mrgsolve: Simulate from ODE-Based Models

Get Started Package Vignette

Kyle T. Baron

2023-01-04

mrgsolve is an R package maintained under the auspices of Metrum Research Group that facilitates simulation from models based on systems of ordinary differential equations (ODE) that are typically employed for understanding pharmacokinetics, pharmacodynamics, and systems biology and pharmacology. mrgsolve consists of computer code written in the R and C++ languages, providing an interface to a C++ translation of the lsoda differential equation solver. This vignette will show you how to get started using mrgsolve.

Keywords: rstats, mrgsolve, simulation, ODE, PK, PKPD, PBPK, QSP

Contents

1 Big picture

In this section, 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 ticket items.

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 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.
```

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 that 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 7.

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")
```

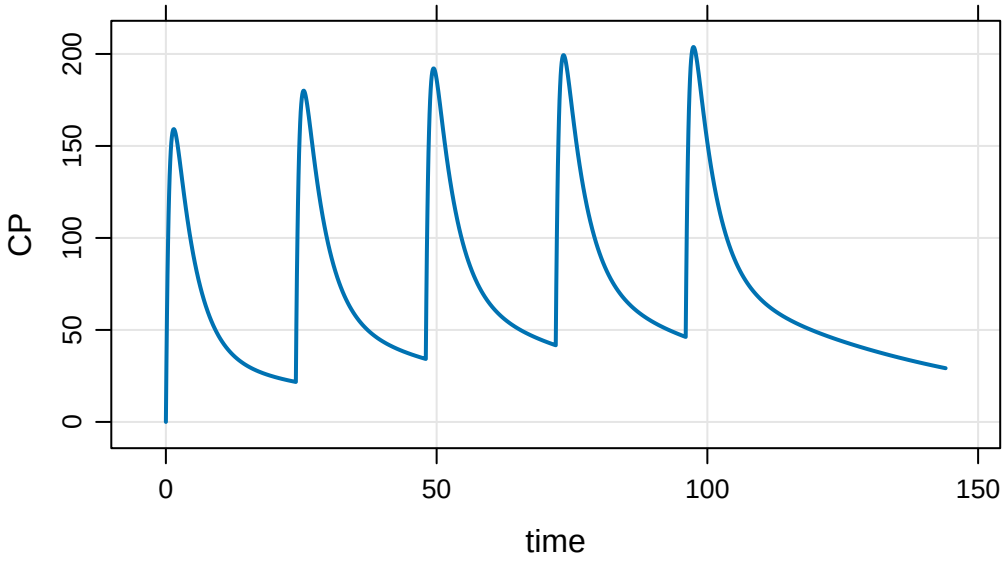


Figure 1: Example simulation of a single profile with an event object.

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 5.

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

When we simulate a population, we want to simulate a collection of individuals (or profiles) in a single simulation. 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.

```
set.seed(9876)

data <- expand.ev(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'll give every subject 250 mg.

```
set.seed(9876)

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

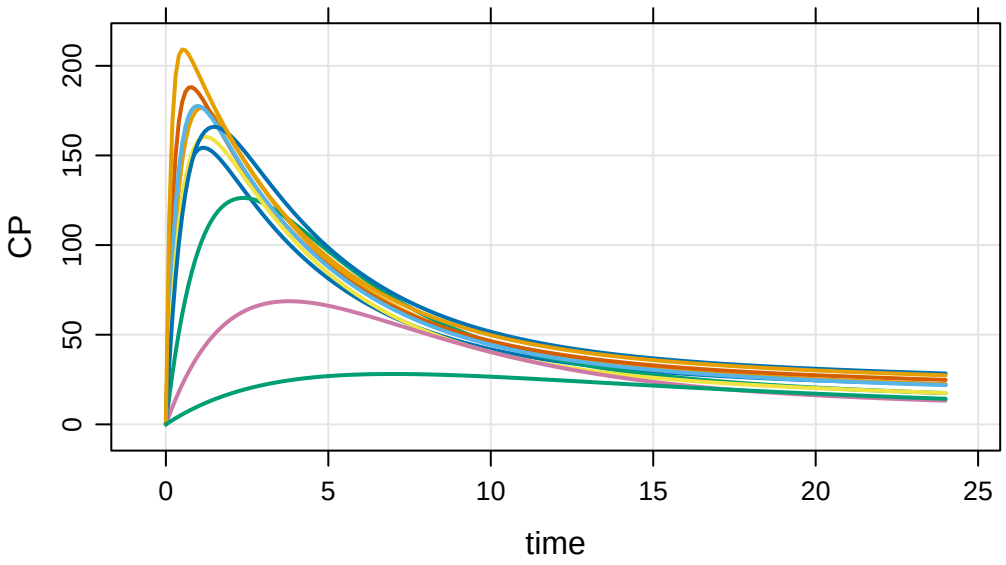


Figure 2: Example population simulation.

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

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. 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")
```

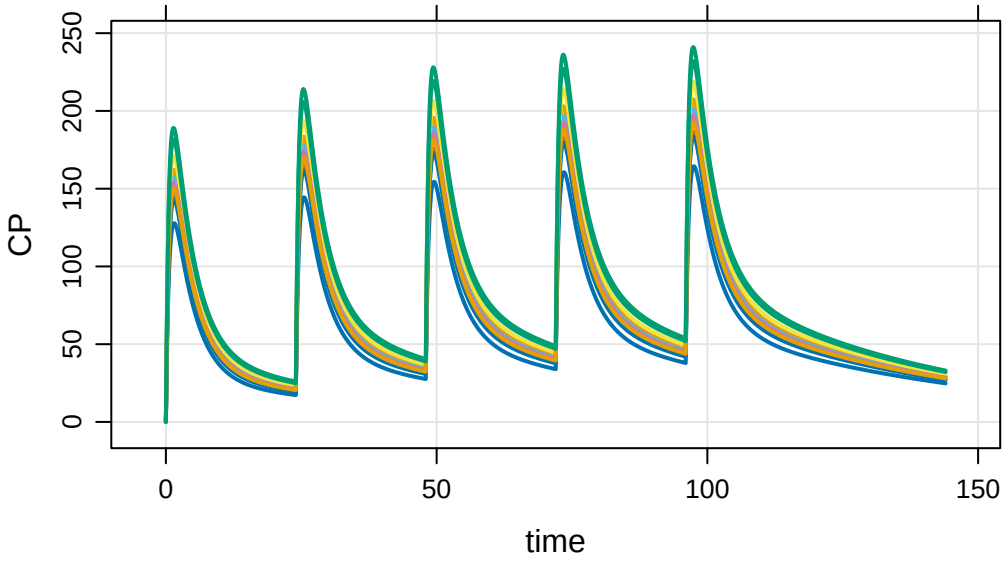


Figure 3: Example simulation with `idata_set`.

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

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 that simulates a data set once and call that repeatedly to get replicate simulated data sets.

1.6 The overall pattern

So the overall 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 = 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
```

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

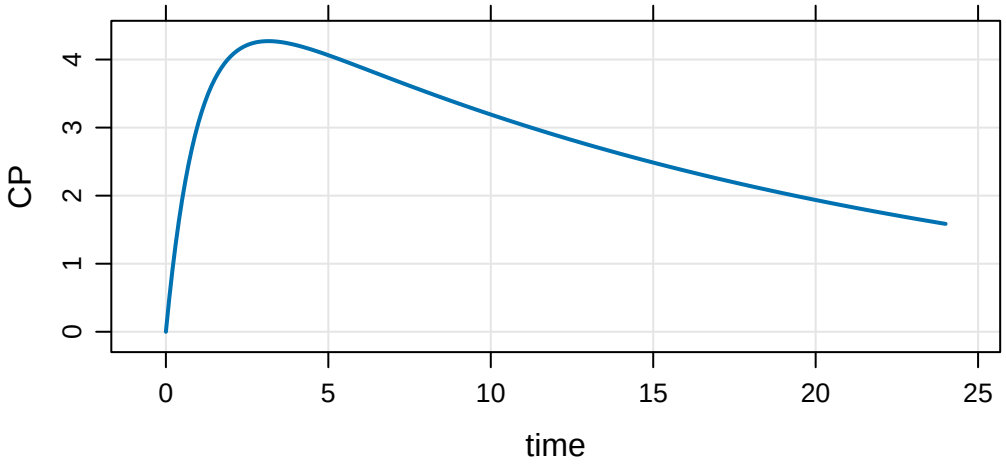


Figure 4: It's easy to get started quickly with mrgsolve.

That was a really simple simulation where we used an event object to initiate a dose into a one-compartment model. See 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 ***model object***.

- The model object contains all information about *the model*
  - Compartments
  - ODE
  - Algebraic relationships
  - Random effects
  - More
- The model object is what you use in R to
  - Query the model
  - Run simulations

3.1 `mread()`

Load a model from a model specification file using `mread()`.

- Don't worry for now what is in that file; we'll show you how to create it.
- 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++` and format the

code in ways that aren't what you usually want. So using `.mod` (or `.txt` can be helpful just to keep your editor from doing too much.

3.1.1 Syntax to load a model

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

Provide the complete path to the file

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

Provide the directory as an argument

Assumes you are keeping all simulation code in the `models` directory

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

Set `project` as an option

All my models are in this directory *for this project*

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

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

Update on load

mrgsolve provides an `update()` method for updating 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.

In this example, we'll

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

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

3.1.2 Read and cache

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

When you load the model the first time, it'll be

```
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; 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.

- Pre-coded models
- Sourced from inside the `mrgsolve` installation directory
- Great way to get models to experiment with
  - But I rarely use these for production work

This code will load a 1-compartment PK model.

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

List out the location of he model library

```
modlib()
```

```
[1] "/Users/kyleb/renv/renv/library/R-4.3/aarch64-apple-darwin20/mrgsolve/models"
```

So the `modlib()` function is equivalent to

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

Find out what models are in the library

```
?modlib
```

3.3 Overview

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

```
mod
```

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

  project: /Users/kyleb/vig...tes/src/model
  shared object: test_mod-so-f0ed61c91d9

  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 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*)

3.5 Compartments

We'll keep working with the model object we loaded in the previous section.

Models also have compartments. Like parameters, compartments have

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

Compartments also have a ***number***

Query the compartment list with `init()`

```
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. But there is a model syntax that allows you to write a model in terms of named compartments (e.g. `A(2)` or `F1`).

3.6 Random effects

```
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 `update()`. To change the simulation time grid we might 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
- Solver settings
  - `atol`, `rtol`
  - `hmax`, `maxsteps`, `mxhnil`, `ixpr`
  - Usually changing `rtol`, `atol`, and maybe `hmax`
- `$OMEGA`, `$SIGMA`
- `tscale` (rescale the output time)
- `digits`

Parameter update

To update parameters, use `param()`. More on this in Section 6

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

3.8 Advanced

Get the value of a parameter or setting

```
mod$CL
```

```
[1] 1
```

```
mod$end
```

```
[1] 240
```

Extract all parameters as a list

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

```
$CL
[1] 1

$V
[1] 20

$TVKA
[1] 1.2
```

Extract the value of one parameter

```
mod$CL
```

```
[1] 1
```

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.

These are like quick and easy data sets.

4.1 Create and 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

See `?ev` for additional details.

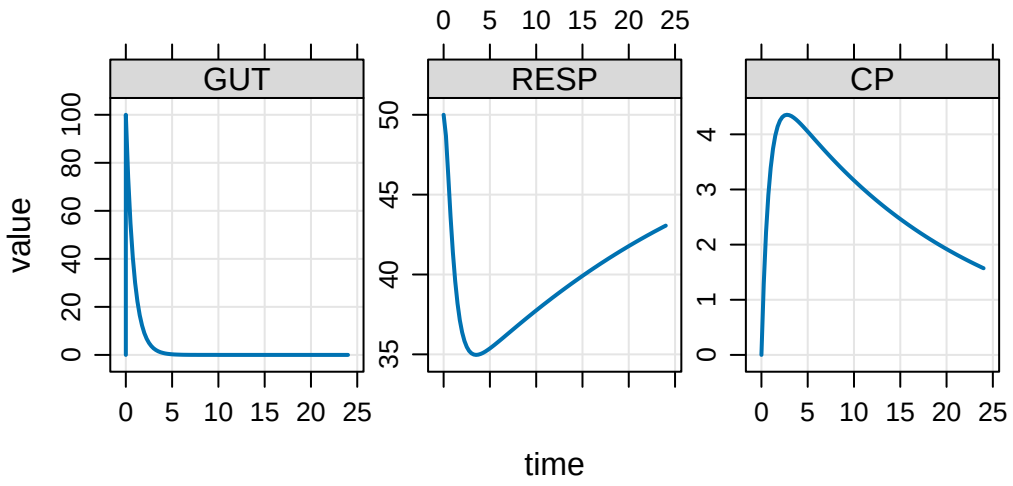
4.2 Invoke event object

4.2.1 Inline

Pipe the model object to `ev()` then simulate.

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

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



4.2.2 As object

You can save the event object out and pass it in.

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

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

This is a good idea when you want to create an intervention and apply it in several different simulation scenarios.

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

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

4.3 Combining event objects

We can create more complex interventions from several simpler event objects.

4.3.1 Simple combination

Use the `c()` operator to concatenate.

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)

c(load, maintenance)
```

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

4.3.2 Sequence

We can make this simpler by putting these in a sequence. 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
```

4.3.3 Expand into multiple subjects

Pass an event object to `ev_rep()` with the IDs you want

```
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
1.1  2    0 100 24    6   1    1
2.1  2  168  50 24   20   1    1
1.2  3    0 100 24    6   1    1
2.2  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 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 cmt evid amt ii addl
1...1    1    0   1    1 100 24    6
1.1...2  2    0   1    1 100 24    6
1...3    3    0   1    1  50 24   20
1.1...4  4    0   1    1  50 24   20
1...5    5    0   1    1 100 24    6
2        5  168   1    1  50 24   20
1.1...7  6    0   1    1 100 24    6
2.1      6  168   1    1  50 24   20
```

4.4 Modifying event objects

You can use a selection of the tidyverse to modify event objects

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

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

ss
```

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

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

4.5 Rx specification

```
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
```

5 Simulation and outputs

This chapter discusses

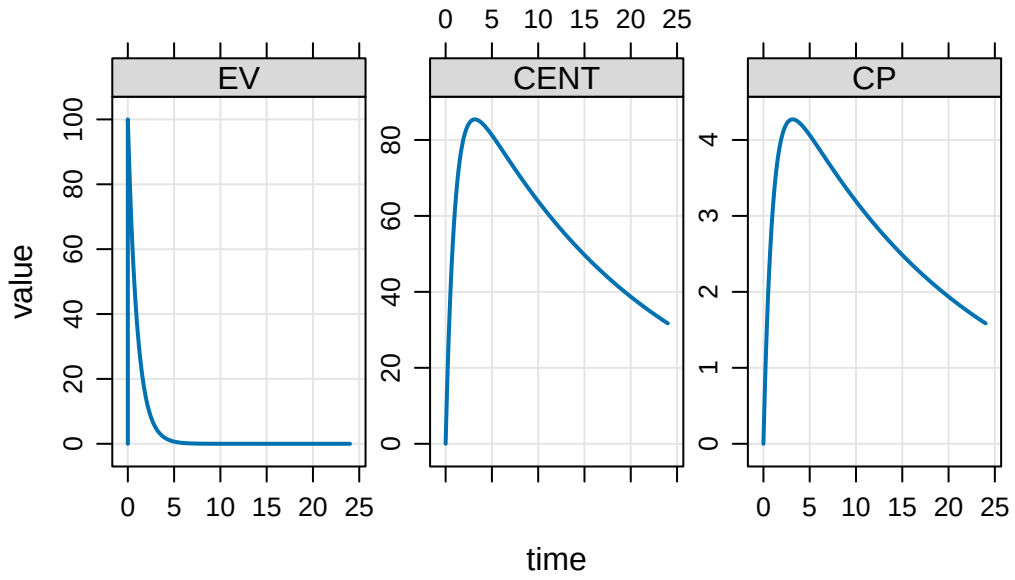
- Simulation from a model object
- Dealing with simulated output

5.1 `mrgsim()`

Use the `mrgsim()` function to actually run the simulation

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

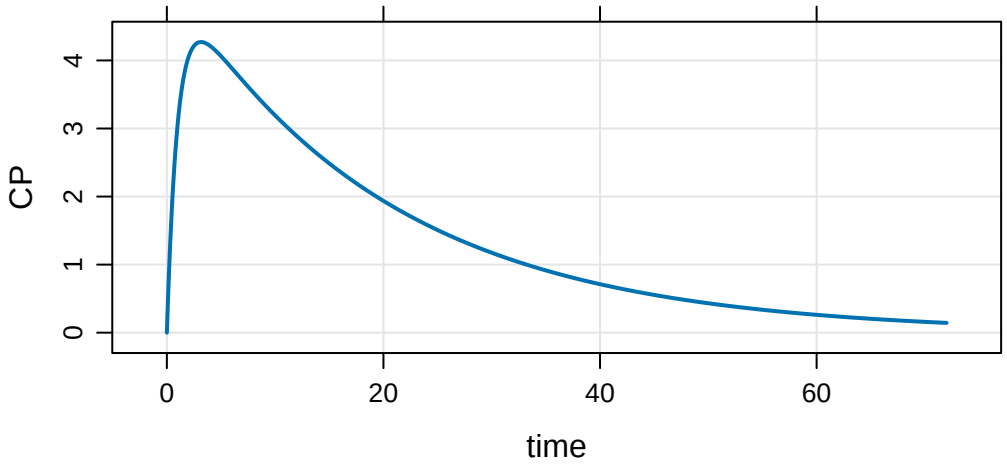
```
mod %>% mrgsim() %>% plot()
```



5.1.1 Update

The `mrgsim()` signature contains `...` which are passed to `update()`

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



5.1.2 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

5.1.3 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). Only really useful when you are simulating repeatedly … like when estimating parameters or doing optimal design.

5.2 Simulated output

`mrgsim()` returns an object with class `mrgsims`; it's a data frame 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
```

5.2.1 Output scope

- First column is always `ID`
- Second column is always `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`)

```
outvars(mod)
```

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

$capture
[1] "CP"
```

You can control which compartments and derived outputs are returned.

Request specific outputs at simulation time

```
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
```

Or make the change persistent

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

outvars(mod2)
```

```
$cmt
character(0)

$capture
[1] "CP"
```

5.2.2 Copy inputs into output

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

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

```
data <- expand.ev(amt = c(100,300)) %>%
  mutate(dose = amt, arm = "100 mg x1", "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  300 100 mg x1 242
```

This will let you copy inputs of *any type* into the output.

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
```

5.3 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))
```

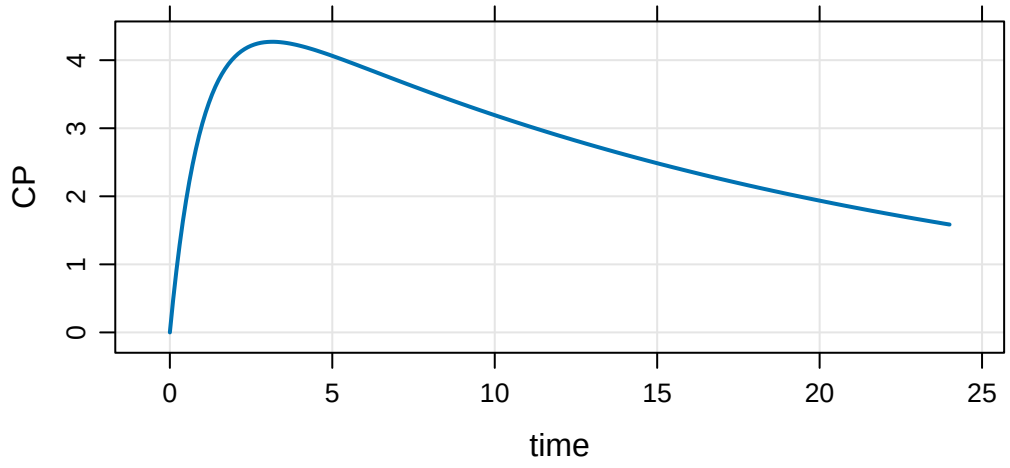
```
out
```

```
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
```

5.3.1 Plot

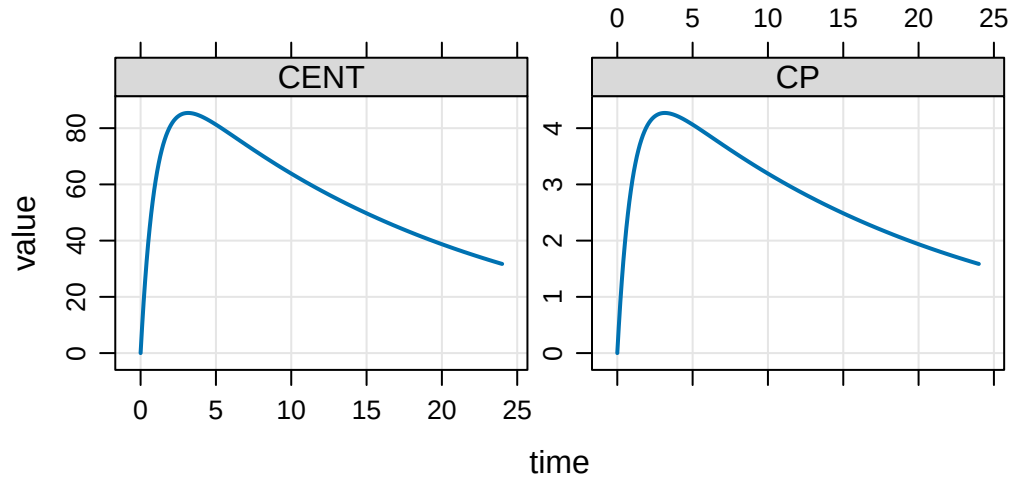
Plot with a formula

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



or a vector of output names

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

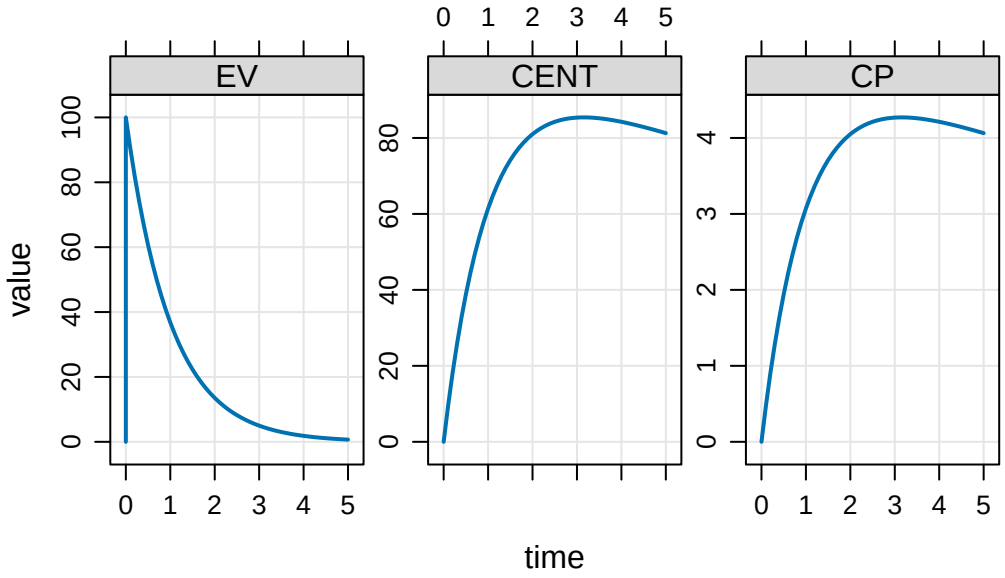


5.3.2 Filter

Use `filter_sims()`

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

plot(out2)
```



5.4 Coerce output

When output is big, these methods are less likely to be useful.

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.

5.4.1 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 include

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

6 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 names and values
  - 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)

6.1 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; specifically, set to the *reference* value in your covariate model.

6.2 Updating parameter values

You can't change the name or number of parameters after you compile the model, but you can change the value.

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

6.2.1 Update *prior to* simulation

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

6.2.1.1 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
```

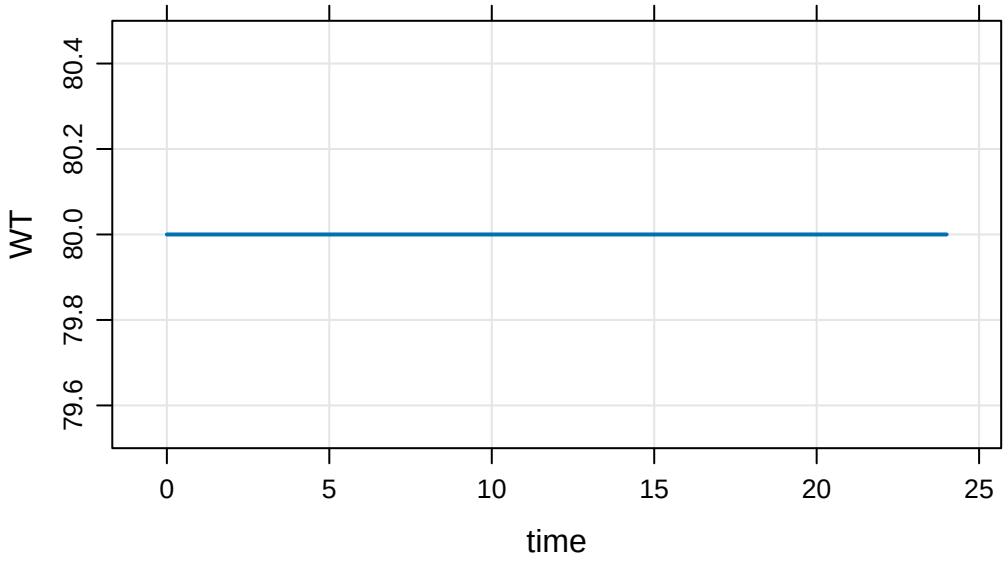
```
mod <- param(mod, WT = 80)

mod$WT
```

```
[1] 80
```

And when we simulate,

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



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

6.2.2 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
```

This will be a very important pattern we'll do later on

6.3 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
```

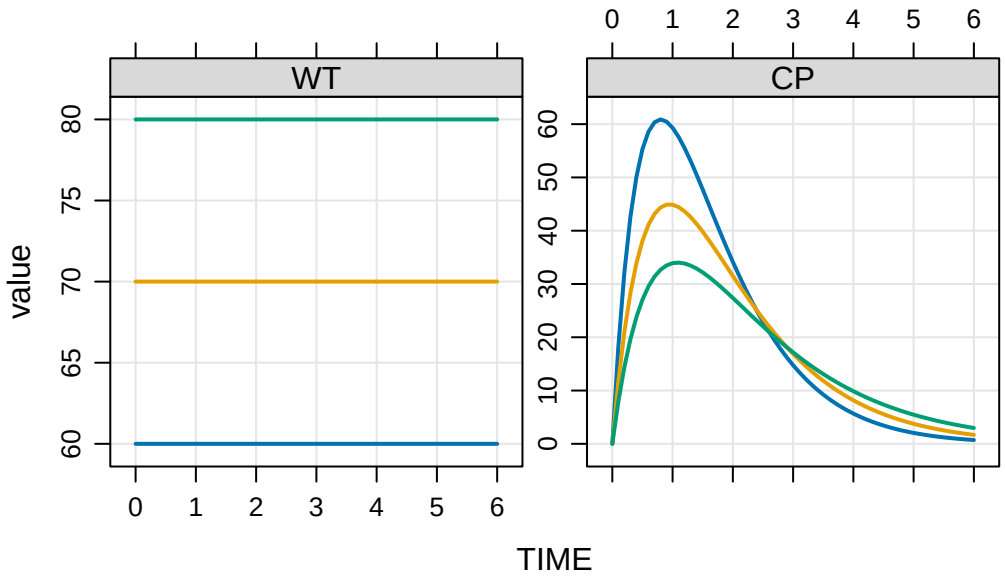
```
  ID TIME AMT CMT WT SEX EGFR EVID
1  1    0 100   1 60   0   60    1
2  2    0 100   1 70   0   60    1
3  3    0 100   1 80   0   60    1
```

- Subjects 1, 2, and 3 have different (increasing) weight
- All subjects have `SEX=0`
- All subjects have `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.

6.4 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`

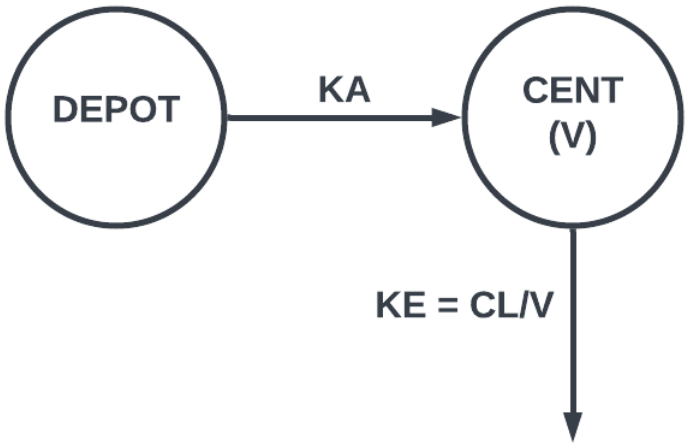
```
data2 <- rename(data, eGFR = EGFR)

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.
```

7 Model Specification

In this chapter, we'll code up a very simple pharmacokinetic model. This is just starter material to get some concepts in place. We'll do more complicated stuff later (see **?@sec-specification-2**).



Parameters

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

Compartments

- `CENT`
- `DEPOT`

7.1 Where to save your code?

- You can use any file name with any extension
- The extension you use may influence how your editor highlights and indents your code
- I use `.mod`

7.2 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 ]`
- etc … they all work

7.2.1 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.

We will show you more `c++` in chapters to come.

7.3 Base model blocks

7.3.1 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 numeric.

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.

7.3.2 Read it in with `mread()`

Point `mread()` at your model file to

- read in the model
- see if it compiles

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

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

We'll learn much more about `mread()` later on, but for now we need a way to check if our model coding is correct.

7.4 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, except

- When putting dosing records into the data set
- You want to use the NONMEM-style coding

7.5 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

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

7.6 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.

7.7 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
```

7.8 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 `;`

7.9 C++ examples

```
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);
```

Be careful of integer division

```
double result = 1/2; # 0
```

Here, `result` will evaluate to `0`.

Good practice to put `.0` behind whole numbers.

```
double result = 1.0/2.0; # 0.5
```

Lots of help on the web http://en.cppreference.com/w/cpp/numeric/math/tgamma

7.10 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`.

7.10.1 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)`.

7.10.2 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.

7.11 Import estimates from NONMEM

- Use `$NMEXT` or `$NMXML`
  - `$NMEXT` reads from the `.ext` file
    - Can be faster
    - Doesn't retain `$OMEGA` and `$SIGMA` structure
  - `$NMXML` reads from the `.xml` file
    - Can be slower
    - 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"
```

7.12 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.

7.13 Plugins

7.13.1 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;
```

7.13.2 nm-vars

`mrgsolve` historically 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()`
- `EXP()` and `exp()`
- `SQRT()` and `sqrt()`

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
```

7.13.3 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: this will slightly increase compile time.

7.14 Other blocks

- Use `$SETUP` to configure the model object on load
  - For example, set the simulation end time
- Use `$ENV` to define a set of R objects that might be evaluated in other model blocks
- Use `$PRED` for other closed form models
- Use `$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

7.15 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/

7.16 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);
```