mrgsolve User Guide

Metrum Research Group

2021-04-21

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 5 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.

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 Wed Apr 21 08:53:37 2021 with mrgsolve version 0.11.0.

Chapter 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: 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 3.1 and in 7.3.

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 7.2 illustrates several of these.

See also: section 7.2 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 1.3.1 and 7.4 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 7.4 for examples and usage.

1.4 Solver settings

`mrgsolve` uses the `DLSODA` solver 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`.

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)
```

```
. Loading model from cache.
```

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

out
```

```
. # A tibble: 6 x 8
.      ID  time expos        T         I         V  logV logChange
.   <dbl> <dbl> <dbl>    <dbl>     <dbl>     <dbl> <dbl>     <dbl>
. 1     1  95    1000 5187630. -2.74e-10 -1.79e-12   NaN       NaN
. 2     1  97.5  1000 5298189. -3.18e-10 -2.09e-12   NaN       NaN
. 3     1 120    1000 6252828. -1.55e-10 -1.01e-12   NaN       NaN
. 4     1 122.   1000 6354547. -2.90e-10 -1.90e-12   NaN       NaN
. 5     1 125    1000 6455422. -1.97e-10 -1.30e-12   NaN       NaN
. 6     1 128.   1000 6555459. -7.81e-11 -5.21e-13   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: 6 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     1 120    1000 6252828. 1.01e-15 6.62e-18 -17.2     -22.9
. 4     1 122.   1000 6354547. 3.94e-16 2.58e-18 -17.6     -23.3
. 5     1 125    1000 6455422. 1.54e-16 1.01e-18 -18.0     -23.7
. 6     1 128.   1000 6555459. 5.99e-17 3.92e-19 -18.4     -24.1
```

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 2.2.10 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.

See 2.2.7 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.

See 2.2.8 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.

See 2.2.9 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).

The user may use the `revar()` function to query both `OMEGA` and `SIGMA`.

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.

Chapter 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:

2.1.1 Separate file

Open a text editor and type the model 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). The extension **MUST** be `.cpp` (`mrgsolve` currently assumes the extension). 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.

2.1.2 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`.

2.2 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.

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.

We note two specific 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:

- `$PARAM`
- `$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 7.1 for an example.

2.2.4 $PARAM

Syntax: R
Multiple allowed: yes
Options: `@annotated`, `@covariates`, `@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.

Example:

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

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 2.2.19 and 2.2.5.

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

2.2.5 $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 2.2.19.

Notes:

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

2.2.6 $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

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.

2.2.7 $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 2.4 for details.

Examples:

```
[ CMT ] CENT RESP

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

[ MAIN ]

RESP_0 = KIN/KOUT;

double ke = CL/VC;
```

2.2.8 $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 2.4 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

2.2.9 $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 2.2.12), 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.

2.2.10 $PREAMBLE

Syntax: C++
Multiple allowed: no

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

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: 2.2.18.

2.2.11 $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 ] TVE0 = 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 the section 3.6 for additional information regarding data sets in use with $PRED block.

2.2.12 $CAPTURE

Syntax: text
Multiple allowed: yes
Options: @annotated

This is a block to identify variables that should be captured in the simulated output.

For example:

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

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

[ CAPTURE ] A B C yes
```

This construct will result in 4 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 `doubles`, 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)
```

New variables may be declared in `$TABLE`. See section 2.4 for details.

2.2.13 $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.

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.

Notes:

- Multiple $OMEGA blocks are allowed

2.2.14 $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.

2.2.15 $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=s(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.

2.2.16 $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 2.4)

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.

2.2.17 $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`.

2.2.18 $PLUGIN

Syntax: text
Multiple allowed: no

Plugins are a way to add extensions to your `mrgsolve` model. When a plugin is recruited into the model, `mrgsolve` may do one or more of the following:

- Link to another `R` package during compilation, including
  - `Rcpp` to allow you to write `Rcpp` code in your specification
  - `mrgx` to provide some extra C++ functions (see below)
  - `RcppArmadillo` to both `Rcpp` and `Armadillo` in your specification
  - `BH` to use `boost` headers
- Include appropriate header files during compilation
  - For example, when the `Rcpp` plugin is called, `mrgsolve` will `#include <Rcpp.h>` at the top of your model file

You recruit a plugin in the `$PLUGIN` block

```
[ PLUGIN ] Rcpp
```

The example above will bring Rcpp headers into the model code.

Plugins that you can use: - Rcpp to write any Rcpp code - mrgx additional functions provided by mrgsolve that help use Rcpp in your model - RcppArmadillo to write any Rcpp or RcppArmadillo code - BH to link to boost headers

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.

Rcpp example

```
[ PLUGIN ] Rcpp

[ MAIN ]

if(NEWIND <=1) {
  double wt = R::rnorm(70,20);
  double sex = R::rbinom(1,0.51);
}
```

mrgx

For example

```
$PLUGIN Rcpp mrgx
```

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)`.

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: 2.2.10.

2.2.19 $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: 2.2.4.

2.2.20 $NMXML

Syntax: R
Multiple allowed: yes

The $NMXML block lets you read and incorporate results from a NONMEM run into your mrgsolve model. From the NONMEM run, THETA will be imported into your parameter list (see 2.2.4 and 1.1), OMEGA will be captured as an $OMEGA block (2.2.13) and SIGMA will be captured as a $SIGMA block (2.2.14). 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 file 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.

```
code <- '
$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 <- mcode("nmxml", code, quiet = TRUE)
```

```
. Loading required namespace: xml2
```

```
mod
```

```
.
.
. ----------------  source: nmxml.cpp  -----------------
.
.   project: /private/var/fol.../T/RtmpI4TSK2
.   shared object: nmxml-so-22457b83661a
.
.   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 2.2.21 for more discussion and examples.

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

See also: 2.2.21.

2.2.21 $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`).

We can load the same mode as the `$NMXML` example above with the following code:

```
path <- file.path(path.package("mrgsolve"),"nonmem")

code <- '
```

```
$NMEXT
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
'
```

```
mod <- mcode("nmext", code, quiet=TRUE)

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   | .      .
```

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`. See the `$NMXML` topic in section 2.2.20 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.

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: 2.2.20.

2.2.22 $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.

2.2.23 $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. Objects inside $ENV can be utilized in different C++ functions (see 2.2.18) or other parts of the simulation process.

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)
}
```

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.

2.3.4 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.

2.3.5 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.

2.3.6 AMT

AMT is the current value of dose amount.

AMT is an alias for self.amt.

2.3.7 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`.

2.3.8 `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);
```

2.3.9 `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;
}
```

2.3.10 `simeta(n)`

The `simeta(n)` function can be used to re-simulate ETA values. As of version `0.11.0`, `simeta()` accepts an integer argument for the ETA number to be re-simulated. For example,

```
$MAIN
simeta(2);
```

will re-simulate `ETA(2)`. Multiple calls to `simeta(n)` may be made. For example, we can resimulate both ETA 2 and ETA 4 by:

```
$MAIN
simeta(2);
simeta(4);
```

It is also possible to resimulate all ETAs by calling `simeta()` with no integer argument. We are currently encouraging use of `simeta(n)` rather than `simeta()` since for most applications only one or two ETAs might need to be re-simulated.

2.3.11 simeps(n)

`simeps(n)` works like `simeta(n)`, but the EPS values are re-simulated rather than the ETA values. We are also encouraging the use of `simeps(n)` rather than `simeps()`. The latter is currently supported but we are evaluating whether this will be supported going forward.

2.3.12 self.nid

The number of IDs in the data set.

2.3.13 self.idn

The current id number. Numbers start at 0 and increase by one to `self.nid-1`.

2.3.14 self.nrow

The number of rows in the output data set.

2.3.15 self.rown

The current row number. Numbers start at 0 and increase by one to `self.rown-1`.

2.3.16 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.

2.3.17 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;
```

2.3.18 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.

2.3.19 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()`.

2.3.20 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().

2.3.21 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.

2.3.22 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 ] @labels ADD PROP
25 0.0025

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

2.3.23 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 2.2.9 and also 2.2.12

2.3.24 F_CMT

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

Example:

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

2.3.25 ALAG_CMT

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

Example:

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

2.3.26 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;
```

2.3.27 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;
```

2.4 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;
```

2.4.1 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.

2.4.2 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);
```

2.5 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.

2.5.1 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;
```

2.5.2 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*CENT;

$CAPTURE CP
```

2.5.3 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 7.8)

```
$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
```

Chapter 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, 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 7.3.

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 sections 2.2.7, 2.3.24,2.3.26, 2.3.27).
- 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 2.2.4 and 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 7.3).

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 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 7.5). 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 <- mrgsolve:::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 IDs 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:

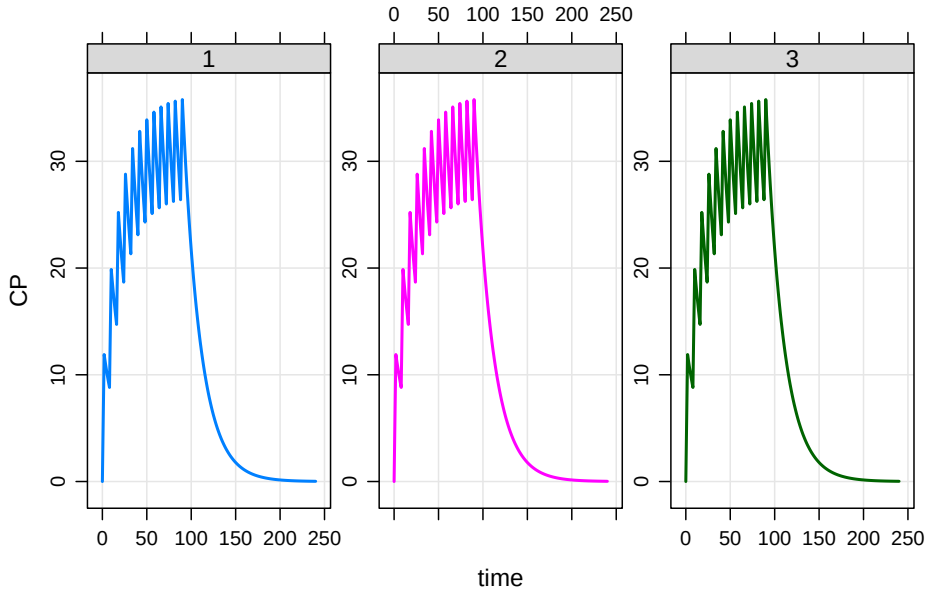
```
data <- expand.ev(ID=1:3, amt=250, rate=125, ii=8, addl=11, cmt=2)

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 <- mrgsolve:::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 7.2.

`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.

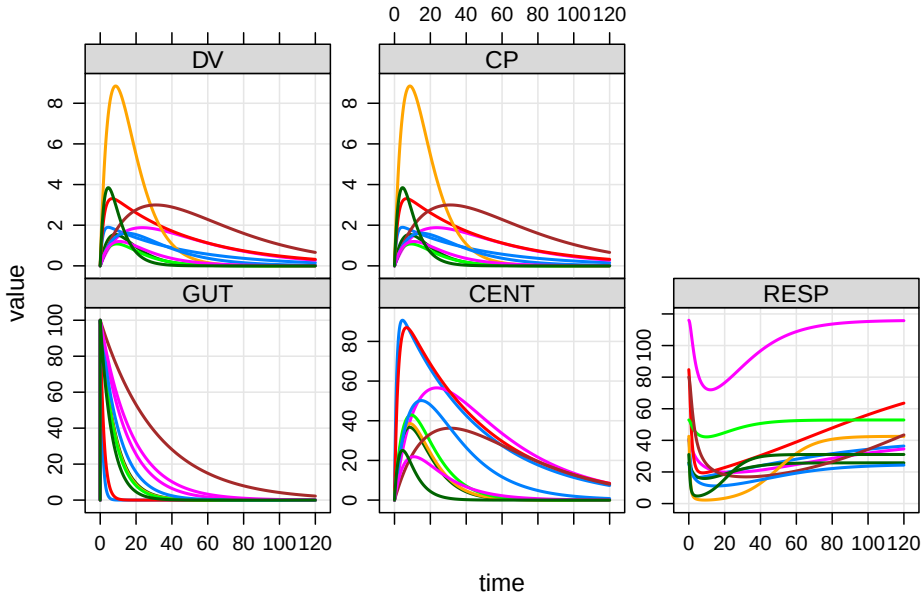
3.3.1 Use case

`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 <- mrgsolve:::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 Data set validation

A the time of simulation, mrgsolve will validate the input data set, removing character column, 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")
```

```
. Loading model from cache.
```

```
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 ...
.  - attr(*, "dimnames")=List of 2
.   ..$ : chr [1:132] "1" "2" "3" "4" ...
.   ..$ : 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.

3.6 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

Chapter 4

Simulated output

When `mrgsim` is used to simulate from a model, it returns an object with class `mrgsims`. It is an S4 object containing a matrix of simulated output and a handful of other pieces of data related to the simulation run.

`mrgsolve` provides several methods for working with `mrgsims` objects or coercing the simulation matrix into other `R` objects.

4.1 Coercion methods

- `as.data.frame` convert to `data.frame`
- `as.matrix` convert to `matrix`
- `as.tbl` convert to `tbl`
- `as_data_frame` convert to `tbl` via `tibble` package

4.2 Query methods

- `head`
- `tail`
- `names`
- `dim`
- `summary`
- `show`
- `$`

4.3 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)

mod <- mrgsolve:::house()

mod %>%
  ev(amt=100) %>%
  mrgsim %>%
  filter(time >=10)
```

```
. # A tibble: 441 x 7
.       ID  time       GUT  CENT  RESP    DV    CP
.    <dbl> <dbl>     <dbl> <dbl> <dbl> <dbl> <dbl>
.  1     1  10   0.000614   63.3  37.7  3.16  3.16
.  2     1  10.2 0.000455   62.5  37.9  3.13  3.13
.  3     1  10.5 0.000337   61.7  38.0  3.09  3.09
.  4     1  10.8 0.000250   61.0  38.1  3.05  3.05
.  5     1  11   0.000185   60.2  38.2  3.01  3.01
.  6     1  11.2 0.000137   59.5  38.3  2.97  2.97
.  7     1  11.5 0.000102   58.7  38.4  2.94  2.94
.  8     1  11.8 0.0000752  58.0  38.5  2.90  2.90
.  9     1  12   0.0000557  57.3  38.7  2.86  2.86
. 10     1  12.2 0.0000413  56.6  38.8  2.83  2.83
. # ... with 431 more rows
```

Here, `mrgsim` returns an `mrgsims` object. When `dplyr` is also loaded, this object can be piped directly to `dplyr::summarise`.

Other `dplyr` functions that can be used with `mrgsims` objects

- `group_by`
- `mutate`
- `filter`
- `summarise`
- `select`
- `slice`
- `summarise.each` (use a `<dot>` not an `<underscore>`)

Chapter 5

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.

5.1 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

5.2 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.

5.3 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

5.4 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 sections 7.3 and 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.

Chapter 6

Steady state

6.1 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.

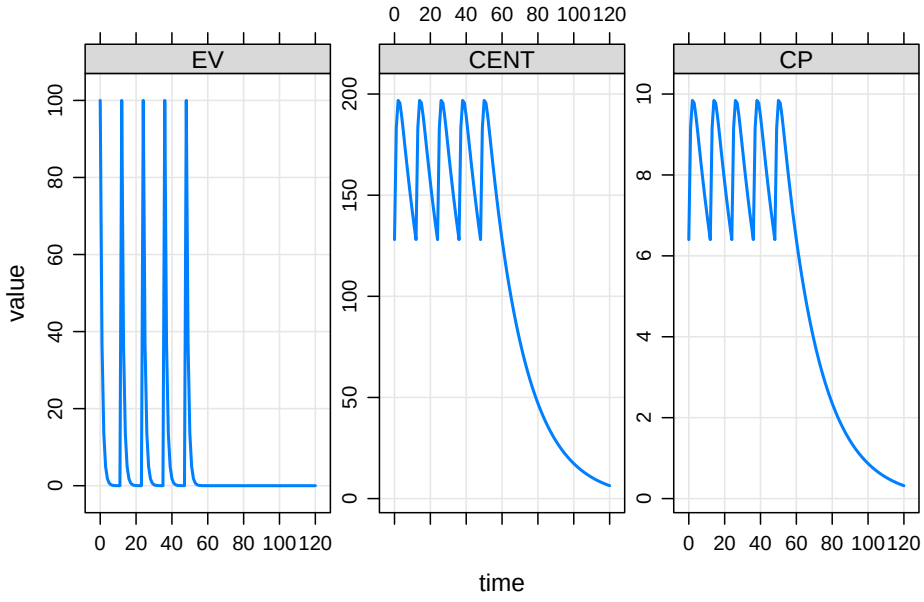
6.1.1 Intermittent doses

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()
```

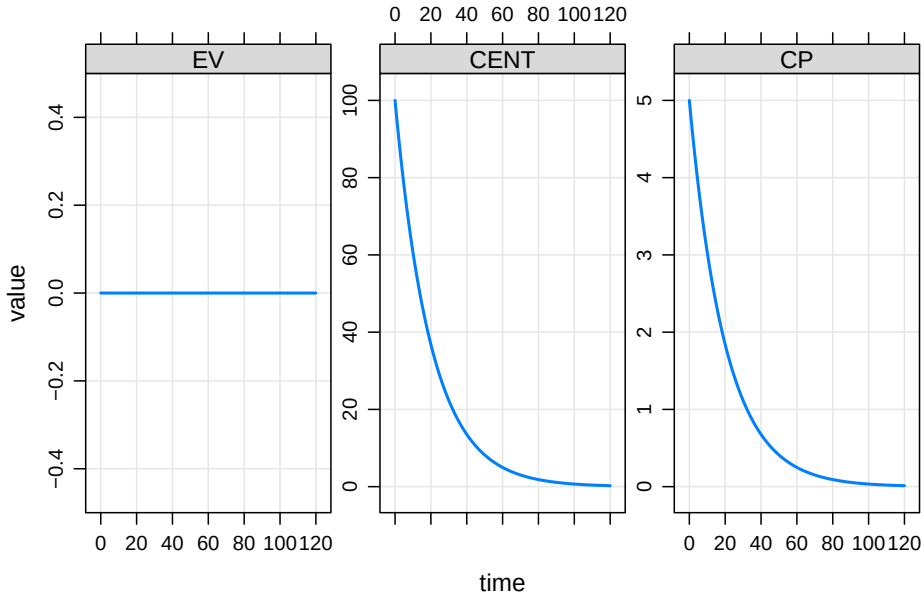


In this example, the `ss=1` flag tells mrgsolve to advance the system to steady state under a dosing regimen of 100 mg every 12 hours (and then give a total of 5 doses). When using the `ss=1` flag, the user is required to indicate the dosing interval (here every 12 hours) and additional doses are optional. Similar behavior can be achieved for intermittent infusions by setting the infusion `rate`.

6.1.2 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.

6.2 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.

6.3 Control advance to SS

6.3.1 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)
```

6.3.2 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, : [stea
.   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.

6.3.3 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.

6.3.4 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.

Chapter 7

Topics

7.1 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
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
PROP: 0.25 : Proportional residual error
ADD : 25   : Additive residual error
```

```
[CMT]
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]
DV  : Concentration (mg/L)
ECL : Random effect on CL
CL  : Individual clearance (L/hr)
```

7.2 Set initial conditions

```
library(mrgsolve)
library(dplyr)
```

7.2.1 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

7.2.2 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

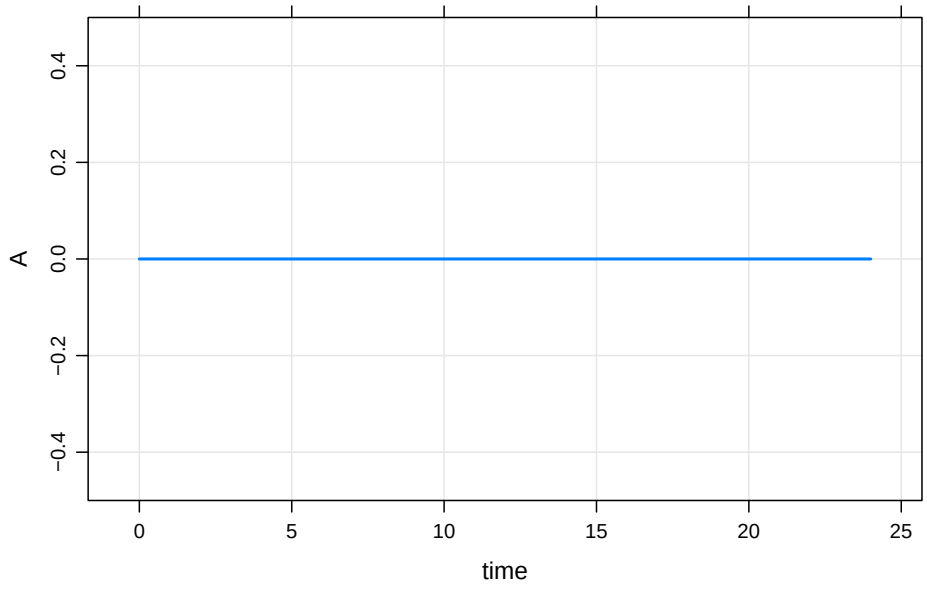
```
init(mod)
```

```
.
.  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
```

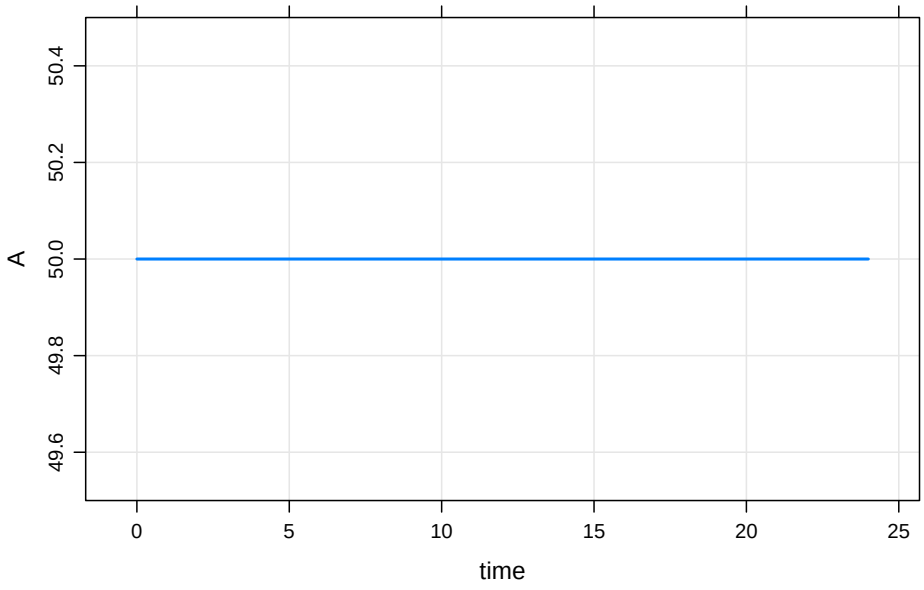


Next, we update the base initial condition for `A` to 50

Note:

- The code in `$MAIN` still doesn't get run because `IFLAG` is 0

```
mod %>% init(A = 50) %>% mrgsim %>% plot
```

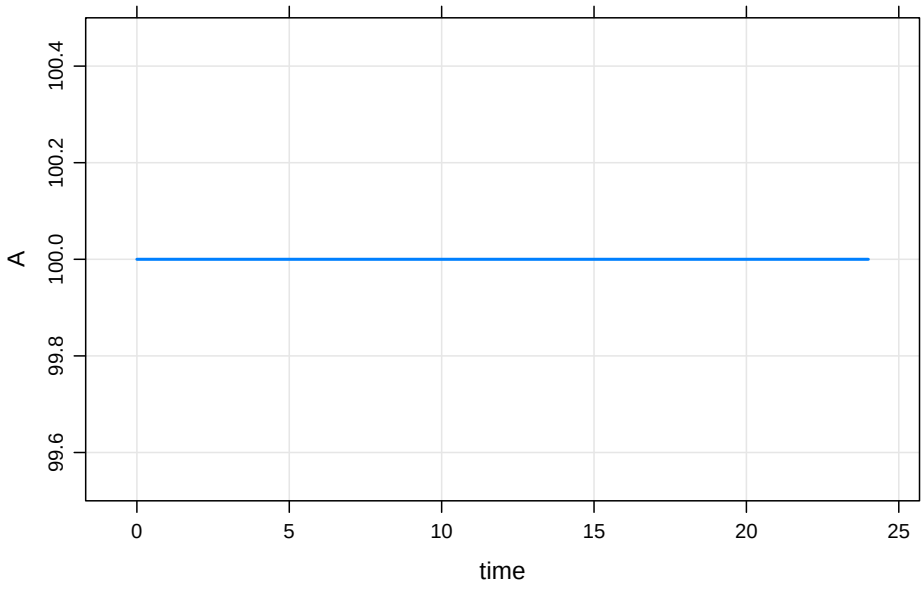


Now, turn on `IFLAG`

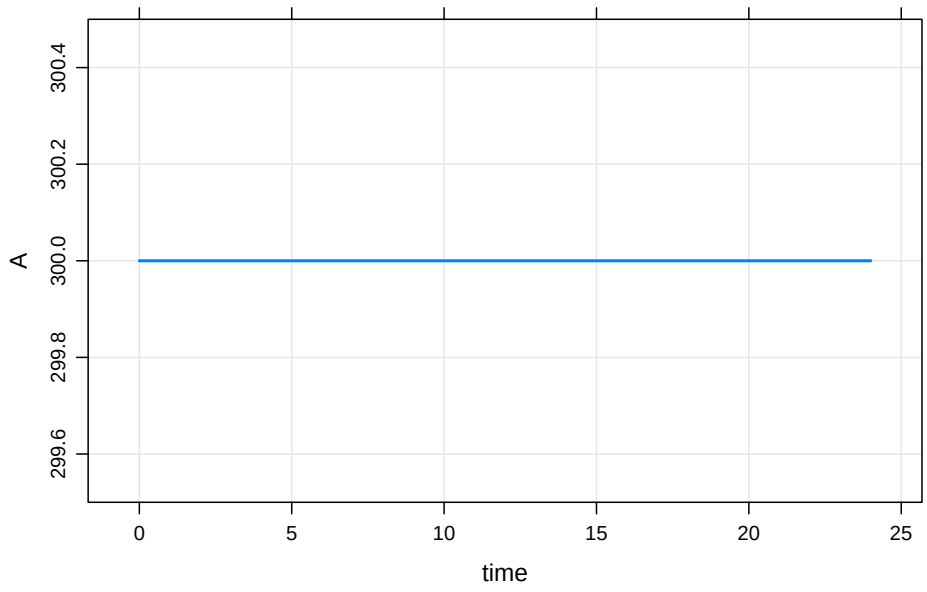
Note:

- 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
```



7.2.3 Example PK/PD model with initial condition

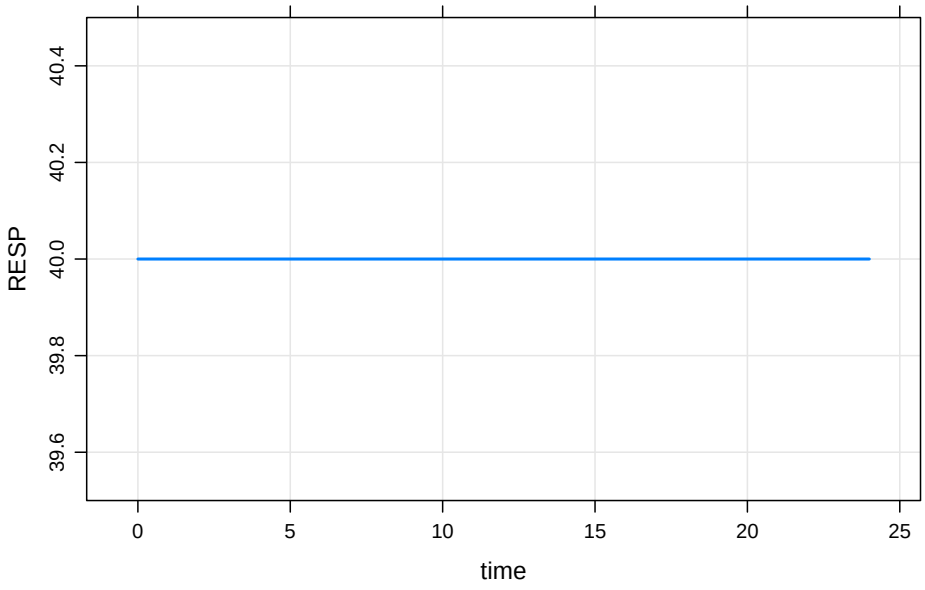
Just to be clear, there is no need to set any sort of flag to set the initial condition as seen here:

```
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
```

7.2.4 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)   0     | . ...   .
```

```
mod %>% param(KIN=100) %>% init
```

```
.
.  Model initial conditions (N=1):
.  name       value . name    value
.  RESP (1)   0     | . ...   .
```

7.2.5 Set initial conditions via `idata`

Go back to house model

```
mod <- mrgsolve:::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.

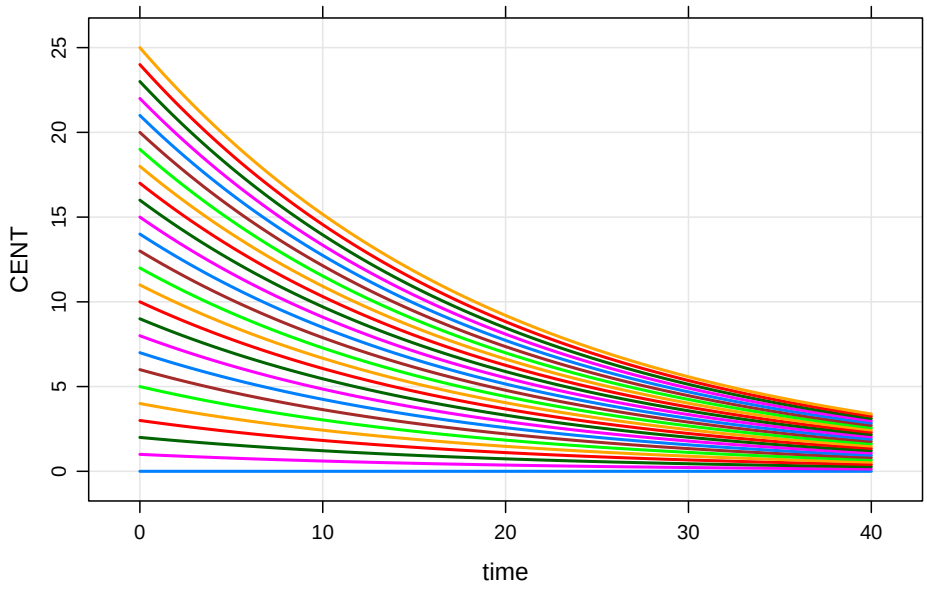
```
idata <- expand.idata(CENT_0 = seq(0,25,1))
```

```
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~.)
```



7.3 Updating parameters

The parameter list was introduced in section 1.1 and the `$PARAM` code block was shown in 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.

7.3.1 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 7.3.2 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.
. 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 sections 7.9 and 5.

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.

7.3.2 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
```

7.4 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)
```

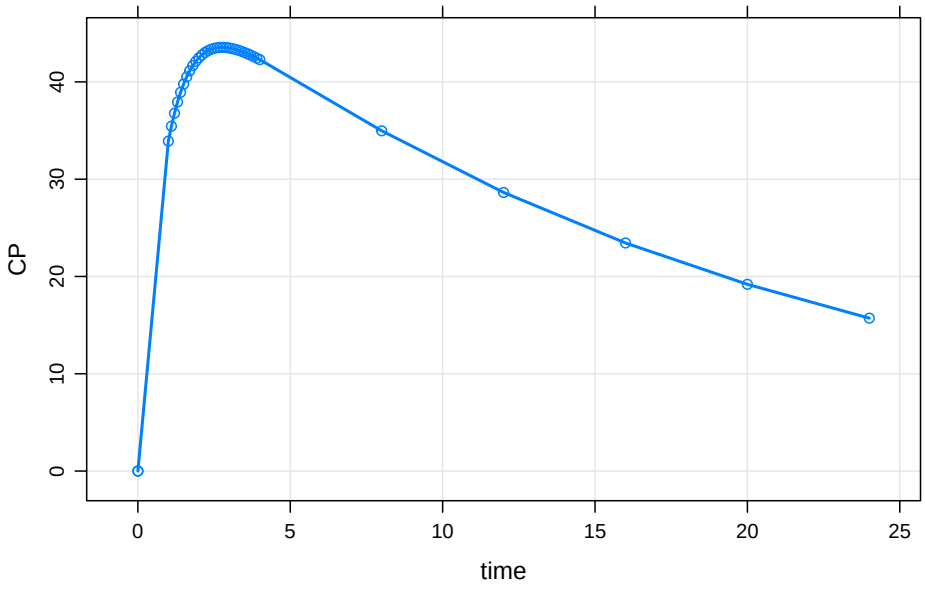
Check this by calling `stime`

```
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.

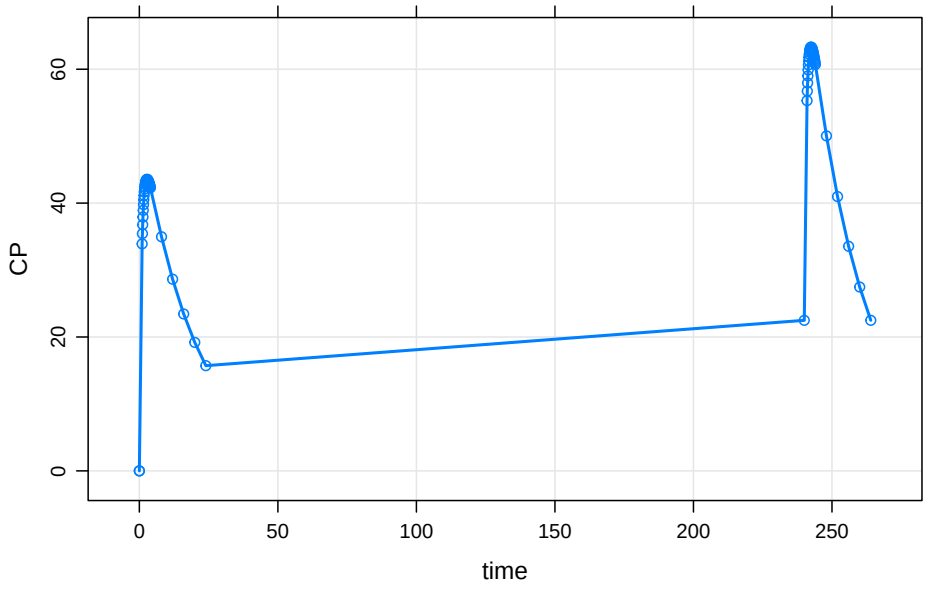
```
mod %>%
  mrgsim(tgrid=day1) %>%
  plot(type='b')
```



Now, look at both day 1 and day 10:

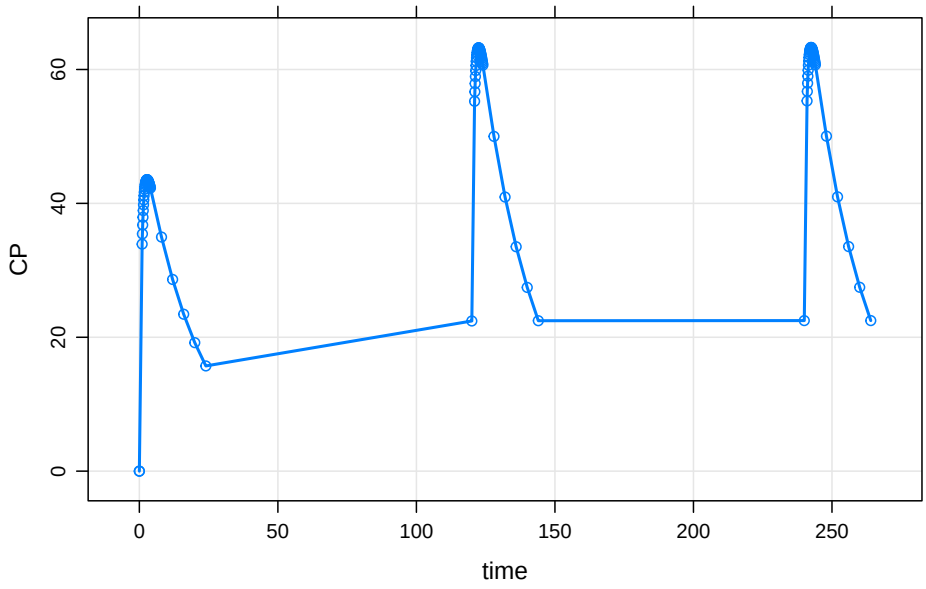
Adding a number to a `tgrid` object will offset those times by that amount.

```
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')
```



7.5 Individualized sampling designs

Here is a PopPK model and a full `data_set`.

```
mod <- mrgsolve:::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 <- df %>% distinct(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
```

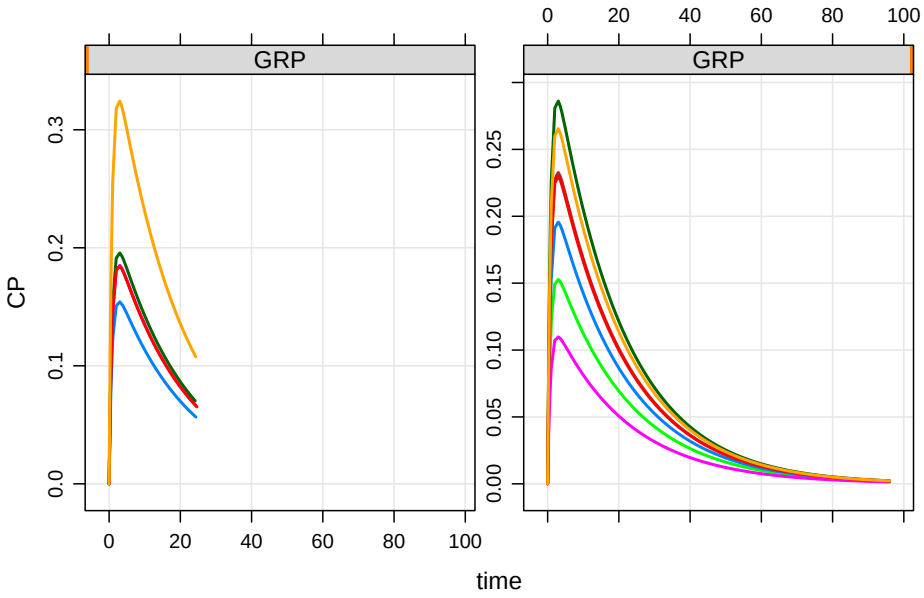
Now, we have two groups in `GRP` in `idata_set` and we have two `tgrid` objects.

- 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)
```



7.6 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.

7.6.1 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;
```

7.6.2 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;
```

7.6.3 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);
```

7.6.4 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
```

7.7 Resimulate ETA and EPS

Call `simeps()` to resimulate `ETA`

- No `$PLUGIN` is required
- `simeta()` takes no arguments

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.   :-0.15363   Min.   :60.03
.  1st Qu.: 25.75   1st Qu.:0   1st Qu.:-0.07708   1st Qu.:64.81
.  Median : 50.50   Median :0   Median :-0.01255   Median :69.13
.  Mean   : 50.50   Mean   :0   Mean   :-0.01125   Mean   :69.43
.  3rd Qu.: 75.25   3rd Qu.:0   3rd Qu.: 0.05667   3rd Qu.:74.08
.  Max.   :100.00   Max.   :0   Max.   : 0.12983   Max.   :79.70
```

Call `simeps()` to resimulate EPS

- No `$PLUGIN` is required
- `simeps()` takes no arguments

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.   : 0.09638
.  1st Qu.:1   1st Qu.:11.25   1st Qu.: 15.93   1st Qu.: 2.18627
.  Median :1   Median :23.50   Median : 29.38   Median : 5.03571
.  Mean   :1   Mean   :23.52   Mean   : 37.47   Mean   : 6.17082
.  3rd Qu.:1   3rd Qu.:35.75   3rd Qu.: 54.21   3rd Qu.: 9.03877
.  Max.   :1   Max.   :48.00   Max.   :100.00   Max.   :16.82901
```

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.

7.8 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.

7.8.1 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 2.2.13). 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.

7.8.2 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()
```

7.8.3 Example: unnamed matrix

Here is a model with only a 3x3 $OMEGA matrix

```
code <- '
$OMEGA
1 2 3
'

mod <- mcode("matrix", code, compile=FALSE)
```

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>
```

7.8.4 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`).

```
code <- '
$OMEGA @name IIV
```

```
1 2 3
$OMEGA @name IOV
4 5
'
mod <- mcode("iov", code, compile=FALSE)

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>
```

7.8.5 Example: unnamed matrices

If we do write the model with unnamed matrices, we can still update them

```
code <- '
$OMEGA
1 2 3

$OMEGA
4 5
'
mod <- mcode("multi", code, compile=FALSE)
```

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
```

7.9 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)
```

Note that time-varying covariates are not possible when using `qsim` simulation.

There is additional information about the sequence of events that takes place during system advance in section 5.

Chapter 8

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/mrgsolve-Installation