mrgsolve: Simulate from ODE-Based Models

Package vignette to get started simulating

Kyle T. Baron

2025-01-01

mrgsolve is an R package for simulation from hierarchical, ordinary differential equation (ODE) based models typically employed in drug development. mrgsolve has been used for a wide variety of model applications, including pharmacokinetics (PK), pharmacokinetics/pharmacodynamics (PK/PD), physiologically-based pharmacokinetic (PBPK) modeling, and quantitative systems pharmacology. This vignette provides a comprehensive introduction to using mrgsolve in a single document. While there is more to learn about mrgsolve, this document will give you a good introduction to the essentials and some ideas for thinking about how all the pieces fit together. Be sure to visit https://mrgsolve.org for additional resources to help you learn and effectively use mrgsolve.

Table of contents

1 Big picture

To start out this package vignette, I want to give you an overhead view of what it is like working with mrgsolve. There are a *huge* number of little details that you might want to eventually know in order to use mrgsolve effectively; but for now, let's get a handle on the big picture of what you need to do to get the simulations you want.

There are 3 (or 4) main simulation workflows that we want to work up to. We can think about the type of **outputs** we want and *then* determine what **inputs** we'll need to create and the **functions** that need to be called in order to get those outputs back.

First, load the package along with any other helper packages we need for this vignette.

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

1.1 You need a model

For every workflow, you need a model. In most cases, is coded in a separate file and read in by `mread()`

```
mod <- mread("azithro-fixed.mod")
```

```
Building azithro-fixed_mod ... done.
```

```
mod
```

```
------------  source: azithro-fixed.mod  ------------

  project: /Users/kyleb/git...olve/vignette
  shared object: azithro-fixed.mod-so-c412206deb60
```

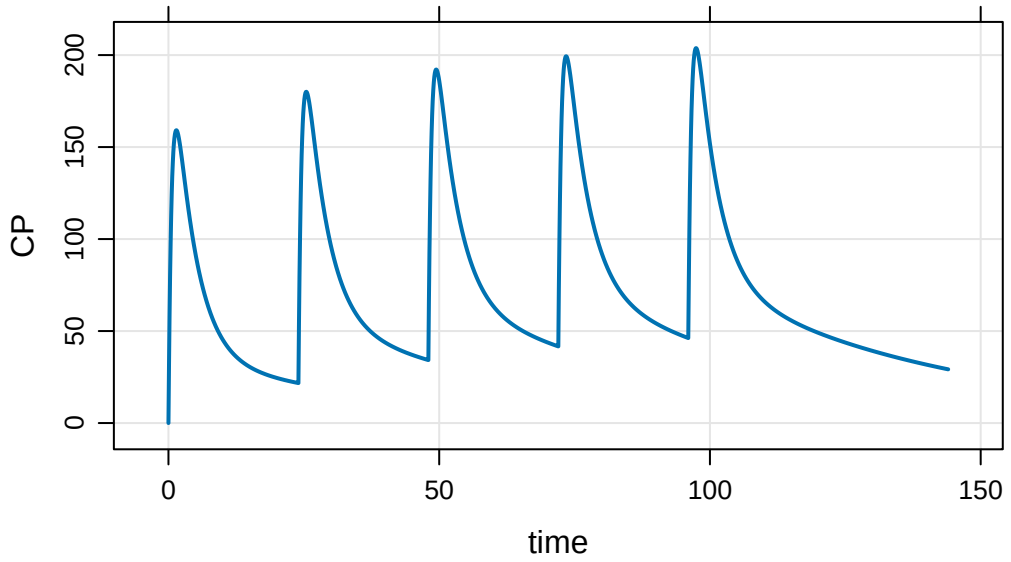
```
  time:           start: 0 end: 240 delta: 0.1
                  add: <none>

  compartments:   GUT CENT PER2 PER3 [4]
  parameters:     TVCL TVV1 TVQ2 TVV2 Q3 V3 KA WT [8]
  captures:       CP [1]
  omega:          2x2
  sigma:          1x1

  solver:         atol: 1e-08 rtol: 1e-08 maxsteps: 20k
-----------------------------------------------------
```

In the above example, we created a file called `azithro-fixed.mod` (azithromycin population PK with fixed effect parameters only) and wrote out the covariate model, differential equations, etc. into that file. We point `mread()` at that file to parse, compile and load the model. More information on using `mread()` and the model object is found in Section 3. We'll start showing you the model syntax in Section 8.

1.2 Single profile

The first and simplest workflow is to generate a single simulated profile from the model. The quickest way we'll do this is using the model object loaded in the previous section along with an *event object*

```
mod %>%
  ev(amt = 250, ii = 24, addl = 4) %>%
  mrgsim(end = 144, delta = 0.1) %>%
  plot("CP")
```



The `mrgsim()` function is called to actually *execute* the simulation and we've introduced some simulation options (like the simulation `end` time) by passing those arguments in. More info on `mrgsim()` can be found in Section 6.

The event object is a quick way to introduce an intervention (like dose administration) into your simulation. More information about event objects is provided in Section 4.

1.3 Population simulation

When we simulate a population, we want to simulate a collection of individuals (or profiles) in a single simulation run. Most often, this involves creating an input data set with dosing or other information for each subject in the population.

In this example, we'll load another azithromycin population PK model

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

```
Building azithro_mod ... done.
```

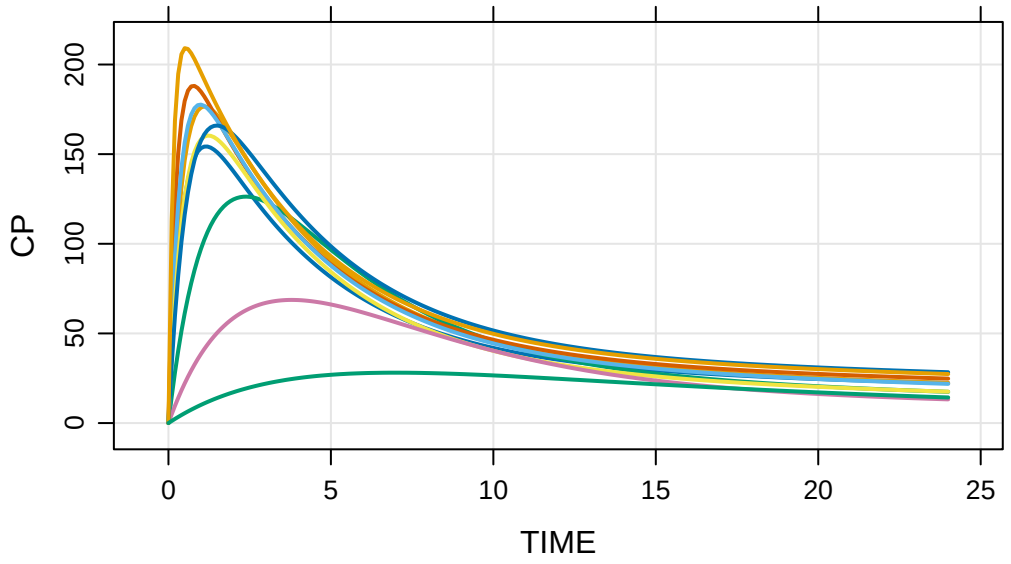
Rather than using an event object as we did for the single profile, we make a data set; in this example, we use `expand.ev()` to help

```
set.seed(9876)

data <- expand.evd(amt = 250, WT = runif(10, 50, 100))

data
```

```
   ID TIME AMT CMT EVID       WT
1   1    0 250   1    1 92.33453
2   2    0 250   1    1 68.39476
3   3    0 250   1    1 56.19846
4   4    0 250   1    1 78.43937
5   5    0 250   1    1 71.22273
6   6    0 250   1    1 63.88194
7   7    0 250   1    1 73.15188
8   8    0 250   1    1 80.16205
9   9    0 250   1    1 75.37584
10 10    0 250   1    1 59.11208
```

In this data set, we see 10 subjects who are differentiated by their different weights (`WT`). For this simulation, we are giving every subject a singe 250 mg dose.

```
set.seed(9876)

mod %>%
  data_set(data) %>%
  mrgsim(end = 24) %>%
  plot("CP")
```



This simulation introduces variability not only through the covariate `WT` but also through random effects (i.e., ETAs) which are simulated when we call `mrgsim()`.

1.4 Batch simulation

You can also simulate a population (or a batch of subjects) with a data set of parameters and an event object. This workflow is *like* the population simulation, but the inputs are configured in a slightly different way where the population is a set of parameters with a common intervention, rather than a data set with (possibly) different interventions (or different parameters) for each subject in the population. Going back to the `azithro-fixed` model

```
mod <- mread("azithro-fixed.mod")
```

```
Building azithro-fixed_mod ... done.
```

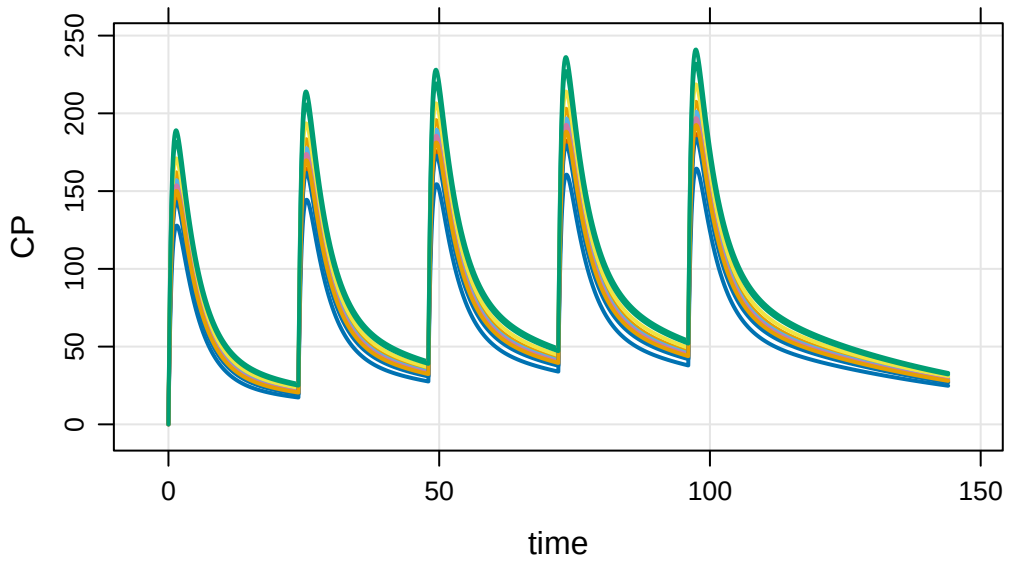
Rather than creating a data set with *doses* for everyone, we just create their parameters

```
set.seed(9876)

data <- expand.idata(WT = runif(10, 50, 100))

data
```

```
   ID       WT
1   1 92.33453
2   2 68.39476
3   3 56.19846
4   4 78.43937
5   5 71.22273
6   6 63.88194
7   7 73.15188
8   8 80.16205
9   9 75.37584
10 10 59.11208
```

Here, we have 10 parameter sets which can also be thought of as 10 people. We can pass this set of parameters as `idata`, or individual-level data, along with an event object

```
mod %>%
  ev(amt = 250, ii = 24, addl = 4) %>%
  idata_set(data) %>%
  mrgsim(end = 144) %>%
  plot("CP")
```



Here, we get the same output as we got for the population simulation, but a slightly different setup. This setup might be more or less convenient or more or less flexible to use compared to the population setup. Either way, the approach is up to you and the needs of your simulation project.

1.5 Replicate simulation

This pattern is just like `data_set`, but we do that in a loop to generate *replicate* simulations. Sometimes we do a simulation like this when we are doing simulation-based model evaluation or maybe we're simulating across draws from a posterior distribution of parameter estimates.

This simulation might look something like this (code not evaluated in this vignette)

```
sim <- function(i, model, data) {
  mod %>%
    data_set(data) %>%
    mrgsim() %>%
    mutate(irep = i)
}

out <- lapply(1:1000, sim, model = mod, data = data) %>% bind_rows()
```

Here, we create a function (`sim()`) that simulates a data set once and then call that function repeatedly to get replicate simulated data sets.

1.6 The general pattern

So the general pattern to working with mrgsolve is

- Code a model
- Load it with `mread()`
- Set up your intervention and population
- Simulate with `mrgsim()`
- Plot or process your output

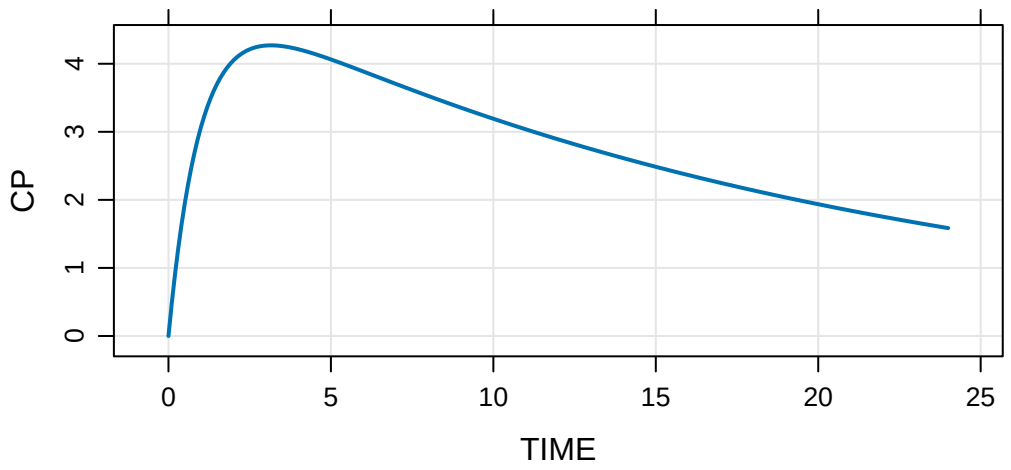
2 Quick start

To quickly get started with mrgsolve, try using the built in model library like this

```
mod <- modlib("pk1", delta = 0.1)

out <- mrgsim(mod, events = evd(amt = 100))

out
```

```
Model:  pk1
Dim:    242 x 5
Time:   0 to 24
ID:     1
    ID TIME     EV   CENT     CP
1:   1  0.0   0.00  0.000 0.0000
2:   1  0.0 100.00  0.000 0.0000
3:   1  0.1  90.48  9.492 0.4746
4:   1  0.2  81.87 18.034 0.9017
5:   1  0.3  74.08 25.715 1.2858
6:   1  0.4  67.03 32.619 1.6309
7:   1  0.5  60.65 38.819 1.9409
8:   1  0.6  54.88 44.383 2.2191
```

```
plot(out, "CP")
```



That was a really simple simulation where we used an event object to initiate a dose into a one-compartment model. Notice how the `plot()` method allows us to quickly visualize what happened in the simulation. See the `?modlib` help topic for more models you can play around with to get comfortable with mrgsolve. Or keep reading to dig into more of the details.

3 Model object

This chapter introduces the mrgsolve **model object**. The model object contains all information about *the model* itself, including

- Compartments
- ODE
- Algebraic relationships
- Random effects
- More

The model object is what you use in R work with the model, including

- Query the model
- Run simulations

3.1 `mread()`

We saw before that you can load a model from a model specification file using the `mread()` function. Don't worry for now what is in that file; we'll show you how to create it in Section 8.

3.1.1 Model file extension

Your model can have any extension. Traditionally, we've used the `.cpp` extension because a lot of the code in that file is `c++`. However, we've moved away from that in recent years because code editors like `Rstudio` see that `.cpp` extension and *think* that *all* the code is `c++`; they then format the code in ways that aren't what you usually want. So using the `.mod` (or `.txt`) file extension can be helpful just to keep your editor from doing too much.

3.1.2 Syntax to load a model

This section walks you though some of the ways you can use `mread()` to load a model.

You can provide the complete path to the file

```
mod <- mread("model/test.mod")
```

You can provide the file name (first argument) and enclosing directory (as `project`); this assumes you are keeping all simulation code in the `models` directory

```
mod <- mread("test.mod", project = "model")
```

This can be a convenient pattern in a larger project because the `project` argument can be pulled from the `mrgsolve.project` R option (see `?options`). For example

```
options(mrgsolve.project = "model")

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

3.1.3 Update the model on load

mrgsolve provides an `update()` method for changing some settings inside a model object. `mread()` will take in arguments and pass them along to `update()` so you can make these changes at the time the model is loaded. For example, we can

- Set the simulation end time to `240`
- Set (increase) ODE solver relative tolerance to `1e-5`

by passing the appropriate arguments through `mread()`

```
mod <- mread("model/test.mod", end = 240, rtol = 1e-5)
```

3.1.4 Read and cache

Use `mread_cache()` to build and cache the model on disk.

When you load the model the first time, you'll see

```
mod <- mread_cache("test.mod", project = "model")
```

```
Building test_mod ... done.
```

When you load it again, you'll see

```
mod <- mread_cache("test.mod", project = "model")
```

```
Loading model from cache.
```

By default, mrgsolve will store the cached model information in the temporary directory that R sets up every time you start a new R session. This is convenient because you don't have to think about what that directory is, but sometimes you want the cached model to sit in a location that you have a little more control over. Look at the `soloc` argument to `mread()`; this will let you place the cached model information in a stable location.

3.2 `modlib()`

Use the `modlib()` function to load a model from an internal model library. These are pre-coded models that can be sourced from within the mrgsolve installation directory. They are a great way to get your hands on different models to experiment with. But note: I rarely use these for production work; *almost* always, my production model is more complicated that what has been coded into these general-purpose library models.

This code will load a 1-compartment PK model

```
mod <- modlib("pk1")
```

So the `modlib()` function is equivalent to

```
mod <- mread("pk1", project = modlib())
```

Check out the `modlib()` help topic for a more detailed listing of the models

```
?modlib
```

3.3 Model overview

You can print `mod` to the R console and see what's going on

```
mod
```

```
-----------------  source: test.mod  -----------------

  project: /Users/kyleb/git...ignette/model
  shared object: test_mod-so-c41235683d50

  time:          start: 0 end: 24 delta: 1
                 add: <none>

  compartments:  GUT CENT [2]
  parameters:    CL V TVKA [3]
  captures:      CP [1]
  omega:         2x2
  sigma:         0x0

  solver:        atol: 1e-08 rtol: 1e-08 maxsteps: 20k
-------------------------------------------------------
```

or summarize

```
summary(mod)
```

```
Model: test_mod
- Parameters: [3]
  CL, V, TVKA
- Compartments: [2]
  GUT, CENT
- Captured: [1]
  CP
- Outputs: [3]
  GUT, CENT, CP
```

or see the model code

```
see(mod)
```

```
Model file:  test.mod
$PARAM CL = 1, V = 20, TVKA = 1.2

$OMEGA 0.1 0.2

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

$MAIN
double KA = TVKA + ETA(1);

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

3.4 Parameters

Parameters are `name=value` pairs that are used in your model. You can *change* the `value` of a parameter in several different ways. Understanding how to do this parameter value update is really important if you want to make interesting simulation outputs.

Query the parameter list with `param()`

```
param(mod)
```

```
Model parameters (N=3):
name value . name value
CL   1     | V    20
TVKA 1.2   | .    .
```

This output shows you there are 3 parameters in the model

- `CL`, with nominal value 1
- `V`, with nominal value 20
- `KA`, with nominal value 1

Note that each parameter has

- A **name**(e.g. `CL`)
- A **value** (must be *numeric* or *evaluate* to a numeric value)

Parameter names can be upper or lower case letters or numbers; the only punctuation allowed in parameter names is underscore (_).

Parameters are unordered; the order in which you code the parameters makes no difference to how you are able to work with the model.

See Section 7 for a deeper discussion of model parameters and their central role in generating simulations to answer questions at hand.

3.5 Compartments

Models also have compartments. Like parameters, compartments have

- A **name**
- A **value**

The same rules hold for compartment names that we discussed for parameter names.

Compartments also have a **number**; they are numbered in the order in which they are entered.

Query the compartment list with `init()`. For example, using the model we loaded in the previous section

```
init(mod)
```

```
Model initial conditions (N=2):
name       value . name      value
CENT (2)   0     | GUT (1)   0
```

Notice that each compartment has a number associated with it; this is mainly used for dosing via `CMT` in your data set. But there is a model syntax that allows you to write a model in terms of named compartments (e.g. `A(2)` or `F1`) as well.

3.6 Random effects

You can see what random effect matrices are available in the model with

```
revar(mod)
```

```
$omega
$...
    [,1] [,2]
1:   0.1  0.0
2:   0.0  0.2


$sigma
No matrices found
```

3.7 Update the model object

We frequently want to *change* or *update* the settings in the model object.

Updates can be made through the `update()` method. For example, use

```
mod <- update(mod, end = 240, delta = 2)
```

to change the simulation end time to 240 hours and the output time interval to every 2 hours. This results in a new model object with updated settings that will be in place whenever you simulate from `mod` until you make more changes.

You can also update on model read

```
mod <- mread("model.mod", end = 240, delta = 2)
```

or at the time of simulation

```
out <- mod %>% mrgsim(end = 240, delta = 2)
```

All of these update mechanisms execute updates to the model object. But only when we save the results back to `mod` are the updates persistent in the model.

What else can I update?

- Time
  - `start`, `end`, `delta`, `add`
- Parameters and compartment initial values
- ODE solver settings
  - `atol`, `rtol`
  - `hmax`, `maxsteps`, `mxhnil`, `ixpr`
  - Usually changing `rtol`, `atol`, and maybe `hmax`
- Settings related to steady state
  - `ss_rtol`, `ss_atol`
- `$OMEGA`, `$SIGMA`
- `tscale` (rescale the output time)
- `digits`
- `outvars` (which compartments or derived quantities should appear in the output)

See `?mrgsolve::update` for more details.

Parameter update

To update parameters, use `param()`. More on this in Section 7

```
mod <- param(mod, CL = 2)
```

3.8 Write a model object to file

Recall that we use `mread()` or `mread_cache()` to read model code from a file into an object in your R session. mrgsolve also allows you to write the model contents out to a file again. The code in the new file will be well formatted, but it will be by necessity different in some ways from the code you originally wrote.

As an example, read `test.mod` back into R

```
mod <- mread_cache("model/test.mod")
```

```
Loading model from cache.
```

We can update this model

```
mod <- update(mod, end = 240, delta = 6, rtol = 1e-4)
mod <- param(mod, V = 15, CL = 1.5)
```

We can write this model back to file in a couple of different formats. First, you can write it in `yaml` format with `mwrite_yaml()`

```
file <- mwrite_yaml(mod, file = "model/test.yaml")
```

Now the model code has been written back to the file `test.yaml` in the `model` directory.

There is no requirements for file extension; we just chose `yaml` to match the format.

To read this file back into R, use `mread_yaml()`

```
mod2 <- mread_yaml("model/test.yaml")
```

```
Building test_yaml_mod ... done.
```

Now, you have a model you can simulate from again (`mod2`).

You can also write the model out in native mrgsolve format

```
mwrite_cpp(mod2, file = "model/test-3.cpp")
```

Of course, we can read this file back in using `mread()` and friends.

The important feature of `mwrite_*` is that it breaks any connections to NONMEM outputs that might be created through the use of `$NMXML` or `$NMEXT` blocks.

For example, model `1005` in `modlib()` is connected to a NONMEM model

```
modx <- modlib("1005")
```

```
Loading required namespace: xml2
```

```
Building 1005 ... done.
```

```
as.list(modx)$nm_import
```

```
[1] "/Users/kyleb/renv/renv/library/R-4.4/aarch64-apple-darwin20/mrgsolve/nonmem/1005/1005.xml
```

This mrgsolve model is reading `THETA`, `OMEGA` and `SIGMA` from `1005.xml`.

When we write the code to, for example, `yaml` format, all parameters and matrices are written into the file as they are, forgetting they came from the NONMEM run.

```
tmp <- tempfile()
mwrite_yaml(modx, file = tmp)
```

We can check that this is true by parsing the yaml file

```
y <- yaml::yaml.load_file(tmp)

names(y)
```

```
 [1] "source"   "mrgsolve" "format"   "version"  "model"    "prob"
 [7] "param"    "init"     "capture"  "omega"    "sigma"    "envir"
[13] "plugin"   "update"   "set"      "code"
```

```
y$param[1:5]
```

```
$SEX
[1] 0

$WT
[1] 70

$THETA1
[1] 9.507886

$THETA2
[1] 22.79099

$THETA3
[1] 0.07143366
```

This functionality can be very helpful when sharing your NONMEM-backed simulation model written in mrgsolve.

3.9 Advanced

This section shows some advanced methods for interacting with the mrgsolve model object.

Get the value of a parameter or setting

```
mod$CL
```

```
[1] 1.5
```

```
mod$end
```

```
[1] 240
```

Extract all parameters as a list

```
as.list(param(mod))
```

```
$CL
[1] 1.5

$V
[1] 15

$TVKA
[1] 1.2
```

Extract the value of one parameter

```
mod$CL
```

```
[1] 1.5
```

Extract everything

You can get the model object contents as a plain list

```
l <- as.list(mod)
```

4 Event objects

Event objects are quick ways to generate an intervention or a sequence of interventions to apply to your model. For example, you have a PK model and want to implement a series of doses into the system during the simulation. Event objects function like quick and easy data sets to accomplish this.

4.1 Create an event object

Use `ev()` and pass NMTRAN data names in lower case.

For example

```
ev(amt = 100, ii = 12, addl = 2)
```

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

You can pass

- `time` time of the event
- `evid` event ID
  - 1 for dose
  - 2 for other type
  - 3 for reset
  - 4 for dose and reset
  - 8 for replace

- `amt` dose amount
- `cmt` compartment for the intervention
  - usually the compartment number
  - can be character compartment name
- `ii` inter-dose interval
- `addl` additional doses (or events)
  - `total` alternative for total number of doses
- `ss` advance to steady-state?
  - 0 don't advance to steady-state
  - 1 advance to steady-state
  - 2 irregular steady-state
- `rate` give the dose zero-order with this rate
  - `tinf` alternative for infusion time
- other `name=value` items that you would like to appear in the data set underlying the simulation

See `?ev` for additional details.

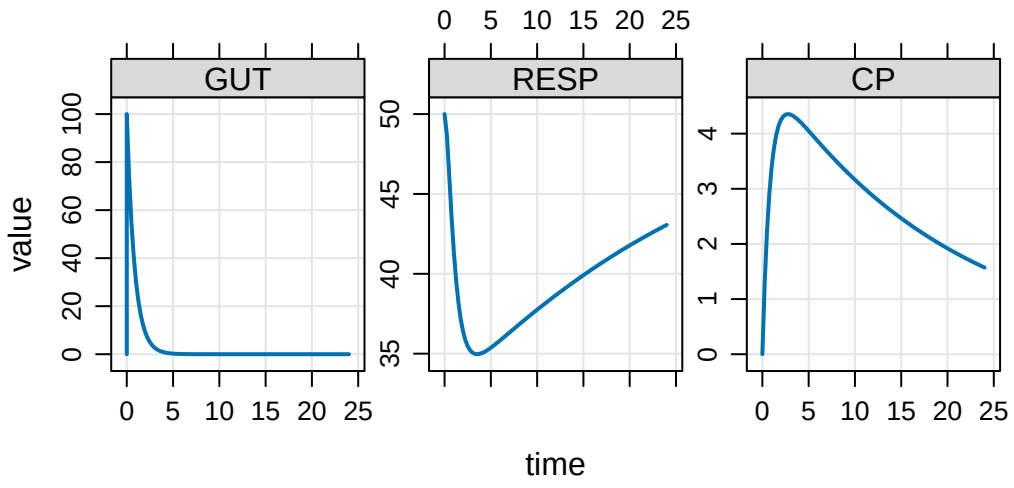
4.2 Invoke event object

There are several ways to create an invoke event objects.

4.2.1 Inline

When the event is simple and can be expressed in a single line, you can pipe the model object to `ev()` and then simulate.

```
mod <- house(outvars = "GUT,CP,RESP", end = 24)

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



This is a common workflow when exploring a model and an intervention.

4.2.2 As object

You can also save the event object and pass it into the pipeline as we did before with the inline setup.

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

mod %>% ev(e) %>% mrgsim() %>% plot()
```

Invoking the event object this way is a good idea when you want to create an intervention and apply it to several different simulation scenarios in the same script.

Alternatively, you can pass it in as the `events` argument for `mrgsim()`.

```
mod %>% mrgsim(events = e) %>% plot()
```

This is functionally the same as passing the (saved) event object into the pipeline via `ev()`.

4.3 Combining event objects

We can create more complex interventions from several, simpler event objects. mrgsolve provides an interface with helper functions to facilitate this.

4.3.1 Simple combination

Use the `c()` operator to concatenate several event objects into a single event object.

For 100 mg loading dose followed by 50 mg daily x6

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

maintenance <- ev(time = 24, amt = 50, ii = 24, addl = 5)

dose <- c(load, maintenance)

dose
```

```
Events:
  time amt cmt evid ii addl
1    0 100   1    1  0    0
2   24  50   1    1 24    5
```

The final event object has all the simpler event object smashed together, with no modification.

4.3.2 Sequence

We can make this simpler by putting these in a sequence using the `seq()` generic. Here is 100 mg daily for a week, followed by 50 mg daily for the rest of the month

```
a <- ev(amt = 100, ii = 24, total =  7)
b <- ev(amt = 50,  ii = 24, total = 21)

seq(a, b)
```

```
Events:
  time amt ii addl cmt evid
1    0 100 24    6   1    1
2  168  50 24   20   1    1
```

The output shows that the `b` event was automatically timed to start once all of the doses from the `a` event were given.

You can also put a waiting period in between event objects in a sequence; to wait for 7 days between `a` and `b` from the example above

```
seq(a, wait = 24*7, b)
```

```
Events:
  time amt ii addl cmt evid
1    0 100 24    6   1    1
2  336  50 24   20   1    1
```

Now, `b` starts one week after `a` ends.

4.3.3 Expand into multiple subjects

We can take any event object and replicate it into several subjects with the `ev_rep()` function.

```
seq(a,b)
```

```
Events:
  time amt ii addl cmt evid
1    0 100 24    6   1    1
2  168  50 24   20   1    1
```

```
seq(a,b) %>% ev_rep(1:3)
```

```
  ID time amt ii addl cmt evid
1  1    0 100 24    6   1    1
2  1  168  50 24   20   1    1
3  2    0 100 24    6   1    1
4  2  168  50 24   20   1    1
5  3    0 100 24    6   1    1
6  3  168  50 24   20   1    1
```

4.3.4 Combine into a data set

Use as_data_set with ev_rep() to create a single data set

```
c <- seq(a,b)

as_data_set(
  a %>% ev_rep(1:2),
  b %>% ev_rep(1:2),
  c %>% ev_rep(1:2)
)
```

```
  ID time amt ii addl cmt evid
1  1    0 100 24    6   1    1
2  2    0 100 24    6   1    1
3  3    0  50 24   20   1    1
4  4    0  50 24   20   1    1
5  5    0 100 24    6   1    1
6  5  168  50 24   20   1    1
7  6    0 100 24    6   1    1
8  6  168  50 24   20   1    1
```

This example gives us two subjects receiving 100 mg for a week, two subjects receiving 50 mg for 3 weeks, and two subjects receiving 100 mg for a week followed by 50 mg for 3 weeks.

4.4 Modifying event objects

You can use a selection of the tidyverse to modify event objects. For example,

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

ss <- mutate(single, ii = 24, ss = 1)

ss
```

```
Events:
  time amt ii ss cmt evid
1    0 100 24  1   1    1
```

Available tidyverse verbs for working on event objects include

- mutate()
- select()
- filter()

4.5 Column name case

By default, event objects have lower case names when they are rendered to a data frame or a data set

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

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

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

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

You can request upper case names by using the `evd()` constructor

```
evd(amt = 100) %>% as.data.frame()
```

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

These are the names you will see in the rendered data set and in the simulated output. Equivalent behavior is seen with

```
evd_expand(amt = 100)
expand.evd(amt = 100)
```

Note that, when working with event objects, always refer to lower case names

```
e <- evd(amt = 100)
e <- mutate(e, ss = 1)
as.data.frame(e)
```

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

You can change the case of any event object *to* upper case (`uctran()`) or *to* lower case (`lctran()`)

```
evd(amt = 100) %>% as.data.frame() %>% lctran()
```

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

In this example, we created an event object using `evd()` and then immediately requested lower case names. This step can also be performed on a raw data frame as well.

4.6 Rx specification

This is an alternate syntax letting you create event objects the same way you might write out a prescription.

```
ev_rx("100 mg x1 then 50 q12h x 10 at 24")
```

```
Events:
  time amt ii addl cmt evid
1    0 100  0    0   1    1
2    0  50 12    9   1    1
```

This syntax will cover many common dosing scenarios. But more complicated scenarios might require creating events as usual with `ev()` and then combining as described above.

5 Data sets

A "data set" is just a data frame with time, dose, and covariate information for one or more subjects in a population. You can make data frame any way you want, as long as the result meets a couple of requirements. That said, mrgsolve contains some convenient functions to make it easier to create these data frames. A data set is a more general form of the event objects you read about in Section 4, so it might help to review that section now if you haven't already.

5.1 Requirements

There are some requirements for input data sets.

- Only numeric data can get passed into a problem; mrgsolve will quietly remove any non-numeric columns before simulating
- Like event objects, NMTRAN data columns can be in upper or lower case, but not both (pick `AMT/CMT/TIME` or `amt/cmt/time`, not `AMT/cmt/TIME`)
- Also like event objects, you can switch between upper and lower case naming with `lctran()` and `uctran()`
- Must contain the columns `ID`, `TIME/time`, `CMT/cmt`, `EVID/evid`
- If you have a column with the same name as a parameter in your model, that column cannot contain missing values (`NA`; see Section 7)
- mrgsolve recognizes a "new" ID in the data set when the value of `ID` changes from record to record; however, we recommend using unique ID numbers for subjects in a data set

5.2 Connection with event objects

There is a natural connection between event objects and data sets: we made event objects to work the way they do so it would be easy to create data sets from them. So, if I have an event object specifying only a single dose

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

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

we can render that into a data set and it will be configured to meet all the requirements

```
  data <- as_data_set(e)
  data
```

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

Here, I didn't specify some required items (like `TIME`), but mrgsolve filled them in with something sensible.

Notice that, in this example, `e` is an event object

```
  class(e)
```

```
[1] "ev"
attr(,"package")
[1] "mrgsolve"
```

but `data` is just a plain old data frame

```
  class(data)
```

```
[1] "data.frame"
```

5.3 Multiple subjects in a data set

Data sets commonly contain information for more than one subject. There is no practical limit to the complexity you can code into the data set for any given individual: one subject might have a single dose whereas another subject may have many doses; doses can be given by different routes; doses may or may not be at steady state etc.

For example, it's not a problem to have this mix up of dosing scenarios in a single data set:

```
   ID TIME AMT RATE II SS ADDL CMT EVID
1   1    0 100    0  0  0    0   1    1
2   2    0 100    0  0  0    0   1    1
3   3    0 100    0  0  0    0   1    1
4   4    0 200   50 24  0    3   1    1
5   5    0  50    0 12  1    3   1    1
6   5   48 100    0 24  0    2   1    1
7   5  120 200    0 48  0    2   1    1
8   6    0  50    0 12  1    3   1    1
9   6   48 100    0 24  0    2   1    1
10  6  120 200    0 48  0    2   1    1
```

5.4 Creating data sets

mrgsolve provides some convenience functions you can use; we've found them helpful when certain input data sets are needed.

5.4.1 `ev_rep()`

One common way to make a data set is to replicate an event object. We can take a very simple event object

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

and replicate it to create several subjects with that dose. To create 3 subjects, call

```
ev_rep(e, 1:3)
```

```
  ID time amt cmt evid
1  1    0 100   1    1
2  2    0 100   1    1
3  3    0 100   1    1
```

Notice that `ev_rep()` converts the event object into a data frame / data set. This function can be convenient when the input event object is complicated like this

```
e1 <- evd(amt = 100, ii = 24, addl = 5)
e2 <- evd(amt = 200, ii = 12, addl = 6)
e3 <- evd(amt = 50, rate = 25)

e <- ev_seq(e1, e2, e3)
e
```

```
Events Data:
  time amt rate ii addl cmt evid
1    0 100    0 24    5   1    1
2  144 200    0 12    6   1    1
3  228  50   25  0    0   1    1
```

It's easy to replicate this into a data set of 3 (or 300) subjects

```
ev_rep(e, 1:3)
```

```
  ID TIME AMT RATE II ADDL CMT EVID
1  1    0 100    0 24    5   1    1
2  1  144 200    0 12    6   1    1
3  1  228  50   25  0    0   1    1
4  2    0 100    0 24    5   1    1
5  2  144 200    0 12    6   1    1
6  2  228  50   25  0    0   1    1
7  3    0 100    0 24    5   1    1
8  3  144 200    0 12    6   1    1
9  3  228  50   25  0    0   1    1
```

5.4.2 as_data_set()

Alternatively, we can use ev_rep() to replicate each event object and then combine them all into a single data frame with as_data_set()

```
as_data_set(ev_rep(e1, 1:2), ev_rep(e, 1:2))
```

```
  ID TIME AMT RATE II ADDL CMT EVID
1  1    0 100    0 24    5   1    1
2  2    0 100    0 24    5   1    1
3  3    0 100    0 24    5   1    1
4  3  144 200    0 12    6   1    1
5  3  228  50   25  0    0   1    1
6  4    0 100    0 24    5   1    1
7  4  144 200    0 12    6   1    1
8  4  228  50   25  0    0   1    1
```

as_data_set() combines these event objects together and ensures there are unique values for each ID while maintain the complexity within each subject.

5.4.3 ev_expand()

The ev_expand() function will create a single data frame / data set for many subjects, but each subject has a single dosing record. The _expand part of the function name indicates that we create all combinations of inputs. For example, we can have 3 individuals at each of 2 doses

```
ev_expand(ID = 1:3, AMT = c(200, 400))
```

```
  ID time amt cmt evid AMT
1  1    0   0   1    1 200
2  2    0   0   1    1 200
3  3    0   0   1    1 200
4  4    0   0   1    1 400
5  5    0   0   1    1 400
6  6    0   0   1    1 400
```

Notice that ID runs from 1 to 6 and each ID appears only on one row. You might see another function called expand.ev(); it does the same thing, but is named to mimic its base R cousin called expand.grid().

5.4.4 See the user guide

See the mrgsolve user guide for more functions you can use to create interesting input data sets. But remember that you don't *have* to use our convenience functions; you might be better off coding your own to get the inputs you need for your simulation.

5.5 What about observation records?

You'll notice our input data sets only include dosing records. When this is the case, mrgsolve will fill in the observation records for you according to the internal simulation time grid maintained in the model object (Section 3.7). You can put observation records into the data set and get exactly the sampling scheme you want for each subject; this is how we simulate from clinical data sets (e.g. those you might use in model development). So this is possible and you might do it. But most of the time when you are simulating *de novo*, you should let mrgsolve fill in the observation records. More on this in the user guide.

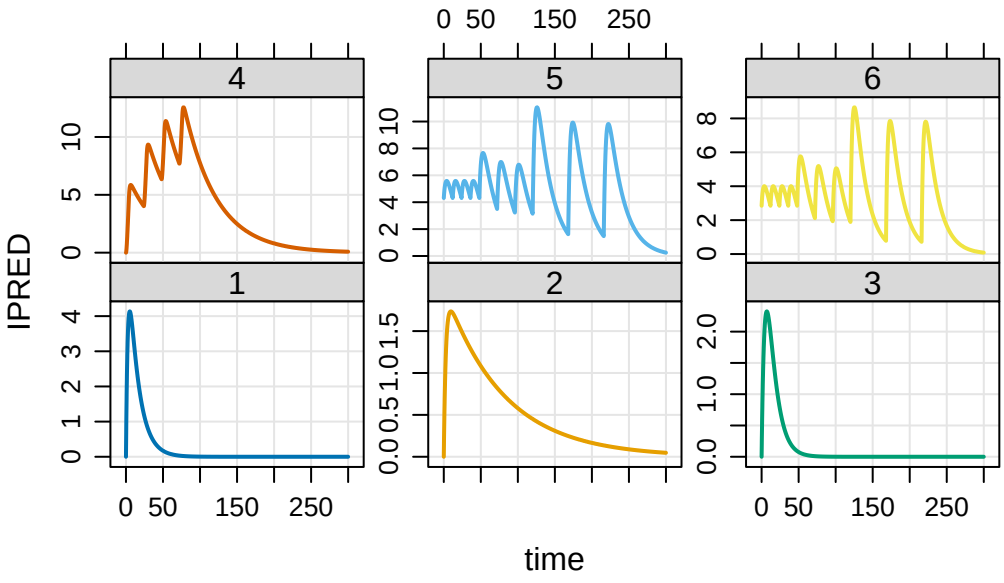
5.6 Invoking data sets

Once you have a model your data set is created, you can pass it into a simulation pipeline with the `data_set()` function. Simulating from the data set we showed in Section 5.3

```
mod <- modlib("popex", delta = 0.1, end = 300)
```

```
Building popex ... done.
```

```
set.seed(2468)
mod %>%
  data_set(data) %>%
  mrgsim(recsort = 3) %>%
  plot(IPRED ~ time|factor(ID))
```



Alternatively, you can just pass the data set to `mrgsim()`

```
mod %>% mrgsim(data = data)
```

6 Simulation and outputs

This section discusses

- Simulation from a model object
- Dealing with simulated output

6.1 mrgsim()

Use the `mrgsim()` function to actually run the simulation. We always pass in the model object as the first argument.

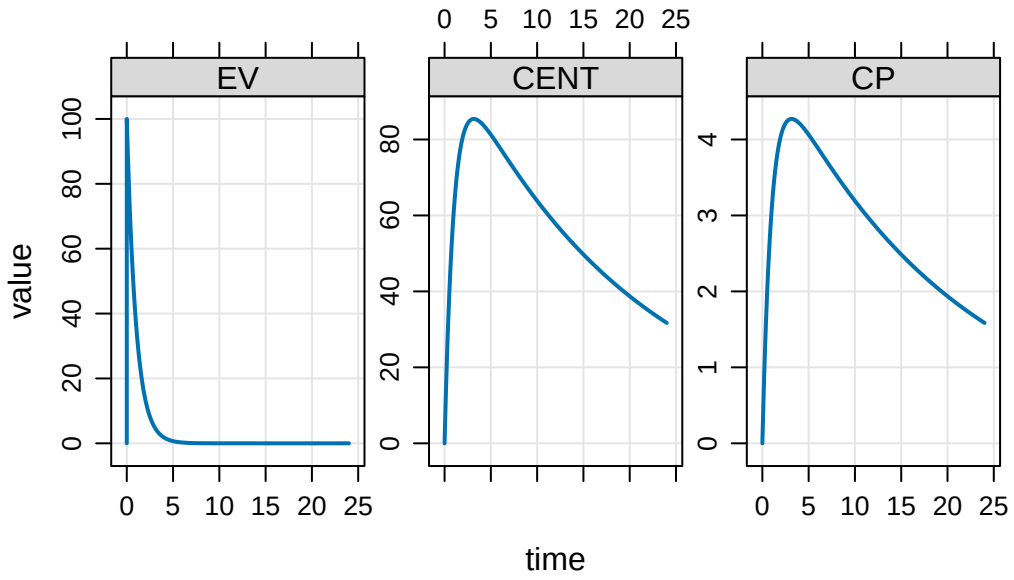
```
mod <- modlib("pk1") %>% ev(amt = 100)
```

```
mrgsim(mod)
```

```
Model:  pk1
Dim:    242 x 5
Time:   0 to 24
ID:     1
    ID time     EV   CENT     CP
1:   1  0.0   0.00  0.000 0.0000
2:   1  0.0 100.00  0.000 0.0000
3:   1  0.1  90.48  9.492 0.4746
4:   1  0.2  81.87 18.034 0.9017
5:   1  0.3  74.08 25.715 1.2858
6:   1  0.4  67.03 32.619 1.6309
7:   1  0.5  60.65 38.819 1.9409
8:   1  0.6  54.88 44.383 2.2191
```

Alternatively, we can execute the simulation by passing the model object in with the pipe

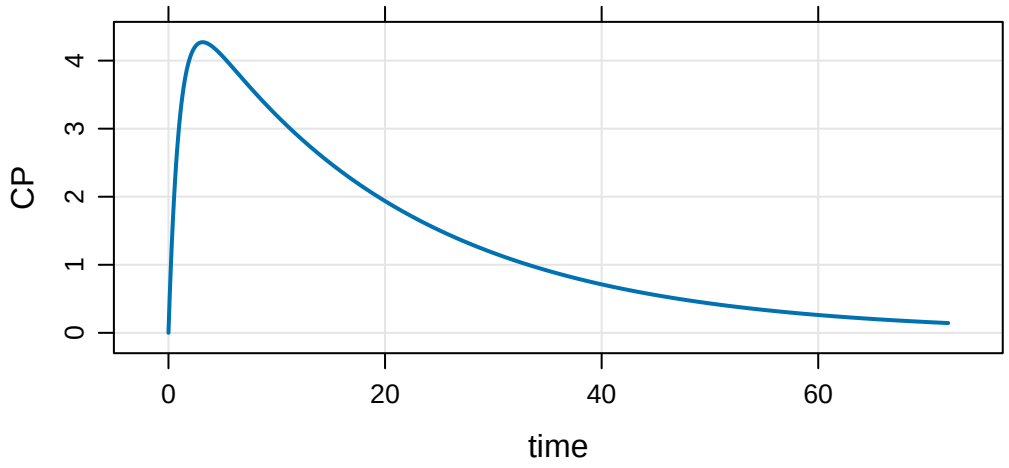
```
mod %>% mrgsim() %>% plot()
```



6.1.1 Update

The `mrgsim()` signature contains `...` which are passed to `update()`. Use this mechanism to customize your simulation or the output on the fly

```
mod %>% mrgsim(outvars = "CP", end = 72, delta = 0.1) %>% plot()
```



In this example, we selected the output variable (`CP`), ran the simulation to 72 hours (`end = 72`) and asked for a finer output time grid (`delta = 0.1`).

6.1.2 Options

There are some options that can *only* be set when you call `mrgsim()`. These are function arguments; you can see them at `?mrgsim`.

- `carry_out`: numeric data columns to *copy* into the simulated output
- `recover`: like `carry_out` but works with any type
- `output`: pass `"df"` to get output as a regular data frame
- `obsonly`: don't return dosing records in the simulated output
- `etasrc`: should ETAs be simulated? or scraped from the data set
- `recsort`: how doses and observations having the same time are ordered
- `tad`: insert time after dose into the output
- `ss_n` and `ss_fixed`: settings for finding steady state
- `nocb`: next observation carry backward; set to `FALSE` for locf

6.1.3 Variants

Inputs

There are `mrgsim()` variants which are specific to the types of inputs

- `mrgsim_e()` - just an event object
- `mrgsim_d()` - just a data set
- `mrgsim_ei()` - event + idata set
- `mrgsim_di()` - data set + idata set
- `mrgsim_i()` - just idata set

Outputs

You can also call `mrgsim_df()`, which is a wrapper for `mrgsim()` that always returns a data frame.

Quick

Call `mrgsim_q()` for a quick turnaround simulation, with minimal overhead (and features). This is only really useful when you are simulating repeatedly, many 100s or 1000s of times or more … like when estimating parameters or doing optimal design. These functions will not make a single simulation run much faster and they won't turn a long-running simulation into a short-running simulation.

6.2 Simulated output

`mrgsim()` returns an object with class `mrgsims`; this is essentially a data frame but with some extra features.

```
out <- mrgsim(mod)

class(out)
```

```
[1] "mrgsims"
attr(,"package")
[1] "mrgsolve"
```

```
head(out)
```

```
  ID time        EV      CENT        CP
1  1  0.0   0.00000  0.000000 0.0000000
2  1  0.0 100.00000  0.000000 0.0000000
3  1  0.1  90.48374  9.492112 0.4746056
4  1  0.2  81.87308 18.033587 0.9016794
5  1  0.3  74.08182 25.715128 1.2857564
6  1  0.4  67.03200 32.618803 1.6309401
```

```
summary(out)
```

```
       ID          time             EV                CENT
 Min.   :1   Min.   : 0.000   Min.   :  0.00000   Min.   : 0.00
 1st Qu.:1   1st Qu.: 5.925   1st Qu.:  0.00000   1st Qu.:41.38
 Median :1   Median :11.950   Median :  0.00059   Median :55.09
 Mean   :1   Mean   :11.950   Mean   :  4.34229   Mean   :56.50
 3rd Qu.:1   3rd Qu.:17.975   3rd Qu.:  0.24198   3rd Qu.:72.26
 Max.   :1   Max.   :24.000   Max.   :100.00000   Max.   :85.41
       CP
 Min.   :0.000
 1st Qu.:2.069
 Median :2.754
 Mean   :2.825
 3rd Qu.:3.613
 Max.   :4.270
```

6.2.1 Output scope

The first column in the simulated output is always `ID`. The second column in the output is always `time` (or `TIME`).

By default, you get simulated values in all compartments and for every derived output *at every* time

```
head(out)
```

```
  ID time        EV      CENT        CP
1  1  0.0   0.00000  0.000000 0.0000000
2  1  0.0 100.00000  0.000000 0.0000000
3  1  0.1  90.48374  9.492112 0.4746056
4  1  0.2  81.87308 18.033587 0.9016794
5  1  0.3  74.08182 25.715128 1.2857564
6  1  0.4  67.03200 32.618803 1.6309401
```

- `EV` and `CENT` are compartments
- `CP` is a derived variable (`CENT/V`)

We can use the `outvars()` function to look at what compartments and derived variables will come back in the simulation

```
outvars(mod)
```

```
$cmt
[1] "EV"   "CENT"

$capture
[1] "CP"
```

You can control which compartments and derived outputs are returned when you do a simulation run. This is a *really* important feature when the simulations become very large: limiting the outputs to those you actually need can make the difference between a simulation that fits within the available memory and one that doesn't.

To request specific outputs at simulation time, set `outvars` in the model object. In this example, we make the selection on the fly

```
mod %>%
  update(outvars = "CP") %>%
  mrgsim()
```

```
Model:  pk1
Dim:    242 x 3
Time:   0 to 24
ID:     1
    ID time     CP
1:   1  0.0 0.0000
2:   1  0.0 0.0000
3:   1  0.1 0.4746
```

```
4:    1  0.2 0.9017
5:    1  0.3 1.2858
6:    1  0.4 1.6309
7:    1  0.5 1.9409
8:    1  0.6 2.2191
```

Alternatively, we can make the change persistent

```
mod2 <- update(mod, outvars = "CP")

outvars(mod2)
```

```
$cmt
character(0)

$capture
[1] "CP"
```

6.2.2 Copy inputs into output

Input data items can be *copied* into the simulated output without passing through the model c++ code itself.

For most applications, use the `recover` argument to `mrgsim()`.

```
data <- expand.ev(amt = c(100, 300))

data <- mutate(
  data,
  dose = amt,
  arm = case_match(
    dose,
    100 ~ "100 mg x1",
    300 ~ "300 mg x1"
  )
)

out <- mrgsim(mod, data, recover = "dose, arm", output = "df")

count(out, dose, arm)
```

```
  dose       arm   n
1  100 100 mg x1 242
2  300 300 mg x1 242
```

This will let you copy inputs of *any type* into the output (for example, character or factor data).

If you just want to get numeric inputs into the output, use `carry_out`

```
data <- expand.ev(amt = c(100, 300)) %>% mutate(dose = amt)

out <- mrgsim(mod, data, carry_out = "dose", output = "df")

count(out, dose)
```

```
  dose   n
1  100 242
2  300 242
```

6.3 Working with `mrgsims` object

The `mrgsims` object can be convenient to work with when the output is small.

```
mod <- modlib("pk1", delta = 0.1)
```

```
Loading model from cache.
```

```
out <- mrgsim(mod, ev(amt = 100))
```

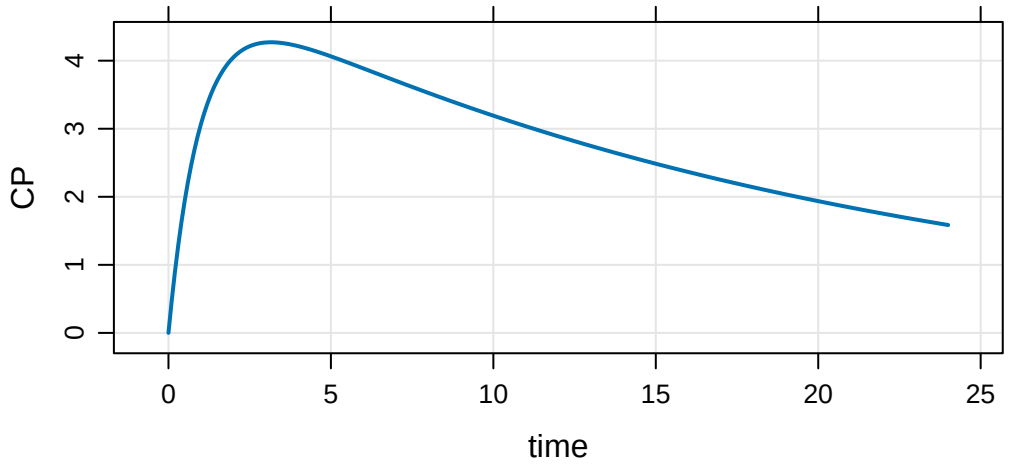
```
out
```

```
Model:  pk1
Dim:    242 x 5
Time:   0 to 24
ID:     1
    ID time     EV   CENT     CP
1:   1  0.0   0.00  0.000 0.0000
2:   1  0.0 100.00  0.000 0.0000
3:   1  0.1  90.48  9.492 0.4746
4:   1  0.2  81.87 18.034 0.9017
5:   1  0.3  74.08 25.715 1.2858
6:   1  0.4  67.03 32.619 1.6309
7:   1  0.5  60.65 38.819 1.9409
8:   1  0.6  54.88 44.383 2.2191
```

6.3.1 Plot

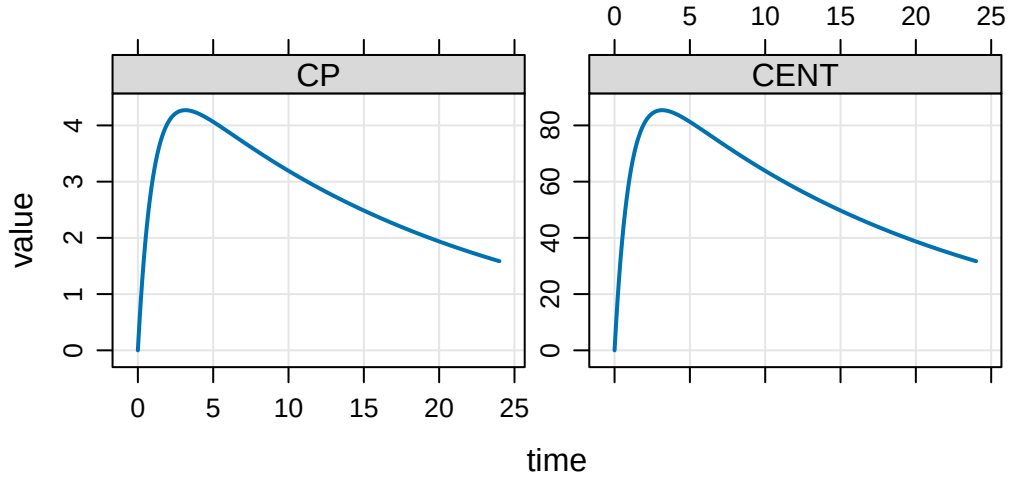
The main benefit from using this object is the ability to easily make plots to see what happened in the simulation. You can plot a single output

```
plot(out, CP ~ time)
```



or a collection of outputs

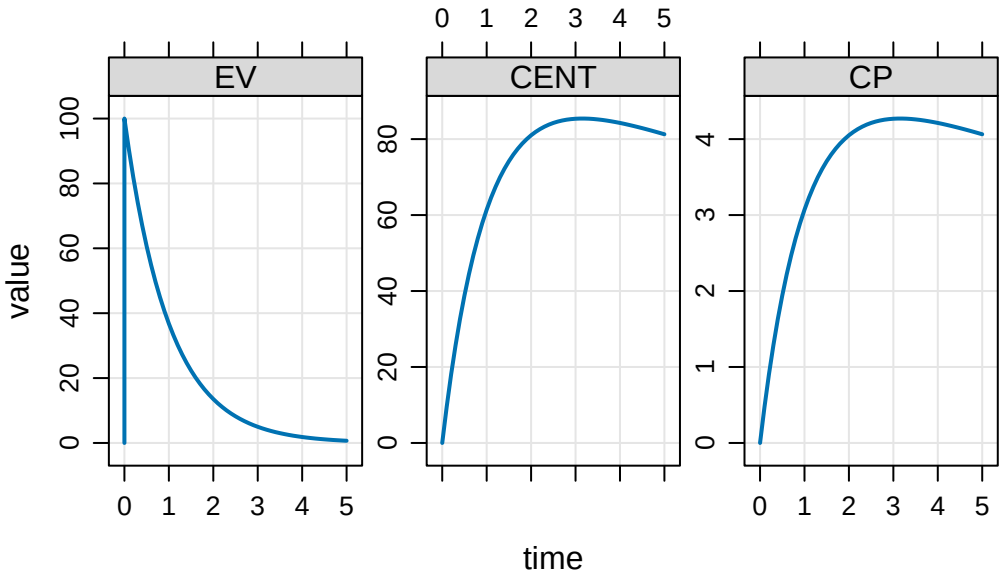
```
plot(out, "CP CENT")
```



6.3.2 Filter

Use `filter_sims()` to limit the rows that are retained in the simulated output

```
out2 <- filter_sims(out, time <= 5)

plot(out2)
```



6.3.3 Mutate

Use `mutate_sims()` to alter the columns in the simulated output

```
mutate_sims(out, week = time/168)
```

```
Model:  pk1
Dim:    242 x 6
Time:   0 to 24
ID:     1
    ID time     EV   CENT     CP       week
1:   1  0.0   0.00  0.000 0.0000 0.0000000
2:   1  0.0 100.00  0.000 0.0000 0.0000000
3:   1  0.1  90.48  9.492 0.4746 0.0005952
4:   1  0.2  81.87 18.034 0.9017 0.0011905
5:   1  0.3  74.08 25.715 1.2858 0.0017857
6:   1  0.4  67.03 32.619 1.6309 0.0023810
7:   1  0.5  60.65 38.819 1.9409 0.0029762
8:   1  0.6  54.88 44.383 2.2191 0.0035714
```

6.4 Coerce output

When output is big, the methods mentioned above are less likely to be useful: what we really want is just a simple data frame to work on. In this case, coerce outputs to data.frame or tibble

```
df <- as.data.frame(out)
df <- as_tibble(out)
head(df)
```

```
# A tibble: 6 x 5
     ID  time    EV  CENT    CP
  <dbl> <dbl> <dbl> <dbl> <dbl>
```

```
1      1   0      0    0     0
2      1   0    100    0     0
3      1   0.1   90.5  9.49 0.475
4      1   0.2   81.9 18.0  0.902
5      1   0.3   74.1 25.7  1.29
6      1   0.4   67.0 32.6  1.63
```

Once the output is coerced to data frame, it is like any other R data frame.

Remember that you can get a data frame directly back from `mrgsim()` with the `output` argument

```
mrgsim(mod, ev(amt = 100), output = "df") %>% class()
```

```
[1] "data.frame"
```

This is what you'll want to do most of the time when doing larger simulations.

6.4.1 dplyr verbs

You can pipe simulated output directly to several dplyr verbs, for example `filter()` or `mutate()`.

```
mod %>% mrgsim(ev(amt = 100)) %>% mutate(rep = 1)
```

```
# A tibble: 242 x 6
      ID  time    EV  CENT    CP   rep
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1     1   0      0    0    0         1
 2     1   0    100    0    0         1
 3     1   0.1   90.5  9.49 0.475     1
 4     1   0.2   81.9 18.0  0.902     1
 5     1   0.3   74.1 25.7  1.29      1
 6     1   0.4   67.0 32.6  1.63      1
 7     1   0.5   60.7 38.8  1.94      1
 8     1   0.6   54.9 44.4  2.22      1
 9     1   0.7   49.7 49.4  2.47      1
10     1   0.8   44.9 53.8  2.69      1
# i 232 more rows
```

This will first coerce the output object to a data frame and then continue to work on the simulated data according to the functions in the pipeline.

Other verbs you can use on an `mrgsims` object include

- `group_by()`
- `mutate()`
- `filter()`
- `summarise()`
- `select()`
- `slice()`
- `pull()`
- `distinct()`

7 Model parameters

Model parameters are `name` / `value` pairs that are used *inside* your model, but they can be varied *outside* the model.

Understanding how mrgsolve handles model "parameters" particularly important for generating interesting and robust simulations.

Big picture

- `mrgsolve` maintains a parameter list, including parameter names and values
  - The parameter list is set at the time the model is compiled; names and number of parameters cannot be changed after compile time
  - This list is used by default if nothing else is done
  - The parameter values in this list can be updated
- `mrgsolve` will check input data sets for *columns* which have the same name as a parameter
  - When a match is made between data set and parameter list, `mrgsolve` will update the value based on what is passed on the data
  - Parameters in `idata` are checked (and parameter list updated) first; after that, the data set is checked (and parameter list updated)

7.1 Coding model parameters

Traditionally, we've used the `$PARAM` block to set parameter names and values

```
$PARAM
WT = 70, SEX = 0, EGFR = 100
```

New in mrgsolve 1.2.0, you can use the `$INPUT` block. This is another way to specify parameters, but they will have a special *tag* on them that we can use later.

```
$INPUT
WT = 70, SEX = 0, EGFR = 100
```

It's best if you can set these to sensible values; this is usually the *reference* value in your covariate model or some other value that gives you a sensible *default* output.

7.2 Updating parameter values

You can't change the names or number of parameters after you compile the model, but you can change the values. You can update parameters either

- *prior to* simulation or
- *during* simulation

We will illustrate with this model

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

```
Building parameters_mod ... done.
```

```
param(mod)
```

```
Model parameters (N=8):
name   value . name   value
EGFR   100   | THETA3 0.262
SEX    0     | THETA4 0.331
THETA1 0     | THETA5 -0.211
THETA2 3     | WT     70
```

There parameters are:

- WT
- SEX
- EGFR
- THETA1 ... THETA5

7.2.1 Update prior to simulation

Use `param()` to update the model object. You can do this in one of two ways.

7.2.1.1 Update with `name=value`

The first way is to pass the new value with the parameter name you want to change. To change `WT`

```
mod$WT
```

```
[1] 70
```

```
mod <- param(mod, WT = 80)

mod$WT
```

```
[1] 80
```

And when we simulate,

```
mrgsim_df(mod) %>% count(WT)
```

```
  WT  n
1 80 25
```

You can also do this via `update()`

```
mod <- update(mod, param = list(WT = 60))

mod$WT
```

```
[1] 60
```

Remember that `mrgsim()` passes to `update()` so you can do the same thing with

```
out <- mrgsim(mod, param = list(WT = 70))
```

This will generate simulated output with `WT` set to 70.

7.2.2 Update with object

If you have a named object, you can pass that in to the update as well. For example, pass in a named list

```
p <- list(WT = 70.2, FOO = 1)

mod <- param(mod, p)

mod$WT
```

```
[1] 70.2
```

Or a data frame

```
data <- data.frame(WT = c(70, 80.1), BAR = 2)

mod <- param(mod, data[2,])

mod$WT
```

```
[1] 80.1
```

7.3 Update during simulation

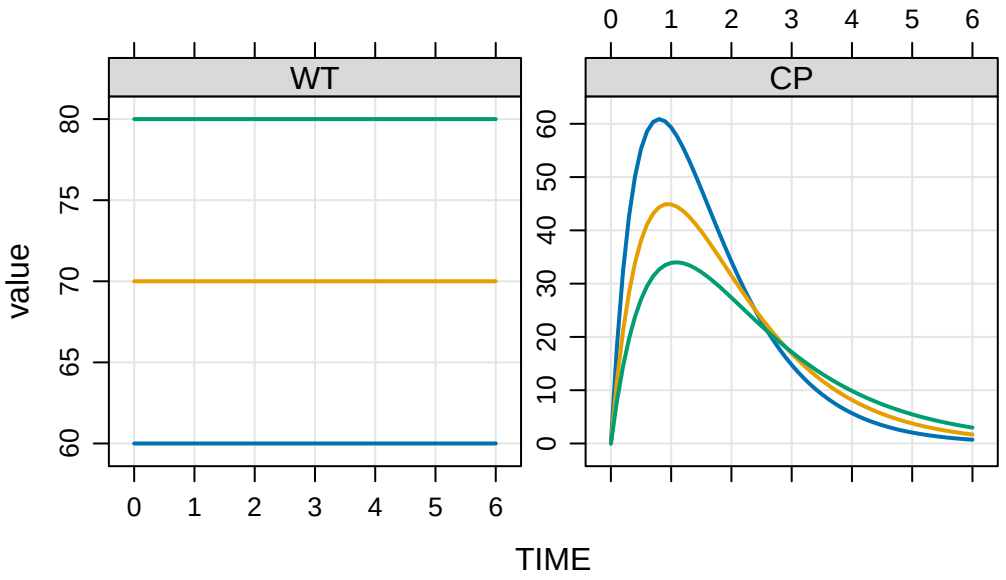
In this approach, we'll add a columns to our input data set with the same names as our parameters and let `mrgsolve` pick up the new values. To illustrate, load a data set from which to simulate

```
data <- read.csv("parameters-data.csv")
data
```

```
  ID TIME AMT CMT WT SEX EGFR EVID
1  1    0 100   1 60   0   60    1
2  2    0 100   1 70   0   60    1
3  3    0 100   1 80   0   60    1
```

In this data set, subjects 1, 2, and 3 have different (increasing) weight; all subjects have `SEX=0` and `EGFR=60`. When we pass this data frame for simulation and plot

```
out <-
  mod %>%
  data_set(data) %>%
  zero_re() %>%
  mrgsim(delta = 0.1, end = 6)

plot(out, "WT,CP")
```



All of this *only* works if the names in the data set match up with the names in the model.

7.4 Check if the names match

Recall that we coded the model covariates using `$INPUT`, rather than `$PARAM`? We can see that these parameters have this special tag

```
param_tags(mod)
```

```
  name   tag
1   WT input
2  SEX input
3 EGFR input
```

They have the `input` tag, which means we expect to find them on the data set *when we ask*. We can check this data set against the parameters in the model

```
check_data_names(data, mod)
```

```
Found all expected parameter names in `data`.
```

Now, modify the data set so it has `eGFR` rather than `EGFR`

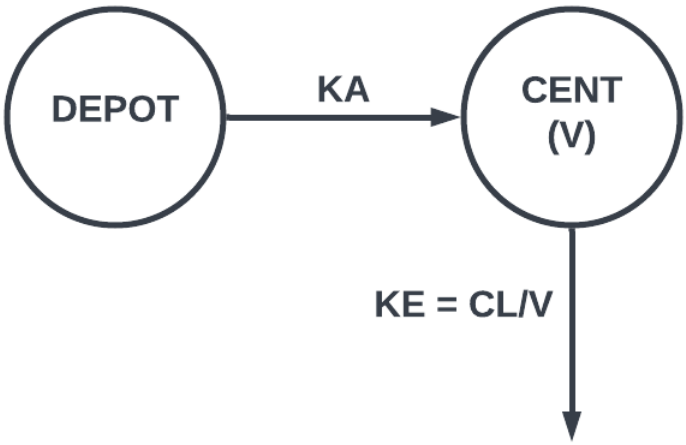
```
data2 <- rename(data, eGFR = EGFR)

check_data_names(data2, mod)
```

```
Warning: Could not find the following parameter names in `data`:
* EGFR (input)
i Please check names in `data` against names in the parameter list.
```

See the `mode` argument to `check_data_names()`; you can warn or inform the user in case parameter names don't look right, or you can issue an error.

8 Model Specification

This chapter gives a broad overview of mrgsolve model specification syntax. We'll start by coding up a pharmacokinetic model. The model will be very simple to start, letting us get some concepts in place. Later on, we'll do more complicated model syntax.



The model parameters are

- `CL`
- `V`
- `KA`

The model compartments are

- `CENT`
- `DEPOT`

8.1 Model specification blocks

Model components are coded into blocks, which are delineated by a specific block syntax. You have a couple of options

NONMEM style

These start with $ and then the block name (`$PK`)

Bracket style

Put the block name in brackets (`[ ERROR ]`)

Upper or lower case

You can use either:

- `$error`
- `[ pk ]`
- `[ DES ]`

etc … they all work.

8.1.1 Syntax

The "type" of code you write will vary from block to block. Sometimes it is an R-like syntax and sometimes it is c++ code.

Don't worry if you don't know c++! We have taken a lot of the complexity out and with a handful of exceptions, the code should be pretty natural and similar to what you write in R.

8.2 Base model blocks

8.2.1 Parameters

Use the `$PARAM` block header.

```
$PARAM
CL = 1, V = 20, KA = 1.1
```

Parameters have a **name** and a **value**, separated by =.

Parameter names can be upper or lower case. If you want punctuation, use underscore _.

Parameter values must *evaluate* to a numeric value.

Parameters can't be functions of other parameters *when writing the `$PARAM` block*. But there is a place where you can do this …we'll see this later on.

Multiple parameters can go on one line, but separate by comma.

8.2.2 Read it in with `mread()`

Point `mread()` at your model file to read it in and see if it compiles.

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

```
Building simple_mod ... done.
```

We suggest writing the model in small sections, interactively checking to see if the model compiles. When you find syntax mistakes (you *will* find them), they will be easier to fix this way.

8.3 Compartments

```
$PARAM
CL = 1, V = 20, KA = 1.1

$CMT DEPOT CENT
```

Compartments are named

- Upper or lower case
- Punctuation use _

Order doesn't matter, but consider listing your default dosing compartment first. This is a convenient pattern to keep so you can just dose into compartment 1 when setting up your data set or event object.

8.4 Differential equations

Now, we'll write ODE using $DES (or $ODE) block.

```
$PARAM
CL = 1, V = 20, KA = 1.1

$CMT DEPOT CENT

$DES
dxdt_DEPOT = -KA * DEPOT;
dxdt_CENT  =  KA * DEPOT - (CL/V)*CENT;
```

Left hand side is `dxdt_<compartment name>`.

Right hand side can reference

- Compartments
- Parameters
- Other quantities derived in $DES or $PK
- Other internal variables

Unlike $PARAM and $CMT, this is c++ code; you can include any valid c++ statement. Also, because this is c++, each line or statement should end in semi-color (;).

8.5 Derived outputs

Like NONMEM, derived can be calculated in the $ERROR block.

```
$PARAM
CL = 1, V = 20, KA = 1.1

$CMT DEPOT CENT

$DES
dxdt_DEPOT = -KA * DEPOT;
```

```
dxdt_CENT  =  KA * DEPOT - (CL/V)*CENT;

$ERROR
double CP = CENT/V;
```

Like `$DES`, this block must be valid `c++` code.

Here we have created a new variable called `CP`, which is the amount in the central compartment divided by the central volume of distribution.

When we create a new variable, we must declare its `type`. Use `double` for a floating point number.

8.6 Capture outputs into the simulated data

mrgsolve has a `$CAPTURE` block that works like NONMEM's `$TABLE`. Just list the names you want copied into the output.

```
$PARAM
CL = 1, V = 20, KA = 1.1

$CMT DEPOT CENT

$DES
dxdt_DEPOT = -KA * DEPOT;
dxdt_CENT  =  KA * DEPOT - (CL/V)*CENT;

$ERROR
double CP = CENT/V;

$CAPTURE CP
```

Rather than putting stuff in `$CAPTURE`, try declaring with type `capture`

```
$ERROR
capture CP = CENT/V;
```

`capture` is identical to type `double`, but tells mrgsolve to include this item in the simulated output.

A little-use feature is renaming items in `$CAPTURE`

```
$ERROR
double DV = CENT/V;

$CAPTURE CP = DV
```

The syntax is `<new-name> = <old-name>`.

8.7 Covariate model

Like NONMEM, we can use `$PK` (or `$MAIN`) to code the covariate model, random effects, F, D, R, and ALAG, and initialize compartments.

```
$PK

double CL = TVCL * pow(WT/70, 0.75) * exp(ETA(1));
```

- Any valid c++ code is allowed
- Each line (statement) should end in semi-colon ;

8.8 C++ examples

You can find all sorts of help with c++ syntax on the web. Here are a few common bits of c++ code that you might need in your model.

```
if(a == 2) b = 2;
if(b <= 2) {
  c=3;
} else {
  c=4;
}
d = a==2 ? 50 : 100;
double d = pow(base,exponent);
double e = exp(3);
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);
```

8.8.1 Integer division

Be careful of dividing two integers; it's usually not what you want to do. When people get bit by this, it's usually when they divide one integer literal by another integer literal in their code. For example, we might *think* the following should evaluate to `0.75`

```
double result = 3/4; # 0
```

but it doesn't. Here, `result` will evaluate to `0` because the c++ compiler will do integer division between the 3 and the 4 and you'll get `0`.

It is good to get in the habit of putting `.0` behind whole numbers.

```
double result = 3.0/4.0; # 0.75
```

Of course, you *might* really want to divide two integers at some point; but for now, please mind this "feature" of c++ when writing your code.

8.9 Random effects

There are times when you *will* need to code this manually. When estimating with NONMEM and simulating with mrgsolve, these matrices will frequently be imported automatically via `$NMXML` or `$NMEXT`.

8.9.1 Omega / ETA

Diagonal matrix

```
$OMEGA
0.1 0.2 0.3
```

This is a 3x3 matrix with 0.1, 0.2, and 0.3 on the diagonal.

Block matrix

```
$OMEGA @block
0.1 0.002 0.3
```

This is a 2x2 matrix matrix with 0.1 and 0.3 on the diagonal. Sometimes it's easier to see when we code it like this

```
$OMEGA @block
0.1
0.002 0.3
```

Random effects simulated from OMEGA are referred to with `ETA(n)`.

8.9.2 Sigma / EPS

Works just like Omega / ETA, but use `$SIGMA` and `EPS(n)`.

For sigma-like theta, code it just as you would in NONMEM.

```
$PARAM THETA12 = 0.025

$SIGMA 1

$ERROR
double W = sqrt(THETA12);
Y = (CENT/V) + W*EPS(1);
```

There is no `FIX` in `mrgsolve`; everything in OMEGA and SIGMA is always fixed.

8.10 Import estimates from NONMEM

- Use $NMEXT or $NMXML
  - $NMEXT reads from the .ext file
    - Can be faster than $NMXML when the root.xml file gets big
    - Doesn't retain $OMEGA and $SIGMA structure
  - $NMXML reads from the .xml file
    - Can be slower than $NMEXT
    - Does retain $OMEGA and $SIGMA structure

This is the safest way to call

```
$NMXML
path = "../nonmem/106/106.xml"
root = "cppfile"
```

You might be able to use this run/project approach as well

```
$NMXML
run = 1006
project = "../sim/"
root = "cppfile"
```

This code will look for 1006/1006.xml under sim, one directory level up from the location of the mrgsolve "cpp' file.

8.11 Models in closed form

mrgsolve will solve one- and two-compartment models with first order input in closed form. This usually results in substantial speed up. Use $PKMODEL.

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

Certain symbols are required to be defined depending on the model. mrgsolve models are always parameterized in terms of clearances and volumes except for absorption, which is in terms of rate constant.

- CL / V
- CL / V / KA
- CL / V2 / Q / V3
- CL / V2 / Q / V3 / KA

These can be defined as a parameter or a derived quantity in $PK.

Compartment names are user-choice; the only thing mrgsolve cares about is the number of compartments.

8.12 Plugins

8.12.1 autodec

Historically, you have had to *declare* the type of any new variable you want to create.

```
$PK
double KE = CL/V;
```

For most models, the numeric variables you declare are likely to be floating point numbers … with type `double`.

We created a plugin that tells mrgsolve to look for new variables and declare them for you.

```
$PLUGIN autodec

$PK
KE = CL/V;
```

8.12.2 nm-vars

mrgsolve historically has used

- `CENT`
- `dxdt_CENT`
- `F_CENT`
- `D_CENT`

etc. When we started mrgsolve, this was a really nice feature because you didn't have to think about compartment *numbers*. However, this made translation of the model more difficult.

When you invoke the `nm-vars` plugin, you can write in a syntax that is much more like NONMEM.

For example

```
$PK
F2 = THETA(3);

ALAG2 = EXP(THETA(4));

$DES
DADT(1) = - KA * A(1);
```

Other convenience syntax

- `LOG()` and `log()`
- `LOG10()` and `log10()`
- `EXP()` and `exp()`
- `DEXP()` and `exp()`
- `SQRT()` and `sqrt()`
- `COS()` and `cos()`

Regardless of whether you have `nm-vars` invoked or not, you can still use `THETA(n)` to refer to parameter `THETAn`.

Try the `nm-like` model in the model library for an example.

```
mod <- modlib("nm-like")

mod@code
```

8.12.3 Rcpp (random numbers)

This gives you functions and data structures that you're used to using in R, but they work in c++.

The main use for this is random number generation. Any d/q/p/r function in R will be available; arguments are the same, but omit `n` (you always get just one draw when calling from c++).

For a draw from U(0,1)

```
$PLUGIN Rcpp

$ERROR
double u = R::runif(0, 1);
```

Note: the model compilation time will slightly increase any time you invoke the Rcpp resources. It's still tolerable, but I just wouldn't include Rcpp if you don't have to.

8.13 Other blocks

- Use `$SET` to configure the model object on load
  - For example, set the simulation end time
- Use `$ENV` to define a set of R objects that might be evaluated in other model blocks
- Use `$PRED` for other user-written closed form models
- Use `$PREAMBLE` for code that gets run once at the start of a problem `NEWIND==0`
- Use `$GLOBAL` to define variables outside of any other block

8.14 Variables and macros

There is too much syntax to mention it all here. You will find all the syntax here

https://mrgsolve.org/user-guide/

8.15 Modeled event times

To get the model to stop at any time (even if not in the data set) with EVID 2

```
double mt1 = self.mtime(1.23 + ETA(1));
```

To get the model to stop at any time with user-specified EVID (e.g. 33)

```
self.mevent(1.23 + ETA(1), 33);
```