model specification.
Let `mrgsolve` do the updating.

`mrgsolve` always reverts to the base parameter set when starting work on a new individual.

### Parameters updated from `idata_set`

When `mrgsolve` finds parameters in `idata`, it will update the base parameter list with those parameters prior to starting that individual.

```
data(exidata)
head(exidata)
```

```
.   ID    CL    VC     KA KOUT  IC50 FOO
. 1  1 1.050 47.80 0.8390 2.45 1.280   4
. 2  2 0.730 30.10 0.0684 2.51 1.840   6
. 3  3 2.820 23.80 0.1180 3.88 2.480   5
. 4  4 0.552 26.30 0.4950 1.18 0.977   2
. 5  5 0.483  4.36 0.1220 2.35 0.483  10
. 6  6 3.620 39.80 0.1260 1.89 4.240   1
```

Notice that there are several columns in `exidata` that match up with the names in the parameter list

```
names(exidata)
```

```
. [1] "ID"   "CL"   "VC"   "KA"   "KOUT" "IC50" "FOO"
```

```
names(param(mod))
```

```
. [1] "CL" "VC" "KA" "KM"
```

The matching names tell `mrgsolve` to update, assigning each individual their individual parameter.

```
out <-
  mod %>%
  idata_set(exidata) %>%
  mrgsim(end=-1 , add=c(0,2))
```

```
out
```

```
. Model:  tmp
. Dim:    20 x 7
. Time:   0 to 2
. ID:     10
.     ID time KYLE    CL   VC     KA KM
. 1:   1    0    0 1.050 47.8 0.8390 10
. 2:   1    2    0 1.050 47.8 0.8390 10
. 3:   2    0    0 0.730 30.1 0.0684 10
. 4:   2    2    0 0.730 30.1 0.0684 10
. 5:   3    0    0 2.820 23.8 0.1180 10
. 6:   3    2    0 2.820 23.8 0.1180 10
. 7:   4    0    0 0.552 26.3 0.4950 10
. 8:   4    2    0 0.552 26.3 0.4950 10
```

### Parameters updated from `data_set`

Like an `idata` set, we can put parameters on a `data` set

```
data <- expand.ev(amt=0, CL=c(1,2,3), VC=30)
```

```
out <-
  mod %>%
  data_set(data) %>%
  obsonly %>%
  mrgsim(end=-1, add=c(0,2))
```

```
out
```

```
. Model:  tmp
. Dim:    6 x 7
. Time:   0 to 2
. ID:     3
.     ID time KYLE CL VC  KA KM
. 1:   1    0    0  1 30 1.7 10
. 2:   1    2    0  1 30 1.7 10
. 3:   2    0    0  2 30 1.7 10
. 4:   2    2    0  2 30 1.7 10
. 5:   3    0    0  3 30 1.7 10
. 6:   3    2    0  3 30 1.7 10
```

This is how we do time-varying parameters:

```
data <-
  data_frame(CL=seq(1,5)) %>%
  mutate(evid=0,ID=1,cmt=1,time=CL-1,amt=0)
```

```
. Warning: `data_frame()` was deprecated in tibble 1.1.0.
. i Please use `tibble()` instead.
```

```
mod %>%
  data_set(data) %>%
  mrgsim(end=-1)
```

```
. Model:  tmp
. Dim:    5 x 7
. Time:   0 to 4
. ID:     1
.     ID time KYLE CL   VC  KA KM
. 1:   1    0    0  1 23.1 1.7 10
. 2:   1    1    0  2 23.1 1.7 10
. 3:   1    2    0  3 23.1 1.7 10
. 4:   1    3    0  4 23.1 1.7 10
. 5:   1    4    0  5 23.1 1.7 10
```

For more information on time-varying covariates (parameters), see Section 11.8 and Chapter 7.

**Parameters are carried back when first record isn't at `time == 0`**

What about this?

```
data <- expand.ev(amt=100,time=24,CL=5,VC=32)
data
```

```
.   ID time amt cmt evid CL VC
. 1  1   24 100   1    1  5 32
```

The first `data` record happens at `time==24`

```
mod %>%
  data_set(data) %>%
  mrgsim(end=-1, add=c(0,2))
```

```
. Model:  tmp
. Dim:    3 x 7
. Time:   0 to 24
. ID:     1
.     ID time KYLE CL VC  KA KM
. 1:   1    0    0  5 32 1.7 10
. 2:   1    2    0  5 32 1.7 10
. 3:   1   24  100  5 32 1.7 10
```

Since the data set doesn't start until `time==5`, we might think that `CL` doesn't change from the base parameter set until then.

But by default, `mrgsolve` carries those parameter values back to the start of the simulation. This is by design … by far the more useful configuration.

If you wanted the base parameter set in play until that first data set record, do this:

```
mod %>%
  data_set(data) %>%
  mrgsim(end=-1,add=c(0,2), filbak=FALSE)
```

```
. Model:  tmp
. Dim:    3 x 7
. Time:   0 to 24
. ID:     1
.     ID time KYLE CL VC  KA KM
. 1:   1    0    0  5 32 1.7 10
. 2:   1    2    0  5 32 1.7 10
. 3:   1   24  100  5 32 1.7 10
```

Will this work?

```
idata <- do.call("expand.idata", as.list(param(mod)))

idata
```

```
.   ID  CL   VC  KA KM
. 1  1 2.1 23.1 1.7 10
```

Here, we'll pass in **both** `data_set` and `idata_set` and they have conflicting values for the parameters.

```
mod %>%
  data_set(data) %>%
  idata_set(idata) %>%
  mrgsim(end=-1,add=c(0,2))
```

```
. Model:  tmp
. Dim:    3 x 7
. Time:   0 to 24
. ID:     1
.     ID time KYLE CL VC  KA KM
. 1:   1    0    0  5 32 1.7 10
. 2:   1    2    0  5 32 1.7 10
. 3:   1   24  100  5 32 1.7 10
```

The data set always gets the last word.

### 11.3.2 Updating the base parameter list

From the previous section

```
param(mod)
```

```
.
.  Model parameters (N=4):
.  name value . name value
.  CL   2.1   | KM   10
.  KA   1.7   | VC   23.1
```

#### Update with `name-value` pairs

We can call `param()` to update the model object, directly naming the parameter to update and the new value to take

```
mod %>% param(CL = 777, KM = 999) %>% param
```

```
.
.  Model parameters (N=4):
.  name value . name value
.  CL   777   | KM   999
.  KA   1.7   | VC   23.1
```

The parameter list can also be updated by scanning the names in a list

```
what <- list(CL = 555, VC = 888, KYLE = 123, MN = 100)

mod %>% param(what) %>% param
```

```
.
.  Model parameters (N=4):
.  name value . name value
.  CL   555   | KM   10
.  KA   1.7   | VC   888
```

`mrgsolve` looks at the names to drive the update. `KYLE` (a compartment name) and `MN` (not in the model anywhere) are ignored.

Alternatively, we can pick a row from a data frame to provide the input for the update

```
d <- data_frame(CL=c(9,10), VC=c(11,12), KTB=c(13,14))

mod %>% param(d[2,]) %>% param
```

```
.
.  Model parameters (N=4):
.  name value . name value
.  CL   10    | KM   10
.  KA   1.7   | VC   12
```

Here the second row in the data frame drives the update. Other names are ignored.

A warning will be issued if an update is attempted, but no matching names are found

```
mod %>% param(ZIP = 1, CODE = 2) %>% param
```

```
Warning message:
Found nothing to update: param
```

## 11.4 Time grid objects

### Simulation times in `mrgsolve`

```
mod <- mrgsolve:::house() %>% Req(CP) %>% ev(amt = 1000, ii = 24, addl = 1000)
```

`mrgsolve` keeps track of a simulation `start` and `end` time and a fixed size step between `start` and `end` (called `delta`). `mrgsolve` also keeps an arbitrary vector of simulation times called `add`.

```
mod %>%
  mrgsim(end = 4, delta = 2, add = c(7,9,50)) %>%
  as.data.frame()
```

```
.   ID time       CP
. 1  1    0  0.00000
. 2  1    0  0.00000
. 3  1    2 42.47580
. 4  1    4 42.28701
. 5  1    7 36.75460
. 6  1    9 33.26649
. 7  1   50 60.97754
```

### `tgrid` objects

The `tgrid` object abstracts this setup and allows us to make complicated sampling designs from elementary building blocks.

### Make a day 1 sampling with intensive sampling around the peak and sparser otherwise

```
peak1 <- tgrid(1, 4, 0.1)
sparse1 <- tgrid(0, 24, 4)
```

### Use the `c` operator to combine simpler designs into more complicated designs

```
day1 <- c(peak1, sparse1)
```

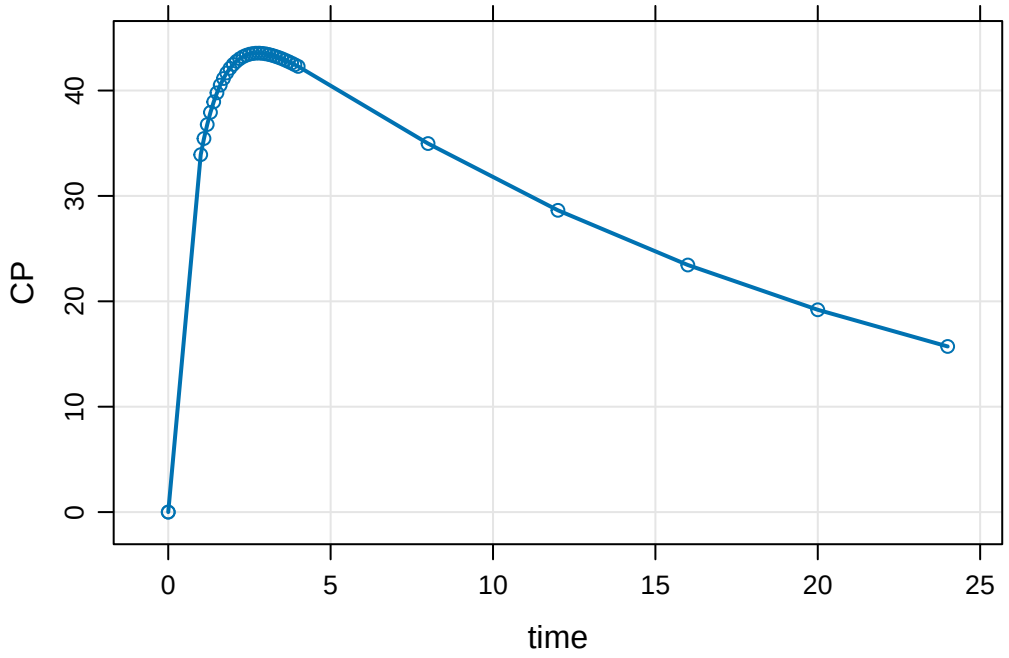
Check this by calling `stime`

```
stime(day1)
```

```
.  [1]  0.0  1.0  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9  2.0  2.1  2.2  2.3
. [16]  2.4  2.5  2.6  2.7  2.8  2.9  3.0  3.1  3.2  3.3  3.4  3.5  3.6  3.7  3.8
. [31]  3.9  4.0  8.0 12.0 16.0 20.0 24.0
```

Pass this object in to `mrgsim` as `tgrid`. It will override the default `start/end/delta/add` sequence.

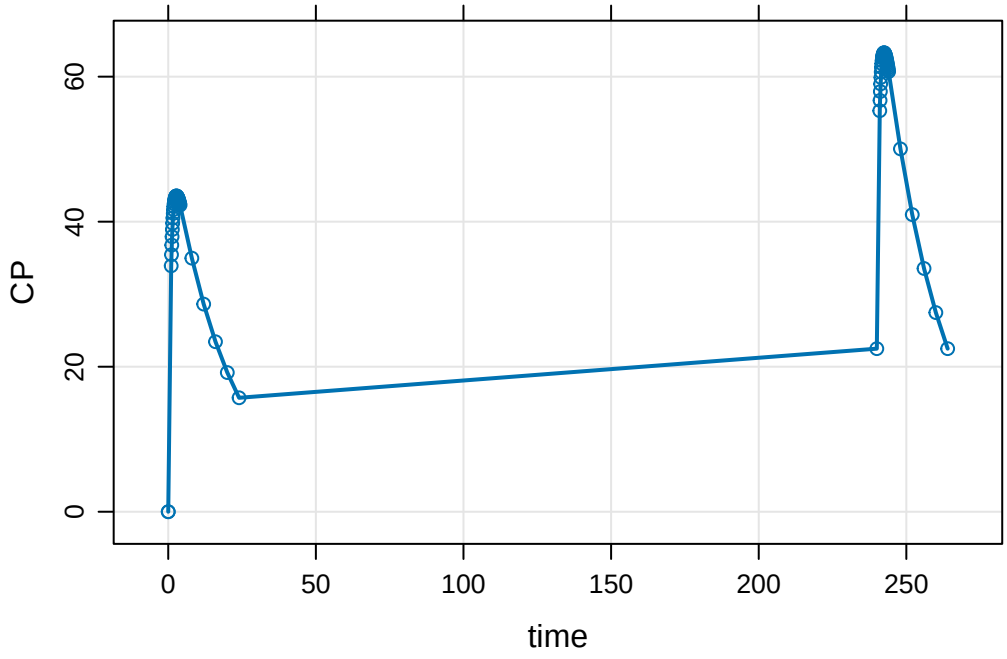
```
mod %>%
  mrgsim(tgrid = day1) %>%
  plot(type = 'b')
```




**Now, look at both day 1 and day 10**:

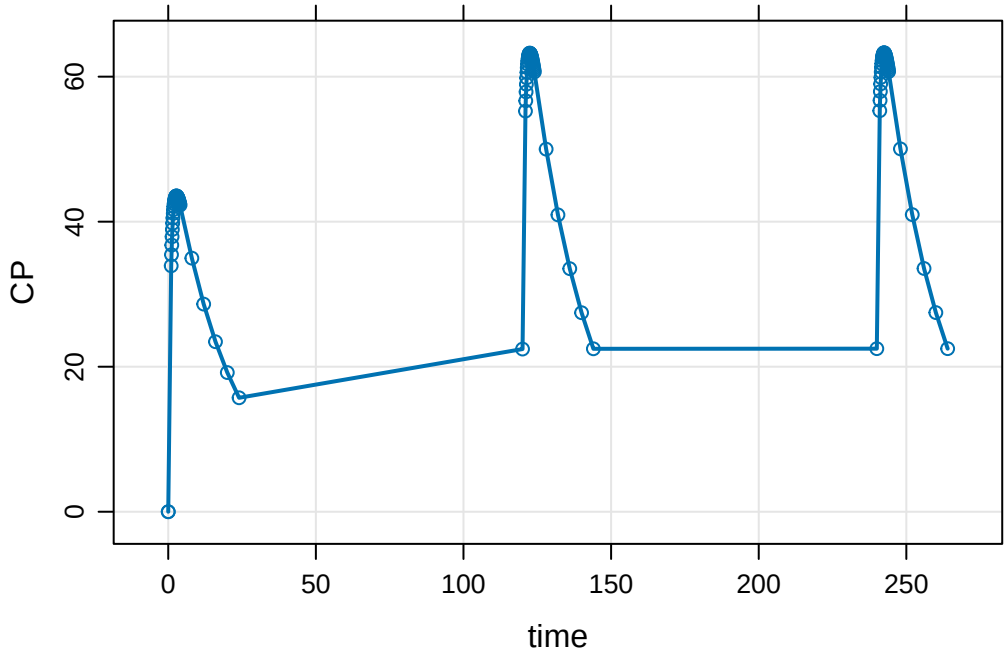
Adding a number to a `tgrid` object will offset those times by that amount.

```
des <- c(day1, day1 + 10*24)

mod %>%
  mrgsim(tgrid = des) %>%
  plot(type = 'b')
```




Pick up day 5 as well

```
des <- c(des, day1 + 5*24)

mod %>%
  mrgsim(tgrid = des) %>%
  plot(type = 'b')
```




## 11.5 Individualized sampling designs

Here is a PopPK model and a full `data_set`.

```
mod <- house()

data(exTheoph)

df <- exTheoph

head(df)
```

```
.   ID   WT Dose time  conc cmt  amt evid
. 1  1 79.6 4.02 0.00  0.00   1 4.02    1
. 2  1 79.6 4.02 0.25  2.84   0 0.00    0
. 3  1 79.6 4.02 0.57  6.57   0 0.00    0
. 4  1 79.6 4.02 1.12 10.50   0 0.00    0
. 5  1 79.6 4.02 2.02  9.66   0 0.00    0
. 6  1 79.6 4.02 3.82  8.58   0 0.00    0
```

```
mod %>%
  Req(CP) %>%
  carry.out(a.u.g) %>%
  data_set(df) %>%
  obsaug() %>%
  mrgsim()
```

```
. Model:  housemodel
. Dim:    5904 x 4
. Time:   0 to 120
. ID:     12
.     ID time a.u.g      CP
. 1:   1 0.00     1 0.00000
. 2:   1 0.00     0 0.00000
. 3:   1 0.25     1 0.04552
. 4:   1 0.25     0 0.04552
. 5:   1 0.50     1 0.07870
. 6:   1 0.57     0 0.08624
. 7:   1 0.75     1 0.10274
. 8:   1 1.00     1 0.12001
```

Now, define two time grid objects: `des1` runs from 0 to 24 and `des2` runs from 0 to 96, both every hour.

```
des1 <- tgrid(0, 24, 1)
des2 <- tgrid(0, 96, 1)

range(stime(des1))
```

```
. [1]  0 24
```

```
range(stime(des2))
```

```
. [1]  0 96
```

Now, derive an `idata_set` after adding a grouping column (`GRP`) that splits the data set into two groups

```
df <-  mutate(df, GRP = as.integer(ID > 5))

id <- distinct(df, ID, GRP)

id
```

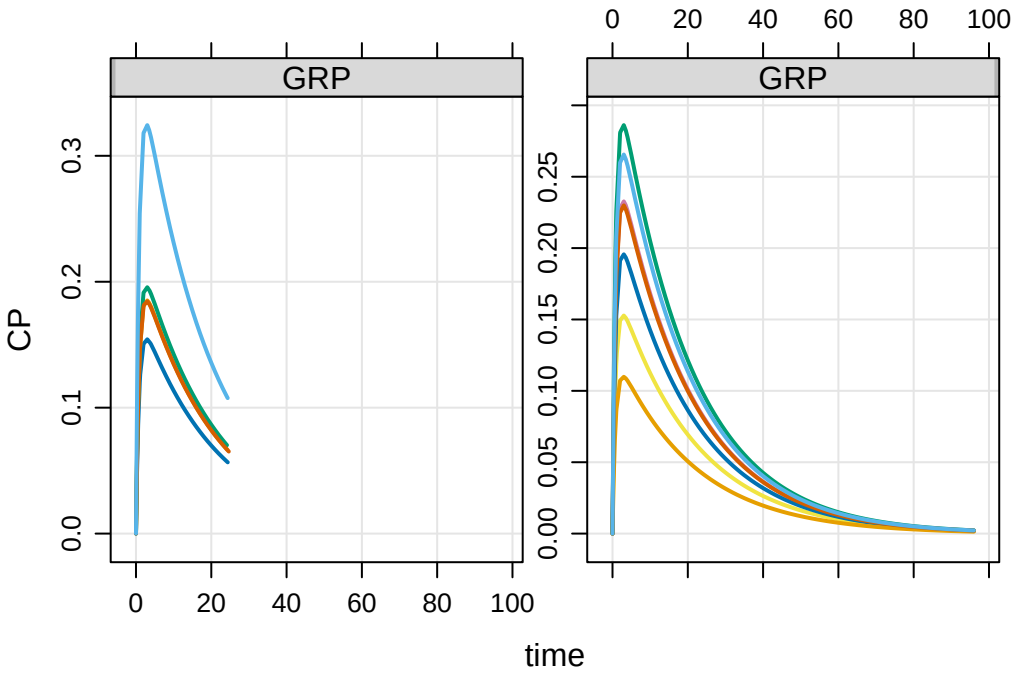
```
.    ID GRP
. 1   1   0
. 2   2   0
. 3   3   0
. 4   4   0
. 5   5   0
. 6   6   1
. 7   7   1
. 8   8   1
. 9   9   1
. 10 10   1
. 11 11   1
. 12 12   1
```

Now, we have two groups in `GRP` in `idata_set` and we have two `tgrid` objects.

- Pass in both the `idata_set` and the `data_set`
- Call `design`
- Identify `GRP` as `descol`; the column **must** be in `idata_set`
- Pass in a list of designs; it **must** be at least two because there are two levels in `GRP`

When we simulate, the individuals in `GRP 1` will get `des1` and those in `GRP 2` will get `des2`

```
out <-
  mod %>%
  Req(CP) %>%
  carry.out(a.u.g,GRP) %>%
  idata_set(id) %>%
  data_set(df) %>%
  design(descol = "GRP", deslist = list(des1, des2)) %>%
  obsaug() %>%
  mrgsim()

plot(out, CP~time|GRP)
```




## 11.6 Some helpful C++

Recall that the following blocks require valid C++ code:

1. $PREAMBLE
2. $MAIN
3. $ODE
4. $TABLE
5. $GLOBAL
6. $PRED

We don't want users to have to be proficient in C++ to be able to use mrgsolve. and we've created several macros to help simplify things as much as possible.
However, it is required to become familiar with some of the basics and certainly additional knowledge of how to do more than just the basics will help you code more and more complicated models in mrgsolve.

There are an unending stream of tutorials, references and help pages on C++ to be found on the interweb. As a general source, I like to use https://en.cppreference.com/. But, again, there many other good resources out there that can suit your needs.

The rest of this section provides a very general reference of the types of C++ code and functions that you might be using in your model.

### 11.6.1 Semi-colons

Every statement in C++ must end with a semi-colon. For example;

```
[ MAIN ]
double CL = exp(log_TVCL + ETA(1));
```

or

```
[ ODE ]

dxdt_DEPOT = -KA * DEPOT;
```

### 11.6.2 if-else

```
if(a == 2) b = 2;
```

```
if(a==2) {
  b = 2;
}
```

```
if(a == 2) {
  b=2;
} else {
  b=3;
}
```

This is the equivalent of `x <- ifelse(c == 4, 8, 10)` in R

```
double x = c == 4 ? 8 : 10;
```

### 11.6.3 Functions

The following functions are hopefully understandable based on the function name. Consult https://cppreference.com for further details.

```
# base^exponent
double d = pow(base,exponent);

double e = exp(3);

# absolute value
double f = fabs(-4);

double g = sqrt(5);

double h = log(6);

double i = log10(7);

double j = floor(4.2);

double k = ceil(4.2);

double l = std::max(0.0, -3.0);

double m = std::min(0.0, -3.0);
```

### 11.6.4 Integer division

The user is warned about division with two integers. In `R`, the following statement evaluates to `0.75`:

```
3/4
```

```
. [1] 0.75
```

But in `C++` it evaluates to 0:

```
double x = 3/4;
```

This is because both the `3` and the `4` are taken as integer literals. This produces the same result as

```
int a = 3;
int b = 4;
double x = a/b;
```

When one integer is divided by another integer, the remainder is discarded (the result is rounded down). This is the way `C++` works. The user is warned.

Note that parameters in mrgsolve are `doubles` so this will evaluate to `0.75`

```
[ PARAM ] a = 3

[ MAIN ]
double x = a/4;
```

Since `a` is a parameter the operation of `a/4` is not integer division and the result is `0.75`.

Unless you are already very comfortable with this concept, users are encouraged to add `.0` suffix to any literal number **written as C++ code**. For example:

```
double x = 3.0 / 4.0;
```

I think it's fair to say that the vast majority of time you want this to evaluate to `0.75` and writing `3.0/4.0` rather than `3/4` will ensure you will not discard any remainder here.

If you would like to experiment with these concepts, try running this code

```
library(mrgsolve)

code <- '
[ param ] a = 3

[ main ]
capture x = 3/4;
capture y = 3.0/4.0;
capture z = a/4;
'
mod <- mcode("foo", code)

mrgsim(mod)
```

```
. Model:  foo
. Dim:    25 x 5
. Time:   0 to 24
. ID:     1
.     ID time x    y    z
. 1:   1    0 0 0.75 0.75
. 2:   1    1 0 0.75 0.75
. 3:   1    2 0 0.75 0.75
. 4:   1    3 0 0.75 0.75
. 5:   1    4 0 0.75 0.75
. 6:   1    5 0 0.75 0.75
. 7:   1    6 0 0.75 0.75
. 8:   1    7 0 0.75 0.75
```

### 11.6.5 Pre-processor directives

Pre-processor directives are global substitutions that are made in your model code at the time the model is compiled. For example

```
$GLOBAL

#define CP (CENT/VC)
```

When you write this into your model, the pre-processor will find every instance of `CP` and **replace** it with `(CENT/VC)`. This substitution happens right as the model is compiled; you won't see this substitution happen anywhere, but think of it as literal replacement of `CP` with `(CENT/VC)`.

**Note**:

- Put all pre-processor directives in `$GLOBAL`.
- It is usually a good idea to enclose the substituted coded in parentheses; this ensures that, for example, `CENT/VC` is evaluated as is, regardless of the surrounding code where it is evaluated.
- Under the hood, mrgsolve uses lots of pre-processor directives to define parameter names, compartment names and other variables; you will see a compiler error if you try to re-define an existing pre-processor directive. If so, just choose another name for your directive.

## 11.7 Resimulate ETA and EPS

### Call simeta() to resimulate ETA

- No $PLUGIN is required
- simeta() takes no arguments; all ETA values are resimulated

For example, we can simulate individual-level covariates within a certain range:

```
code <- '
$PARAM TVCL = 1, TVWT = 70

$MAIN
capture WT = TVWT*exp(EWT);

int i = 0;

while((WT < 60) || (WT > 80)) {
  if(++i > 100) break;
  simeta();
  WT = TVWT*exp(EWT);
}

$OMEGA @labels EWT
4

$CAPTURE EWT WT
'

mod <- mcode("simeta", code)

out <- mod %>% mrgsim(nid=100, end=-1)

sum <- summary(out)

sum
```

```
.        ID              time        EWT                 WT
.  Min.   :  1.00   Min.   :0   Min.   :-0.15326   Min.   :60.05
.  1st Qu.: 25.75   1st Qu.:0   1st Qu.:-0.08303   1st Qu.:64.42
.  Median : 50.50   Median :0   Median :-0.01841   Median :68.72
.  Mean   : 50.50   Mean   :0   Mean   :-0.01482   Mean   :69.19
```

```
.  3rd Qu.: 75.25   3rd Qu.:0   3rd Qu.: 0.05510   3rd Qu.:73.97
.  Max.   :100.00   Max.   :0   Max.   : 0.13246   Max.   :79.91
```

### Call `simeps()` to resimulate `EPS`

- No `$PLUGIN` is required
- `simeps()` takes no arguments; all `EPS` values are resimulated

For example, we can resimulate until all concentrations are greater than zero:

```
code <- '
$PARAM CL = 1, V = 20,

$CMT CENT

$SIGMA 50

$PKMODEL ncmt=1

$TABLE
capture CP = CENT/V + EPS(1);

int i = 0;

while(CP < 0 && i < 100) {
  simeps();
  CP = CENT/V + EPS(1);
  ++i;
}

'

mod <- mcode("simeps", code)

out <- mod %>% ev(amt=100) %>% mrgsim(end=48)
sum <- summary(out)

sum
```

```
.        ID         time            CENT               CP
.  Min.   :1   Min.   : 0.00   Min.   :  0.00   Min.   : 0.08704
.  1st Qu.:1   1st Qu.:11.25   1st Qu.: 15.93   1st Qu.: 2.54011
```

```
.  Median :1   Median :23.50   Median : 29.38   Median : 5.74223
.  Mean   :1   Mean   :23.52   Mean   : 37.47   Mean   : 6.47759
.  3rd Qu.:1   3rd Qu.:35.75   3rd Qu.: 54.21   3rd Qu.: 9.50077
.  Max.   :1   Max.   :48.00   Max.   :100.00   Max.   :17.52213
```

**A safety check is recommended** Note that in both examples, we implement a safety check: an integer counter is incremented every time we resimulated. The resimulation process stops if we don't reach the desired condition within 100 replicates. You might also consider issuing a message or a flag in the simulated data if you are not able to reach the desired condition.

## 11.8 Time varying covariates

A note in a previous section showed how to implement time-varying covariates or other time-varying parameters by including those parameters as column in the data set.

By default, `mrgsolve` performs next observation carried backward (`nocb`) when processing time-varying covariates. That is, when the system advances from `TIME1` to `TIME2`, and the advance is a function of a covariate found in the data set, the system advances using the covariate value `COV2` rather than the covariate `COV1`.

The user can change the behavior to last observation carried forward (`locf`), so that the system uses the value of `COV1` to advance from `TIME1` to `TIME2`. To use `locf` advance, set `nocb` to `FALSE` when calling `mrgsim`. For example,

```
mod %>% mrgsim(nocb = FALSE)
```

There is additional information about the sequence of events that takes place during system advance in Chapter 7.

# 12 Questions and Answers

I'm using this chapter as a place to provide miscellaneous information that might not have an obvious place to live. I'd call this a FAQ, but not all of the questions are asked frequently or at all.

## 12.1 Can I interrupt a simulation?

Starting with `mrgsolve` version `0.11.1`, you can interrupt a long simulation by pressing `Control-C` or `Esc`, the standard way to pass an interrupt signal through R. `mrgsolve` will stop every so often to look for the interrupt signal.

You can control the frequency with which `mrgsolve` looks for the interrupt signal through an argument to `mrgsim` (default: 256 simulation records). Increase to check less frequently, increase to check more frequently (this might be needed for a model where a large amount of work is required to advance one step) or set to negative number to never check.

## 12.2 Can I pass compiler flags to my model?

Compiler flags can be passed to your model by setting `PKG_CXXFLAGS` in `$ENV`. For example

```
$ENV
PKG_CXXFLAGS = "-std=c++11"
```

will compile your model according to `C++11` standard (but note that there is a special plugin that will do this automatically for you; see Section 9.5.

## 12.3 Can I compile my model with `C++11`?

Yes, you can do this by invoking the `CXX11` plugin (Section 9.5).

## 12.4 How can I calculate time after dose?

There are three approaches

**tad argument to mrgsim()**

To get time after dose into your output you can call

```
mrgsim(mod, tad = TRUE)
```

and the output will have a `tad` column. Note this does not let you interact with the `tad` value inside your model.

**Simple calculation in the model**

Most applications will call `self.tad()` (Section 2.3.21). For example

```
[ main ]
double tad = self.tad();
```

**More complicated calculation in the model**

You can get more control and track `tad` in a specific compartment by using the `tad` plugin. See Section 9.3 for details.

## 12.5 My model failed to compile; what now?

The model can fail to compile for a variety of reasons, including an error in the `C++` code or inability of R to find the compiler and other pieces of the tool chain.

If your model is not compiling, try the `recover` argument to `mread()` (or `mcode()`)

```
mod <- mread(..., recover = TRUE)
```

You will see a warning on the console and `mread()` will return a list of build information. You can look into that information or share it in the mrgsolve issue tracker on GitHub.

If your model has `C++` syntax problems, the errors should be printed on the console. If you possibly have problems with the compiler or the rest of the toolchain, take a look at the `pkgbuild` package, which provides some helpful tools, especially if you are working on a Windows platform

```
pkgbuild::check_build_tools()
pkgbuild::has_build_tools()
pkgbuild::has_rtools()
```

## 12.6 Can I run mrgsolve on a network drive?

No; do not run mrgsolve on a network drive. Your R installation, mrgsolve installation, and R working directory should be on a local hard disk.

## 12.7 Can I run mrgsolve on a cloud-synced folder?

No; do not run mrgsolve in a synced folder for cloud services like OneDrive, GoogleDrive, DropBox etc. Your R installation, mrgsolve installation, and R working directory should be on a local hard disk.

## 12.8 Can I run mrgsolve in a path that includes spaces?

No; do not run mrgsolve in a path that includes spaces. Your R installation, mrgsolve installation, and R working directory should all be in locations whose paths do not include spaces.

# 13 Installation

The most up to date installation instructions can be found on our `github` site:

https://github.com/metrumresearchgroup/mrgsolve/ https://github.com/metrumresearchgroup/mrgsolve/wiki
Installation