RxODE user manual

Matthew Fidler, Melissa Hallow, Wenping Wang

2021-02-25

Contents

Chapter 1

Introduction

Welcome to the RxODE user guide; **RxODE** is an R package for solving and simulating from ode-based models. These models are convert the RxODE mini-language to C and create a compiled dll for fast solving. ODE solving using RxODE has a few key parts:

- `RxODE()` which creates the C code for fast ODE solving based on a simple syntax (Chapter 6) related to Leibnitz notation.

- The event data, which can be:
  - a `NONMEM` or `deSolve` compatible data frame (Chapter 7), or
  - created with `et()` or `EventTable()` for easy simulation of events (Chapter 11)
  - The data frame can be augmented by adding time varying or adding individual covariates (`iCov=` as needed)

- `rxSolve()` which solves the system of equations using initial conditions and parameters to make predictions
  - With multiple subject data, this may be parallelized.
  - With single subject the output data frame is adaptive
  - Covariances and other metrics of uncertainty can be used to simulate while solving.

While this is the user guide, there are other places that you can visit for help:

- `RxODE` github pkgdown page
- `RxODE` tutorial (accessible in tutorials in Rstudio 1.3+)
- `RxODE` github discussions

This book was assembled on Thu Feb 25 21:11:52 2021 with RxODE version 1.0.5 automatically by github actions.

Chapter 2

Authors and Acknowledgments

2.1 Authors

- Matthew L. Fidler (core team/developer/manual)
- Melissa Hallow (tutorial writer)
- Wenping Wang (core team/developer)

2.2 Contributors

- Zufar Mulyukov – Wrote initial version of `rxShiny()` with modifications from Matthew Fidler
- Alan Hindmarsh – Lsoda author
- Awad H. Al-Mohy – Al-Mohy matrix exponential author
- Ernst Hairer – dop853 author
- Gerhard Wanner – dop853 author
- Goro Fuji – Timsort author
- Hadley Wickham – Author of original findLhs in RxODE, also original author of .s3register (used with permission to anyone, both changed by Matthew Fidler)
- Jack Dongarra – LApack author; CRAN asked us to add
- Linda Petzold – LSODA
- Martin Maechler – expm author, used routines from there for inductive linearization
- Morwenn – Timsort author
- Nicholas J. Higham – Author of Al-mohy matrix exponential
- Roger B. Sidje – expokit matrix exponential author
- Simon Frost – thread safe C implementation of liblsoda
- Kevin Ushey – Original author of fast factor, modified by Matthew Filder

- Yu Feng – thread safe liblsoda
- Matt Dowle – forder primary author (version modified by Matthew Fidler to allow different type of threading and exclude grouping)
- Cleve Moler – LApack author; CRAN asked us to add
- David Cooley – Author of fast_factor which was modified and now is used RxODE to quickly create factors for IDs without sorting them like R does
- Daniel C. Dillon – Author of stripping utility to reduce size of RxODE/nlmixr executable
- Drew Schmidt – Drew Schmidt author of edits for exponential matrix utility taken from R package expm
- Arun Srinivasan – forder secondary author (version modified by Matthew Fidler to allow different type of threading, indexing and exclude grouping)

2.3   RxODE acknowledgments:

- Sherwin Sy – Weight based dosing example
- Justin Wilkins – Documentation updates, logo and testing
- Emma Schwager – R IJK distribution author
- J Coligne – dop853 fortran author
- Bill Denney – Documentation updates, manual and minor bug fixes
- Tim Waterhouse – Fixed one bug with mac working directories
- Richard Upton – Helped with solving the ADVAN linCmt() solutions
- Ross Ihaka – R author
- Robert Gentleman – R author
- R core team – R authors

Chapter 3

Related R packages

3.1 ODE solving

This is a brief comparison of pharmacometric ODE solving R packages to `RxODE`.

There are several R packages for differential equations. The most popular is deSolve.

However for pharmacometrics-specific ODE solving, there are only 2 packages other than RxODE released on CRAN. Each uses compiled code to have faster ODE solving.

- mrgsolve, which uses C++ lsoda solver to solve ODE systems. The user is required to write hybrid R/C++ code to create a mrgsolve model which is translated to C++ for solving.

  In contrast, `RxODE` has a R-like mini-language that is parsed into C code that solves the ODE system.

  Unlike `RxODE`, `mrgsolve` does not currently support symbolic manipulation of ODE systems, like automatic Jacobian calculation or forward sensitivity calculation (`RxODE` currently supports this and this is the basis of nlmixr's FOCEi algorithm)

- dMod, which uses a unique syntax to create "reactions". These reactions create the underlying ODEs and then created c code for a compiled deSolve model.

  In contrast `RxODE` defines ODE systems at a lower level. `RxODE`'s parsing of the mini-language comes from C, whereas `dMod`'s parsing comes from R.

  Like `RxODE`, `dMod` supports symbolic manipulation of ODE systems and calculates forward sensitivities and adjoint sensitivities of systems.

  Unlike `RxODE`, `dMod` is not thread-safe since `deSolve` is not yet thread-safe.

And there is one package that is not released on CRAN:

- PKPDsim which defines models in an R-like syntax and converts the system to compiled code.

  Like `mrgsolve`, PKPDsim does not currently support symbolic manipulation of ODE systems.

  PKPDsim is not thread-safe.

The open pharmacometrics open source community is fairly friendly, and the Rx-ODE maintainers has had positive interactions with all of the ODE-solving pharmacometric projects listed.

3.2   PK Solved systems

`RxODE` supports 1-3 compartment models with gradients (using stan math's auto-differentiation). This currently uses the same equations as `PKADVAN` to allow time-varying covariates.

`RxODE` can mix ODEs and solved systems.

3.2.1   The following packages for solved PK systems are on CRAN

- mrgsolve currently has 1-2 compartment (poly-exponential models) models built-in. The solved systems and ODEs cannot currently be mixed.
- pmxTools currently have 1-3 compartment (super-positioning) models built-in. This is a R-only implementation.
- PKPDmodels has a one-compartment model with gradients.

3.2.2   Non-CRAN libraries:

- PKADVAN Provides 1-3 compartment models using non-superpositioning. This allows time-varying covariates.

Chapter 4

Installation

You can install the released version of RxODE from CRAN with:

```r
install.packages("RxODE")
```

You can install the development version of RxODE with

```r
devtools::install_github("nlmixrdevelopment/RxODE")
```

To build models with RxODE, you need a working c compiler. To use parallel threaded solving in RxODE, this c compiler needs to support open-mp.

You can check to see if R has working c compiler you can check with:

```r
## install.packages("pkgbuild")
pkgbuild::has_build_tools(debug = TRUE)
```

If you do not have the toolchain, you can set it up as described by the platform information below:

4.0.1 Windows

In windows you may simply use installr to install rtools:

```r
install.packages("installr")
library(installr)
install.rtools()
```

Alternatively you can download and install rtools directly.

4.0.2   Mac OSX

To get the most speed you need OpenMP enabled and compile RxODE with that compiler. There are various options and the most up to date discussion about this is likely the data.table installation faq for MacOS. The last thing to keep in mind is that RxODE uses the code very similar to the original lsoda which requires the gfortran compiler to be setup as well as the OpenMP compilers.s

4.0.3   Linux

To install on linux make sure you install gcc (with openmp support) and gfortran using your distribution's package manager.

4.1   Development Version

Since the development version of RxODE uses StanHeaders, you will need to make sure your compiler is setup to support C++14, as described in the rstan setup page. For R 4.0, I do not believe this requires modifying the windows toolchain any longer (so it is much easier to setup).

Once the C++ toolchain is setup appropriately, you can install the development version from GitHub with:

```
# install.packages("devtools")
devtools::install_github("nlmixrdevelopment/RxODE")
```

Chapter 5

Getting Started

The model equations can be specified through a text string, a model file or an R expression. Both differential and algebraic equations are permitted. Differential equations are specified by `d/dt(var_name) =`. Each equation can be separated by a semicolon.

To load `RxODE` package and compile the model:

```
library(RxODE)
```

```
#> RxODE 1.0.5 using 4 threads (see ?getRxThreads)
```

```
mod1 <-RxODE({
    C2 = centr/V2;
    C3 = peri/V3;
    d/dt(depot) =-KA*depot;
    d/dt(centr) = KA*depot - CL*C2 - Q*C2 + Q*C3;
    d/dt(peri)  =                   Q*C2 - Q*C3;
    d/dt(eff)  = Kin - Kout*(1-C2/(EC50+C2))*eff;
})
```

```
#> qs v0.23.6.
```

5.1 Specify ODE parameters and initial conditions

Model parameters can be defined as named vectors. Names of parameters in the vector must be a superset of parameters in the ODE model, and the order of parameters within the vector is not important.

```r
theta <-
   c(KA=2.94E-01, CL=1.86E+01, V2=4.02E+01, # central
     Q=1.05E+01,  V3=2.97E+02,              # peripheral
     Kin=1, Kout=1, EC50=200)              # effects
```

Initial conditions (ICs) can be defined through a vector as well. If the elements are not specified, the initial condition for the compartment is assumed to be zero.

```r
inits <- c(eff=1);
```

If you want to specify the initial conditions in the model you can add:

```r
eff(0) = 1
```

5.2   Specify Dosing and sampling in RxODE

RxODE provides a simple and very flexible way to specify dosing and sampling through functions that generate an event table. First, an empty event table is generated through the "eventTable()" function:

```r
ev <- eventTable(amount.units='mg', time.units='hours')
```

Next, use the `add.dosing()` and `add.sampling()` functions of the `EventTable` object to specify the dosing (amounts, frequency and/or times, etc.) and observation times at which to sample the state of the system. These functions can be called multiple times to specify more complex dosing or sampling regiments. Here, these functions are used to specify 10mg BID dosing for 5 days, followed by 20mg QD dosing for 5 days:

```r
ev$add.dosing(dose=10000, nbr.doses=10, dosing.interval=12)
ev$add.dosing(dose=20000, nbr.doses=5, start.time=120,
              dosing.interval=24)
ev$add.sampling(0:240)
```

If you wish you can also do this with the `mattigr` pipe operator %>%

```r
ev <- eventTable(amount.units="mg", time.units="hours") %>%
  add.dosing(dose=10000, nbr.doses=10, dosing.interval=12) %>%
  add.dosing(dose=20000, nbr.doses=5, start.time=120,
             dosing.interval=24) %>%
  add.sampling(0:240)
```

The functions `get.dosing()` and `get.sampling()` can be used to retrieve information from the event table.

```
head(ev$get.dosing())
```

```
#>   id low time high       cmt   amt rate ii addl evid ss dur
#> 1  1  NA    0   NA (default) 10000    0 12    9    1  0   0
#> 2  1  NA  120   NA (default) 20000    0 24    4    1  0   0
```

```
head(ev$get.sampling())
```

```
#>   id low time high   cmt amt rate ii addl evid ss dur
#> 1  1  NA    0   NA (obs)  NA   NA NA   NA    0 NA  NA
#> 2  1  NA    1   NA (obs)  NA   NA NA   NA    0 NA  NA
#> 3  1  NA    2   NA (obs)  NA   NA NA   NA    0 NA  NA
#> 4  1  NA    3   NA (obs)  NA   NA NA   NA    0 NA  NA
#> 5  1  NA    4   NA (obs)  NA   NA NA   NA    0 NA  NA
#> 6  1  NA    5   NA (obs)  NA   NA NA   NA    0 NA  NA
```

You may notice that these are similar to NONMEM event tables; If you are more familiar with NONMEM data and events you could use them directly with the event table function `et`

```
ev  <- et(amountUnits="mg", timeUnits="hours") %>%
  et(amt=10000, addl=9,ii=12,cmt="depot") %>%
  et(time=120, amt=2000, addl=4, ii=14, cmt="depot") %>%
  et(0:240) # Add sampling
```

You can see from the above code, you can dose to the compartment named in the RxODE model. This slight deviation from NONMEM can reduce the need for compartment renumbering.

These events can also be combined and expanded (to multi-subject events and complex regimens) with `rbind`, `c`, `seq`, and `rep`. For more information about creating complex dosing regimens using RxODE see the RxODE events section.

5.3   Solving ODEs

The ODE can now be solved by calling the model object's `run` or `solve` function. Simulation results for all variables in the model are stored in the output matrix x.

```
x <- mod1$solve(theta, ev, inits);
knitr::kable(head(x))
```

time	C2	C3	depot	centr	peri	eff
0	0.00000	0.0000000	10000.000	0.000	0.0000	1.000000
1	44.37555	0.9198298	7452.765	1783.897	273.1895	1.084664
2	54.88296	2.6729825	5554.370	2206.295	793.8758	1.180825
3	51.90343	4.4564927	4139.542	2086.518	1323.5783	1.228914
4	44.49738	5.9807076	3085.103	1788.795	1776.2702	1.234610
5	36.48434	7.1774981	2299.255	1466.670	2131.7169	1.214742

You can also solve this and create a RxODE data frame:

```
x <- mod1 %>% rxSolve(theta, ev, inits);
x
```

```
#> _____ Solved RxODE object _____
#> -- Parameters (x$params): ----------------------------------
#>       V2      V3      KA      CL       Q     Kin    Kout    EC50
#>   40.200 297.000   0.294  18.600  10.500   1.000   1.000 200.000
#> -- Initial Conditions (x$inits): ---------------------------
#> depot centr  peri    eff
#>     0     0     0      1
#> -- First part of data (object): ----------------------------
#> # A tibble: 241 x 7
#>     time    C2    C3  depot centr  peri    eff
#>      [h] <dbl> <dbl>  <dbl> <dbl> <dbl>  <dbl>
#> 1     0   0    0      10000     0     0  1
#> 2     1  44.4 0.920   7453. 1784.  273.  1.08
#> 3     2  54.9 2.67    5554. 2206.  794.  1.18
#> 4     3  51.9 4.46    4140. 2087. 1324.  1.23
#> 5     4  44.5 5.98    3085. 1789. 1776.  1.23
#> 6     5  36.5 7.18    2299. 1467. 2132.  1.21
#> # ... with 235 more rows
#> _____
```

This returns a modified data frame. You can see the compartment values in the plot below:

```
library(ggplot2)
plot(x,C2) + ylab("Central Concentration")
```

Or,
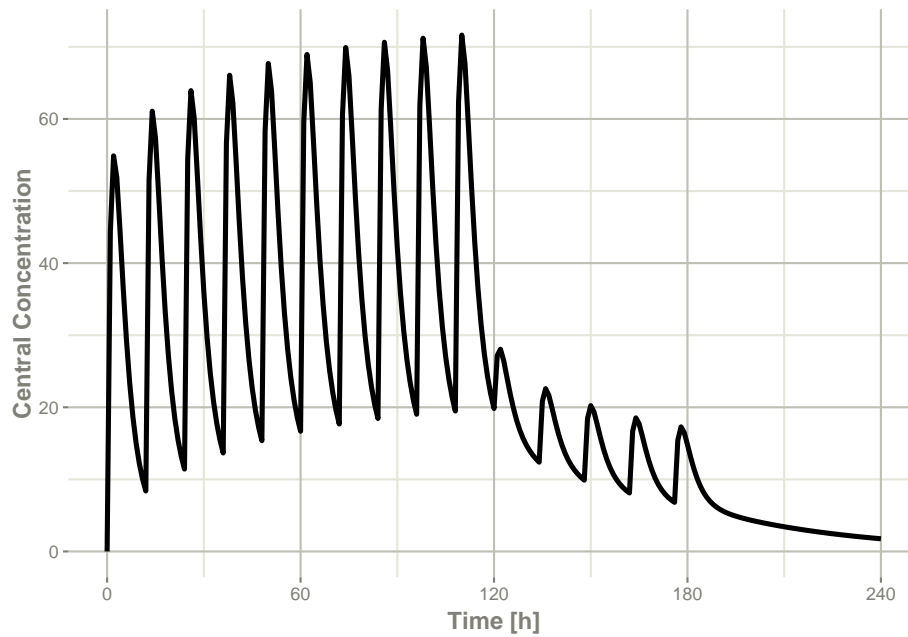
```r
plot(x,eff) + ylab("Effect")
```
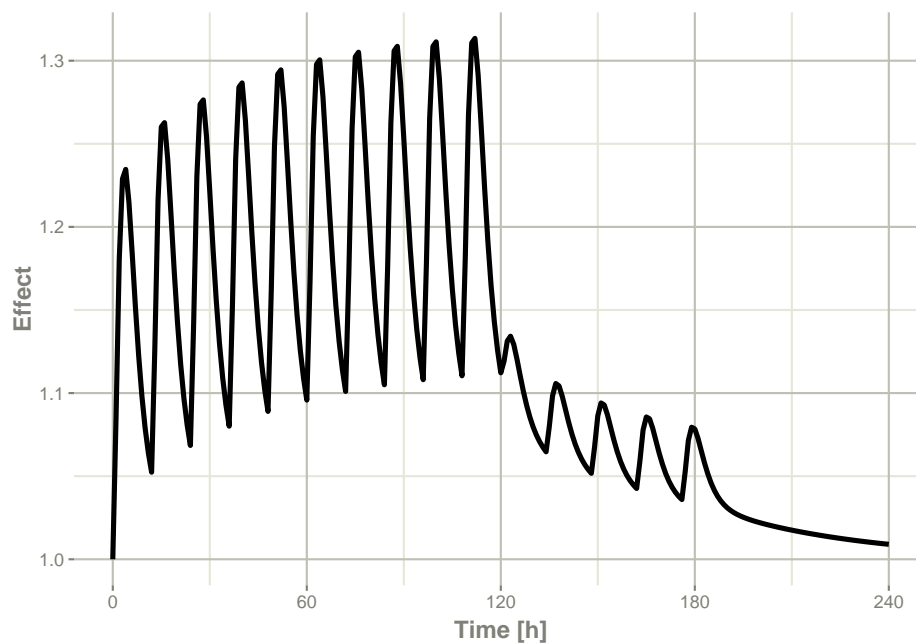

Note that the labels are automatically labeled with the units from the initial event

table. RxODE extracts `units` to label the plot (if they are present).

Chapter 6

RxODE syntax

This briefly describes the syntax used to define models that RxODE will translate into R-callable compiled code. It also describes the communication of variables between R and the RxODE modeling specification.

6.1 Example

```
# An RxODE model specification (this line is a comment).

if(comed==0){   # concomitant medication (con-med)?
   F = 1.0;     # full bioavailability w.o. con-med
}
else {
   F = 0.80;    # 20% reduced bioavailability
}

C2 = centr/V2;  # concentration in the central compartment
C3 = peri/V3;   # concentration in the peripheral compartment

# ODE describing the PK and PD

d/dt(depot) = -KA*depot;
d/dt(centr) = F*KA*depot - CL*C2 - Q*C2 + Q*C3;
d/dt(peri)  =                       Q*C2 - Q*C3;
d/dt(eff)   = Kin - Kout*(1-C2/(EC50+C2))*eff;
```

6.2   Syntax

An RxODE model specification consists of one or more statements optionally terminated by semi-colons ; and optional comments (comments are delimited by # and an end-of-line).

A block of statements is a set of statements delimited by curly braces, { ... }.

Statements can be either assignments, conditional if/else if/else, while loops (can be exited by break), special statements, or printing statements (for debugging/testing)

Assignment statements can be:

- **simple** assignments, where the left hand is an identifier (i.e., variable)

- special **time-derivative** assignments, where the left hand specifies the change of the amount in the corresponding state variable (compartment) with respect to time e.g., d/dt(depot):

- special **initial-condition** assignments where the left hand specifies the compartment of the initial condition being specified, e.g. depot(0) = 0

- special model event changes including **bioavailability** (f(depot)=1), **lag time** (alag(depot)=0), **modeled rate** (rate(depot)=2) and **modeled duration** (dur(depot)=2). An example of these model features and the event specification for the modeled infusions the RxODE data specification is found in RxODE events section.

- special **change point syntax, or model times**. These model times are specified by mtime(var)=time

- special **Jacobian-derivative** assignments, where the left hand specifies the change in the compartment ode with respect to a variable. For example, if d/dt(y) = dy, then a Jacobian for this compartment can be specified as df(y)/dy(dy) = 1. There may be some advantage to obtaining the solution or specifying the Jacobian for very stiff ODE systems. However, for the few stiff systems we tried with LSODA, this actually slightly slowed down the solving.

Note that assignment can be done by =, <- or ~.

When assigning with the ~ operator, the **simple assignments** and **time-derivative** assignments will not be output.

Special statements can be:

- **Compartment declaration statements**, which can change the default dosing compartment and the assumed compartment number(s) as well as add extra compartment names at the end (useful for multiple-endpoint nlmixr models); These are specified by cmt(compartmentName)

- **Parameter declaration statements**, which can make sure the input parameters are in a certain order instead of ordering the parameters by the order they are parsed. This is useful for keeping the parameter order the same when using 2 different ODE models. These are specified by `param(par1, par2,...)`

An example model is shown below:

```
# simple assignment
C2 = centr/V2;

# time-derivative assignment
d/dt(centr) = F*KA*depot - CL*C2 - Q*C2 + Q*C3;
```

Expressions in assignment and `if` statements can be numeric or logical, however, no character nor integer expressions are currently supported.

Numeric expressions can include the following numeric operators `+`, `-`, `*`, `/`, `^` and those mathematical functions defined in the C or the R math libraries (e.g., `fabs`, `exp`, `log`, `sin`, `abs`).

You may also access the R's functions in the R math libraries, like `lgammafn` for the log gamma function.

The `RxODE` syntax is case-sensitive, i.e., `ABC` is different than `abc`, `Abc`, `ABc`, etc.

6.2.1 Identifiers

Like R, Identifiers (variable names) may consist of one or more alphanumeric, underscore `_` or period `.` characters, but the first character cannot be a digit or underscore `_`.

Identifiers in a model specification can refer to:

- State variables in the dynamic system (e.g., compartments in a pharmacokinetics model).
- Implied input variable, `t` (time), `tlast` (last time point), and `podo` (oral dose, in the undocumented case of absorption transit models).
- Special constants like `pi` or R's predefined constants.
- Model parameters (e.g., `ka` rate of absorption, `CL` clearance, etc.)
- Others, as created by assignments as part of the model specification; these are referred as *LHS* (left-hand side) variable.

Currently, the `RxODE` modeling language only recognizes system state variables and "parameters", thus, any values that need to be passed from R to the ODE model (e.g., age) should be either passed in the `params` argument of the integrator function `rxSolve()` or be in the supplied event data-set.

There are certain variable names that are in the RxODE event tables. To avoid confusion, the following event table-related items cannot be assigned, or used as a state but can be accessed in the RxODE code:

- cmt
- dvid
- addl
- ss
- rate
- id

However the following variables are cannot be used in a model specification:

- evid
- ii

Sometimes RxODE generates variables that are fed back to RxODE. Similarly, nlmixr generates some variables that are used in nlmixr estimation and simulation. These variables start with the either the rx or nlmixr prefixes. To avoid any problems, it is suggested to not use these variables starting with either the rx or nlmixr prefixes.

6.3 Logical Operators

Logical operators support the standard R operators ==, != >= <= > and <. Like R these can be in if() or while() statements, ifelse() expressions. Additionally they can be in a standard assignment. For instance, the following is valid:

```
cov1 = covm*(sexf == "female") + covm*(sexf != "female")
```

Notice that you can also use character expressions in comparisons. This convenience comes at a cost since character comparisons are slower than numeric expressions. Unlike R, as.numeric or as.integer for these logical statements is not only not needed, but will cause an syntax error if you try to use the function.

6.4 cmt() changing compartment numbers for states

The compartment order can be changed with the cmt() syntax in the model. To understand what the cmt() can do you need to understand how RxODE numbers the compartments.

Below is an example of how RxODE numbers compartments

6.4.1 How RxODE numbers compartments

RxODE automatically assigns compartment numbers when parsing. For example, with the Mavoglurant PBPK model the following model may be used:

```
library(RxODE)
pbpk <- RxODE({
    KbBR = exp(lKbBR)
    KbMU = exp(lKbMU)
    KbAD = exp(lKbAD)
    CLint= exp(lCLint + eta.LClint)
    KbBO = exp(lKbBO)
    KbRB = exp(lKbRB)

    ## Regional blood flows
    # Cardiac output (L/h) from White et al (1968)
    CO  = (187.00*WT^0.81)*60/1000;
    QHT = 4.0 *CO/100;
    QBR = 12.0*CO/100;
    QMU = 17.0*CO/100;
    QAD = 5.0 *CO/100;
    QSK = 5.0 *CO/100;
    QSP = 3.0 *CO/100;
    QPA = 1.0 *CO/100;
    QLI = 25.5*CO/100;
    QST = 1.0 *CO/100;
    QGU = 14.0*CO/100;
    # Hepatic artery blood flow
    QHA = QLI - (QSP + QPA + QST + QGU);
    QBO = 5.0 *CO/100;
    QKI = 19.0*CO/100;
    QRB = CO - (QHT + QBR + QMU + QAD + QSK + QLI + QBO + QKI);
    QLU = QHT + QBR + QMU + QAD + QSK + QLI + QBO + QKI + QRB;

    ## Organs' volumes = organs' weights / organs' density
    VLU = (0.76 *WT/100)/1.051;
    VHT = (0.47 *WT/100)/1.030;
    VBR = (2.00 *WT/100)/1.036;
    VMU = (40.00*WT/100)/1.041;
    VAD = (21.42*WT/100)/0.916;
    VSK = (3.71 *WT/100)/1.116;
    VSP = (0.26 *WT/100)/1.054;
    VPA = (0.14 *WT/100)/1.045;
    VLI = (2.57 *WT/100)/1.040;
    VST = (0.21 *WT/100)/1.050;
```

```
VGU = (1.44 *WT/100)/1.043;
VBO = (14.29*WT/100)/1.990;
VKI = (0.44 *WT/100)/1.050;
VAB = (2.81 *WT/100)/1.040;
VVB = (5.62 *WT/100)/1.040;
VRB = (3.86 *WT/100)/1.040;

## Fixed parameters
BP = 0.61;        # Blood:plasma partition coefficient
fup = 0.028;      # Fraction unbound in plasma
fub = fup/BP;     # Fraction unbound in blood

KbLU = exp(0.8334);
KbHT = exp(1.1205);
KbSK = exp(-.5238);
KbSP = exp(0.3224);
KbPA = exp(0.3224);
KbLI = exp(1.7604);
KbST = exp(0.3224);
KbGU = exp(1.2026);
KbKI = exp(1.3171);

##----------------------------------------
S15 = VVB*BP/1000;
C15 = Venous_Blood/S15

##----------------------------------------
d/dt(Lungs) = QLU*(Venous_Blood/VVB - Lungs/KbLU/VLU);
d/dt(Heart) = QHT*(Arterial_Blood/VAB - Heart/KbHT/VHT);
d/dt(Brain) = QBR*(Arterial_Blood/VAB - Brain/KbBR/VBR);
d/dt(Muscles) = QMU*(Arterial_Blood/VAB - Muscles/KbMU/VMU);
d/dt(Adipose) = QAD*(Arterial_Blood/VAB - Adipose/KbAD/VAD);
d/dt(Skin) = QSK*(Arterial_Blood/VAB - Skin/KbSK/VSK);
d/dt(Spleen) = QSP*(Arterial_Blood/VAB - Spleen/KbSP/VSP);
d/dt(Pancreas) = QPA*(Arterial_Blood/VAB - Pancreas/KbPA/VPA);
d/dt(Liver) = QHA*Arterial_Blood/VAB + QSP*Spleen/KbSP/VSP +
  QPA*Pancreas/KbPA/VPA + QST*Stomach/KbST/VST +
  QGU*Gut/KbGU/VGU - CLint*fub*Liver/KbLI/VLI - QLI*Liver/KbLI/VLI;
d/dt(Stomach) = QST*(Arterial_Blood/VAB - Stomach/KbST/VST);
d/dt(Gut) = QGU*(Arterial_Blood/VAB - Gut/KbGU/VGU);
d/dt(Bones) = QBO*(Arterial_Blood/VAB - Bones/KbBO/VBO);
d/dt(Kidneys) = QKI*(Arterial_Blood/VAB - Kidneys/KbKI/VKI);
d/dt(Arterial_Blood) = QLU*(Lungs/KbLU/VLU - Arterial_Blood/VAB);
d/dt(Venous_Blood) = QHT*Heart/KbHT/VHT + QBR*Brain/KbBR/VBR +
  QMU*Muscles/KbMU/VMU + QAD*Adipose/KbAD/VAD + QSK*Skin/KbSK/VSK +
```

```
       QLI*Liver/KbLI/VLI + QBO*Bones/KbBO/VBO + QKI*Kidneys/KbKI/VKI +
       QRB*Rest_of_Body/KbRB/VRB - QLU*Venous_Blood/VVB;
    d/dt(Rest_of_Body) = QRB*(Arterial_Blood/VAB - Rest_of_Body/KbRB/VRB);
})
```

If you look at the summary, you can see where RxODE assigned the compartment
number(s)

```
summary(pbpk)
```

```
#> RxODE 1.0.5 model named rx_58b79e8d26bd79e8d2dfb854b8892b2a model (ready).
#> DLL: /home/matt/.cache/R/RxODE/rx_58b79e8d26bd79e8d2dfb854b8892b2a__.rxd/rx_58b79e8d26bd79e8d2
#> NULL
#>
#> Calculated Variables:
#>  [1] "KbBR"   "KbMU"   "KbAD"   "CLint"  "KbBO"   "KbRB"   "CO"     "QHT"    "QBR"
#> [10] "QMU"    "QAD"    "QSK"    "QSP"    "QPA"    "QLI"    "QST"    "QGU"    "QHA"
#> [19] "QBO"    "QKI"    "QRB"    "QLU"    "VLU"    "VHT"    "VBR"    "VMU"    "VAD"
#> [28] "VSK"    "VSP"    "VPA"    "VLI"    "VST"    "VGU"    "VBO"    "VKI"    "VAB"
#> [37] "VVB"    "VRB"    "fub"    "KbLU"   "KbHT"   "KbSK"   "KbSP"   "KbPA"   "KbLI"
#> [46] "KbST"   "KbGU"   "KbKI"   "S15"    "C15"
#> _____ RxODE Model Syntax _____
#> RxODE({
#>     KbBR = exp(lKbBR)
#>     KbMU = exp(lKbMU)
#>     KbAD = exp(lKbAD)
#>     CLint = exp(lCLint + eta.LClint)
#>     KbBO = exp(lKbBO)
#>     KbRB = exp(lKbRB)
#>     CO = (187 * WT^0.81) * 60/1000
#>     QHT = 4 * CO/100
#>     QBR = 12 * CO/100
#>     QMU = 17 * CO/100
#>     QAD = 5 * CO/100
#>     QSK = 5 * CO/100
#>     QSP = 3 * CO/100
#>     QPA = 1 * CO/100
#>     QLI = 25.5 * CO/100
#>     QST = 1 * CO/100
#>     QGU = 14 * CO/100
#>     QHA = QLI - (QSP + QPA + QST + QGU)
#>     QBO = 5 * CO/100
#>     QKI = 19 * CO/100
#>     QRB = CO - (QHT + QBR + QMU + QAD + QSK + QLI + QBO + QKI)
```

```
#>      QLU = QHT + QBR + QMU + QAD + QSK + QLI + QBO + QKI + QRB
#>      VLU = (0.76 * WT/100)/1.051
#>      VHT = (0.47 * WT/100)/1.03
#>      VBR = (2 * WT/100)/1.036
#>      VMU = (40 * WT/100)/1.041
#>      VAD = (21.42 * WT/100)/0.916
#>      VSK = (3.71 * WT/100)/1.116
#>      VSP = (0.26 * WT/100)/1.054
#>      VPA = (0.14 * WT/100)/1.045
#>      VLI = (2.57 * WT/100)/1.04
#>      VST = (0.21 * WT/100)/1.05
#>      VGU = (1.44 * WT/100)/1.043
#>      VBO = (14.29 * WT/100)/1.99
#>      VKI = (0.44 * WT/100)/1.05
#>      VAB = (2.81 * WT/100)/1.04
#>      VVB = (5.62 * WT/100)/1.04
#>      VRB = (3.86 * WT/100)/1.04
#>      BP = 0.61
#>      fup = 0.028
#>      fub = fup/BP
#>      KbLU = exp(0.8334)
#>      KbHT = exp(1.1205)
#>      KbSK = exp(-0.5238)
#>      KbSP = exp(0.3224)
#>      KbPA = exp(0.3224)
#>      KbLI = exp(1.7604)
#>      KbST = exp(0.3224)
#>      KbGU = exp(1.2026)
#>      KbKI = exp(1.3171)
#>      S15 = VVB * BP/1000
#>      C15 = Venous_Blood/S15
#>      d/dt(Lungs) = QLU * (Venous_Blood/VVB - Lungs/KbLU/VLU)
#>      d/dt(Heart) = QHT * (Arterial_Blood/VAB - Heart/KbHT/VHT)
#>      d/dt(Brain) = QBR * (Arterial_Blood/VAB - Brain/KbBR/VBR)
#>      d/dt(Muscles) = QMU * (Arterial_Blood/VAB - Muscles/KbMU/VMU)
#>      d/dt(Adipose) = QAD * (Arterial_Blood/VAB - Adipose/KbAD/VAD)
#>      d/dt(Skin) = QSK * (Arterial_Blood/VAB - Skin/KbSK/VSK)
#>      d/dt(Spleen) = QSP * (Arterial_Blood/VAB - Spleen/KbSP/VSP)
#>      d/dt(Pancreas) = QPA * (Arterial_Blood/VAB - Pancreas/KbPA/VPA)
#>      d/dt(Liver) = QHA * Arterial_Blood/VAB + QSP * Spleen/KbSP/VSP +
#>          QPA * Pancreas/KbPA/VPA + QST * Stomach/KbST/VST + QGU *
#>          Gut/KbGU/VGU - CLint * fub * Liver/KbLI/VLI - QLI * Liver/KbLI/VLI
#>      d/dt(Stomach) = QST * (Arterial_Blood/VAB - Stomach/KbST/VST)
#>      d/dt(Gut) = QGU * (Arterial_Blood/VAB - Gut/KbGU/VGU)
#>      d/dt(Bones) = QBO * (Arterial_Blood/VAB - Bones/KbBO/VBO)
#>      d/dt(Kidneys) = QKI * (Arterial_Blood/VAB - Kidneys/KbKI/VKI)
```

```
#>      d/dt(Arterial_Blood) = QLU * (Lungs/KbLU/VLU - Arterial_Blood/VAB)
#>      d/dt(Venous_Blood) = QHT * Heart/KbHT/VHT + QBR * Brain/KbBR/VBR +
#>          QMU * Muscles/KbMU/VMU + QAD * Adipose/KbAD/VAD + QSK *
#>          Skin/KbSK/VSK + QLI * Liver/KbLI/VLI + QBO * Bones/KbBO/VBO +
#>          QKI * Kidneys/KbKI/VKI + QRB * Rest_of_Body/KbRB/VRB -
#>          QLU * Venous_Blood/VVB
#>      d/dt(Rest_of_Body) = QRB * (Arterial_Blood/VAB - Rest_of_Body/KbRB/VRB)
#> })
#> _____
```

In this case, `Venous_Blood` is assigned to compartment 15. Figuring this out can be inconvenient and also lead to re-numbering compartment in simulation or estimation datasets. While it is easy and probably clearer to specify the compartment by name, other tools only support compartment numbers. Therefore, having a way to number compartment easily can lead to less data modification between multiple tools.

6.4.2   Changing compartments by pre-declaring with `cmt()`

To add the compartments to the RxODE model in the order you desire you simply need to pre-declare the compartments with `cmt`. For example specifying is `Venous_Blood` and `Skin` to be the 1st and 2nd compartments, respectively, is simple:

```
pbpk2 <- RxODE({
  ## Now this is the first compartment, ie cmt=1
  cmt(Venous_Blood)
  ## Skin may be a compartment you wish to dose to as well,
  ##  so it is now cmt=2
  cmt(Skin)
  KbBR = exp(lKbBR)
  KbMU = exp(lKbMU)
  KbAD = exp(lKbAD)
  CLint= exp(lCLint + eta.LClint)
  KbBO = exp(lKbBO)
  KbRB = exp(lKbRB)

  ## Regional blood flows
  # Cardiac output (L/h) from White et al (1968)m
  CO  = (187.00*WT^0.81)*60/1000;
  QHT = 4.0 *CO/100;
  QBR = 12.0*CO/100;
  QMU = 17.0*CO/100;
  QAD = 5.0 *CO/100;
```

```
QSK = 5.0 *CO/100;
QSP = 3.0 *CO/100;
QPA = 1.0 *CO/100;
QLI = 25.5*CO/100;
QST = 1.0 *CO/100;
QGU = 14.0*CO/100;
QHA = QLI - (QSP + QPA + QST + QGU); # Hepatic artery blood flow
QBO = 5.0 *CO/100;
QKI = 19.0*CO/100;
QRB = CO - (QHT + QBR + QMU + QAD + QSK + QLI + QBO + QKI);
QLU = QHT + QBR + QMU + QAD + QSK + QLI + QBO + QKI + QRB;

## Organs' volumes = organs' weights / organs' density
VLU = (0.76 *WT/100)/1.051;
VHT = (0.47 *WT/100)/1.030;
VBR = (2.00 *WT/100)/1.036;
VMU = (40.00*WT/100)/1.041;
VAD = (21.42*WT/100)/0.916;
VSK = (3.71 *WT/100)/1.116;
VSP = (0.26 *WT/100)/1.054;
VPA = (0.14 *WT/100)/1.045;
VLI = (2.57 *WT/100)/1.040;
VST = (0.21 *WT/100)/1.050;
VGU = (1.44 *WT/100)/1.043;
VBO = (14.29*WT/100)/1.990;
VKI = (0.44 *WT/100)/1.050;
VAB = (2.81 *WT/100)/1.040;
VVB = (5.62 *WT/100)/1.040;
VRB = (3.86 *WT/100)/1.040;

## Fixed parameters
BP = 0.61;      # Blood:plasma partition coefficient
fup = 0.028;    # Fraction unbound in plasma
fub = fup/BP;   # Fraction unbound in blood

KbLU = exp(0.8334);
KbHT = exp(1.1205);
KbSK = exp(-.5238);
KbSP = exp(0.3224);
KbPA = exp(0.3224);
KbLI = exp(1.7604);
KbST = exp(0.3224);
KbGU = exp(1.2026);
KbKI = exp(1.3171);
```

```
  ##----------------------------------------
  S15 = VVB*BP/1000;
  C15 = Venous_Blood/S15

  ##----------------------------------------
  d/dt(Lungs) = QLU*(Venous_Blood/VVB - Lungs/KbLU/VLU);
  d/dt(Heart) = QHT*(Arterial_Blood/VAB - Heart/KbHT/VHT);
  d/dt(Brain) = QBR*(Arterial_Blood/VAB - Brain/KbBR/VBR);
  d/dt(Muscles) = QMU*(Arterial_Blood/VAB - Muscles/KbMU/VMU);
  d/dt(Adipose) = QAD*(Arterial_Blood/VAB - Adipose/KbAD/VAD);
  d/dt(Skin) = QSK*(Arterial_Blood/VAB - Skin/KbSK/VSK);
  d/dt(Spleen) = QSP*(Arterial_Blood/VAB - Spleen/KbSP/VSP);
  d/dt(Pancreas) = QPA*(Arterial_Blood/VAB - Pancreas/KbPA/VPA);
  d/dt(Liver) = QHA*Arterial_Blood/VAB + QSP*Spleen/KbSP/VSP +
    QPA*Pancreas/KbPA/VPA + QST*Stomach/KbST/VST + QGU*Gut/KbGU/VGU -
    CLint*fub*Liver/KbLI/VLI - QLI*Liver/KbLI/VLI;
  d/dt(Stomach) = QST*(Arterial_Blood/VAB - Stomach/KbST/VST);
  d/dt(Gut) = QGU*(Arterial_Blood/VAB - Gut/KbGU/VGU);
  d/dt(Bones) = QBO*(Arterial_Blood/VAB - Bones/KbBO/VBO);
  d/dt(Kidneys) = QKI*(Arterial_Blood/VAB - Kidneys/KbKI/VKI);
  d/dt(Arterial_Blood) = QLU*(Lungs/KbLU/VLU - Arterial_Blood/VAB);
  d/dt(Venous_Blood) = QHT*Heart/KbHT/VHT + QBR*Brain/KbBR/VBR +
    QMU*Muscles/KbMU/VMU + QAD*Adipose/KbAD/VAD + QSK*Skin/KbSK/VSK +
    QLI*Liver/KbLI/VLI + QBO*Bones/KbBO/VBO + QKI*Kidneys/KbKI/VKI +
    QRB*Rest_of_Body/KbRB/VRB - QLU*Venous_Blood/VVB;
  d/dt(Rest_of_Body) = QRB*(Arterial_Blood/VAB - Rest_of_Body/KbRB/VRB);
})
```

You can see this change in the simple printout

```
pbpk2
```

```
#> RxODE 1.0.5 model named rx_2dca6ec86251c10d68a5c48169890253 model (ready).
#> x$state: Venous_Blood, Skin, Lungs, Heart, Brain, Muscles, Adipose, Spleen, Pancreas, Liver, S
#> x$params: lKbBR, lKbMU, lKbAD, lCLint, eta.LClint, lKbBO, lKbRB, WT, BP, fup
#> x$lhs: KbBR, KbMU, KbAD, CLint, KbBO, KbRB, CO, QHT, QBR, QMU, QAD, QSK, QSP, QPA, QLI, QST, Q
```

The first two compartments are Venous_Blood followed by Skin.

6.4.3   Appending compartments to the model with `cmt()`

You can also append "compartments" to the model. Because of the ODE solving internals, you cannot add fake compartments to the model until after all the differential equations are defined.

For example this is legal:

```
ode.1c.ka <- RxODE({
    C2 = center/V;
    d / dt(depot) = -KA * depot
    d/dt(center) = KA * depot - CL*C2
    cmt(eff);
})
print(ode.1c.ka)
```

```
#> RxODE 1.0.5 model named rx_30a6f29377900e11d0549d668255825f model (ready).
#> $state: depot, center
#> $stateExtra: eff
#> $params: V, KA, CL
#> $lhs: C2
```

But compartments defined before all the differential equations is not supported;
So the model below:

```
ode.1c.ka <- RxODE({
    cmt(eff);
    C2 = center/V;
    d / dt(depot) = -KA * depot
    d/dt(center) = KA * depot - CL*C2
})
```

will give an error:

```
Error in rxModelVars_(obj) :
  Evaluation error: Compartment 'eff' needs differential equations defined.
```

Chapter 7

RxODE events

7.1 RxODE event tables

In general, RxODE event tables follow NONMEM dataset convention with the exceptions:

- The compartment data item (`cmt`) can be a string/factor with compartment names
    - You may turn off a compartment with a negative compartment number or "-cmt" where cmt is the compartment name.
    - The compartment data item (`cmt`) can still be a number, the number of the compartment is defined by the appearance of the compartment name in the model. This can be tedious to count, so you can specify compartment numbers easier by using the `cmt(cmtName)` at the beginning of the model.

- An additional column, `dur` can specify the duration of infusions;
    - Bioavailability changes will change the rate of infusion since `dur/amt` are fixed in the input data.
    - Similarly, when specifying `rate/amt` for an infusion, the bioavailability will change the infusion duration since `rate/amt` are fixed in the input data.

- Some infrequent NONMEM columns are not supported: `pcmt`, `call`.
- Additional events are supported:
    - `evid=5` or replace event; This replaces the value of a compartment with the value specified in the `amt` column. This is equivalent to `deSolve=replace`.

- – `evid=6` or multiply event; This multiplies the value in the compartment with the value specified by the `amt` column. This is equivalent to `deSolve=multiply`.

Here are the legal entries to a data table:

Data Item	Meaning	Notes
id	Individual identifier	Can be a integer, factor, character, or numeric
time	Individual time	Numeric for each time.
amt	dose amount	Positive for doses zero/NA for observations
rate	infusion rate	When specified the infusion duration will be dur=amt/rate
		rate = -1, rate modeled; rate = -2, duration modeled
dur	infusion duration	When specified the infusion rate will be rate = amt/dur
evid	event ID	0=Observation; 1=Dose; 2=Other; 3=Reset; 4=Reset+Dose; 5=Replace; 6=Multiply
cmt	Compartment	Represents compartment #/name for dose/observation
ss	Steady State Flag	0 = non-steady-state; 1=steady state; 2=steady state +prior states
ii	Inter-dose Interval	Time between doses.
addl	# of additional doses	Number of doses like the current dose.

Other notes:

- The `evid` can be the classic RxODE (described here) or the `NONMEM`-style `evid` described above.
- `NONMEM`'s `DV` is not required; `RxODE` is a ODE solving framework.
- `NONMEM`'s `MDV` is not required, since it is captured in `EVID`.
- Instead of `NONMEM`-compatible data, it can accept `deSolve` compatible data-frames.

When returning the RxODE solved data-set there are a few additional event ids (`EVID`) that you may see depending on the solving options:

- `EVID = -1` is when a modeled rate ends (corresponds to `rate = -1`)
- `EVID = -2` is when a modeled duration ends (corresponds to `rate=-2`)
- `EVID = -10` when a rate specified zero-order infusion ends (corresponds to `rate > 0`)

- EVID = -20 when a duration specified zero-order infusion ends (corresponds to dur > 0)
- EVID = 101, 102, 103,... These correspond to the 1, 2, 3, ... modeled time (mtime).

These can only be accessed when solving with the option combination addDosing=TRUE and subsetNonmem=FALSE. If you want to see the classic EVID equivalents you can use addDosing=NA.

To illustrate the event types we will use the model from the original RxODE tutorial.

```r
library(RxODE)
### Model from RxODE tutorial
m1 <-RxODE({
    KA=2.94E-01;
    CL=1.86E+01;
    V2=4.02E+01;
    Q=1.05E+01;
    V3=2.97E+02;
    Kin=1;
    Kout=1;
    EC50=200;
    ## Added modeled bioavaiblity, duration and rate
    fdepot = 1;
    durDepot = 8;
    rateDepot = 1250;
    C2 = centr/V2;
    C3 = peri/V3;
    d/dt(depot) =-KA*depot;
    f(depot) = fdepot
    dur(depot) = durDepot
    rate(depot) = rateDepot
    d/dt(centr) = KA*depot - CL*C2 - Q*C2 + Q*C3;
    d/dt(peri)  =                    Q*C2 - Q*C3;
    d/dt(eff)  = Kin - Kout*(1-C2/(EC50+C2))*eff;
    eff(0) = 1
});
```

7.2 Bolus/Additive Doses

A bolus dose is the default type of dose in RxODE and only requires the amt/dose. Note that this uses the convenience function et() described in the RxODE event tables

```r
ev <- et(timeUnits="hr") %>%
    et(amt=10000, ii=12,until=24) %>%
    et(seq(0, 24, length.out=100))

ev
```

```
#> ---------------- EventTable with 101 records ---------------
#>
#>    1 dosing records (see x$get.dosing(); add with add.dosing or et)
#>    100 observation times (see x$get.sampling(); add with add.sampling or et)
#>    multiple doses in `addl` columns, expand with x$expand(); or etExpand(x)
#> -- First part of x: --------------------------------------
#> # A tibble: 101 x 5
#>         time    amt    ii  addl evid
#>          [h] <dbl>   [h] <int> <evid>
#>  1 0.0000000    NA    NA    NA 0:Observation
#>  2 0.0000000 10000    12     2 1:Dose (Add)
#>  3 0.2424242    NA    NA    NA 0:Observation
#>  4 0.4848485    NA    NA    NA 0:Observation
#>  5 0.7272727    NA    NA    NA 0:Observation
#>  6 0.9696970    NA    NA    NA 0:Observation
#>  7 1.2121212    NA    NA    NA 0:Observation
#>  8 1.4545455    NA    NA    NA 0:Observation
#>  9 1.6969697    NA    NA    NA 0:Observation
#> 10 1.9393939    NA    NA    NA 0:Observation
#> # ... with 91 more rows
```

```r
rxSolve(m1, ev) %>% plot(C2) +
    xlab("Time")
```
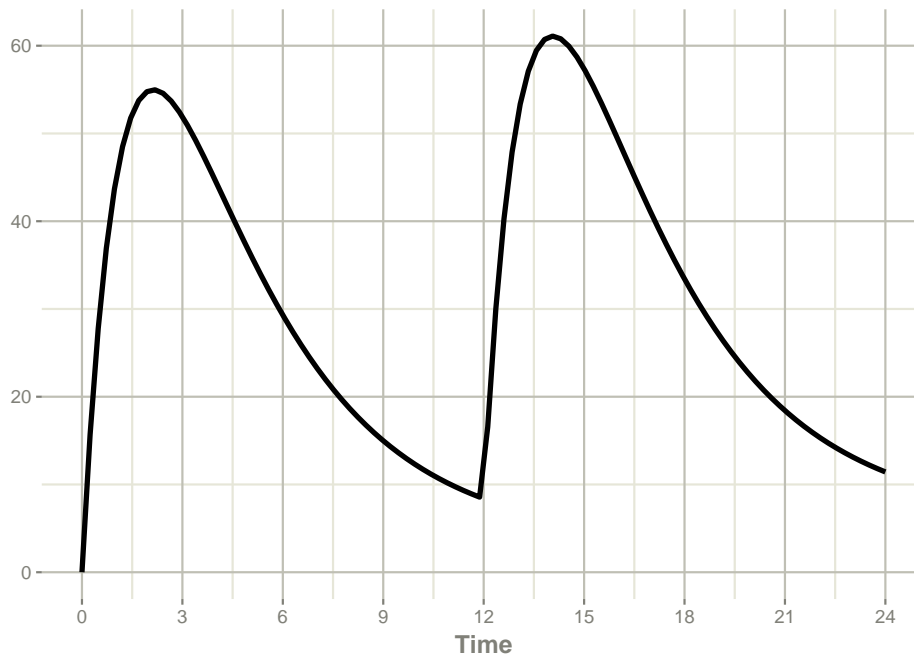
7.3 Infusion Doses

There are a few different type of infusions that RxODE supports:

- Constant Rate Infusion (`rate`)
- Constant Duration Infusion (`dur`)
- Estimated Rate of Infusion
- Estimated Duration of Infusion

7.3.1 Constant Infusion (in terms of duration and rate)

The next type of event is an infusion; There are two ways to specify an infusion; The first is the `dur` keyword.

An example of this is:

```
ev <- et(timeUnits="hr") %>%
    et(amt=10000, ii=12,until=24, dur=8) %>%
    et(seq(0, 24, length.out=100))

ev
```

```
#> ---------------- EventTable with 101 records ---------------
#>
#>    1 dosing records (see x$get.dosing(); add with add.dosing or et)
#>    100 observation times (see x$get.sampling(); add with add.sampling or et)
#>    multiple doses in `addl` columns, expand with x$expand(); or etExpand(x)
#> -- First part of x: -------------------------------------
#> # A tibble: 101 x 6
#>          time    amt    ii  addl evid            dur
#>          [h] <dbl>   [h] <int> <evid>            [h]
#>  1 0.0000000     NA    NA    NA 0:Observation    NA
#>  2 0.0000000  10000    12     2 1:Dose (Add)      8
#>  3 0.2424242     NA    NA    NA 0:Observation    NA
#>  4 0.4848485     NA    NA    NA 0:Observation    NA
#>  5 0.7272727     NA    NA    NA 0:Observation    NA
#>  6 0.9696970     NA    NA    NA 0:Observation    NA
#>  7 1.2121212     NA    NA    NA 0:Observation    NA
#>  8 1.4545455     NA    NA    NA 0:Observation    NA
#>  9 1.6969697     NA    NA    NA 0:Observation    NA
#> 10 1.9393939     NA    NA    NA 0:Observation    NA
#> # ... with 91 more rows
```

```r
rxSolve(m1, ev) %>% plot(depot, C2) +
    xlab("Time")
```
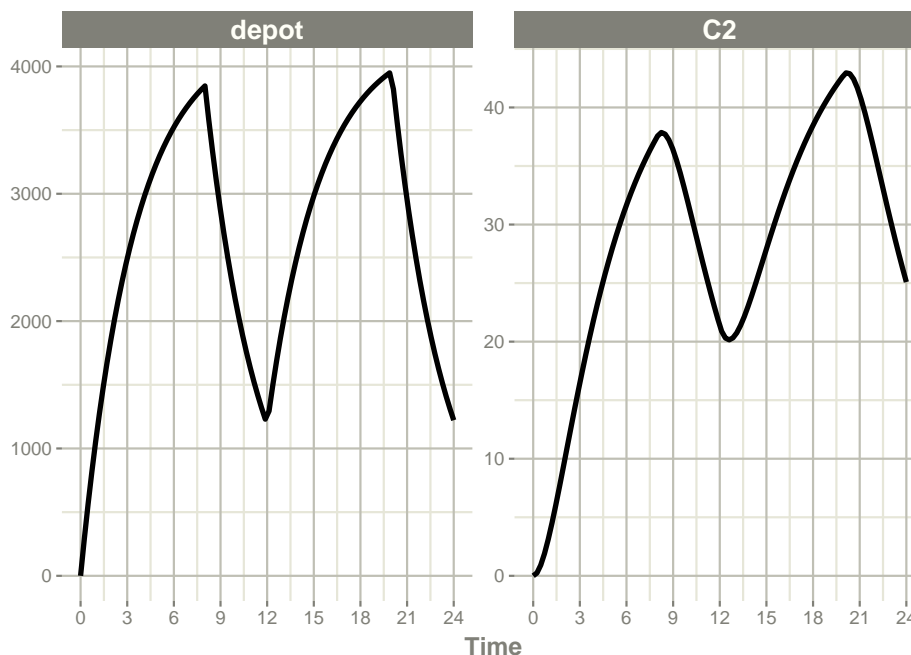

It can be also specified by the rate component:

```
ev <- et(timeUnits="hr") %>%
    et(amt=10000, ii=12,until=24, rate=10000/8) %>%
    et(seq(0, 24, length.out=100))


ev
```

```
#> ---------------- EventTable with 101 records ---------------
#>
#>     1 dosing records (see x$get.dosing(); add with add.dosing or et)
#>     100 observation times (see x$get.sampling(); add with add.sampling or et)
#>     multiple doses in `addl` columns, expand with x$expand(); or etExpand(x)
#> -- First part of x: -------------------------------------
#> # A tibble: 101 x 6
#>           time    amt rate                ii  addl evid
#>            [h] <dbl> <rate/dur>          [h] <int> <evid>
#>  1 0.0000000     NA NA                    NA    NA 0:Observation
#>  2 0.0000000 10000   1250                 12     2 1:Dose (Add)
#>  3 0.2424242     NA NA                    NA    NA 0:Observation
#>  4 0.4848485     NA NA                    NA    NA 0:Observation
#>  5 0.7272727     NA NA                    NA    NA 0:Observation
#>  6 0.9696970     NA NA                    NA    NA 0:Observation
#>  7 1.2121212     NA NA                    NA    NA 0:Observation
#>  8 1.4545455     NA NA                    NA    NA 0:Observation
#>  9 1.6969697     NA NA                    NA    NA 0:Observation
#> 10 1.9393939     NA NA                    NA    NA 0:Observation
#> # ... with 91 more rows
```

```
rxSolve(m1, ev) %>% plot(depot, C2) +
    xlab("Time")
```
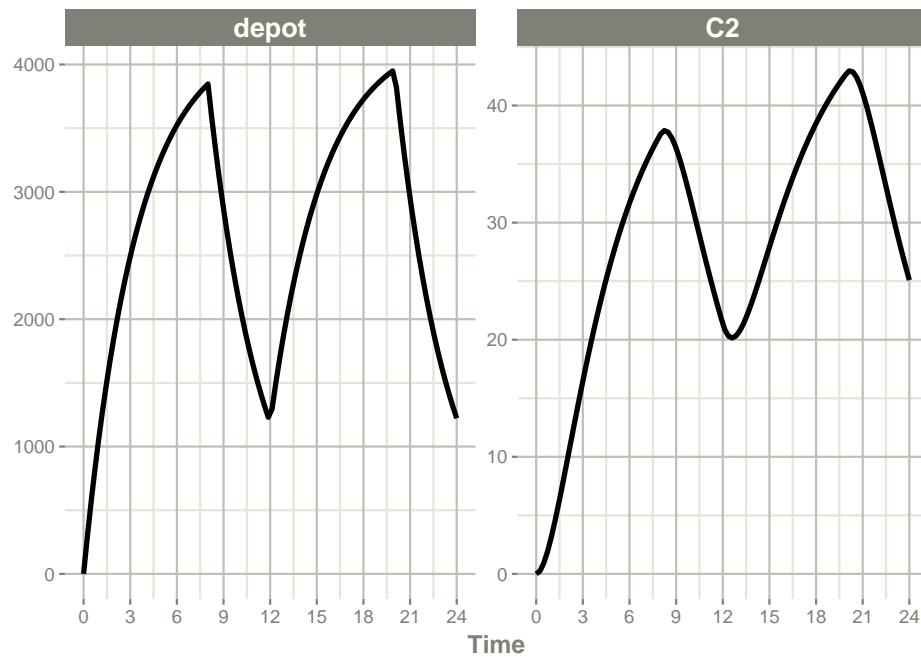
These are the same with the exception of how bioavailability changes the infusion.

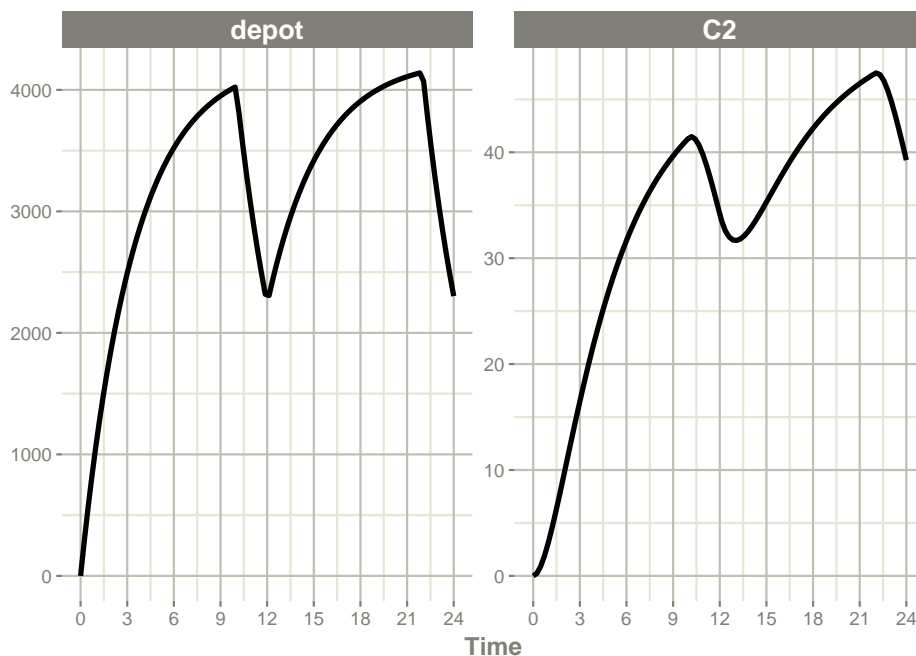In the case of modeling `rate`, a bioavailability decrease, decreases the infusion duration, as in NONMEM. For example:

```
rxSolve(m1, ev, c(fdepot=0.25)) %>% plot(depot, C2) +
    xlab("Time")
```
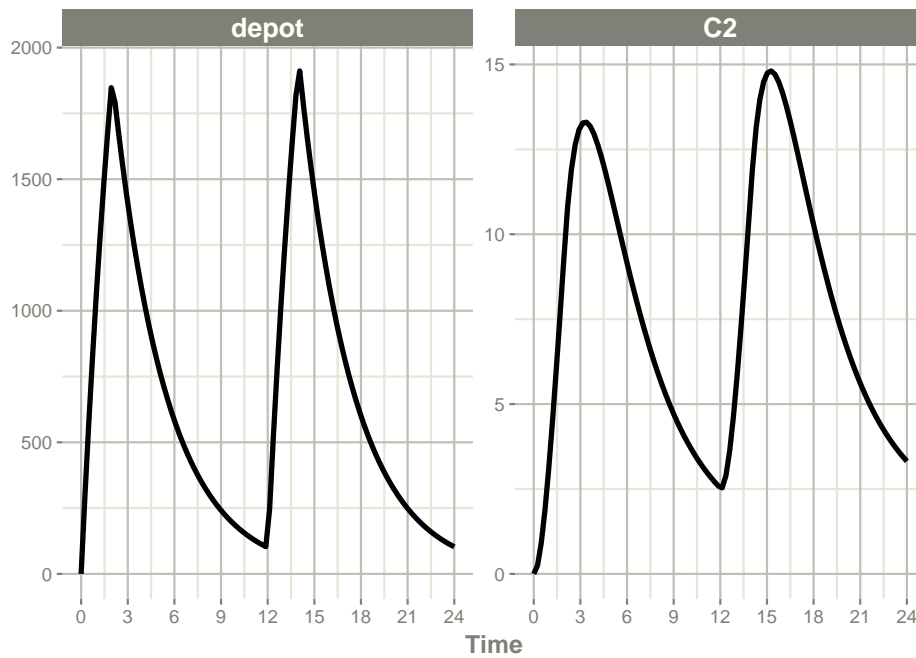
Similarly increasing the bioavailability increases the infusion duration.

```r
rxSolve(m1, ev, c(fdepot=1.25)) %>% plot(depot, C2) +
    xlab("Time")
```

The rationale for this behavior is that the `rate` and `amt` are specified by the event table, so the only thing that can change with a bioavailability increase is the duration of the infusion.

If you specify the `amt` and `dur` components in the event table, bioavailability changes affect the `rate` of infusion.

```
ev <- et(timeUnits="hr") %>%
    et(amt=10000, ii=12,until=24, dur=8) %>%
    et(seq(0, 24, length.out=100))
```

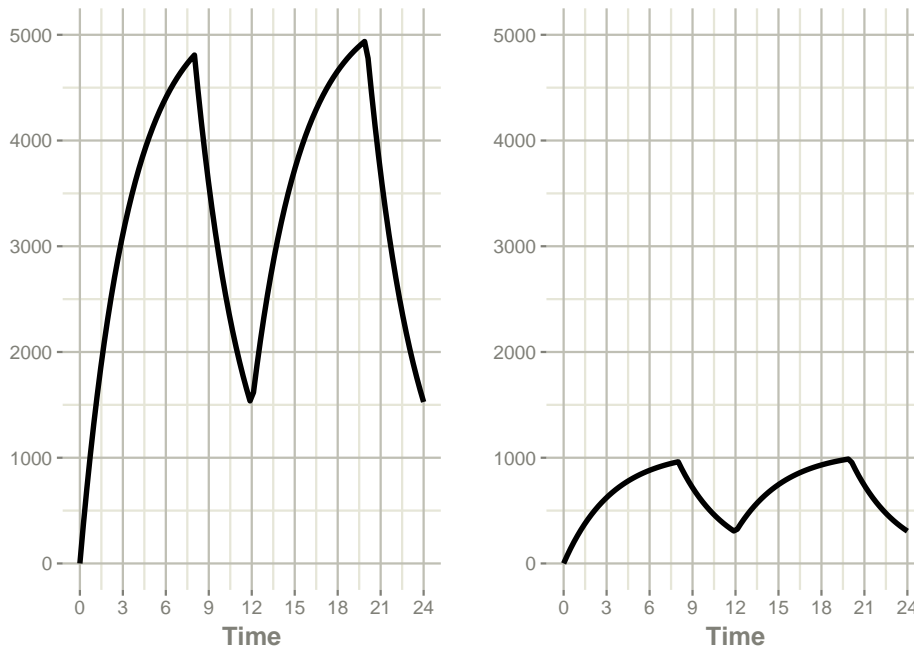You can see the side-by-side comparison of bioavailability changes affecting `rate` instead of duration with these records in the following plots:

```
library(ggplot2)
library(patchwork)

p1 <- rxSolve(m1, ev, c(fdepot=1.25)) %>% plot(depot) +
    xlab("Time") + ylim(0,5000)

p2 <- rxSolve(m1, ev, c(fdepot=0.25)) %>% plot(depot) +
    xlab("Time")+ ylim(0,5000)

### Use patchwork syntax to combine plots
p1 * p2
```

7.3.2   Modeled Rate and Duration of Infusion

You can model the duration, which is equivalent to NONMEM's `rate=-2`. As a mnemonic you can use the `dur=model` instead of `rate=-2`

```
ev <- et(timeUnits="hr") %>%
    et(amt=10000, ii=12,until=24, dur=model) %>%
    et(seq(0, 24, length.out=100))

ev
```

```
#> ---------------- EventTable with 101 records ---------------
#>
#>    1 dosing records (see x$get.dosing(); add with add.dosing or et)
#>    100 observation times (see x$get.sampling(); add with add.sampling or et)
#>    multiple doses in `addl` columns, expand with x$expand(); or etExpand(x)
#> -- First part of x: ------------------------------------
#> # A tibble: 101 x 6
#>       time    amt rate            ii  addl evid
#>        [h] <dbl> <rate/dur>      [h] <int> <evid>
#>  1 0.0000000    NA NA            NA    NA 0:Observation
#>  2 0.0000000 10000 -2:dur        12     2 1:Dose (Add)
#>  3 0.2424242    NA NA            NA    NA 0:Observation
```

```
#>  4 0.4848485    NA NA           NA     NA 0:Observation
#>  5 0.7272727    NA NA           NA     NA 0:Observation
#>  6 0.9696970    NA NA           NA     NA 0:Observation
#>  7 1.2121212    NA NA           NA     NA 0:Observation
#>  8 1.4545455    NA NA           NA     NA 0:Observation
#>  9 1.6969697    NA NA           NA     NA 0:Observation
#> 10 1.9393939    NA NA           NA     NA 0:Observation
#> # ... with 91 more rows
```

```r
rxSolve(m1, ev, c(durDepot=7)) %>% plot(depot, C2) +
    xlab("Time")
```


Similarly, you may also model rate. This is equivalent to NONMEM's `rate=-1` and is how RxODE's event table specifies the data item as well. You can also use `rate=model` as a mnemonic:
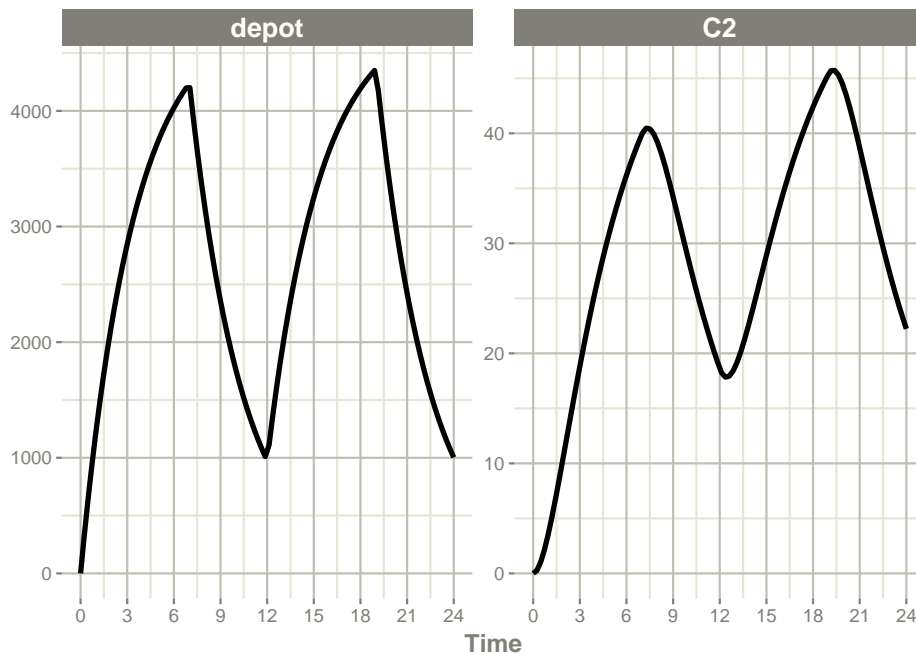
```r
ev <- et(timeUnits="hr") %>%
    et(amt=10000, ii=12,until=24, rate=model) %>%
    et(seq(0, 24, length.out=100))

ev
```

```
#> ---------------- EventTable with 101 records ---------------
#>
```

```
#>    1 dosing records (see x$get.dosing(); add with add.dosing or et)
#>    100 observation times (see x$get.sampling(); add with add.sampling or et)
#>    multiple doses in `addl` columns, expand with x$expand(); or etExpand(x)
#> -- First part of x: --------------------------------------
#> # A tibble: 101 x 6
#>        time    amt rate            ii  addl evid
#>         [h] <dbl> <rate/dur>      [h] <int> <evid>
#>  1 0.0000000    NA NA             NA    NA 0:Observation
#>  2 0.0000000 10000 -1:rate        12     2 1:Dose (Add)
#>  3 0.2424242    NA NA             NA    NA 0:Observation
#>  4 0.4848485    NA NA             NA    NA 0:Observation
#>  5 0.7272727    NA NA             NA    NA 0:Observation
#>  6 0.9696970    NA NA             NA    NA 0:Observation
#>  7 1.2121212    NA NA             NA    NA 0:Observation
#>  8 1.4545455    NA NA             NA    NA 0:Observation
#>  9 1.6969697    NA NA             NA    NA 0:Observation
#> 10 1.9393939    NA NA             NA    NA 0:Observation
#> # ... with 91 more rows
```

```
rxSolve(m1, ev, c(rateDepot=10000/3)) %>% plot(depot, C2) +
    xlab("Time")
```
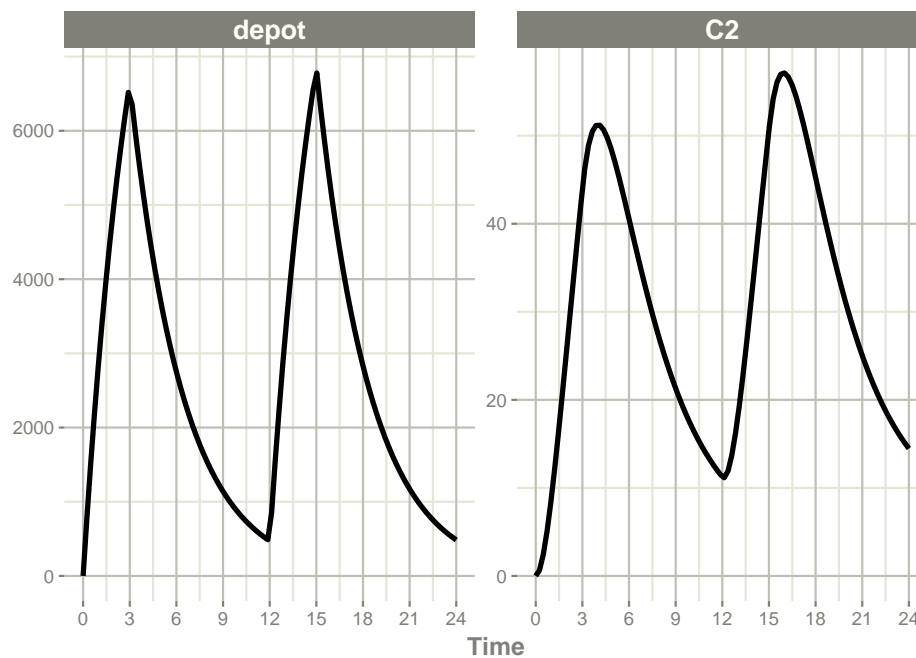
7.4   Steady State

These doses are solved until a steady state is reached with a constant inter-dose interval.

```
ev <- et(timeUnits="hr") %>%
    et(amt=10000, ii=12, ss=1) %>%
    et(seq(0, 24, length.out=100))

ev
```

```
#> ---------------- EventTable with 101 records ---------------
#>
#>     1 dosing records (see x$get.dosing(); add with add.dosing or et)
#>     100 observation times (see x$get.sampling(); add with add.sampling or et)
#> -- First part of x: -------------------------------------
#> # A tibble: 101 x 5
#>          time    amt    ii evid                ss
#>           [h] <dbl>   [h] <evid>          <int>
#>  1 0.0000000    NA    NA 0:Observation     NA
#>  2 0.0000000 10000    12 1:Dose (Add)       1
#>  3 0.2424242    NA    NA 0:Observation     NA
#>  4 0.4848485    NA    NA 0:Observation     NA
#>  5 0.7272727    NA    NA 0:Observation     NA
#>  6 0.9696970    NA    NA 0:Observation     NA
#>  7 1.2121212    NA    NA 0:Observation     NA
#>  8 1.4545455    NA    NA 0:Observation     NA
#>  9 1.6969697    NA    NA 0:Observation     NA
#> 10 1.9393939    NA    NA 0:Observation     NA
#> # ... with 91 more rows
```

```
rxSolve(m1, ev) %>% plot(C2)
```
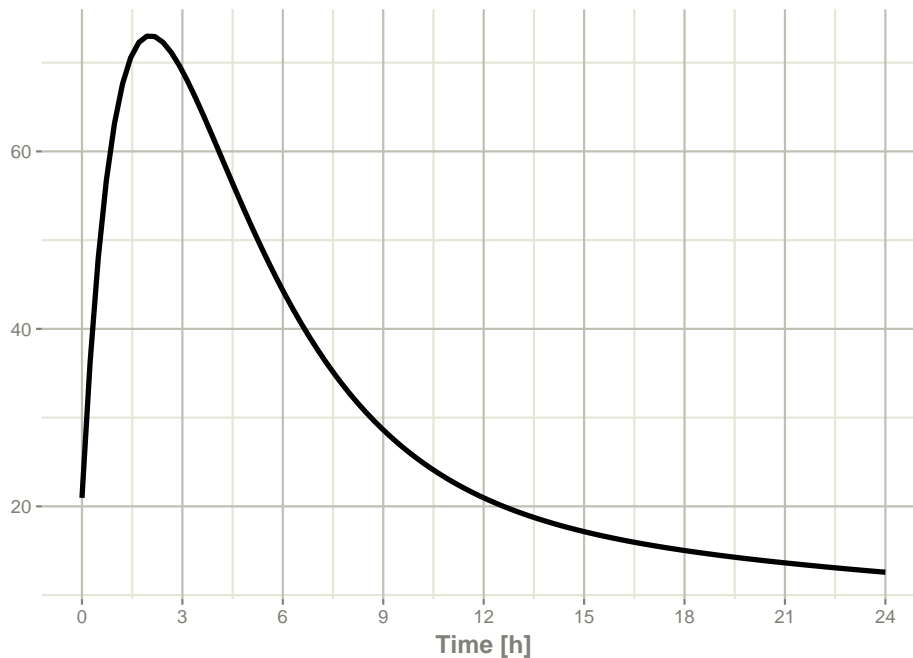
7.4.1 Steady state for complex dosing

By using the `ss=2` flag, you can use the super-positioning principle in linear kinetics to get steady state nonstandard dosing (i.e. morning 100 mg vs evening 150 mg). This is done by:

- Saving all the state values
- Resetting all the states and solving the system to steady state
- Adding back all the prior state values

```
ev <- et(timeUnits="hr") %>%
    et(amt=10000, ii=24, ss=1) %>%
    et(time=12, amt=15000, ii=24, ss=2) %>%
    et(time=24, amt=10000, ii=24, addl=3) %>%
    et(time=36, amt=15000, ii=24, addl=3) %>%
    et(seq(0, 64, length.out=500))

library(ggplot2)

rxSolve(m1, ev,maxsteps=10000) %>% plot(C2) +
  annotate("rect", xmin=0, xmax=24, ymin=-Inf, ymax=Inf,
           alpha=0.2) +
```

```
annotate("text", x=12.5, y=7,
         label="Initial Steady State Period") +
annotate("text", x=44,   y=7,
         label="Steady State AM/PM dosing")
```
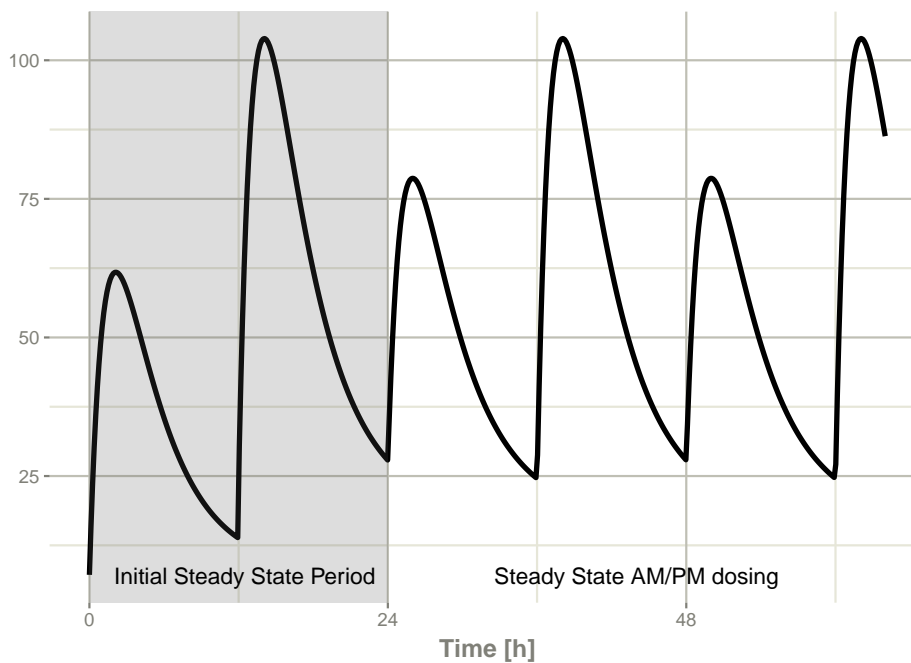

You can see that it takes a full dose cycle to reach the true complex steady state dosing.

7.4.2   Steady state for constant infusion or zero order processes

The last type of steady state that RxODE supports is steady-state constant infusion rate. This can be specified the same way as NONMEM, that is:

- No inter-dose interval `ii=0`
- A steady state dose, ie `ss=1`
- Either a positive rate (`rate>0`) or a estimated rate `rate=-1`.
- A zero dose, ie `amt=0`
- Once the steady-state constant infusion is achieved, the infusion is turned off when using this record, just like NONMEM.

Note that `rate=-2` where we model the duration of infusion doesn't make much sense since we are solving the infusion until steady state. The duration is specified by the steady state solution.

Also note that bioavailability changes on this steady state infusion also do not make sense because they neither change the `rate` or the duration of the steady state infusion. Hence modeled bioavailability on this type of dosing event is ignored.

Here is an example:

```
ev <- et(timeUnits="hr") %>%
    et(amt=0, ss=1,rate=10000/8)

p1 <- rxSolve(m1, ev) %>% plot(C2, eff)


ev <- et(timeUnits="hr") %>%
    et(amt=200000, rate=10000/8) %>%
    et(0, 250, length.out=1000)

p2 <- rxSolve(m1, ev) %>% plot(C2, eff)

library(patchwork)

p1 / p2
```
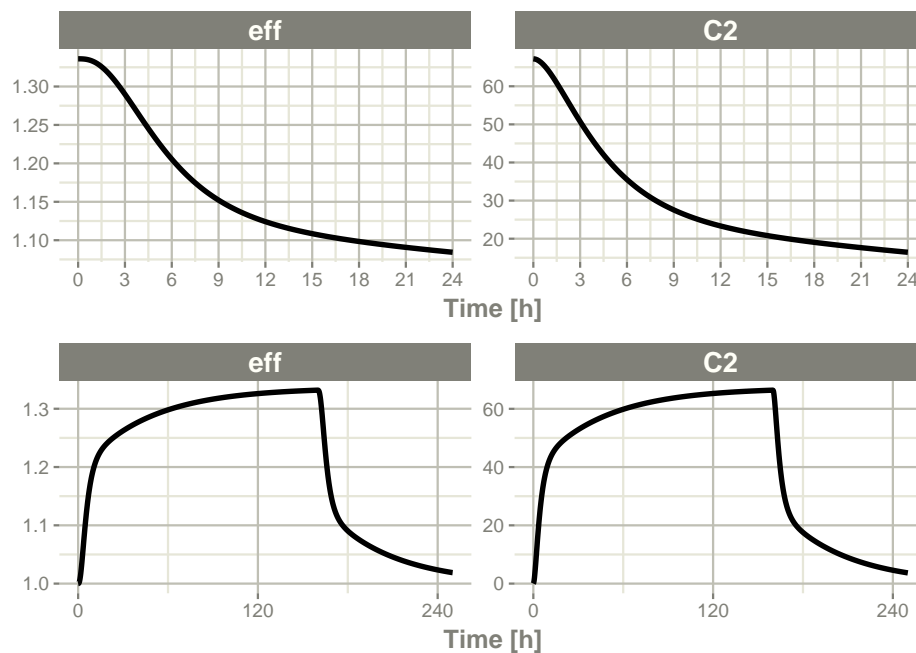

Not only can this be used for PK, it can be used for steady-state disease processes.

7.5   Reset Events

Reset events are implemented by `evid=3` or `evid=reset`, for reset and `evid=4` for
reset and dose.

```r
ev <- et(timeUnits="hr") %>%
    et(amt=10000, ii=12, addl=3) %>%
    et(time=6, evid=reset) %>%
    et(seq(0, 24, length.out=100))

ev
```

```
#> ---------------- EventTable with 102 records ---------------
#>
#>    2 dosing records (see x$get.dosing(); add with add.dosing or et)
#>    100 observation times (see x$get.sampling(); add with add.sampling or et)
#>    multiple doses in `addl` columns, expand with x$expand(); or etExpand(x)
#> -- First part of x: -------------------------------------
#> # A tibble: 102 x 5
#>         time    amt    ii   addl evid
#>          [h] <dbl>   [h] <int> <evid>
#>  1 0.0000000     NA    NA     NA 0:Observation
#>  2 0.0000000 10000    12      3 1:Dose (Add)
#>  3 0.2424242     NA    NA     NA 0:Observation
#>  4 0.4848485     NA    NA     NA 0:Observation
#>  5 0.7272727     NA    NA     NA 0:Observation
#>  6 0.9696970     NA    NA     NA 0:Observation
#>  7 1.2121212     NA    NA     NA 0:Observation
#>  8 1.4545455     NA    NA     NA 0:Observation
#>  9 1.6969697     NA    NA     NA 0:Observation
#> 10 1.9393939     NA    NA     NA 0:Observation
#> # ... with 92 more rows
```

The solving show what happens in this system when the system is reset at 6 hours
post-dose.

```r
rxSolve(m1, ev) %>% plot(depot,C2, eff)
```
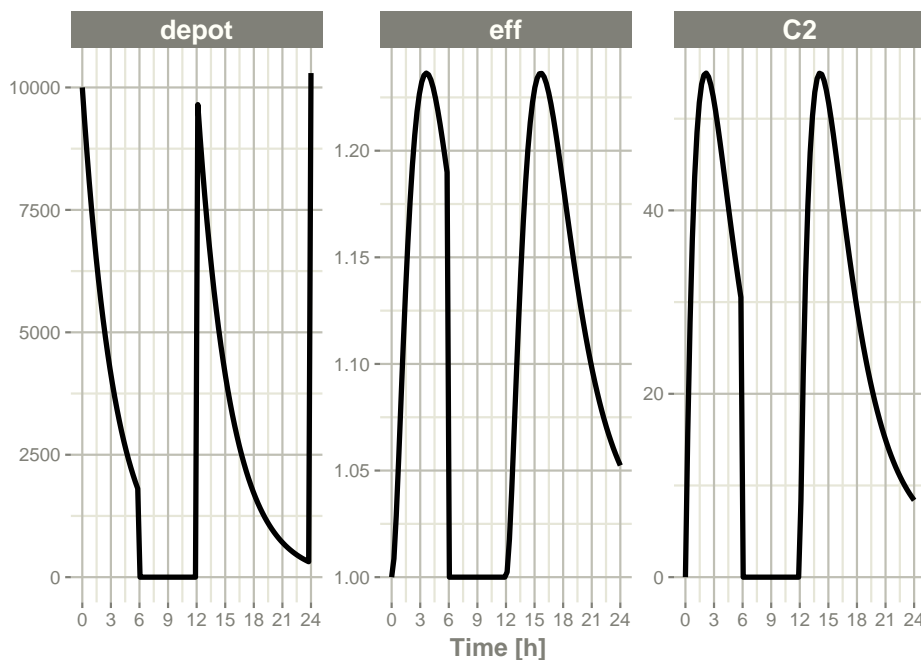
You can see all the compartments are reset to their initial values. The next dose start the dosing cycle over.

```
ev <- et(timeUnits="hr") %>%
    et(amt=10000, ii=12, addl=3) %>%
    et(time=6, amt=10000, evid=4) %>%
    et(seq(0, 24, length.out=100))

ev
```

```
#> ---------------- EventTable with 102 records ---------------
#>
#>    2 dosing records (see x$get.dosing(); add with add.dosing or et)
#>    100 observation times (see x$get.sampling(); add with add.sampling or et)
#>    multiple doses in `addl` columns, expand with x$expand(); or etExpand(x)
#> -- First part of x: ------------------------------------
#> # A tibble: 102 x 5
#>         time    amt    ii  addl evid
#>          [h] <dbl>   [h] <int> <evid>
#> 1 0.0000000    NA    NA    NA 0:Observation
#> 2 0.0000000 10000    12     3 1:Dose (Add)
#> 3 0.2424242    NA    NA    NA 0:Observation
#> 4 0.4848485    NA    NA    NA 0:Observation
#> 5 0.7272727    NA    NA    NA 0:Observation
```

```
#>  6 0.9696970    NA    NA    NA 0:Observation
#>  7 1.2121212    NA    NA    NA 0:Observation
#>  8 1.4545455    NA    NA    NA 0:Observation
#>  9 1.6969697    NA    NA    NA 0:Observation
#> 10 1.9393939    NA    NA    NA 0:Observation
#> # ... with 92 more rows
```

In this case, the whole system is reset and the dose is given

```
rxSolve(m1, ev) %>% plot(depot,C2, eff)
```
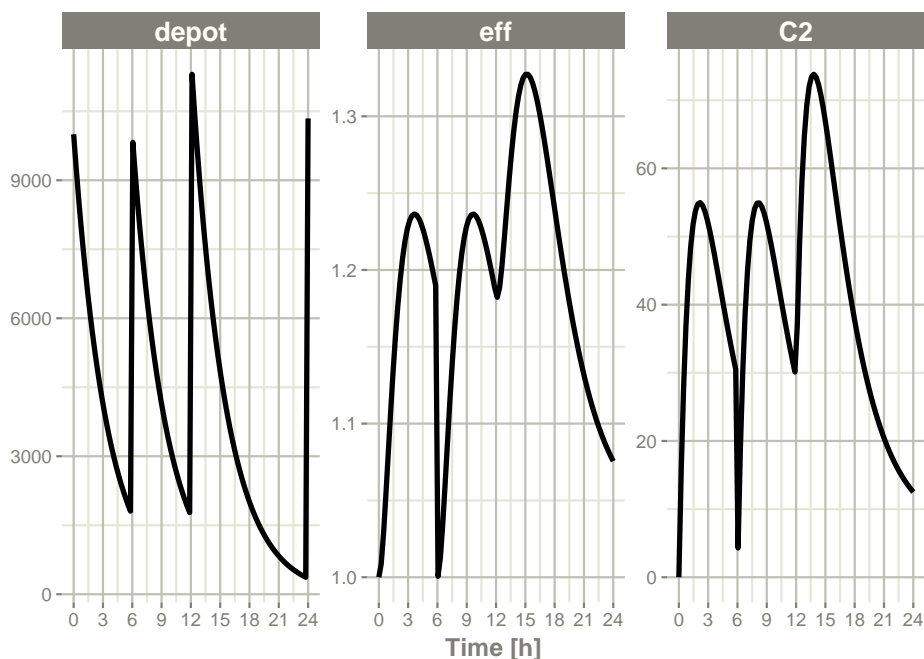

7.6   Turning off compartments

You may also turn off a compartment, which is similar to a reset event.

```
ev <- et(timeUnits="hr") %>%
    et(amt=10000, ii=12, addl=3) %>%
    et(time=6, cmt="-depot", evid=2) %>%
    et(seq(0, 24, length.out=100))

ev
```

```
#> ---------------- EventTable with 102 records ---------------
#>
#>    2 dosing records (see x$get.dosing(); add with add.dosing or et)
#>    100 observation times (see x$get.sampling(); add with add.sampling or et)
#>    multiple doses in `addl` columns, expand with x$expand(); or etExpand(x)
#> -- First part of x: ------------------------------------
#> # A tibble: 102 x 6
#>         time cmt        amt    ii  addl evid
#>          [h] <chr>      <dbl>  [h] <int> <evid>
#>  1 0.0000000 (obs)         NA   NA    NA 0:Observation
#>  2 0.0000000 (default) 10000   12     3 1:Dose (Add)
#>  3 0.2424242 (obs)         NA   NA    NA 0:Observation
#>  4 0.4848485 (obs)         NA   NA    NA 0:Observation
#>  5 0.7272727 (obs)         NA   NA    NA 0:Observation
#>  6 0.9696970 (obs)         NA   NA    NA 0:Observation
#>  7 1.2121212 (obs)         NA   NA    NA 0:Observation
#>  8 1.4545455 (obs)         NA   NA    NA 0:Observation
#>  9 1.6969697 (obs)         NA   NA    NA 0:Observation
#> 10 1.9393939 (obs)         NA   NA    NA 0:Observation
#> # ... with 92 more rows
```

Solving shows what this does in the system:

```
rxSolve(m1, ev) %>% plot(depot,C2, eff)
```
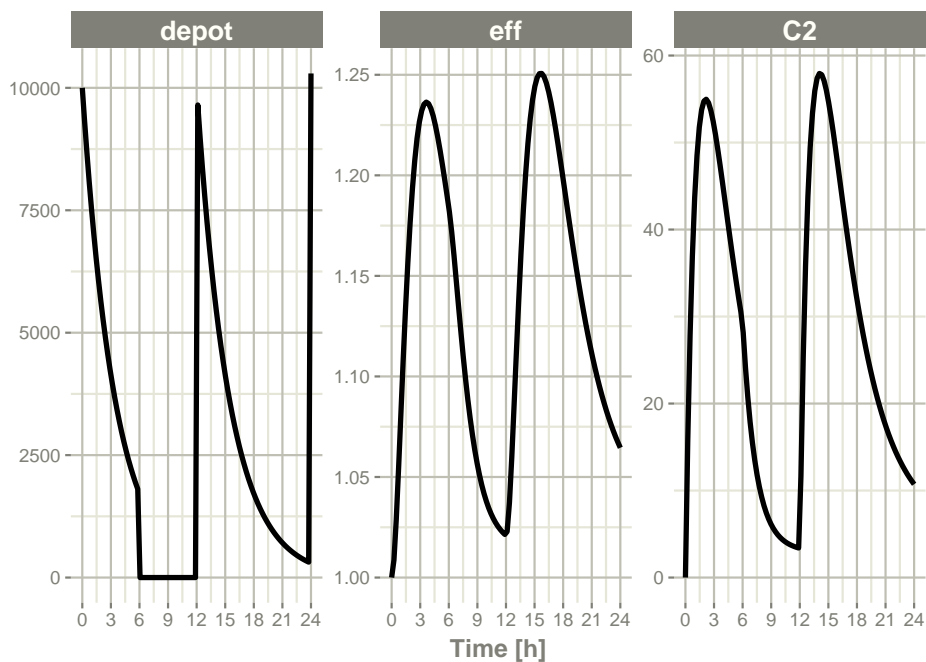
In this case, the depot is turned off, and the depot compartment concentrations are set to the initial values but the other compartment concentrations/levels are not reset. When another dose to the depot is administered the depot compartment is turned back on.

Note that a dose to a compartment only turns back on the compartment that was dosed. Hence if you turn off the effect compartment, it continues to be off after another dose to the depot.

```
ev <- et(timeUnits="hr") %>%
    et(amt=10000, ii=12, addl=3) %>%
    et(time=6, cmt="-eff", evid=2) %>%
    et(seq(0, 24, length.out=100))

rxSolve(m1, ev) %>% plot(depot,C2, eff)
```
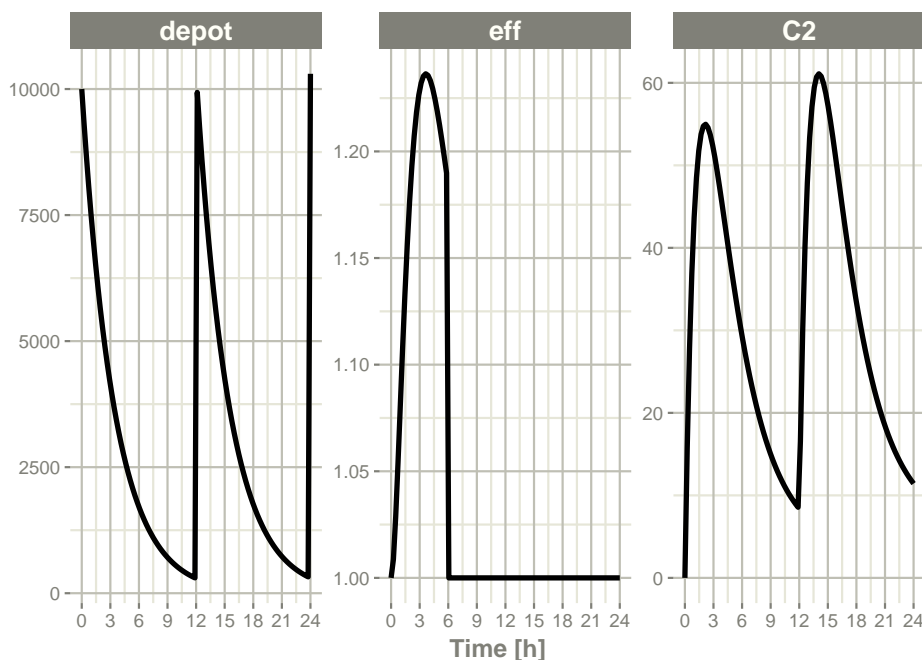

To turn back on the compartment, a zero-dose to the compartment or a evid=2 with the compartment would be needed.

```
ev <- et(timeUnits="hr") %>%
    et(amt=10000, ii=12, addl=3) %>%
    et(time=6, cmt="-eff", evid=2) %>%
    et(time=12,cmt="eff",evid=2) %>%
    et(seq(0, 24, length.out=100))
```

```r
rxSolve(m1, ev) %>% plot(depot,C2, eff)
```
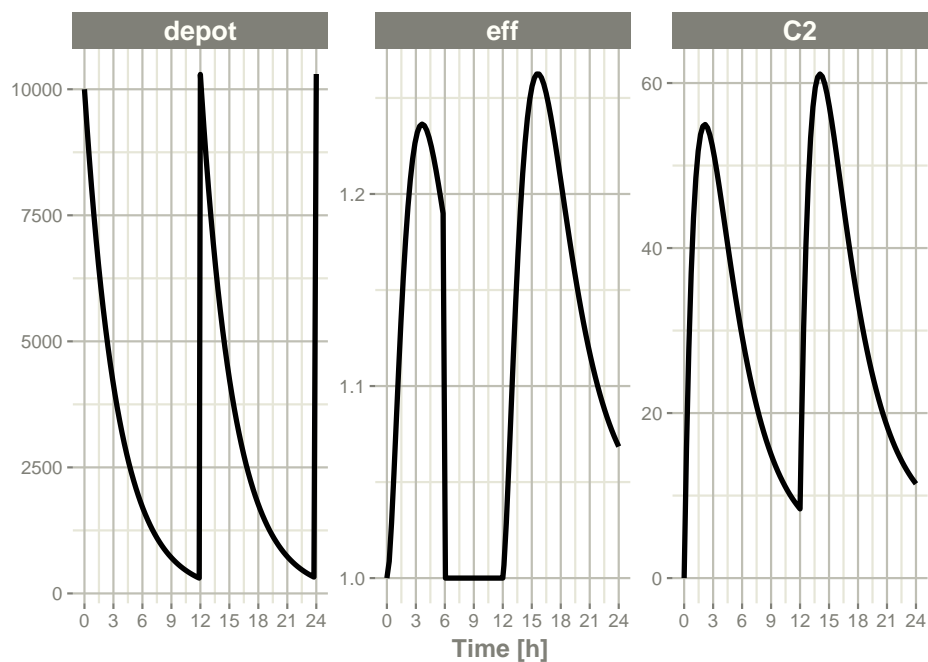

7.7 Classic RxODE events

Originally RxODE supported compound event IDs; RxODE still supports these parameters, but it is often more useful to use the the normal NONMEM dataset standard that is used by many modeling tools like NONMEM, Monolix and nlmixr, described in the RxODE event types article.

Classically, RxODE supported event coding in a single event id `evid` described in the following table.

100+ cmt	Infusion/Event Flag	<99 Cmt	SS flag & Turning of Compartment
100+ cmt	0 = bolus dose	< 99 cmt	1 = dose
	1 = infusion (rate)		10 = Steady state 1 (equivalent to SS=1)
	2 = infusion (dur)		20 = Steady state 2 (equivalent to SS=2)
	6 = turn off modeled duration		30 = Turn off a compartment (equivalent to -CMT w/EVID=2)

100+ cmt	Infusion/Event Flag	<99 Cmt	SS flag & Turning of Compartment
	7 = turn off modeled rate 8 = turn on modeled duration 9 = turn on modeled rate 4 = replace event 5 = multiply event		

The classic EVID concatenate the numbers in the above table, so an infusion would to compartment 1 would be `10101` and an infusion to compartment 199 would be `119901`.

EVID = 0 (observations), EVID=2 (other type event) and EVID=3 are all supported. Internally an EVID=9 is a non-observation event and makes sure the system is initialized to zero; EVID=9 should not be manually set. EVID 10-99 represents modeled time interventions, similar to NONMEM's MTIME. This along with amount (amt) and time columns specify the events in the ODE system.

For infusions specified with EVIDs > 100 the amt column represents the rate value.

For Infusion flags 1 and 2 `+amt` turn on the infusion to a specific compartment `-amt` turn off the infusion to a specific compartment. To specify a dose/duration you place the dosing records at the time the duration starts or stops.

For modeled rate/duration infusion flags the on infusion flag must be followed by an off infusion record.

These number are concatenated together to form a full RxODE event ID, as shown in the following examples:

7.7.1   Bolus Dose Examples

A 100 bolus dose to compartment #1 at time 0

time	evid	amt
0	101	100
0.5	0	0
1	0	0

A 100 bolus dose to compartment #99 at time 0

time	evid	amt
0	9901	100
0.5	0	0
1	0	0

A 100 bolus dose to compartment #199 at time 0

time	evid	amt
0	109901	100
0.5	0	0
1	0	0

7.7.2 Infusion Event Examples

Bolus infusion with rate 50 to compartment 1 for 1.5 hr, (modeled bioavailability changes duration of infusion)

time	evid	amt
0	10101	50
0.5	0	0
1	0	0
1.5	10101	-50

Bolus infusion with rate 50 to compartment 1 for 1.5 hr (modeled bioavailability changes rate of infusion)

time	evid	amt
0	20101	50
0.5	0	0
1	0	0
1.5	20101	-50

Modeled rate with amount of 50

time	evid	amt
0	90101	50
0	70101	50
0.5	0	0

time	evid	amt
1	0	0

Modeled duration with amount of 50

time	evid	amt
0	80101	50
0	60101	50
0.5	0	0
1	0	0

7.7.3   Steady State for classic RxODE EVID example

Steady state dose to cmt 1

time	evid	amt
0	110	50

Steady State with super-positioning principle for am 50 and pm 100 dose

time	evid	amt
0	110	50
12	120	100

7.7.4   Turning off a compartment with classic RxODE EVID

Turn off the first compartment at time 12

time	evid	amt
0	110	50
12	130	NA

Event coding in RxODE is encoded in a single event number `evid`. For compartments under 100, this is coded as:

- This event is `0` for observation events.
- For a specified compartment a bolus dose is defined as:

- **–** 100*(Compartment Number) + 1
- **–** The dose is then captured in the `amt`

- For IV bolus doses the event is defined as:

  - **–** 10000 + 100*(Compartment Number) + 1
  - **–** The infusion rate is captured in the `amt` column
  - **–** The infusion is turned off by subtracting `amt` with the same `evid` at the stop of the infusion.

For compartments greater or equal to 100, the 100s place and above digits are transferred to the 100,000th place digit. For doses to the 99th compartment the `evid` for a bolus dose would be `9901` and the `evid` for an infusion would be `19901`. For a bolus dose to the 199th compartment the `evid` for the bolus dose would be `109901`. An infusion dosing record for the 199th compartment would be `119901`.

Chapter 8

Easily creating RxODE events

An event table in RxODE is a specialized data frame that acts as a container for all of RxODE's events and observation times.

To create an RxODE event table you may use the code `eventTable()`, `et()`, or even create your own data frame with the right event information contained in it. This is closely related to the types of events that RxODE supports.

```
library(RxODE)
library(units)
```

```
#> udunits database from /usr/share/xml/udunits/udunits2.xml
```

```
(ev <- eventTable())
```

```
#> ----------------- EventTable with 0 records ----------------
#>
#>    0 dosing records (see x$get.dosing(); add with add.dosing or et)
#>    0 observation times (see x$get.sampling(); add with add.sampling or et)
```

or

```
(ev <- et())
```

```
#> ----------------- EventTable with 0 records ----------------
#>
#>    0 dosing records (see x$get.dosing(); add with add.dosing or et)
#>    0 observation times (see x$get.sampling(); add with add.sampling or et)
```

With this event table you can add sampling/observations or doses by piping or direct access.

This is a short table of the two main functions to create dosing

add.dosing()	et()	Description
dose	amt	Dose/Rate/Duration amount
nbr.doses	addl	Additional doses or number of doses
dosing.interval	ii	Dosing Interval
dosing.to	cmt	Dosing Compartment
rate	rate	Infusion rate
start.time	time	Dosing start time
	dur	Infusion Duration

Sampling times can be added with `add.sampling(` sampling times `)` or `et(` sampling times `)`. Dosing intervals and sampling windows are also supported.

For these models, we can illustrate by using the model shared in the RxODE tutorial:

```
## Model from RxODE tutorial
m1 <-RxODE({
    KA=2.94E-01;
    CL=1.86E+01;
    V2=4.02E+01;
    Q=1.05E+01;
    V3=2.97E+02;
    Kin=1;
    Kout=1;
    EC50=200;
    ## Added modeled bioavaiblity, duration and rate
    fdepot = 1;
    durDepot = 8;
    rateDepot = 1250;
    C2 = centr/V2;
    C3 = peri/V3;
    d/dt(depot) =-KA*depot;
    f(depot) = fdepot
    dur(depot) = durDepot
    rate(depot) = rateDepot
    d/dt(centr) = KA*depot - CL*C2 - Q*C2 + Q*C3;
    d/dt(peri)  =                     Q*C2 - Q*C3;
    d/dt(eff)  = Kin - Kout*(1-C2/(EC50+C2))*eff;
    eff(0) = 1
})
```

8.1 Adding doses to the event table

Once created you can add dosing to the event table by the `add.dosing()`, and `et()` functions.

Using the `add.dosing()` function you have:

argument	meaning
dose	dose amount
nbr.doses	Number of doses; Should be at least 1.
dosing.interval	Dosing interval; By default this is 24.
dosing.to	Compartment where dose is administered.
rate	Infusion rate
start.time	The start time of the dose

```r
ev <- eventTable(amount.units="mg", time.units="hr")

## The methods ar attached to the event table, so you can use
## them directly
ev$add.dosing(dose=10000, nbr.doses = 3)# loading doses
## Starts at time 0; Default dosing interval is 24

## You can also pipe the event tables to these methods.
ev <- ev %>%
  add.dosing(dose=5000, nbr.doses=14,
             dosing.interval=12)# maintenance

ev
```

```
#> ---------------- EventTable with 2 records ----------------
#>
#>    2 dosing records (see x$get.dosing(); add with add.dosing or et)
#>    0 observation times (see x$get.sampling(); add with add.sampling or et)
#>    multiple doses in `addl` columns, expand with x$expand(); or etExpand(x)
#> -- First part of x: -------------------------------------
#> # A tibble: 2 x 5
#>    time    amt    ii  addl evid
#>     [h]   [mg]   [h] <int> <evid>
#> 1     0  10000    24     2 1:Dose (Add)
#> 2     0   5000    12    13 1:Dose (Add)
```

Notice that the units were specified in the table. When specified, the units use the `units` package to keep track of the units and convert them if needed. Additionally,

ggforce uses them to label the `ggplot` axes. The `set_units` and `drop_units` are useful to set and drop the RxODE event table units.

In this example, you can see the time axes is labeled:

```r
rxSolve(m1, ev) %>% plot(C2)
```


Time [h]

If you are more familiar with the NONMEM/RxODE event records, you can also specify dosing using `et` with the dose elements directly:

```r
ev <- et(timeUnits="hr") %>%
  et(amt=10000, until = set_units(3, days),
     ii=12) # loading doses

ev
```
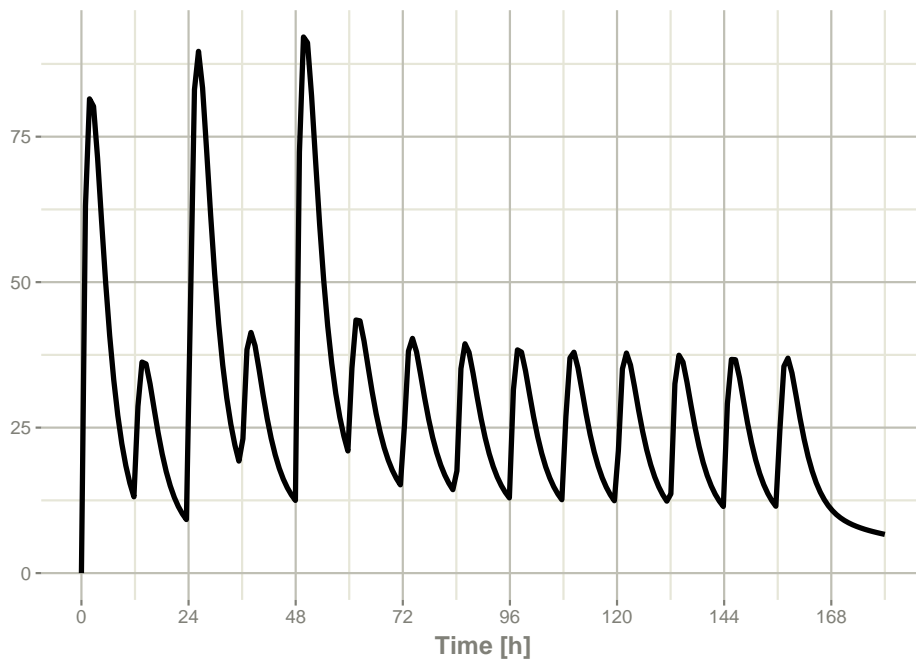
```
#> ------------------ EventTable with 1 records ----------------
#>
#>    1 dosing records (see x$get.dosing(); add with add.dosing or et)
#>    0 observation times (see x$get.sampling(); add with add.sampling or et)
#>    multiple doses in `addl` columns, expand with x$expand(); or etExpand(x)
#> -- First part of x: --------------------------------------
#> # A tibble: 1 x 5
#>    time    amt    ii  addl evid
```

```
#>      [h] <dbl>   [h] <int> <evid>
#> 1     0 10000    12    6 1:Dose (Add)
```

Which gives:

```
rxSolve(m1, ev) %>% plot(C2)
```


This shows how easy creating event tables can be.

8.2 Adding sampling to an event table

If you notice in the above examples, RxODE generated some default sampling times since there was not any sampling times. If you wish more control over the sampling time, you should add the samples to the RxODE event table by add.sampling or et

```
ev <- eventTable(amount.units="mg", time.units="hr")

## The methods ar attached to the event table, so you can use them
## directly
ev$add.dosing(dose=10000, nbr.doses = 3)# loading doses
```

```
ev$add.sampling(seq(0,24,by=4))

ev
```

```
#> ---------------- EventTable with 8 records ----------------
#>
#>     1 dosing records (see x$get.dosing(); add with add.dosing or et)
#>     7 observation times (see x$get.sampling(); add with add.sampling or et)
#>     multiple doses in `addl` columns, expand with x$expand(); or etExpand(x)
#> -- First part of x: -------------------------------------
#> # A tibble: 8 x 5
#>     time    amt    ii  addl evid
#>      [h]   [mg]   [h] <int> <evid>
#> 1      0     NA    NA    NA 0:Observation
#> 2      0  10000    24     2 1:Dose (Add)
#> 3      4     NA    NA    NA 0:Observation
#> 4      8     NA    NA    NA 0:Observation
#> 5     12     NA    NA    NA 0:Observation
#> 6     16     NA    NA    NA 0:Observation
#> 7     20     NA    NA    NA 0:Observation
#> 8     24     NA    NA    NA 0:Observation
```

Which gives:

```
solve(m1, ev) %>% plot(C2)
```
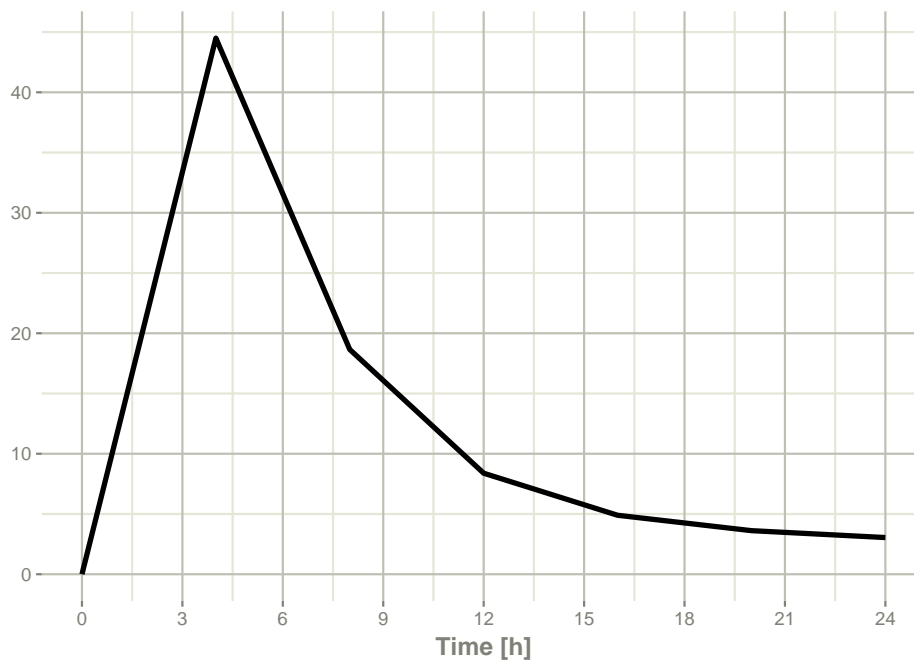
Or if you use `et` you can simply add them in a similar way to `add.sampling`:

```
ev <- et(timeUnits="hr") %>%
  et(amt=10000, until = set_units(3, days),
     ii=12) %>% # loading doses
  et(seq(0,24,by=4))

ev
```
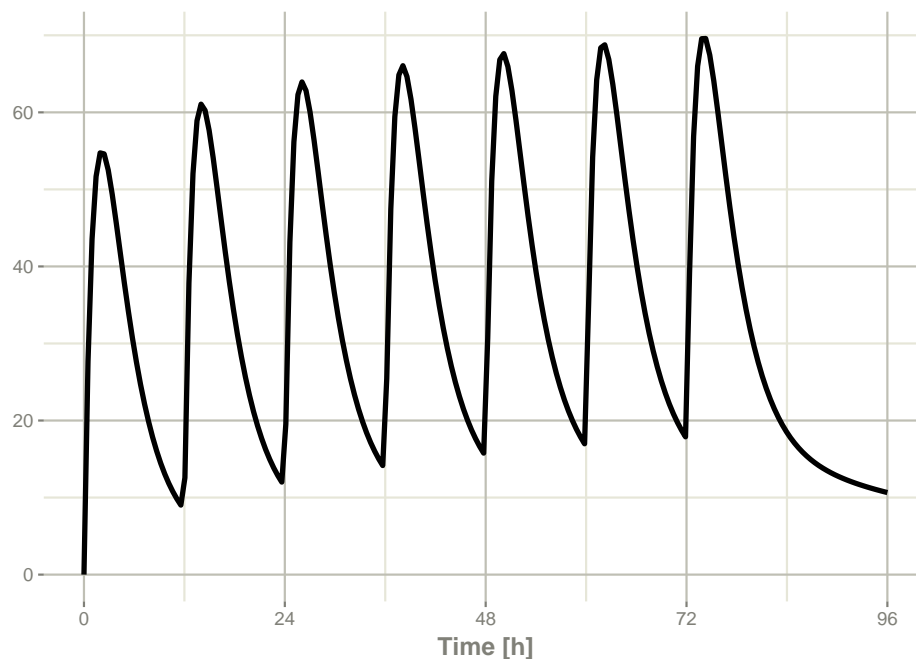
```
#> ---------------- EventTable with 8 records ----------------
#>
#>    1 dosing records (see x$get.dosing(); add with add.dosing or et)
#>    7 observation times (see x$get.sampling(); add with add.sampling or et)
#>    multiple doses in `addl` columns, expand with x$expand(); or etExpand(x)
#> -- First part of x: -------------------------------------
#> # A tibble: 8 x 5
#>    time    amt   ii  addl evid
#>     [h] <dbl>   [h] <int> <evid>
#> 1     0    NA    NA    NA 0:Observation
#> 2     0 10000    12     6 1:Dose (Add)
#> 3     4    NA    NA    NA 0:Observation
#> 4     8    NA    NA    NA 0:Observation
#> 5    12    NA    NA    NA 0:Observation
#> 6    16    NA    NA    NA 0:Observation
```

```
#> 7     20     NA     NA     NA 0:Observation
#> 8     24     NA     NA     NA 0:Observation
```

which gives the following RxODE solve:

```
solve(m1, ev) %>% plot(C2)
```
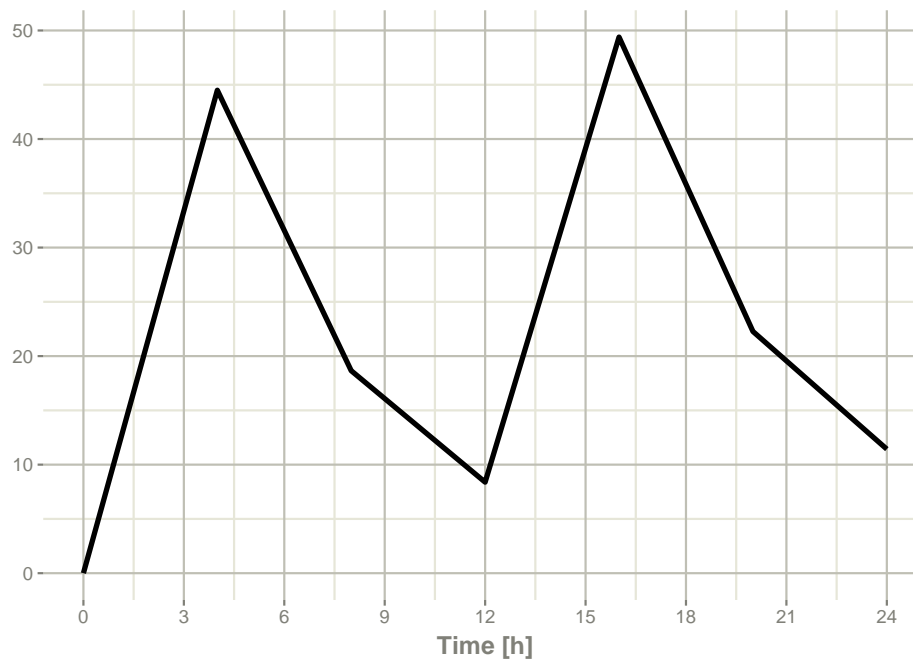

Note the jagged nature of these plots since there was only a few sample times.

8.3   Expand the event table to a multi-subject event table.

The only thing that is needed to expand an event table is a list of IDs that you want to expand;

```
ev <- et(timeUnits="hr") %>%
  et(amt=10000, until = set_units(3, days),
     ii=12) %>% # loading doses
  et(seq(0,48,length.out=200)) %>%
  et(id=1:4)

ev
```

```
#> ---------------- EventTable with 804 records ---------------
#>    4 individuals
#>    4 dosing records (see x$get.dosing(); add with add.dosing or et)
#>    800 observation times (see x$get.sampling(); add with add.sampling or et)
#>    multiple doses in `addl` columns, expand with x$expand(); or etExpand(x)
#> -- First part of x: -------------------------------------
#> # A tibble: 804 x 6
#>       id      time    amt      ii   addl evid
#>    <int>      [h] <dbl>     [h] <int> <evid>
#>  1     1 0.0000000     NA      NA     NA 0:Observation
#>  2     1 0.0000000  10000      12      6 1:Dose (Add)
#>  3     1 0.2412060     NA      NA     NA 0:Observation
#>  4     1 0.4824121     NA      NA     NA 0:Observation
#>  5     1 0.7236181     NA      NA     NA 0:Observation
#>  6     1 0.9648241     NA      NA     NA 0:Observation
#>  7     1 1.2060302     NA      NA     NA 0:Observation
#>  8     1 1.4472362     NA      NA     NA 0:Observation
#>  9     1 1.6884422     NA      NA     NA 0:Observation
#> 10     1 1.9296482     NA      NA     NA 0:Observation
#> # ... with 794 more rows
```

You can see in the following simulation there are 4 individuals that are solved for:

```
set.seed(42)
solve(m1, ev,
      params=data.frame(KA=0.294*exp(rnorm(4)),
                        18.6*exp(rnorm(4)))) %>%
    plot(C2)
```

```
#> Warning: 'ID' missing in 'parameters' dataset
#> individual parameters are assumed to have the same order as the event dataset
```

8.4   Add doses and samples within a sampling window

In addition to adding fixed doses and fixed sampling times, you can have windows where you sample and draw doses from. For dosing windows you specify the time as an ordered numerical vector with the lowest dosing time and the highest dosing time inside a list.

In this example, you start with a dosing time with a 6 hour dosing window:

```r
set.seed(42)
ev <- et(timeUnits="hr") %>%
  et(time=list(c(0,6)), amt=10000, until = set_units(2, days),
     ii=12) %>% # loading doses
  et(id=1:4)

ev
```
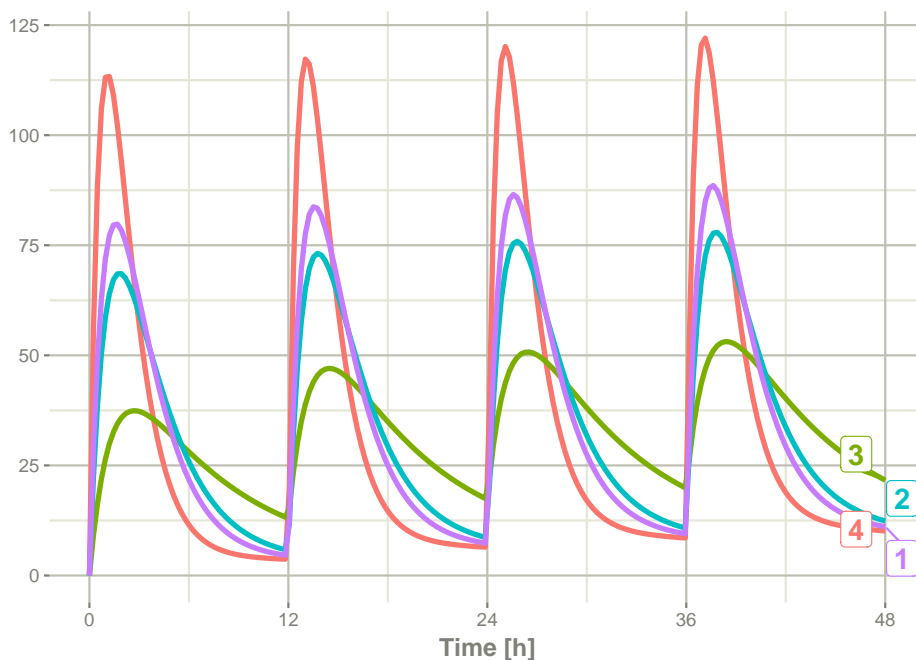
```
#> ---------------- EventTable with 16 records ----------------
#>     4 individuals
#>     16 dosing records (see x$get.dosing(); add with add.dosing or et)
#>     0 observation times (see x$get.sampling(); add with add.sampling or et)
#> -- First part of x: --------------------------------------
```

```
#> # A tibble: 16 x 6
#>         id  low        time  high    amt evid
#>      <int>  [h]         [h]   [h] <dbl> <evid>
#>  1     1     0   5.4888363     6 10000 1:Dose (Add)
#>  2     1    12  16.9826858    18 10000 1:Dose (Add)
#>  3     1    24  25.7168372    30 10000 1:Dose (Add)
#>  4     1    36  41.6224525    42 10000 1:Dose (Add)
#>  5     2     0   4.3146735     6 10000 1:Dose (Add)
#>  6     2    12  14.7464507    18 10000 1:Dose (Add)
#>  7     2    24  28.2303887    30 10000 1:Dose (Add)
#>  8     2    36  39.9419537    42 10000 1:Dose (Add)
#>  9     3     0   0.8079996     6 10000 1:Dose (Add)
#> 10     3    12  16.4195299    18 10000 1:Dose (Add)
#> 11     3    24  27.1145757    30 10000 1:Dose (Add)
#> 12     3    36  39.8504731    42 10000 1:Dose (Add)
#> 13     4     0   4.9826858     6 10000 1:Dose (Add)
#> 14     4    12  13.7168372    18 10000 1:Dose (Add)
#> 15     4    24  29.6224525    30 10000 1:Dose (Add)
#> 16     4    36  41.4888363    42 10000 1:Dose (Add)
```

You can clearly see different dosing times in the following simulation:

```
ev <- ev %>% et(seq(0,48,length.out=200))

solve(m1, ev,
      params=data.frame(KA=0.294*exp(rnorm(4)),
                        18.6*exp(rnorm(4)))) %>%
  plot(C2)
```

```
#> Warning: 'ID' missing in 'parameters' dataset
#> individual parameters are assumed to have the same order as the event dataset
```
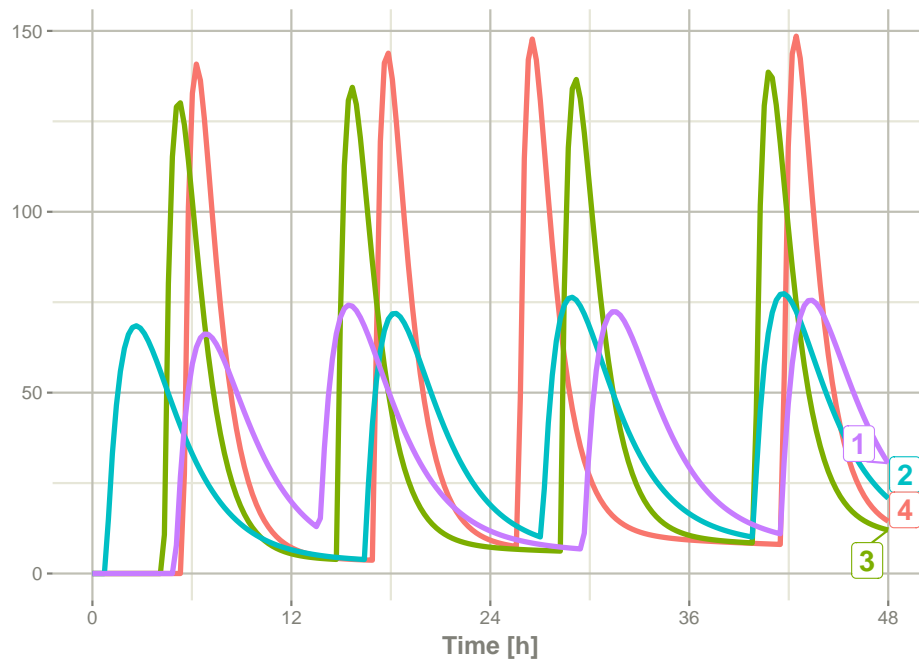
Of course in reality the dosing interval may only be 2 hours:

```
set.seed(42)
ev <- et(timeUnits="hr") %>%
  et(time=list(c(0,2)), amt=10000, until = set_units(2, days),
     ii=12) %>% # loading doses
  et(id=1:4) %>%
  et(seq(0,48,length.out=200))

solve(m1, ev,
      params=data.frame(KA=0.294*exp(rnorm(4)),
                        18.6*exp(rnorm(4)))) %>%
  plot(C2)
```

```
#> Warning: 'ID' missing in 'parameters' dataset
#> individual parameters are assumed to have the same order as the event dataset
```
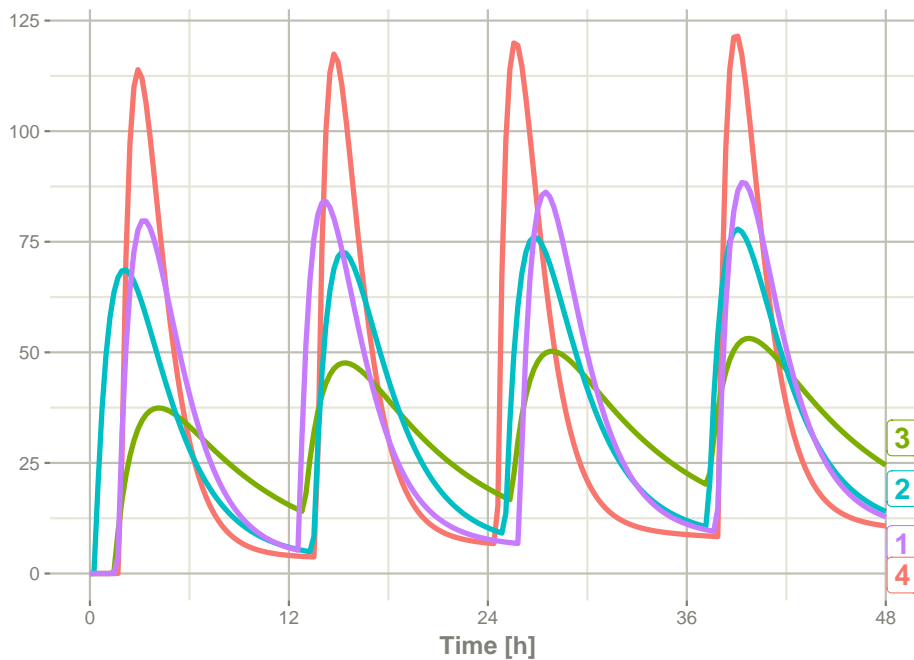
The same sort of thing can be specified with sampling times. To specify the sampling times in terms of a sampling window, you can create a list of the sampling times. Each sampling time will be a two element ordered numeric vector.

```
set.seed(42)
ev <- et(timeUnits="hr") %>%
  et(time=list(c(0,2)), amt=10000, until = set_units(2, days),
     ii=12) %>% # loading doses
  et(id=1:4)

## Create 20 samples in the first 24 hours and 20 samples in the
## second 24 hours
samples <- c(lapply(1:20, function(...){c(0,24)}),
             lapply(1:20, function(...){c(20,48)}))

## Add the random collection to the event table
ev <- ev %>% et(samples)

library(ggplot2)
solve(m1, ev, params=data.frame(KA=0.294*exp(rnorm(4)),
                                18.6*exp(rnorm(4)))) %>%
  plot(C2) + geom_point()
```

```
#> Warning: 'ID' missing in 'parameters' dataset
```

```
#> individual parameters are assumed to have the same order as the event dataset
```
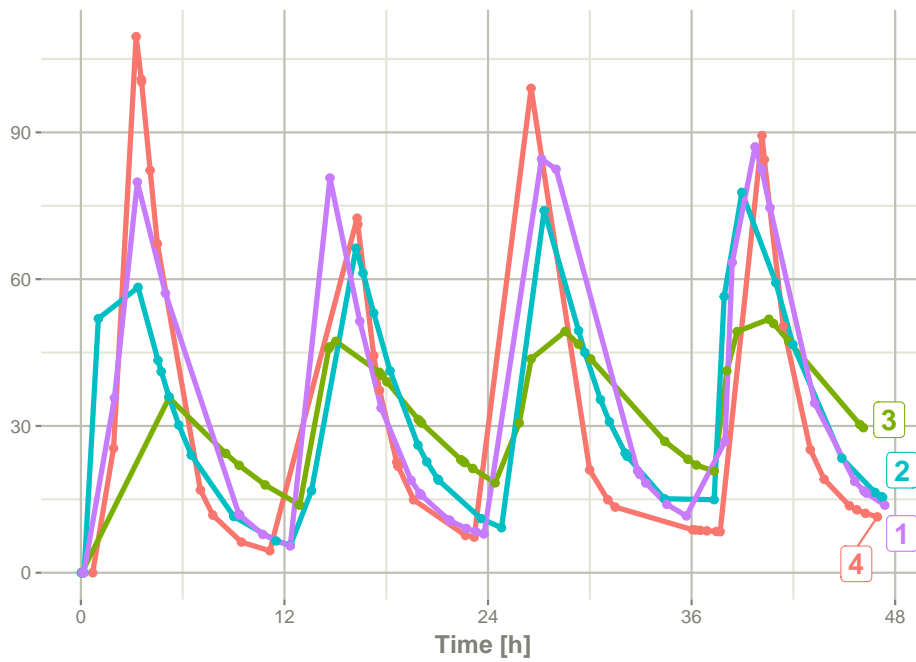

This shows the flexibility in dosing and sampling that the RxODE event tables allow.

8.5   Combining event tables

Since you can create dosing records and sampling records, you can create any complex dosing regimen you wish. In addition, RxODE allows you to combine event tables by c, seq, rep, and rbind.

8.6   Sequencing event tables

One way to combine event table is to sequence them by c, seq or etSeq. This takes the two dosing groups and adds at least one inter-dose interval between them:

```r
## bid for 5 days
bid <- et(timeUnits="hr") %>%
       et(amt=10000,ii=12,until=set_units(5, "days"))

## qd for 5 days
qd <- et(timeUnits="hr") %>%
```

```
      et(amt=20000,ii=24,until=set_units(5, "days"))

## bid for 5 days followed by qd for 5 days
et <- seq(bid,qd) %>% et(seq(0,11*24,length.out=100));

rxSolve(m1, et) %>% plot(C2)
```
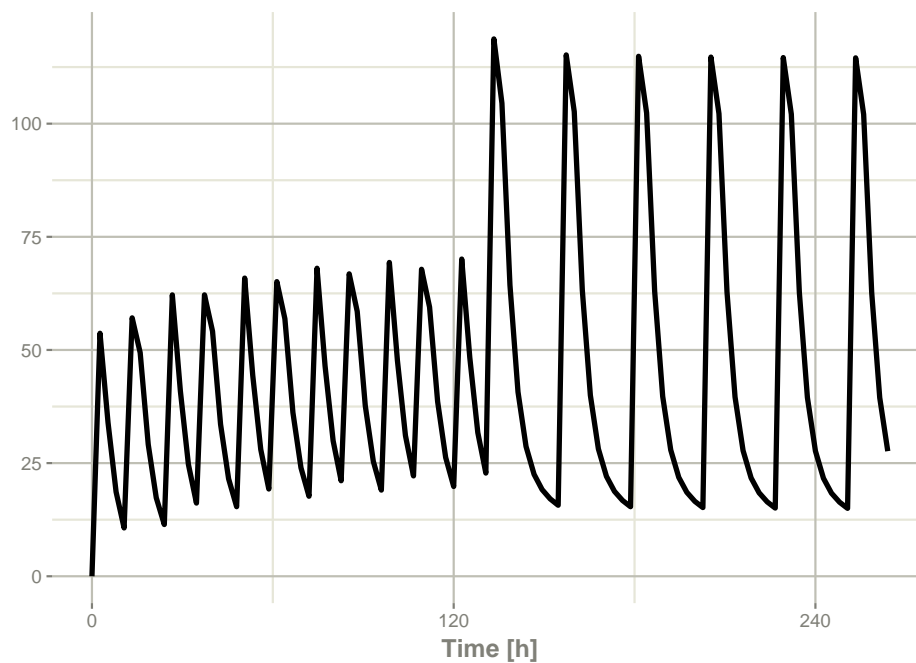

When sequencing events, you can also separate this sequence by a period of time; For example if you wanted to separate this by a week, you could easily do that with the following sequence of event tables:

```
## bid for 5 days followed by qd for 5 days
et <- seq(bid,set_units(1, "week"), qd) %>%
    et(seq(0,18*24,length.out=100));

rxSolve(m1, et) %>% plot(C2)
```
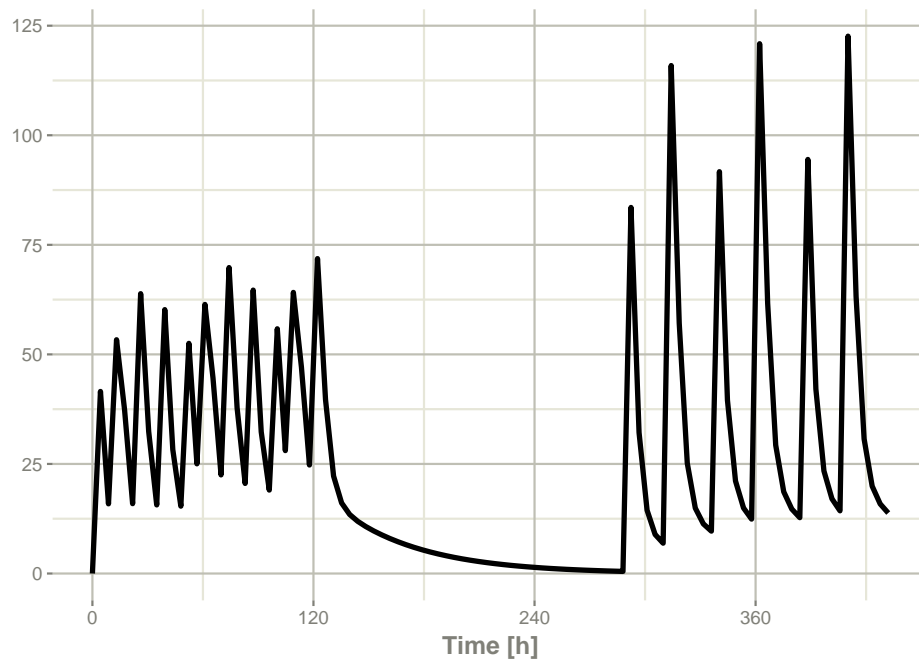
Note that in this example the time between the bid and the qd event tables is exactly one week, not 1 week plus 24 hours because of the inter-dose interval. If you want that behavior, you can sequence it using the `wait="+ii"`.

```
## bid for 5 days followed by qd for 5 days
et <- seq(bid,set_units(1, "week"), qd,wait="+ii") %>%
    et(seq(0,18*24,length.out=100));

rxSolve(m1, et) %>% plot(C2)
```
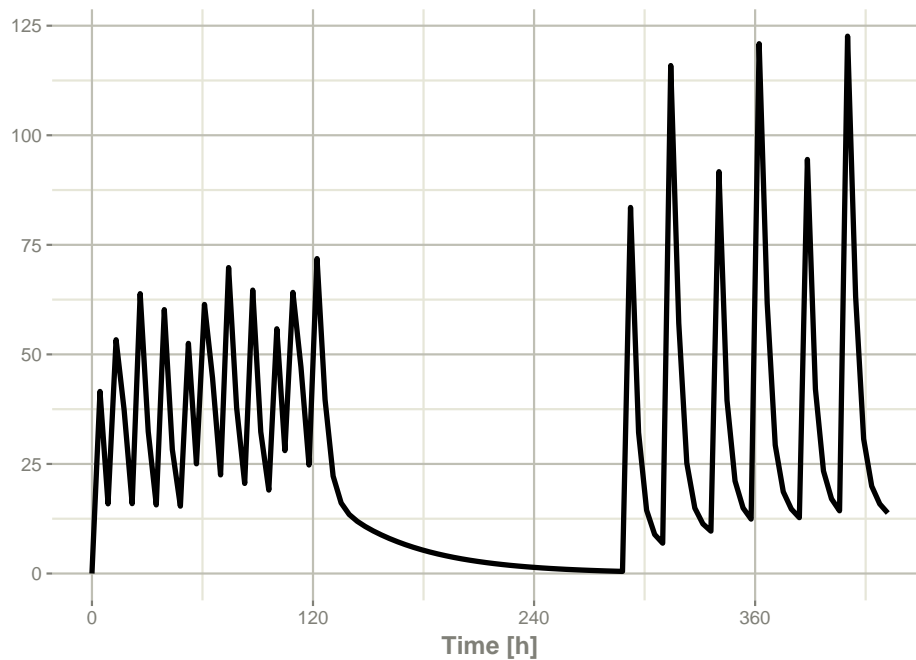
Also note, that RxODE assumes that the dosing is what you want to space the event tables by, and clears out any sampling records when you combine the event tables. If that is not true, you can also use the option `samples="use"`

8.7 Repeating event tables

You can have an event table that you can repeat with `etRep` or `rep`. For example 4 rounds of 2 weeks on QD therapy and 1 week off of therapy can be simply specified:

```
qd <-et(timeUnits = "hr") %>%
  et(amt=10000, ii=24, until=set_units(2, "weeks"), cmt="depot")

et <- rep(qd, times=4, wait=set_units(1,"weeks")) %>%
      add.sampling(set_units(seq(0, 12.5,by=0.005),weeks))

rxSolve(m1, et)  %>% plot(C2)
```
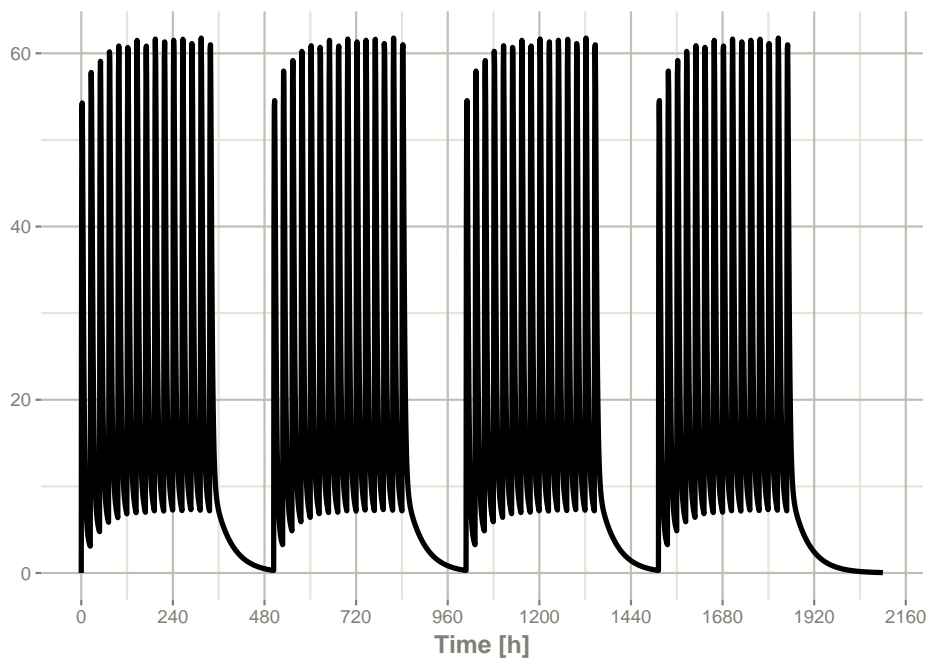
This is a simplified way to use a sequence of event tables. Therefore, many of the same options still apply; That is `samples` are cleared unless you use `samples="use"`, and the time between event tables is at least the inter-dose interval. You can adjust the timing by the `wait` option.

8.8   Combining event tables with rbind

You may combine event tables with `rbind`. This does not consider the event times when combining the event tables, but keeps them the same times. If you space the event tables by a waiting period, it also does not consider the inter-dose interval.

Using the previous `seq` you can clearly see the difference. Here was the sequence:

```r
## bid for 5 days
bid <- et(timeUnits="hr") %>%
       et(amt=10000,ii=12,until=set_units(5, "days"))

## qd for 5 days
qd <- et(timeUnits="hr") %>%
      et(amt=20000,ii=24,until=set_units(5, "days"))

et <- seq(bid,qd) %>%
    et(seq(0,18*24,length.out=500));
```

```
rxSolve(m1, et) %>% plot(C2)
```
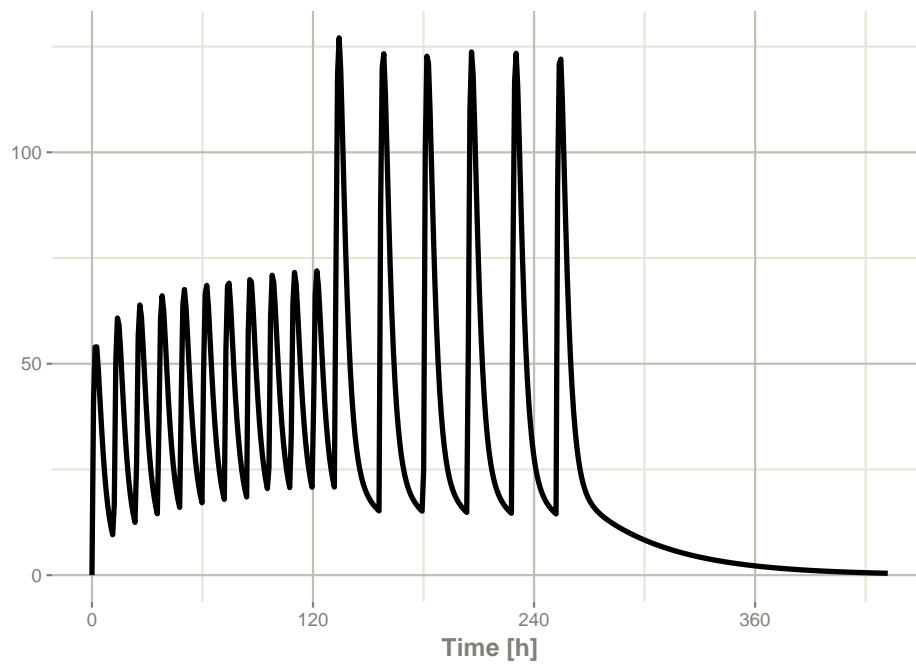

But if you bind them together with `rbind`

```
## bid for 5 days
et <- rbind(bid,qd) %>%
    et(seq(0,18*24,length.out=500));

rxSolve(m1, et) %>% plot(C2)
```
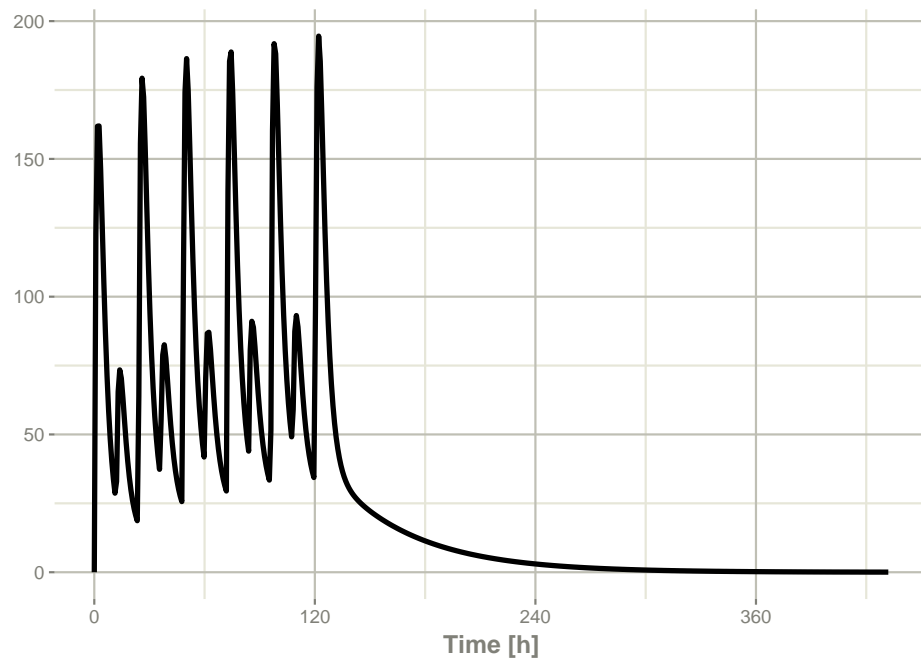
Still the waiting period applies (but does not consider the inter-dose interval)

```
et <- rbind(bid,wait=set_units(10,days),qd) %>%
    et(seq(0,18*24,length.out=500));

rxSolve(m1, et) %>% plot(C2)
```
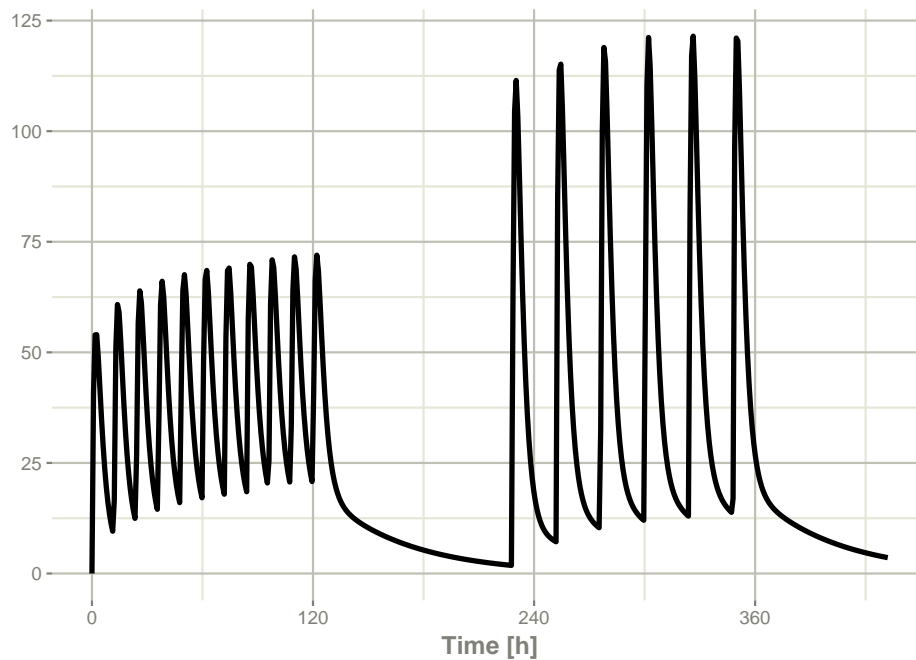
You can also bind the tables together and make each ID in the event table unique; This can be good to combine cohorts with different expected dosing and sampling times. This requires the id="unique" option; Using the first example shows how this is different in this case:

```
## bid for 5 days
et <- etRbind(bid,qd, id="unique") %>%
    et(seq(0,150,length.out=500));

library(ggplot2)
rxSolve(m1, et) %>% plot(C2) + facet_wrap( ~ id)
```
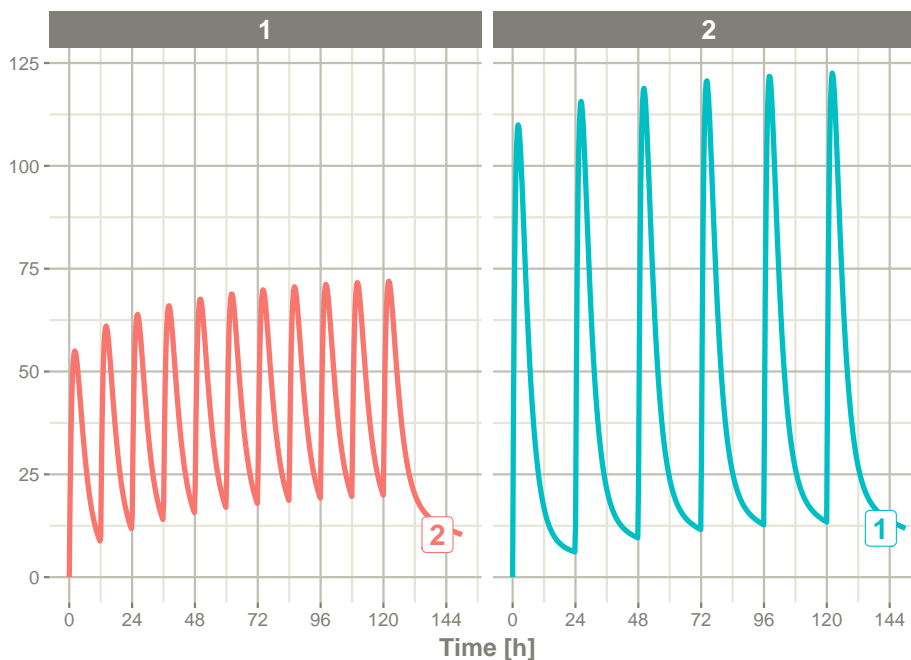
8.9   Expanding events

Event tables can be expanded so they contain an addl data item, like the following
example:

```
ev <- et() %>%
  et(dose=50, ii=8, until=48)

ev
```

```
#> ----------------- EventTable with 1 records ----------------
#>
#>    1 dosing records (see x$get.dosing(); add with add.dosing or et)
#>    0 observation times (see x$get.sampling(); add with add.sampling or et)
#>    multiple doses in `addl` columns, expand with x$expand(); or etExpand(x)
#> -- First part of x: -------------------------------------
#> # A tibble: 1 x 5
#>    time    amt    ii  addl evid
#>   <dbl> <dbl> <dbl> <int> <evid>
#> 1     0    50     8     6 1:Dose (Add)
```

You can expand the events so they do not have the addl items by $expand() or
etExpand(ev):

The first, `etExpand(ev)` expands the event table without modifying the original data frame:

```
etExpand(ev)
```

```
#> ----------------- EventTable with 7 records ----------------
#>
#>    7 dosing records (see x$get.dosing(); add with add.dosing or et)
#>    0 observation times (see x$get.sampling(); add with add.sampling or et)
#> -- First part of x: -------------------------------------
#> # A tibble: 7 x 4
#>    time   amt    ii evid
#>   <dbl> <dbl> <dbl> <evid>
#> 1     0    50     0 1:Dose (Add)
#> 2     8    50     0 1:Dose (Add)
#> 3    16    50     0 1:Dose (Add)
#> 4    24    50     0 1:Dose (Add)
#> 5    32    50     0 1:Dose (Add)
#> 6    40    50     0 1:Dose (Add)
#> 7    48    50     0 1:Dose (Add)
```

You can see the `addl` events were expanded, however the original data frame remained intact:

```
print(ev)
```

```
#> ----------------- EventTable with 1 records ----------------
#>
#>    1 dosing records (see $get.dosing(); add with add.dosing or et)
#>    0 observation times (see $get.sampling(); add with add.sampling or et)
#>    multiple doses in `addl` columns, expand with $expand(); or etExpand()
#> -- First part of : -------------------------------------
#> # A tibble: 1 x 5
#>    time   amt    ii  addl evid
#>   <dbl> <dbl> <dbl> <int> <evid>
#> 1     0    50     8     6 1:Dose (Add)
```

If you use `ev$expand()` it will modify the `ev` object. This is similar to an object-oriented method:
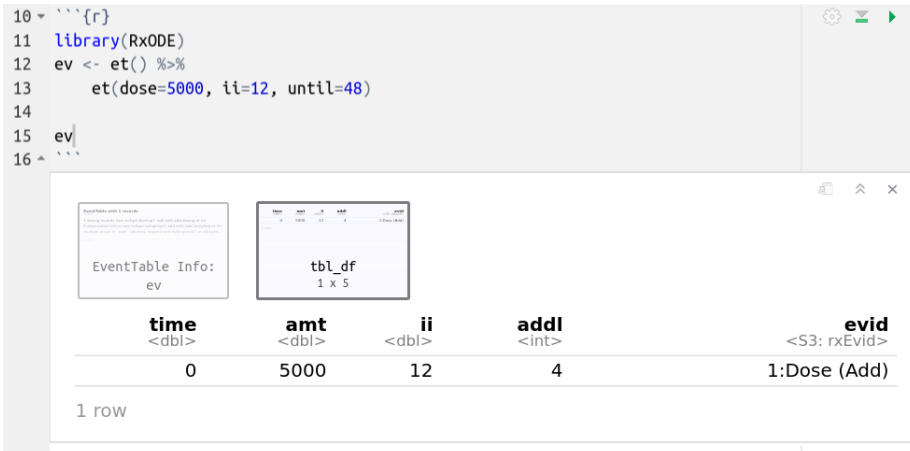
```
ev$expand()
ev
```

```
#> ----------------- EventTable with 7 records ----------------
#>
#>    7 dosing records (see x$get.dosing(); add with add.dosing or et)
#>    0 observation times (see x$get.sampling(); add with add.sampling or et)
#> -- First part of x: --------------------------------------
#> # A tibble: 7 x 4
#>     time    amt    ii evid
#>    <dbl> <dbl> <dbl> <evid>
#> 1     0    50     0 1:Dose (Add)
#> 2     8    50     0 1:Dose (Add)
#> 3    16    50     0 1:Dose (Add)
#> 4    24    50     0 1:Dose (Add)
#> 5    32    50     0 1:Dose (Add)
#> 6    40    50     0 1:Dose (Add)
#> 7    48    50     0 1:Dose (Add)
```

8.10   Event tables in Rstudio Notebooks

In addition to the output in the console which has been shown in the above examples, Rstudio notebook output is different and can be seen in the following screenshots;

The first screenshot shows how the event table looks after evaluating it in the Rstduio notebook

This is a simple dataframe that allows you to page through the contents. If you click on the first box in the Rstudio notebook output, it will have the notes about the event table:

```r
10 ▾ ```{r}
11    library(RxODE)
12    ev <- et() %>%
13        et(dose=5000, ii=12, until=48)
14
15    ev
16 ▴ ```
```

EventTable Info:
ev

tbl_df
1 x 5

EventTable with 1 records
<chr>

1 dosing records (see ev$get.dosing(); add with add.dosing or et)

0 observation times (see ev$get.sampling(); add with add.sampling or et)

multiple doses in `addl` columns, expand with ev$expand(); or etExpand(ev)

3 rows

Chapter 9

Solving and solving options

In general, ODEs are solved using a combination of:

- A compiled model specification from `RxODE()`, specified with `object=`

- Input parameters, specified with `params=` (and could be blank)

- Input data or event table, specified with `events=`

- Initial conditions, specified by `inits=` (and possibly in the model itself by `state(0)=`)

The solving options are given in the sections below:

9.1 General Solving Options

9.1.1 object

`object` is a either a RxODE family of objects, or a file-name with a RxODE model specification, or a string with a RxODE model specification.

9.1.2 params

`params` a numeric named vector with values for every parameter in the ODE system; the names must correspond to the parameter identifiers used in the ODE specification;

9.1.3   events

events an `eventTable` object describing the input (e.g., doses) to the dynamic system and observation sampling time points (see [eventTable()]);

9.1.4   inits

`inits` a vector of initial values of the state variables (e.g., amounts in each compartment), and the order in this vector must be the same as the state variables (e.g., PK/PD compartments);

9.1.5   method

`method` The method for solving ODEs. Currently this supports:

- `"liblsoda"` thread safe lsoda. This supports parallel thread-based solving, and ignores user Jacobian specification.
- `"lsoda"` – LSODA solver. Does not support parallel thread-based solving, but allows user Jacobian specification.
- `"dop853"` – DOP853 solver. Does not support parallel thread-based solving nor user Jacobain specification
- `"indLin"` – Solving through inductive linearization. The RxODE dll must be setup specially to use this solving routine.

9.1.6   stiff

`stiff` a logical (`TRUE` by default) indicating whether the ODE system is stiff or not.

For stiff ODE systems (`stiff = TRUE`), RxODE uses the LSODA (Livermore Solver for Ordinary Differential Equations) Fortran package, which implements an automatic method switching for stiff and non-stiff problems along the integration interval, authored by Hindmarsh and Petzold (2003).

For non-stiff systems (`stiff = FALSE`), RxODE uses DOP853, an explicit Runge-Kutta method of order 8(5, 3) of Dormand and Prince as implemented in C by Hairer and Wanner (1993).

If stiff is not specified, the `method` argument is used instead.

9.2   lsoda/dop solving options

9.2.1 atol

`atol` a numeric absolute tolerance (`1e-8` by default) used by the ODE solver to determine if a good solution has been achieved; This is also used in the solved linear model to check if prior doses do not add anything to the solution.

9.2.2 rtol

`rtol` a numeric relative tolerance (`1e-6` by default) used by the ODE solver to determine if a good solution has been achieved. This is also used in the solved linear model to check if prior doses do not add anything to the solution.

9.2.3 maxsteps

`maxsteps` maximum number of (internally defined) steps allowed during one call to the solver. (5000 by default)

9.2.4 hmin

`hmin` The minimum absolute step size allowed. The default value is 0.

9.2.5 hmax

`hmax` The maximum absolute step size allowed. When `hmax=NA` (default), uses the average difference + hmaxSd*sd in times and sampling events. The `hmaxSd` is a user specified parameter and which defaults to zero. When `hmax=NULL` RxODE uses the maximum difference in times in your sampling and events. The value 0 is equivalent to infinite maximum absolute step size.

9.2.6 hmaxSd

`hmaxSd` The number of standard deviations of the time difference to add to hmax. The default is 0

9.2.7 hini

`hini` The step size to be attempted on the first step. The default value is determined by the solver (when `hini = 0`)

9.2.8   maxordn

`maxordn` The maximum order to be allowed for the nonstiff (Adams) method. The default is 12. It can be between 1 and 12.

9.2.9   maxords

`maxords` The maximum order to be allowed for the stiff (BDF) method. The default value is 5. This can be between 1 and 5.

9.2.10   mxhnil

`mxhnil` maximum number of messages printed (per problem) warning that `T + H` = `T` on a step (`H` = step size). This must be positive to result in a non-default value. The default value is 0 (or infinite).

9.2.11   hmxi

`hmxi` inverse of the maximum absolute value of `H` to are used. hmxi = 0.0 is allowed and corresponds to an infinite `hmax1 (default).hminandhmximay be changed at any time, but will not take effect until the next change of`H`is considered. This option is only considered with`method`="liblsoda"'`.

9.2.12   istateReset

`istateReset` When `TRUE`, reset the `ISTATE` variable to 1 for lsoda and liblsoda with doses, like `deSolve`; When `FALSE`, do not reset the `ISTATE` variable with doses.

9.3   Inductive Linerization Options

9.3.1   indLinMatExpType

`indLinMatExpType` This is them matrix exponential type that is use for RxODE. Currently the following are supported:

- `Al-Mohy` Uses the exponential matrix method of Al-Mohy Higham (2009)

- `arma` Use the exponential matrix from RcppArmadillo

- `expokit` Use the exponential matrix from Roger B. Sidje (1998)

9.3.2 indLinMatExpOrder

`indLinMatExpOrder` an integer, the order of approximation to be used, for the `Al-Mohy` and `expokit` values. The best value for this depends on machine precision (and slightly on the matrix). We use 6 as a default.

9.3.3 indLinPhiTol

`indLinPhiTol` the requested accuracy tolerance on exponential matrix.

9.3.4 indLinPhiM

`indLinPhiM` the maximum size for the Krylov basis

9.4 Steady State Solving Options

9.4.1 minSS

`minSS` Minimum number of iterations for a steady-state dose

9.4.2 maxSS

`maxSS` Maximum number of iterations for a steady-state dose

9.4.3 strictSS

`strictSS` Boolean indicating if a strict steady-state is required. If a strict steady-state is (TRUE) required then at least `minSS` doses are administered and the total number of steady states doses will continue until `maxSS` is reached, or `atol` and `rtol` for every compartment have been reached. However, if ODE solving problems occur after the `minSS` has been reached the whole subject is considered an invalid solve. If `strictSS` is FALSE then as long as `minSS` has been reached the last good solve before ODE solving problems occur is considered the steady state, even though either `atol`, `rtol` or `maxSS` have not been achieved.

9.4.4 infSSstep

`infSSstep` Step size for determining if a constant infusion has reached steady state. By default this is large value, 420.

9.4.5   ssAtol

`ssAtol` Steady state atol convergence factor. Can be a vector based on each state.

9.4.6   ssRtol

`ssRtol` Steady state rtol convergence factor. Can be a vector based on each state.

9.5   RxODE numeric stability options

9.5.1   maxAtolRtolFactor

`maxAtolRtolFactor` The maximum `atol/rtol` that FOCEi and other routines may adjust to. By default 0.1

9.5.2   stateTrim

`stateTrim` When amounts/concentrations in one of the states are above this value, trim them to be this value. By default Inf. Also trims to -stateTrim for large negative amounts/concentrations. If you want to trim between a range say `c(0, 2000000)` you may specify 2 values with a lower and upper range to make sure all state values are in the reasonable range.

9.5.3   safeZero

`safeZero` Use safe zero divide and log routines. By default this is turned on but you may turn it off if you wish.

9.5.4   sumType

`sumType` Sum type to use for `sum()` in RxODE code blocks.

`pairwise` uses the pairwise sum (fast, default)

`fsum` uses Python's fsum function (most accurate)

`kahan` uses Kahan correction

`neumaier` uses Neumaier correction

`c` uses no correction: default/native summing

9.5.5 prodType

prodType Product to use for `prod()` in RxODE blocks

`long double` converts to long double, performs the multiplication and then converts back.

`double` uses the standard double scale for multiplication.

9.5.6 maxwhile

`maxwhile` represents the maximum times a while loop is evaluated before exiting. By default this is 100000

9.5.7 transitAbs

`transitAbs` boolean indicating if this is a transit compartment absorption

9.6 Linear compartment model sensitivity options

9.6.1 sensType

sensType Sensitivity type for `linCmt()` model:

advan Use the direct advan solutions

autodiff Use the autodiff advan solutions

forward Use forward difference solutions

central Use central differences

9.6.2 linDiff

`linDiff` This gives the linear difference amount for all the types of linear compartment model parameters where sensitivities are not calculated. The named components of this numeric vector are:

- `"lag"` Central compartment lag
- `"f"` Central compartment bioavailability
- `"rate"` Central compartment modeled rate
- `"dur"` Central compartment modeled duration
- `"lag2"` Depot compartment lag
- `"f2"` Depot compartment bioavailability

- `"rate2"` Depot compartment modeled rate
- `"dur2"` Depot compartment modeled duration

9.6.3   linDiffCentral

`linDiffCentral` This gives the which parameters use central differences for the linear compartment model parameters. The are the same components as `linDiff`

9.7   Covariate Solving Options

9.7.1   iCov

`iCov` A data frame of individual non-time varying covariates to combine with the `params` to form a parameter data.frame.

9.7.2   covsInterpolation

`covsInterpolation` specifies the interpolation method for time-varying covariates. When solving ODEs it often samples times outside the sampling time specified in `events`. When this happens, the time varying covariates are interpolated. Currently this can be:

- `"linear"` interpolation, which interpolates the covariate by solving the line between the observed covariates and extrapolating the new covariate value.

- `"constant"` – Last observation carried forward (the default).

- `"NOCB"` – Next Observation Carried Backward. This is the same method that NONMEM uses.

- `"midpoint"` Last observation carried forward to midpoint; Next observation carried backward to midpoint.

9.7.3   addCov

`addCov` A boolean indicating if covariates should be added to the output matrix or data frame. By default this is disabled.

9.8 Simulation options

9.8.1 seed

`seed` an object specifying if and how the random number generator should be initialized

9.8.2 nsim

`nsim` represents the number of simulations. For RxODE, if you supply single subject event tables (created with [`eventTable()`])

9.8.3 thetaMat

`thetaMat` Named theta matrix.

9.8.4 thetaLower

`thetaLower` Lower bounds for simulated population parameter variability (by default `-Inf`)

9.8.5 thetaUpper

`thetaUpper` Upper bounds for simulated population unexplained variability (by default `Inf`)

9.8.6 thetaDf

`thetaDf` The degrees of freedom of a t-distribution for simulation. By default this is `NULL` which is equivalent to `Inf` degrees, or to simulate from a normal distribution instead of a t-distribution.

9.8.7 thetaIsChol

`thetaIsChol` Indicates if the `theta` supplied is a Cholesky decomposed matrix instead of the traditional symmetric matrix.

9.8.8   nStud

nStud Number virtual studies to characterize uncertainty in estimated parameters.

9.8.9   omega

omega Estimate of Covariance matrix. When omega is a list, assume it is a block matrix and convert it to a full matrix for simulations.

9.8.10   omegaIsChol

omegaIsChol Indicates if the omega supplied is a Cholesky decomposed matrix instead of the traditional symmetric matrix.

9.8.11   omegaSeparation

omegaSeparation Omega separation strategy

Tells the type of separation strategy when simulating covariance with parameter uncertainty with standard deviations modeled in the thetaMat matrix.

- "lkj" simulates the correlation matrix from the rLKJ1 matrix with the distribution parameter eta equal to the degrees of freedom nu by (nu-1)/2

- "separation" simulates from the identity inverse Wishart covariance matrix with nu degrees of freedom. This is then converted to a covariance matrix and augmented with the modeled standard deviations. While computationally more complex than the "lkj" prior, it performs better when the covariance matrix size is greater or equal to 10

- "auto" chooses "lkj" when the dimension of the matrix is less than 10 and "separation" when greater than equal to 10.

9.8.12   omegaXform

omegaXform When taking omega values from the thetaMat simulations (using the separation strategy for covariance simulation), how should the thetaMat values be turned int standard deviation values:

- identity This is when standard deviation values are directly modeled by the params and thetaMat matrix

- `variance` This is when the `params` and `thetaMat` simulates the variance that are directly modeled by the `thetaMat` matrix

- `log` This is when the `params` and `thetaMat` simulates `log(sd)`

- `nlmixrSqrt` This is when the `params` and `thetaMat` simulates the inverse cholesky decomposed matrix with the `x^2` modeled along the diagonal. This only works with a diagonal matrix.

- `nlmixrLog` This is when the `params` and `thetaMat` simulates the inverse cholesky decomposed matrix with the `exp(x^2)` along the diagonal. This only works with a diagonal matrix.

- `nlmixrIdentity` This is when the `params` and `thetaMat` simulates the inverse cholesky decomposed matrix. This only works with a diagonal matrix.

9.8.13 omegaLower

`omegaLower` Lower bounds for simulated ETAs (by default -Inf)

9.8.14 omegaUpper

`omegaUpper` Upper bounds for simulated ETAs (by default Inf)

9.8.15 omegaDf

`omegaDf` The degrees of freedom of a t-distribution for simulation. By default this is `NULL` which is equivalent to `Inf` degrees, or to simulate from a normal distribution instead of a t-distribution.

9.8.16 nSub

`nSub` Number between subject variabilities (`ETAs`) simulated for every realization of the parameters.

9.8.17 dfSub

`dfSub` Degrees of freedom to sample the between subject variability matrix from the inverse Wishart distribution (scaled) or scaled inverse chi squared distribution.

9.8.18   sigma

`sigma` Named sigma covariance or Cholesky decomposition of a covariance matrix. The names of the columns indicate parameters that are simulated. These are simulated for every observation in the solved system.

9.8.19   sigmaLower

`sigmaLower` Lower bounds for simulated unexplained variability (by default -Inf)

9.8.20   sigmaUpper

`sigmaUpper` Upper bounds for simulated unexplained variability (by default Inf)

9.8.21   sigmaXform

`sigmaXform` When taking `sigma` values from the `thetaMat` simulations (using the separation strategy for covariance simulation), how should the `thetaMat` values be turned int standard deviation values:

- `identity` This is when standard deviation values are directly modeled by the `params` and `thetaMat` matrix

- `variance` This is when the `params` and `thetaMat` simulates the variance that are directly modeled by the `thetaMat` matrix

- `log` This is when the `params` and `thetaMat` simulates $\log(sd)$

- `nlmixrSqrt` This is when the `params` and `thetaMat` simulates the inverse cholesky decomposed matrix with the x^2 modeled along the diagonal. This only works with a diagonal matrix.

- `nlmixrLog` This is when the `params` and `thetaMat` simulates the inverse cholesky decomposed matrix with the $\exp(x^2)$ along the diagonal. This only works with a diagonal matrix.

- `nlmixrIdentity` This is when the `params` and `thetaMat` simulates the inverse cholesky decomposed matrix. This only works with a diagonal matrix.

9.8.22   sigmaDf

`sigmaDf` Degrees of freedom of the sigma t-distribution. By default it is equivalent to `Inf`, or a normal distribution.

9.8.23 sigmaIsChol

`sigmaIsChol` Boolean indicating if the sigma is in the Cholesky decomposition instead of a symmetric covariance

9.8.24 sigmaSeparation

`sigmaSeparation` separation strategy for sigma;

Tells the type of separation strategy when simulating covariance with parameter uncertainty with standard deviations modeled in the `thetaMat` matrix.

- `"lkj"` simulates the correlation matrix from the `rLKJ1` matrix with the distribution parameter `eta` equal to the degrees of freedom `nu` by `(nu-1)/2`

- `"separation"` simulates from the identity inverse Wishart covariance matrix with `nu` degrees of freedom. This is then converted to a covariance matrix and augmented with the modeled standard deviations. While computationally more complex than the `"lkj"` prior, it performs better when the covariance matrix size is greater or equal to 10

- `"auto"` chooses `"lkj"` when the dimension of the matrix is less than 10 and `"separation"` when greater than equal to 10.

9.8.25 dfObs

`dfObs` Degrees of freedom to sample the unexplained variability matrix from the inverse Wishart distribution (scaled) or scaled inverse chi squared distribution.

9.8.26 resample

`resample` A character vector of model variables to resample from the input dataset; This sampling is done with replacement. When `NULL` or `FALSE` no resampling is done. When `TRUE` resampling is done on all covariates in the input dataset

9.8.27 resampleID

`resampleID` boolean representing if the resampling should be done on an individual basis `TRUE` (ie. a whole patient is selected) or each covariate is resampled independent of the subject identifier `FALSE`. When `resampleID=TRUE` correlations of parameters are retained, where as when `resampleID=FALSE` ignores patient covariate correaltions. Hence the default is `resampleID=TRUE`.

9.9   RxODE output options

9.9.1   returnType

`returnType` This tells what type of object is returned. The currently supported types are:

- `"rxSolve"` (default) will return a reactive data frame that can change easily change different pieces of the solve and update the data frame. This is the currently standard solving method in RxODE, is used for `rxSolve(object, ...)`, `solve(object,...)`,

- `"data.frame"` – returns a plain, non-reactive data frame; Currently very slightly faster than `returnType="matrix"`

- `"matrix"` – returns a plain matrix with column names attached to the solved object. This is what is used `object$run` as well as `object$solve`

- `"data.table"` – returns a `data.table`; The `data.table` is created by reference (ie `setDt()`), which should be fast.

- `"tbl"` or `"tibble"` returns a tibble format.

9.9.2   addDosing

`addDosing` Boolean indicating if the solve should add RxODE EVID and related columns. This will also include dosing information and estimates at the doses. Be default, RxODE only includes estimates at the observations. (default `FALSE`). When `addDosing` is `NULL`, only include EVID=0 on solve and exclude any model-times or EVID=2. If `addDosing` is `NA` the classic `RxODE` EVID events are returned. When `addDosing` is `TRUE` add the event information in NONMEM-style format; If `subsetNonmem=FALSE` RxODE will also include extra event types (EVID) for ending infusion and modeled times:

- EVID=-1 when the modeled rate infusions are turned off (matches `rate=-1`)

- EVID=-2 When the modeled duration infusions are turned off (matches `rate=-2`)

- EVID=-10 When the specified `rate` infusions are turned off (matches `rate>0`)

- EVID=-20 When the specified `dur` infusions are turned off (matches `dur>0`)

- EVID=101,102,103,... Modeled time where 101 is the first model time, 102 is the second etc.

9.9.3 keep

`keep` Columns to keep from either the input dataset or the `iCov` dataset. With the `iCov` dataset, the column is kept once per line. For the input dataset, if any records are added to the data LOCF (Last Observation Carried forward) imputation is performed.

9.9.4 drop

`drop` Columns to drop from the output

9.9.5 idFactor

`idFactor` This boolean indicates if original ID values should be maintained. This changes the default sequentially ordered ID to a factor with the original ID values in the original dataset. By default this is enabled.

9.9.6 subsetNonmem

`subsetNonmem` subset to NONMEM compatible EVIDs only. By default `TRUE`.

9.9.7 matrix

`matrix` A boolean indicating if a matrix should be returned instead of the RxODE's solved object.

9.9.8 scale

`scale` a numeric named vector with scaling for ode parameters of the system. The names must correspond to the parameter identifiers in the ODE specification. Each of the ODE variables will be divided by the scaling factor. For example `scale=c(center=2)` will divide the center ODE variable by 2.

9.9.9 amountUnits

`amountUnits` This supplies the dose units of a data frame supplied instead of an event table. This is for importing the data as an RxODE event table.

9.9.10   timeUnits

`timeUnits` This supplies the time units of a data frame supplied instead of an event table. This is for importing the data as an RxODE event table.

9.9.11   theta

`theta` A vector of parameters that will be named `THETA\[#\]` and added to parameters

9.9.12   eta

`eta` A vector of parameters that will be named `ETA\[#\]` and added to parameters

9.9.13   from

`from` When there is no observations in the event table, start observations at this value. By default this is zero.

9.9.14   to

`to` When there is no observations in the event table, end observations at this value. By default this is 24 + maximum dose time.

9.9.15   length.out

`length.out` The number of observations to create if there isn't any observations in the event table. By default this is 200.

9.9.16   by

`by` When there are no observations in the event table, this is the amount to increment for the observations between `from` and `to`.

9.9.17   warnIdSort

`warnIdSort` Warn if the ID is not present and RxODE assumes the order of the parameters/iCov are the same as the order of the parameters in the input dataset.

9.9.18   warnDrop

`warnDrop` Warn if column(s) were supposed to be dropped, but were not present.

9.10   Internal RxODE options

9.10.1   nDisplayProgress

`nDisplayProgress` An integer indicating the minimum number of c-based solves before a progress bar is shown. By default this is 10,000.

9.10.2   …

. . . Other arguments including scaling factors for each compartment. This includes S# = numeric will scale a compartment # by a dividing the compartment amount by the scale factor, like NONMEM.

9.10.3   a

`a` when using `solve()`, this is equivalent to the `object` argument. If you specify `object` later in the argument list it overwrites this parameter.

9.10.4   b

`b` when using `solve()`, this is equivalent to the `params` argument. If you specify `params` as a named argument, this overwrites the output

9.10.5   updateObject

`updateObject` This is an internally used flag to update the RxODE solved object (when supplying an RxODE solved object) as well as returning a new object. You probably should not modify it's `FALSE` default unless you are willing to have unexpected results.

9.11   Parallel/Threaded Solve

9.11.1   cores

`cores` Number of cores used in parallel ODE solving. This is equivalent to calling [setRxThreads()]

9.11.2   nCoresRV

`nCoresRV` Number of cores used for the simulation of the sigma variables. By default this is 1. To reproduce the results you need to run on the same platform with the same number of cores. This is the reason this is set to be one, regardless of what the number of cores are used in threaded ODE solving.

Chapter 10

RxODE output

10.1 Using RxODE data frames

10.1.1 Creating an interactive data frame

RxODE supports returning a solved object that is a modified data-frame. This is done by the predict(), solve(), or rxSolve() methods.

```
library(RxODE)
library(units)

### Setup example model
mod1 <-RxODE({
  C2 = centr/V2;
  C3 = peri/V3;
  d/dt(depot) =-KA*depot;
  d/dt(centr) = KA*depot - CL*C2 - Q*C2 + Q*C3;
  d/dt(peri)  =                   Q*C2 - Q*C3;
  d/dt(eff)  = Kin - Kout*(1-C2/(EC50+C2))*eff;
})

### Seup parameters and initial conditions

theta <-
  c(KA=2.94E-01, CL=1.86E+01, V2=4.02E+01, # central
    Q=1.05E+01,  V3=2.97E+02,              # peripheral
    Kin=1, Kout=1, EC50=200)              # effects

inits <- c(eff=1)
```

```r
### Setup dosing event information
ev <- eventTable(amount.units="mg", time.units="hours") %>%
  add.dosing(dose=10000, nbr.doses=10, dosing.interval=12) %>%
  add.dosing(dose=20000, nbr.doses=5, start.time=120,
             dosing.interval=24) %>%
  add.sampling(0:240);


### Now solve
x <- predict(mod1,theta, ev, inits)
print(x)
```

```
#> _____ Solved RxODE object _____
#> -- Parameters ($params): -----------------------------------
#>       V2       V3      KA       CL       Q      Kin     Kout     EC50
#>   40.200 297.000    0.294  18.600  10.500   1.000   1.000 200.000
#> -- Initial Conditions ($inits): ----------------------------
#> depot centr  peri    eff
#>     0     0     0      1
#> -- First part of data (object): ----------------------------
#> # A tibble: 241 x 7
#>    time     C2     C3   depot centr   peri    eff
#>     [h]  <dbl>  <dbl>   <dbl> <dbl>  <dbl>  <dbl>
#> 1     0    0    0      10000     0      0    1
#> 2     1  44.4 0.920    7453. 1784.   273.  1.08
#> 3     2  54.9 2.67     5554. 2206.   794.  1.18
#> 4     3  51.9 4.46     4140. 2087.  1324.  1.23
#> 5     4  44.5 5.98     3085. 1789.  1776.  1.23
#> 6     5  36.5 7.18     2299. 1467.  2132.  1.21
#> # ... with 235 more rows
#> _____
```

or

```r
x <- solve(mod1,theta, ev, inits)
print(x)
```

```
#> _____ Solved RxODE object _____
#> -- Parameters ($params): -----------------------------------
#>       V2       V3      KA       CL       Q      Kin     Kout     EC50
#>   40.200 297.000    0.294  18.600  10.500   1.000   1.000 200.000
#> -- Initial Conditions ($inits): ----------------------------
#> depot centr  peri    eff
```

```
#>      0    0    0    1
#> -- First part of data (object): ---------------------------
#> # A tibble: 241 x 7
#>    time    C2    C3  depot centr  peri   eff
#>     [h] <dbl> <dbl>  <dbl> <dbl> <dbl> <dbl>
#> 1     0   0    0    10000     0     0   1
#> 2     1  44.4 0.920  7453. 1784.  273.  1.08
#> 3     2  54.9 2.67   5554. 2206.  794.  1.18
#> 4     3  51.9 4.46   4140. 2087. 1324.  1.23
#> 5     4  44.5 5.98   3085. 1789. 1776.  1.23
#> 6     5  36.5 7.18   2299. 1467. 2132.  1.21
#> # ... with 235 more rows
#>  ------------------------------------------------------------
```

Or with mattigr

```
x <- mod1 %>% solve(theta, ev, inits)
print(x)
```

```
#>  _____ Solved RxODE object _____
#> -- Parameters ($params): ----------------------------------
#>      V2      V3      KA      CL       Q     Kin    Kout    EC50
#>  40.200 297.000   0.294  18.600  10.500   1.000   1.000 200.000
#> -- Initial Conditions ($inits): ---------------------------
#> depot centr  peri   eff
#>      0    0    0    1
#> -- First part of data (object): ---------------------------
#> # A tibble: 241 x 7
#>    time    C2    C3  depot centr  peri   eff
#>     [h] <dbl> <dbl>  <dbl> <dbl> <dbl> <dbl>
#> 1     0   0    0    10000     0     0   1
#> 2     1  44.4 0.920  7453. 1784.  273.  1.08
#> 3     2  54.9 2.67   5554. 2206.  794.  1.18
#> 4     3  51.9 4.46   4140. 2087. 1324.  1.23
#> 5     4  44.5 5.98   3085. 1789. 1776.  1.23
#> 6     5  36.5 7.18   2299. 1467. 2132.  1.21
#> # ... with 235 more rows
#>  ------------------------------------------------------------
```

10.1.2 RxODE solved object properties

10.1.3 Using the solved object as a simple data frame

The solved object acts as a data.frame or tbl that can be filtered by dpylr. For example you could filter it easily.

```r
library(dplyr)
```

```
#>
#> Attaching package: 'dplyr'

#> The following objects are masked from 'package:stats':
#>
#>      filter, lag

#> The following objects are masked from 'package:base':
#>
#>      intersect, setdiff, setequal, union
```

```r
### You can  drop units for comparisons and filtering
x <- mod1 %>% solve(theta,ev,inits) %>%
    drop_units %>% filter(time <= 3) %>% as.tbl
```

```
#> Warning: `as.tbl()` was deprecated in dplyr 1.0.0.
#> Please use `tibble::as_tibble()` instead.
```

```r
### or keep them and compare with the proper units.
x <- mod1 %>% solve(theta,ev,inits) %>%
    filter(time <= set_units(3, hr)) %>% as.tbl
x
```

```
#> # A tibble: 4 x 7
#>    time     C2     C3  depot centr   peri    eff
#>     [h] <dbl> <dbl>  <dbl> <dbl>  <dbl>  <dbl>
#> 1     0    0    0    10000     0      0      1
#> 2     1  44.4 0.920  7453. 1784.  273.   1.08
#> 3     2  54.9 2.67   5554. 2206.  794.   1.18
#> 4     3  51.9 4.46   4140. 2087. 1324.   1.23
```

10.2   Updating the data-set interactively

However it isn't just a simple data object. You can use the solved object to update parameters on the fly, or even change the sampling time.

First we need to recreate the original solved system:

```
x <- mod1 %>% solve(theta,ev,inits);
print(x)
```

```
#> _____ Solved RxODE object _____
#> -- Parameters ($params): ------------------------------------
#>       V2       V3       KA       CL        Q      Kin     Kout     EC50
#>   40.200  297.000    0.294   18.600   10.500    1.000    1.000  200.000
#> -- Initial Conditions ($inits): ---------------------------
#> depot centr  peri    eff
#>     0     0     0      1
#> -- First part of data (object): ---------------------------
#> # A tibble: 241 x 7
#>     time     C2     C3   depot centr   peri     eff
#>      [h] <dbl> <dbl>   <dbl> <dbl>  <dbl>   <dbl>
#> 1     0    0     0     10000     0      0     1
#> 2     1  44.4 0.920   7453.  1784.   273.   1.08
#> 3     2  54.9 2.67    5554.  2206.   794.   1.18
#> 4     3  51.9 4.46    4140.  2087.  1324.   1.23
#> 5     4  44.5 5.98    3085.  1789.  1776.   1.23
#> 6     5  36.5 7.18    2299.  1467.  2132.   1.21
#> # ... with 235 more rows
#> _____
```

10.2.1 Modifying initial conditions

To examine or change initial conditions, you can use the syntax `cmt.0`, `cmt0`, or `cmt_0`. In the case of the `eff` compartment defined by the model, this is:

```
x$eff0
```

```
#> [1] 1
```

which shows the initial condition of the effect compartment. If you wished to change this initial condition to 2, this can be done easily by:

```
x$eff0 <- 2
print(x)
```

```
#> _____ Solved RxODE object _____
#> -- Parameters ($params): ------------------------------------
#>       V2       V3       KA       CL        Q      Kin     Kout     EC50
#>   40.200  297.000    0.294   18.600   10.500    1.000    1.000  200.000
```

```
#> -- Initial Conditions ($inits): ----------------------------
#> depot centr  peri    eff
#>    0     0     0     2
#> -- First part of data (object): ----------------------------
#> # A tibble: 241 x 7
#>   time    C2    C3  depot centr  peri    eff
#>    [h] <dbl> <dbl>  <dbl> <dbl> <dbl> <dbl>
#> 1    0   0   0     10000     0     0   2
#> 2    1  44.4 0.920  7453. 1784.  273.  1.50
#> 3    2  54.9 2.67   5554. 2206.  794.  1.37
#> 4    3  51.9 4.46   4140. 2087. 1324.  1.31
#> 5    4  44.5 5.98   3085. 1789. 1776.  1.27
#> 6    5  36.5 7.18   2299. 1467. 2132.  1.23
#> # ... with 235 more rows
#> _____
```

```
plot(x)
```
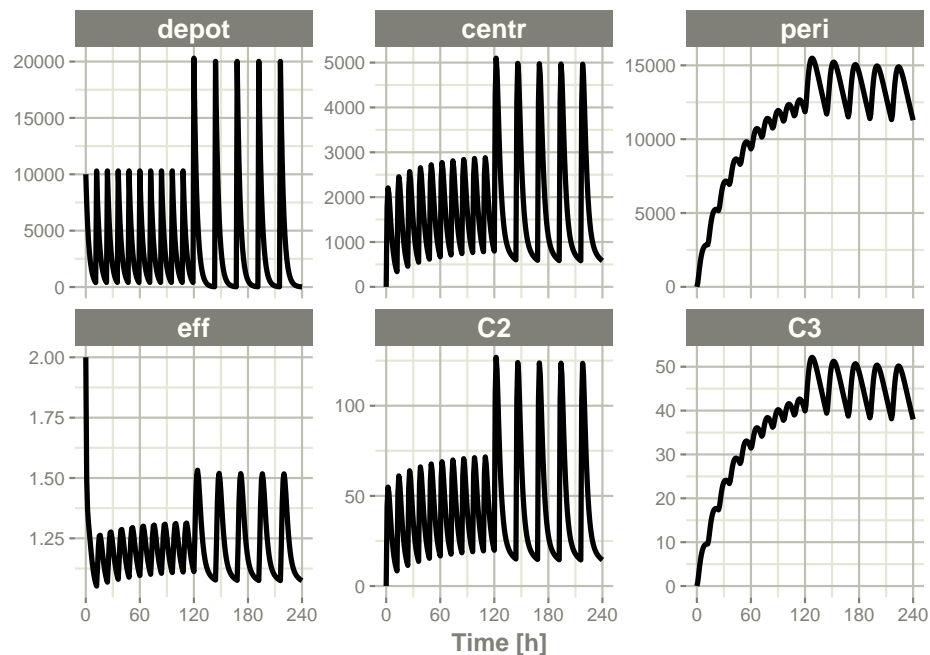

10.2.2   Modifying observation times for RxODE

Notice that the initial effect is now 2.

You can also change the sampling times easily by this method by changing t or time. For example:

```r
x$t <- seq(0,5,length.out=20)
print(x)
```

```
#> _____ Solved RxODE object _____
#> -- Parameters ($params): ----------------------------------
#>      V2      V3      KA      CL       Q     Kin    Kout    EC50
#>   40.200 297.000   0.294  18.600  10.500   1.000   1.000 200.000
#> -- Initial Conditions ($inits): ---------------------------
#> depot centr  peri    eff
#>     0     0     0      2
#> -- First part of data (object): ---------------------------
#> # A tibble: 20 x 7
#>        time    C2      C3   depot centr  peri    eff
#>         [h] <dbl>   <dbl>   <dbl> <dbl> <dbl> <dbl>
#> 1 0.0000000    0   0       10000     0     0     2
#> 2 0.2631579  16.8 0.0817   9255.  677.  24.3  1.79
#> 3 0.5263158  29.5 0.299    8566. 1187.  88.7  1.65
#> 4 0.7894737  38.9 0.615    7929. 1562. 183.   1.55
#> 5 1.0526316  45.5 1.00     7338. 1830. 298.   1.49
#> 6 1.3157895  50.1 1.44     6792. 2013. 427.   1.44
#> # ... with 14 more rows
#> _____
```

```r
plot(x)
```
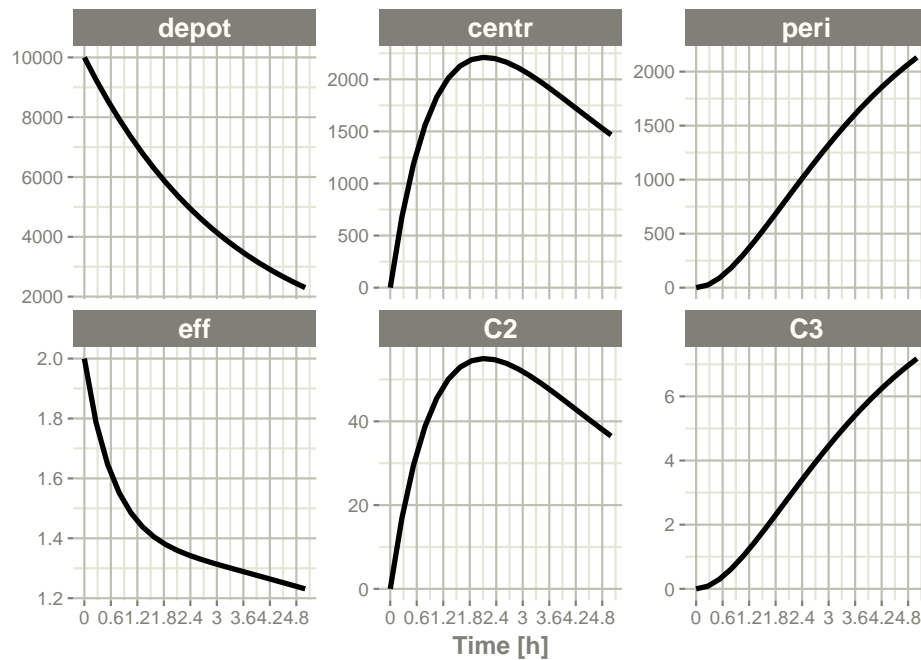
10.2.3   Modifying simulation parameters

You can also access or change parameters by the $ operator. For example, accessing KA can be done by:

```
x$KA
```

```
#> [1] 0.294
```

And you may change it by assigning it to a new value.

```
x$KA <- 1
print(x)
```

```
#> _____ Solved RxODE object _____
#> -- Parameters ($params): ---------------------------------
#>    V2    V3    KA    CL     Q   Kin  Kout  EC50
#>  40.2 297.0   1.0  18.6  10.5   1.0   1.0 200.0
#> -- Initial Conditions ($inits): --------------------------
#> depot centr  peri   eff
#>     0     0     0     2
#> -- First part of data (object): --------------------------
```

```
#> # A tibble: 20 x 7
#>        time    C2    C3  depot centr   peri   eff
#>         [h] <dbl> <dbl>  <dbl> <dbl>  <dbl> <dbl>
#> 1 0.0000000    0   0     10000    0      0    2
#> 2 0.2631579  52.2 0.261  7686. 2098.  77.6  1.82
#> 3 0.5263158  83.3 0.900  5908. 3348.  267.  1.74
#> 4 0.7894737  99.8 1.75   4541. 4010.  519.  1.69
#> 5 1.0526316 106.  2.69   3490. 4273.  800.  1.67
#> 6 1.3157895 106.  3.66   2683. 4272. 1086.  1.64
#> # ... with 14 more rows
#> _____
```

```
plot(x)
```
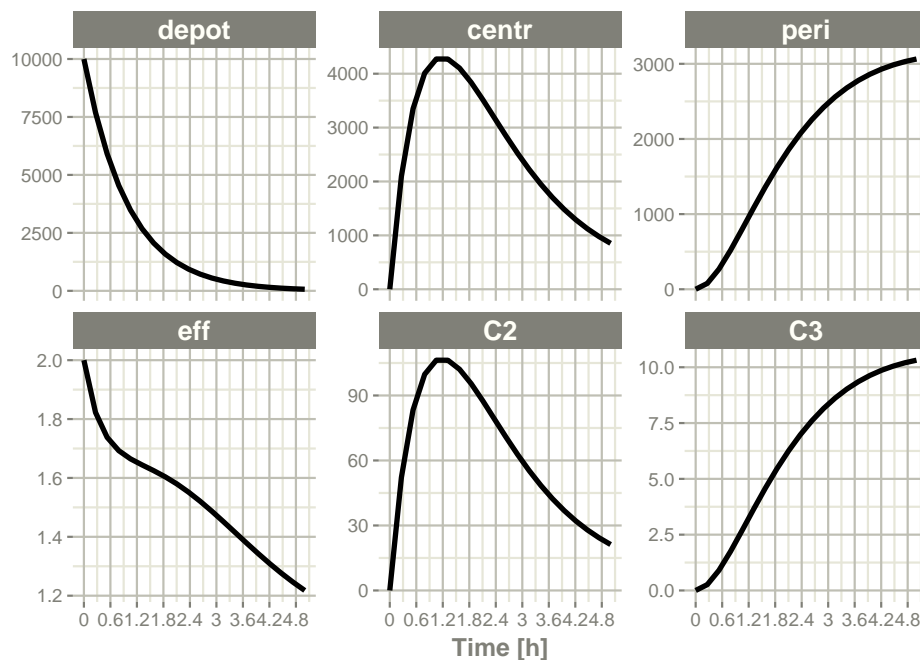

You can access/change all the parameters, initialization(s) or events with the `$params`, `$inits`, `$events` accessor syntax, similar to what is used above.

This syntax makes it easy to update and explore the effect of various parameters on the solved object.

Chapter 11

Simulation

Chapter 12

Examples

This section is for example models to get you started in common simulation scenarios.

12.1 Prediction only models

Prediction only models are simple to create. You use the RxODE syntax without any ODE systems in them. A very simple example is a one-compartment model.

```
library(RxODE)
mod <- RxODE({
    ipre <- 10 * exp(-ke * t);
})
mod
```

```
#> RxODE 1.0.5 model named rx_686859e10a9ee0bacc9a573af33b8219 model (ready).
#> x$params: ke
#> x$lhs: ipre
```

Solving the RxODE models are the same as saving the simple ODE system, but faster of course.

```
et   <- et(seq(0,24,length.out=50))
cmt1 <- rxSolve(mod,et,params=c(ke=0.5))
cmt1
```

```
#> _____ Solved RxODE object _____
```

```
#> -- Parameters (x$params): -----------------------------------
#>  ke
#> 0.5
#> -- Initial Conditions (x$inits): ---------------------------
#> named numeric(0)
#> -- First part of data (object): ----------------------------
#> # A tibble: 50 x 2
#>    time  ipre
#>   <dbl> <dbl>
#> 1 0      10
#> 2 0.490  7.83
#> 3 0.980  6.13
#> 4 1.47   4.80
#> 5 1.96   3.75
#> 6 2.45   2.94
#> # ... with 44 more rows
#> _____
```

12.2   Solved compartment models

Solved models are also simple to create. You simply place the `linCmt()` psuedo-function into your code. The `linCmt()` function figures out the type of model to use based on the parameter names specified.

Most often, pharmacometric models are parameterized in terms of volume and clearances. Clearances are specified by NONMEM-style names of `CL`, `Q`, `Q1`, `Q2`, etc. or distributional clearances `CLD`, `CLD2`. Volumes are specified by Central (`VC` or `V`), Peripheral/Tissue (`VP`, `VT`). While more translations are available, some example translations are below:

```
clLinCmt <- read.csv("../../vignettes/cl-lincmt.csv");
library(DT)
datatable(clLinCmt, rownames = FALSE, filter="top", options = list(pageLength = 5, scr
```

Show 5 ▾ entries   Search: [   ]

par1 ⇕	par2 ⇕	par3 ⇕	par4 ⇕	par5 ⇕	par6 ⇕	par7 ⇕	ncmt ⇕
All	All	All	All	All	All	All	All
ka	cl	q	q2	v	vp	vp2	3
cl	q	q2	v	vp	vp2		3
ka	cl	q	q2	vc	vp	vp2	3
cl	q	q2	vc	vp	vp2		3
ka	cl	q1	q2	v	vp	vp2	3

Showing 1 to 5 of 210 entries   Previous 1 2 3 4 5 ... 42 Next

Another popular parameterization is in terms of micro-constants. RxODE assumes compartment 1 is the central compartment. The elimination constant would be specified by `K`, `Ke` or `Kel`. Some example translations are below:

```
kelLinCmt <- read.csv("../../vignettes/kel-lincmt.csv");
datatable(kelLinCmt, rownames = FALSE, filter="top", options = list(pageLength = 5, scrollX=T) )
```

Show 5 entries   Search:

par1	par2	par3	par4	par5	par6	par7	ncmt
All	All	All	All	All	All	All	All
ka	v	k	k12	k21	k13	k31	3
v	k	k12	k21	k13	k31		3
ka	vc	k	k12	k21	k13	k31	3
vc	k	k12	k21	k13	k31		3
ka	v1	k	k12	k21	k13	k31	3

Showing 1 to 5 of 54 entries   Previous 1 2 3 4 5 ... 11 Next

The last parameterization possible is using `alpha` and `V` and/or `A/B/C`. Some example translations are below:

```
alphaLinCmt <- read.csv("../../vignettes/alpha-lincmt.csv");
datatable(alphaLinCmt, rownames = FALSE, filter="top", options = list(pageLength = 5, scrollX=T)
```

Show 5 entries   Search:

par1	par2	par3	par4	par5	par6	par7	ncmt
All	All	All	All	All	All	All	All
ka	v	alpha	beta	aob			1
v	alpha	beta	aob				1
ka	vc	alpha	beta	aob			1
vc	alpha	beta	aob				1
ka	v1	alpha	beta	aob			1

Showing 1 to 5 of 24 entries   Previous 1 2 3 4 5 Next

Once the `linCmt()` sleuthing is complete, the 1, 2 or 3 compartment model solution is used as the value of `linCmt()`.

The compartments where you can dose in a linear solved system are `depot` and `central` when there is an linear absorption constant in the model `ka`. Without any additional ODEs, these compartments are numbered `depot=1` and `central=2`.

When the absorption constant `ka` is missing, you may only dose to the `central` compartment. Without any additional ODEs the compartment number is `central=1`.

These compartments take the same sort of events that a ODE model can take, and are discussed in the RxODE events vignette.

```
mod <- RxODE({
    ke <- 0.5
    V <- 1
    ipre <- linCmt();
})
mod
```

```
#> RxODE 1.0.5 model named rx_31e0f7e072ff9bae06525cb46024b427 model (ready).
#> x$stateExtra: central
#> x$params: ke, V
#> x$lhs: ipre
```

This then acts as an ODE model; You specify a dose to the depot compartment and
then solve the system:

```
et  <- et(amt=10,time=0,cmt=depot) %>%
    et(seq(0,24,length.out=50))
cmt1 <- rxSolve(mod,et,params=c(ke=0.5))
cmt1
```

```
#> _____ Solved RxODE object _____
#> -- Parameters (x$params): -----------------------------------
#>  ke    V
#> 0.5 1.0
#> -- Initial Conditions (x$inits): --------------------------
#> named numeric(0)
#> -- First part of data (object): ---------------------------
#> # A tibble: 50 x 2
#>    time  ipre
#>   <dbl> <dbl>
#> 1 0      10
#> 2 0.490  7.83
#> 3 0.980  6.13
#> 4 1.47   4.80
#> 5 1.96   3.75
#> 6 2.45   2.94
#> # ... with 44 more rows
#> -----------------------------------------------------------
```

12.3   Mixing Solved Systems and ODEs

In addition to pure ODEs, you may mix solved systems and ODEs. The prior 2-
compartment indirect response model can be simplified with a linCmt() function:

```
library(RxODE)
## Setup example model
mod1 <-RxODE({
    C2 = centr/V2;
    C3 = peri/V3;
    d/dt(depot) =-KA*depot;
    d/dt(centr) = KA*depot - CL*C2 - Q*C2 + Q*C3;
    d/dt(peri)  =                    Q*C2 - Q*C3;
    d/dt(eff)  = Kin - Kout*(1-C2/(EC50+C2))*eff;
});

## Seup parameters and initial conditions

theta <-
    c(KA=2.94E-01, CL=1.86E+01, V2=4.02E+01, # central
      Q=1.05E+01,  V3=2.97E+02,               # peripheral
      Kin=1, Kout=1, EC50=200)                # effects

inits <- c(eff=1);

## Setup dosing event information
ev <- eventTable(amount.units="mg", time.units="hours") %>%
    add.dosing(dose=10000, nbr.doses=10, dosing.interval=12) %>%
    add.dosing(dose=20000, nbr.doses=5, start.time=120,dosing.interval=24) %>%
    add.sampling(0:240);

## Setup a mixed solved/ode system:
mod2 <- RxODE({
    ## the order of variables do not matter, the type of compartmental
    ## model is determined by the parameters specified.
    C2   = linCmt(KA, CL, V2, Q, V3);
    eff(0) = 1  ## This specifies that the effect compartment starts at 1.
    d/dt(eff) =  Kin - Kout*(1-C2/(EC50+C2))*eff;
})
```

This allows the indirect response model above to assign the 2-compartment model to the C2 variable and the used in the indirect response model.

When mixing the solved systems and the ODEs, the solved system's compartment is always the last compartment. This is because the solved system technically isn't a compartment to be solved. Adding the dosing compartment to the end will not interfere with the actual ODE to be solved.

Therefore,in the two-compartment indirect response model, the effect compartment is compartment #1 while the PK dosing compartment for the depot is compartment #2.

This compartment model requires a new event table since the compartment number changed:

```
ev <- eventTable(amount.units='mg', time.units='hours') %>%
    add.dosing(dose=10000, nbr.doses=10, dosing.interval=12,dosing.to=2) %>%
    add.dosing(dose=20000, nbr.doses=5, start.time=120,dosing.interval=24,dosing.to=2)
    add.sampling(0:240);
```

This can be solved with the following command:

```
x <- mod2 %>%  solve(theta, ev)
print(x)
```

```
#> _____ Solved RxODE object _____
#> -- Parameters ($params): -----------------------------------
#>       CL       V2       Q       V3       KA      Kin     Kout      EC50
#>   18.600   40.200   10.500 297.000    0.294    1.000    1.000 200.000
#> -- Initial Conditions ($inits): ----------------------------
#> eff
#>    1
#> -- First part of data (object): ----------------------------
#> # A tibble: 241 x 3
#>    time    C2    eff
#>     [h] <dbl> <dbl>
#> 1     0 249.     1
#> 2     1 121.   1.35
#> 3     2  60.3  1.38
#> 4     3  31.0  1.28
#> 5     4  17.0  1.18
#> 6     5  10.2  1.11
#> # ... with 235 more rows
#> _____
```

Note this solving did not require specifying the effect compartment initial condition to be 1. Rather, this is already pre-specified by eff(0)=1.

This can be solved for different initial conditions easily:

```
x <- mod2 %>%  solve(theta, ev,c(eff=2))
print(x)
```

```
#> _____ Solved RxODE object _____
#> -- Parameters ($params): -----------------------------------
#>       CL       V2       Q       V3       KA      Kin     Kout      EC50
```

```
#>  18.600  40.200  10.500 297.000    0.294   1.000    1.000 200.000
#> -- Initial Conditions ($inits): ----------------------------
#> eff
#>    2
#> -- First part of data (object): ----------------------------
#> # A tibble: 241 x 3
#>    time    C2   eff
#>     [h] <dbl> <dbl>
#> 1     0 249.    2
#> 2     1 121.    1.93
#> 3     2  60.3   1.67
#> 4     3  31.0   1.41
#> 5     4  17.0   1.23
#> 6     5  10.2   1.13
#> # ... with 235 more rows
#> _____
```

The RxODE detective also does not require you to specify the variables in the
`linCmt()` function if they are already defined in the block. Therefore, the following
function will also work to solve the same system.

```
mod3 <- RxODE({
    KA=2.94E-01;
    CL=1.86E+01;
    V2=4.02E+01;
    Q=1.05E+01;
    V3=2.97E+02;
    Kin=1;
    Kout=1;
    EC50=200;
    ## The linCmt() picks up the variables from above
    C2   = linCmt();
    eff(0) = 1   ## This specifies that the effect compartment starts at 1.
    d/dt(eff) =  Kin - Kout*(1-C2/(EC50+C2))*eff;
})

x <- mod3 %>%  solve(ev)
print(x)
```

```
#> _____ Solved RxODE object _____
#> -- Parameters ($params): -----------------------------------
#>      KA     CL     V2      Q     V3    Kin   Kout    EC50
#>   0.294  18.600  40.200  10.500 297.000   1.000   1.000 200.000
#> -- Initial Conditions ($inits): ----------------------------
#> eff
```

```
#>     1
#> -- First part of data (object): ----------------------------
#> # A tibble: 241 x 3
#>     time    C2    eff
#>      [h] <dbl> <dbl>
#> 1      0 249.    1
#> 2      1 121.   1.35
#> 3      2  60.3  1.38
#> 4      3  31.0  1.28
#> 5      4  17.0  1.18
#> 6      5  10.2  1.11
#> # ... with 235 more rows
#>  _____
```

Note that you do not specify the parameters when solving the system since they are built into the model, but you can override the parameters:

```
x <- mod3 %>%  solve(c(KA=10),ev)
print(x)
```

```
#>  _____ Solved RxODE object _____
#> -- Parameters ($params): ----------------------------------
#>    KA    CL    V2    Q    V3   Kin  Kout   EC50
#>  10.0  18.6  40.2  10.5 297.0   1.0   1.0  200.0
#> -- Initial Conditions ($inits): ---------------------------
#> eff
#>     1
#> -- First part of data (object): ----------------------------
#> # A tibble: 241 x 3
#>     time    C2    eff
#>      [h] <dbl> <dbl>
#> 1      0 249.    1
#> 2      1 121.   1.35
#> 3      2  60.3  1.38
#> 4      3  31.0  1.28
#> 5      4  17.0  1.18
#> 6      5  10.2  1.11
#> # ... with 235 more rows
#>  _____
```

12.4   Weight based dosing

This is an example model for weight based dosing of daptomycin. Daptomycin is a cyclic lipopeptide antibiotic from fermented *Streptomyces roseosporus*.

There are 3 stages for weight-based dosing simulations: - Create RxODE model - Simulate Covariates - Create event table with weight-based dosing (merged back to covariates)

12.4.1 Creating a 2-compartment model in RxODE

```
library(RxODE)

## Note the time covariate is not included in the simulation
m1 <- RxODE({
  CL ~ (1-0.2*SEX)*(0.807+0.00514*(CRCL-91.2))*exp(eta.cl)
  V1 ~ 4.8*exp(eta.v1)
  Q ~ (3.46+0.0593*(WT-75.1))*exp(eta.q);
  V2 ~ 1.93*(3.13+0.0458*(WT-75.1))*exp(eta.v2)
  A1 ~ centr;
  A2 ~ peri;
  d/dt(centr) ~ - A1*(CL/V1 + Q/V1) + A2*Q/V2;
  d/dt(peri) ~ A1*Q/V1 - A2*Q/V2;
  DV = centr / V1 * (1 + prop.err)
})
```

12.4.2 Simulating Covariates

This simulation correlates age, sex, and weight. Since we will be using weight based dosing, this needs to be simulated first

```
set.seed(42)
library(dplyr)
nsub=30
### Simulate Weight based on age and gender
AGE<-round(runif(nsub,min=18,max=70))
SEX<-round(runif(nsub,min=0,max=1))
HTm<-round(rnorm(nsub,176.3,0.17*sqrt(4482)),digits=1)
HTf<-round(rnorm(nsub,162.2,0.16*sqrt(4857)),digits=1)
WTm<-round(exp(3.28+1.92*log(HTm/100))*exp(rnorm(nsub,0,0.14)),digits=1)
WTf<-round(exp(3.49+1.45*log(HTf/100))*exp(rnorm(nsub,0,0.17)),digits=1)
WT<-ifelse(SEX==1,WTf,WTm)
CRCL<-round(runif(nsub,30,140))
## id is in lower case to match the event table
cov.df <- tibble(id=seq_along(AGE), AGE=AGE, SEX=SEX, WT=WT, CRCL=CRCL)
print(cov.df)
```

```
#> # A tibble: 30 x 5
#>       id   AGE   SEX    WT  CRCL
#>    <int> <dbl> <dbl> <dbl> <dbl>
#>  1     1    66     1  49.4    83
#>  2     2    67     1  52.5    79
#>  3     3    33     0  97.9    37
#>  4     4    61     1  63.8    66
#>  5     5    51     0  71.8   127
#>  6     6    45     1  69.6   132
#>  7     7    56     0  61      73
#>  8     8    25     0  57.7    47
#>  9     9    52     1  58.7    65
#> 10    10    55     1  73.1    64
#> # ... with 20 more rows
```

12.4.3   Creating weight based event table

```
s<-c(0,0.25,0.5,0.75,1,1.5,seq(2,24,by=1))
s <- lapply(s, function(x){.x <- 0.1 * x; c(x - .x, x + .x)})

e <- et() %>%
    ## Specify the id and weight based dosing from covariate data.frame
    ## This requires RxODE XXX
    et(id=cov.df$id, amt=6*cov.df$WT, rate=6 * cov.df$WT) %>%
    ## Sampling is added for each ID
    et(s) %>%
    as.data.frame %>%
    ## Merge the event table with the covarite information
    merge(cov.df, by="id") %>%
    as_tibble

e
```

```
#> # A tibble: 900 x 12
#>       id   low  time  high cmt          amt  rate  evid   AGE   SEX    WT  CRCL
#>    <int> <dbl> <dbl> <dbl> <chr>      <dbl> <dbl> <int> <dbl> <dbl> <dbl> <dbl>
#> 1      1 0     0     0     (obs)         NA    NA     0    66     1  49.4    83
#> 2      1 NA    0     NA    (default)   296.  296.     1    66     1  49.4    83
#> 3      1 0.225 0.246 0.275 (obs)         NA    NA     0    66     1  49.4    83
#> 4      1 0.45  0.516 0.55  (obs)         NA    NA     0    66     1  49.4    83
#> 5      1 0.675 0.729 0.825 (obs)         NA    NA     0    66     1  49.4    83
#> 6      1 0.9   0.921 1.1   (obs)         NA    NA     0    66     1  49.4    83
```

```
#>  7     1  1.35  1.42   1.65  (obs)      NA    NA    0   66    1  49.4    83
#>  8     1  1.8   1.82   2.2   (obs)      NA    NA    0   66    1  49.4    83
#>  9     1  2.7   2.97   3.3   (obs)      NA    NA    0   66    1  49.4    83
#> 10     1  3.6   3.87   4.4   (obs)      NA    NA    0   66    1  49.4    83
#> # ... with 890 more rows
```

12.4.4 Solving Daptomycin simulation

```
data <- rxSolve(m1, e,
             ## Lotri uses lower-triangular matrix rep. for named matrix
             omega=lotri(eta.cl ~ .306,
                         eta.q ~0.0652,
                         eta.v1 ~.567,
                         eta.v2 ~ .191),
             sigma=lotri(prop.err ~ 0.15),
             addDosing = TRUE, addCov = TRUE)

print(data)
```

```
#> _____ Solved RxODE object _____
#> -- Parameters ($params): -----------------------------------------------------
#> # A tibble: 30 x 5
#>    id    eta.cl   eta.v1    eta.q   eta.v2
#>    <fct>  <dbl>    <dbl>    <dbl>    <dbl>
#>  1 1      0.563   0.580    0.0360 -0.246
#>  2 2      0.0341  0.406   -0.139  -0.481
#>  3 3     -0.447   0.0952  -0.185  -0.249
#>  4 4     -0.988   0.248   -0.131  -0.449
#>  5 5      0.144  -1.14     0.106   0.360
#>  6 6     -0.689   0.407   -0.193  -0.200
#>  7 7     -0.426  -0.706   -0.190  -0.234
#>  8 8     -0.212   0.728    0.335   0.0665
#>  9 9      0.0884 -0.934    0.337   0.154
#> 10 10    -0.557   1.29     0.0163 -0.140
#> # ... with 20 more rows
#> -- Initial Conditions ($inits): ----------------------------------------------
#> centr  peri
#>     0     0
#> -- First part of data (object): ----------------------------------------------
#> # A tibble: 900 x 9
#>       id  evid   cmt   amt  time    DV   SEX    WT  CRCL
#>    <int> <int> <int> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1     1     1     1     1 296.  0     0        1  49.4    83
```

```
#> 2      1      0      NA    NA  0        0        1   49.4    83
#> 3      1      0      NA    NA  0.246  2.32      1   49.4    83
#> 4      1      0      NA    NA  0.516 19.6       1   49.4    83
#> 5      1      0      NA    NA  0.729 23.2       1   49.4    83
#> 6      1      0      NA    NA  0.921 21.1       1   49.4    83
#> # ... with 894 more rows
#> _____
```

```
plot(data, log="y")
```

```
#> Warning in self$trans$transform(x): NaNs produced
```

```
#> Warning: Transformation introduced infinite values in continuous y-axis
```
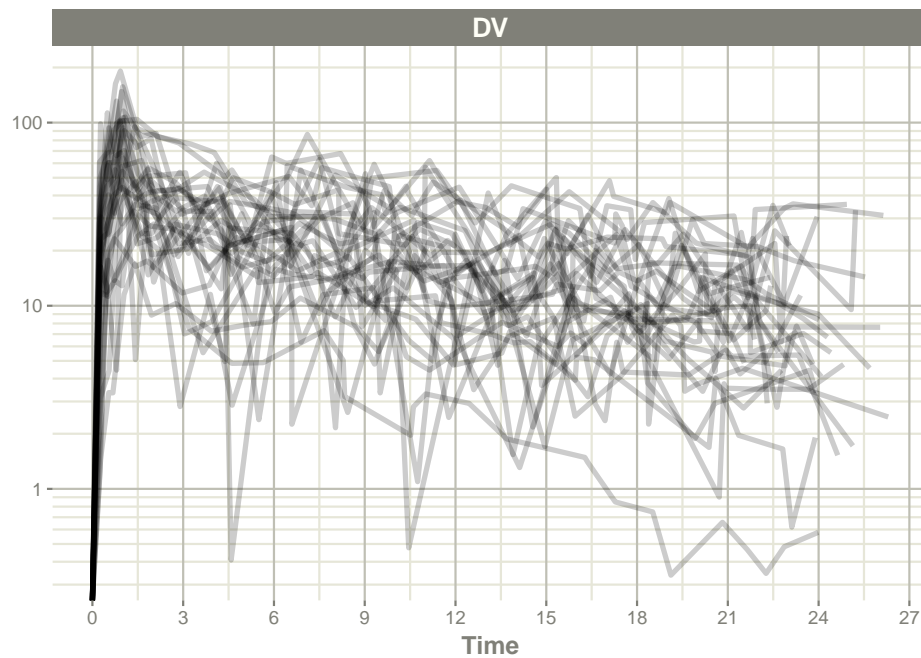

12.4.5   Daptomycin Reference

This weight-based simulation is adapted from the Daptomycin article below:

Dvorchik B, Arbeit RD, Chung J, Liu S, Knebel W, Kastrissios H. Population pharmacokinetics of daptomycin. Antimicrob Agents Che mother 2004; 48: 2799-2807. doi:(10.1128/AAC.48.8.2799-2807.2004)[https://dx.doi.org/10.1128%2FAAC.48.8.2799-2807.2004]

This simulation example was made available from the work of Sherwin Sy with modifications by Matthew Fidler

12.5 Inter-occasion and other nesting examples

More than one level of nesting is possible in RxODE; In this example we will be using the following uncertainties and sources of variability:

Level	Variable	Matrix specified	Integrated Matrix
Model uncertainty	NA	`thetaMat`	`thetaMat`
Investigator	`inv.Cl, inv.Ka`	`omega`	`theta`
Subject	`eta.Cl, eta.Ka`	`omega`	`omega`
Eye	`eye.Cl, eye.Ka`	`omega`	`omega`
Occasion	`iov.Cl, occ.Ka`	`omega`	`omega`
Unexplained Concentration	`prop.sd`	`sigma`	`sigma`
Unexplained Effect	`add.sd`	`sigma`	`sigma`

12.5.1 Event table

This event table contains nesting variables:

- inv: investigator id
- id: subject id
- eye: eye id (left or right)
- occ: occasion

```r
library(RxODE)
library(dplyr)

et(amountUnits="mg", timeUnits="hours") %>%
  et(amt=10000, addl=9,ii=12,cmt="depot") %>%
  et(time=120, amt=2000, addl=4, ii=14, cmt="depot") %>%
  et(seq(0, 240, by=4)) %>% # Assumes sampling when there is no dosing information
  et(seq(0, 240, by=4) + 0.1) %>% ## adds 0.1 for separate eye
  et(id=1:20) %>%
  ## Add an occasion per dose
  mutate(occ=cumsum(!is.na(amt))) %>%
  mutate(occ=ifelse(occ == 0, 1, occ)) %>%
  mutate(occ=2- occ %% 2) %>%
  mutate(eye=ifelse(round(time) == time, 1, 2)) %>%
  mutate(inv=ifelse(id < 10, 1, 2)) %>% as_tibble ->
  ev
```

12.5.2   RxODE model

This creates the `RxODE` model with multi-level nesting. Note the variables `inv.Cl`, `inv.Ka`, `eta.Cl` etc; You only need one variable for each level of nesting.

```r
mod <- RxODE({
  ## Clearance with individuals
  eff(0) = 1
  C2 = centr/V2*(1+prop.sd);
  C3 = peri/V3;
  CL =  TCl*exp(eta.Cl + eye.Cl + iov.Cl + inv.Cl)
  KA = TKA * exp(eta.Ka + eye.Ka + iov.Cl + inv.Ka)
  d/dt(depot) =-KA*depot;
  d/dt(centr) = KA*depot - CL*C2 - Q*C2 + Q*C3;
  d/dt(peri)  =                    Q*C2 - Q*C3;
  d/dt(eff)  = Kin - Kout*(1-C2/(EC50+C2))*eff;
  ef0 = eff + add.sd
})
```

12.5.3   Uncertainty in Model parameters

```r
theta <- c("TKA"=0.294, "TCl"=18.6, "V2"=40.2,
           "Q"=10.5, "V3"=297, "Kin"=1, "Kout"=1, "EC50"=200)

## Creating covariance matrix
tmp <- matrix(rnorm(8^2), 8, 8)
tMat <- tcrossprod(tmp, tmp) / (8 ^ 2)
dimnames(tMat) <- list(names(theta), names(theta))

tMat
```

```
#>                TKA          TCl          V2            Q           V3
#> TKA    0.084901793  0.008491535 -0.028211905 -0.083460075  0.058187918
#> TCl    0.008491535  0.074011602 -0.047388281 -0.049711532  0.009450000
#> V2    -0.028211905 -0.047388281  0.094485793 -0.001142105 -0.031424916
#> Q     -0.083460075 -0.049711532 -0.001142105  0.271685090 -0.042012559
#> V3     0.058187918  0.009450000 -0.031424916 -0.042012559  0.089992792
#> Kin   -0.046169545 -0.016205743 -0.014705754  0.094478654  0.001534101
#> Kout  -0.023946567 -0.040201822  0.008377087  0.062087105 -0.053276590
#> EC50  -0.017733886 -0.011145761 -0.004038295  0.025496213  0.006360558
#>                Kin         Kout         EC50
#> TKA   -0.046169545 -0.023946567 -0.017733886
#> TCl   -0.016205743 -0.040201822 -0.011145761
```

```
#> V2   -0.014705754  0.008377087 -0.004038295
#> Q     0.094478654  0.062087105  0.025496213
#> V3    0.001534101 -0.053276590  0.006360558
#> Kin   0.102223150  0.035112776  0.030704006
#> Kout  0.035112776  0.109746574  0.019060142
#> EC50  0.030704006  0.019060142  0.030188688
```

12.5.4 Nesting Variability

To specify multiple levels of nesting, you can specify it as a nested `lotri` matrix; When using this approach you use the condition operator | to specify what variable nesting occurs on; For the Bayesian simulation we need to specify how much information we have for each parameter; For RxODE this is the `nu` parameter.

In this case: - id, nu=100 or the model came from 100 subjects - eye, nu=200 or the model came from 200 eyes - occ, nu=200 or the model came from 200 occasions - inv, nu=10 or the model came from 10 investigators

To specify this in `lotri` you can use | `var(nu=X)`, or:

```
omega <- lotri(lotri(eta.Cl ~ 0.1,
                     eta.Ka ~ 0.1) | id(nu=100),
               lotri(eye.Cl ~ 0.05,
                     eye.Ka ~ 0.05) | eye(nu=200),
               lotri(iov.Cl ~ 0.01,
                     iov.Ka ~ 0.01) | occ(nu=200),
               lotri(inv.Cl ~ 0.02,
                     inv.Ka ~ 0.02) | inv(nu=10))
omega
```

```
#> $id
#>        eta.Cl eta.Ka
#> eta.Cl    0.1    0.0
#> eta.Ka    0.0    0.1
#>
#> $eye
#>        eye.Cl eye.Ka
#> eye.Cl   0.05   0.00
#> eye.Ka   0.00   0.05
#>
#> $occ
#>        iov.Cl iov.Ka
#> iov.Cl   0.01   0.00
#> iov.Ka   0.00   0.01
#>
```

```
#> $inv
#>         inv.Cl inv.Ka
#> inv.Cl   0.02   0.00
#> inv.Ka   0.00   0.02
#>
#> Properties: nu
```

12.5.5   Unexplained variability

The last piece of variability to specify is the unexplained variability

```
sigma <- lotri(prop.sd ~ .25,
               add.sd~ 0.125)
```

12.5.6   Solving the problem

```
s <- rxSolve(mod, theta, ev,
             thetaMat=tMat, omega=omega,
             sigma=sigma, sigmaDf=400,
             nStud=400)
```

```
#> unhandled error message: EE:[lsoda] 70000 steps taken before reaching tout
#>  @(lsoda.c:748
```

```
#> Warning: some ID(s) could not solve the ODEs correctly; These values are
#> replaced with 'NA'
```

```
print(s)
```

```
#> _____ Solved RxODE object _____
#> -- Parameters ($params): --------------------------------------------------
#> # A tibble: 8,000 x 24
#>    sim.id id    `inv.Cl(inv==1)` `inv.Cl(inv==2)` `inv.Ka(inv==1)`
#>     <int> <fct>            <dbl>            <dbl>            <dbl>
#> 1       1 1               0.0186            0.116            0.188
#> 2       1 2               0.0186            0.116            0.188
#> 3       1 3               0.0186            0.116            0.188
#> 4       1 4               0.0186            0.116            0.188
#> 5       1 5               0.0186            0.116            0.188
#> 6       1 6               0.0186            0.116            0.188
#> 7       1 7               0.0186            0.116            0.188
```

```
#>  8      1 8                0.0186              0.116               0.188
#>  9      1 9                0.0186              0.116               0.188
#> 10      1 10               0.0186              0.116               0.188
#> # ... with 7,990 more rows, and 19 more variables: inv.Ka(inv==2) <dbl>,
#> #   eye.Cl(eye==1) <dbl>, eye.Cl(eye==2) <dbl>, eye.Ka(eye==1) <dbl>,
#> #   eye.Ka(eye==2) <dbl>, iov.Cl(occ==1) <dbl>, iov.Cl(occ==2) <dbl>,
#> #   iov.Ka(occ==1) <dbl>, iov.Ka(occ==2) <dbl>, V2 <dbl>, V3 <dbl>, TCl <dbl>,
#> #   eta.Cl <dbl>, TKA <dbl>, eta.Ka <dbl>, Q <dbl>, Kin <dbl>, Kout <dbl>,
#> #   EC50 <dbl>
#> -- Initial Conditions ($inits): -----------------------------------------------
#> depot centr  peri   eff
#>     0     0     0     1
#>
#> Simulation with uncertainty in:
#> * parameters (s$thetaMat for changes)
#> * omega matrix (s$omegaList)
#>
#> -- First part of data (object): -----------------------------------------------
#> # A tibble: 976,000 x 18
#>   sim.id    id  time inv.Cl inv.Ka eye.Cl eye.Ka iov.Cl  iov.Ka    C2      C3
#>    <int> <int>   [h]  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>   <dbl> <dbl>   <dbl>
#> 1      1     1   0.0 0.0186  0.188 -0.104 0.0786 0.0247 -0.0530     0       0
#> 2      1     1   0.1 0.0186  0.188 -0.296 0.0588 0.0247 -0.0530  3.00  0.0281
#> 3      1     1   4.0 0.0186  0.188 -0.104 0.0786 0.0247 -0.0530  22.4    9.90
#> 4      1     1   4.1 0.0186  0.188 -0.296 0.0588 0.0247 -0.0530  57.9    10.1
#> 5      1     1   8.0 0.0186  0.188 -0.104 0.0786 0.0247 -0.0530  20.1    12.4
#> 6      1     1   8.1 0.0186  0.188 -0.296 0.0588 0.0247 -0.0530  15.6    12.4
#> # ... with 975,994 more rows, and 7 more variables: CL <dbl>, KA <dbl>,
#> #   ef0 <dbl>, depot <dbl>, centr <dbl>, peri <dbl>, eff <dbl>
#> -------------------------------------------------------------------------------
```

There are multiple investigators in a study; Each investigator has a number of individuals enrolled at their site. RxODE automatically determines the number of investigators and then will simulate an effect for each investigator. With the output, inv.Cl(inv==1) will be the inv.Cl for investigator 1, inv.Cl(inv==2) will be the inv.Cl for investigator 2, etc.

inv.Cl(inv==1), inv.Cl(inv==2), etc will be simulated for each study and then combined to form the between investigator variability. In equation form these represent the following:

```
inv.Cl = (inv == 1) * `inv.Cl(inv==1)` + (inv == 2) * `inv.Cl(inv==2)`
```

If you look at the simulated parameters you can see inv.Cl(inv==1) and inv.Cl(inv==2) are in the s$params; They are the same for each study:

```r
print(head(s$params))
```

```
#>   sim.id id inv.Cl(inv==1) inv.Cl(inv==2) inv.Ka(inv==1) inv.Ka(inv==2)
#> 1      1  1     0.01864161      0.1159198      0.1878234      -0.292727
#> 2      1  2     0.01864161      0.1159198      0.1878234      -0.292727
#> 3      1  3     0.01864161      0.1159198      0.1878234      -0.292727
#> 4      1  4     0.01864161      0.1159198      0.1878234      -0.292727
#> 5      1  5     0.01864161      0.1159198      0.1878234      -0.292727
#> 6      1  6     0.01864161      0.1159198      0.1878234      -0.292727
#>   eye.Cl(eye==1) eye.Cl(eye==2) eye.Ka(eye==1) eye.Ka(eye==2) iov.Cl(occ==1)
#> 1    -0.10410790    -0.29632211     0.07855205     0.05884539     0.02466606
#> 2    -0.06820792     0.06585538     0.34603340     0.20141355    -0.06976537
#> 3    -0.04885836     0.13135196    -0.13256387     0.21645151    -0.03121109
#> 4     0.20667975    -0.08775327    -0.01404241    -0.04239568    -0.08207797
#> 5     0.04877033    -0.22890756    -0.21685969     0.04846680    -0.01393029
#> 6     0.19830134    -0.33204702    -0.16960164     0.06823678    -0.17462695
#>   iov.Cl(occ==2) iov.Ka(occ==1) iov.Ka(occ==2)       V2       V3      TCl
#> 1    -0.11264526   -0.052971164   -0.106088075 39.57653 297.3736 18.81115
#> 2     0.03970507   -0.073742566    0.090882718 39.57653 297.3736 18.81115
#> 3    -0.23892944   -0.136470596    0.067412442 39.57653 297.3736 18.81115
#> 4    -0.02134625   -0.061910605   -0.072601879 39.57653 297.3736 18.81115
#> 5     0.05580236    0.099876044   -0.094708943 39.57653 297.3736 18.81115
#> 6    -0.03152016   -0.002074008   -0.004758332 39.57653 297.3736 18.81115
#>         eta.Cl       TKA      eta.Ka        Q      Kin      Kout     EC50
#> 1 -0.02740884 0.4375889  0.07529548 10.97588 1.179938 0.9161172 200.2625
#> 2 -0.11896272 0.4375889  0.07490355 10.97588 1.179938 0.9161172 200.2625
#> 3 -0.61026874 0.4375889 -0.15964154 10.97588 1.179938 0.9161172 200.2625
#> 4 -0.17447915 0.4375889 -0.19377239 10.97588 1.179938 0.9161172 200.2625
#> 5  0.26213020 0.4375889 -0.38954283 10.97588 1.179938 0.9161172 200.2625
#> 6 -0.22932331 0.4375889 -0.49123723 10.97588 1.179938 0.9161172 200.2625
```

```r
print(head(s$params %>% filter(sim.id == 2)))
```

```
#>   sim.id id inv.Cl(inv==1) inv.Cl(inv==2) inv.Ka(inv==1) inv.Ka(inv==2)
#> 1      2  1    -0.01105301     -0.1209402      0.3370577      0.0902204
#> 2      2  2    -0.01105301     -0.1209402      0.3370577      0.0902204
#> 3      2  3    -0.01105301     -0.1209402      0.3370577      0.0902204
#> 4      2  4    -0.01105301     -0.1209402      0.3370577      0.0902204
#> 5      2  5    -0.01105301     -0.1209402      0.3370577      0.0902204
#> 6      2  6    -0.01105301     -0.1209402      0.3370577      0.0902204
#>   eye.Cl(eye==1) eye.Cl(eye==2) eye.Ka(eye==1) eye.Ka(eye==2) iov.Cl(occ==1)
#> 1    -0.01262553    -0.08161227   -0.238499594     0.17178813     0.08330981
#> 2    -0.06778157     0.29410669   -0.003700213    -0.03805489    -0.12869095
#> 3     0.06059738    -0.16831575   -0.085582067     0.22970053     0.05711749
```

```
#> 4    -0.13086494      0.02748735   -0.056454551    -0.23331112      0.07216869
#> 5    -0.23416424     -0.13568099   -0.436719663    -0.03106162     -0.13191139
#> 6     0.24092815      0.66166495   -0.345840539     0.13552870      0.03987511
#>   iov.Cl(occ==2) iov.Ka(occ==1) iov.Ka(occ==2)      V2        V3       TCl
#> 1   -0.148592318     0.10100830    0.050123275 40.32538 296.5826 18.65152
#> 2   -0.002200205    -0.04045931   -0.077835601 40.32538 296.5826 18.65152
#> 3   -0.185116411     0.02903611    0.076384740 40.32538 296.5826 18.65152
#> 4    0.101902663     0.04680555    0.054662894 40.32538 296.5826 18.65152
#> 5    0.103696663     0.02589958    0.100946619 40.32538 296.5826 18.65152
#> 6    0.023244426    -0.03067335   -0.009347601 40.32538 296.5826 18.65152
#>       eta.Cl      TKA     eta.Ka       Q     Kin     Kout     EC50
#> 1 -0.2518665 -0.2160777  0.33092617 11.02462 1.122102 1.022177 199.9981
#> 2  0.3495104 -0.2160777 -0.35774607 11.02462 1.122102 1.022177 199.9981
#> 3 -0.3101379 -0.2160777 -0.09014428 11.02462 1.122102 1.022177 199.9981
#> 4 -0.1665144 -0.2160777 -0.11974060 11.02462 1.122102 1.022177 199.9981
#> 5  0.3184297 -0.2160777 -0.06982612 11.02462 1.122102 1.022177 199.9981
#> 6 -0.1216137 -0.2160777  0.30275205 11.02462 1.122102 1.022177 199.9981
```

For between eye variability and between occasion variability each individual sim-
ulates a number of variables that become the between eye and between occasion
variability; In the case of the eye:

```
eye.Cl = (eye == 1) * `eye.Cl(eye==1)` + (eye == 2) * `eye.Cl(eye==2)`
```

So when you look the simulation each of these variables (ie `eye.Cl(eye==1)`,
`eye.Cl(eye==2)`, etc) they change for each individual and when combined make
the between eye variability or the between occasion variability that can be seen in
some pharamcometric models.

Chapter 13

Advanced & Miscellaneous Topics

This covers advanced or miscellaneous topics in `RxODE`

13.1 Using RxODE with a pipeline

13.1.1 Setting up the RxODE model for the pipeline

In this example we will show how to use RxODE in a simple pipeline.

We can start with a model that can be used for the different simulation workflows that RxODE can handle:

```
library(RxODE)

Ribba2012 <- RxODE({
    k = 100

    tkde = 0.24
    eta.tkde = 0
    kde ~ tkde*exp(eta.tkde)

    tkpq = 0.0295
    eta.kpq = 0
    kpq ~ tkpq * exp(eta.kpq)

    tkqpp = 0.0031
```

```
    eta.kqpp = 0
    kqpp ~ tkqpp * exp(eta.kqpp)

    tlambdap = 0.121
    eta.lambdap = 0
    lambdap ~ tlambdap*exp(eta.lambdap)

    tgamma = 0.729
    eta.gamma = 0
    gamma ~ tgamma*exp(eta.gamma)

    tdeltaqp = 0.00867
    eta.deltaqp = 0
    deltaqp ~ tdeltaqp*exp(eta.deltaqp)

    prop.err <- 0
    pstar <- (pt+q+qp)*(1+prop.err)
    d/dt(c) = -kde * c
    d/dt(pt) = lambdap * pt *(1-pstar/k) + kqpp*qp -
        kpq*pt - gamma*c*kde*pt
    d/dt(q) = kpq*pt -gamma*c*kde*q
    d/dt(qp) = gamma*c*kde*q - kqpp*qp - deltaqp*qp
#### initial conditions
    tpt0 = 7.13
    eta.pt0 = 0
    pt0 ~ tpt0*exp(eta.pt0)
    tq0 = 41.2
    eta.q0 = 0
    q0 ~ tq0*exp(eta.q0)
    pt(0) = pt0
    q(0) = q0
})
```

This is a tumor growth model described in Ribba 2012. In this case, we compiled the model into an R object Ribba2012, though in an RxODE simulation pipeline, you do not *have* to assign the compiled model to any object, though I think it makes sense.
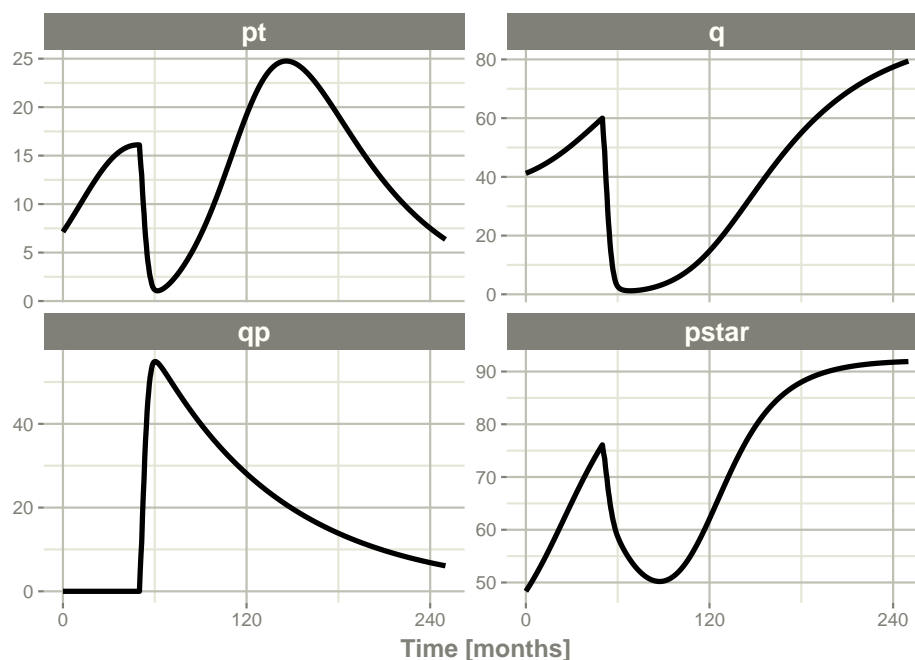
13.1.2   Simulating one event table

Simulating a single event table is quite simple:

- You pipe the RxODE simulation object into an event table object by et().

- When the events are completely specified, you simply solve the ODE system with `rxSolve()`.
- In this case you can pipe the output to `plot()` to conveniently view the results.
- Note for the plot we are only selecting the selecting following:
    - `pt` (Proliferative Tissue),
    - `q` (quiescent tissue)
    - `qp` (DNA-Damaged quiescent tissue) and
    - `pstar` (total tumor tissue)

```
Ribba2012 %>% # Use RxODE
    et(time.units="months") %>% # Pipe to a new event table
    et(amt=1, time=50, until=58, ii=1.5) %>% # Add dosing every 1.5 months
    et(0, 250, by=0.5) %>% # Add some sampling times (not required)
    rxSolve() %>% # Solve the simulation
    plot(pt, q, qp, pstar) # Plot it, plotting the variables of interest
```


13.1.3 Simulating multiple subjects from a single event table

13.1.3.1 Simulating with between subject variability

The next sort of simulation that may be useful is simulating multiple patients with the same treatments. In this case, we will use the `omega` matrix specified by the

paper:

```
#### Add CVs from paper for individual simulation
#### Uses exact formula:

lognCv = function(x){log((x/100)^2+1)}

library(lotri)
#### Now create omega matrix
#### I'm using lotri to quickly specify names/diagonals
omega <- lotri(eta.pt0 ~ lognCv(94),
               eta.q0 ~ lognCv(54),
               eta.lambdap ~ lognCv(72),
               eta.kqp ~ lognCv(76),
               eta.qpp ~ lognCv(97),
               eta.deltaqp ~ lognCv(115),
               eta.kde ~ lognCv(70))

omega
```

```
#>                 eta.pt0      eta.q0 eta.lambdap    eta.kqp    eta.qpp eta.deltaqp
#> eta.pt0       0.6331848 0.0000000   0.0000000 0.0000000 0.0000000   0.0000000
#> eta.q0        0.0000000 0.2558818   0.0000000 0.0000000 0.0000000   0.0000000
#> eta.lambdap   0.0000000 0.0000000   0.4176571 0.0000000 0.0000000   0.0000000
#> eta.kqp       0.0000000 0.0000000   0.0000000 0.4559047 0.0000000   0.0000000
#> eta.qpp       0.0000000 0.0000000   0.0000000 0.0000000 0.6631518   0.0000000
#> eta.deltaqp   0.0000000 0.0000000   0.0000000 0.0000000 0.0000000   0.8426442
#> eta.kde       0.0000000 0.0000000   0.0000000 0.0000000 0.0000000   0.0000000
#>                 eta.kde
#> eta.pt0       0.0000000
#> eta.q0        0.0000000
#> eta.lambdap   0.0000000
#> eta.kqp       0.0000000
#> eta.qpp       0.0000000
#> eta.deltaqp   0.0000000
#> eta.kde       0.3987761
```

With this information, it is easy to simulate 3 subjects from the model-based parameters:

```
set.seed(1089)
Ribba2012 %>% # Use RxODE
    et(time.units="months") %>% # Pipe to a new event table
    et(amt=1, time=50, until=58, ii=1.5) %>% # Add dosing every 1.5 months
    et(0, 250, by=0.5) %>% # Add some sampling times (not required)
```

```
rxSolve(nSub=3, omega=omega) %>% # Solve the simulation
plot(pt, q, qp, pstar) # Plot it, plotting the variables of interest
```
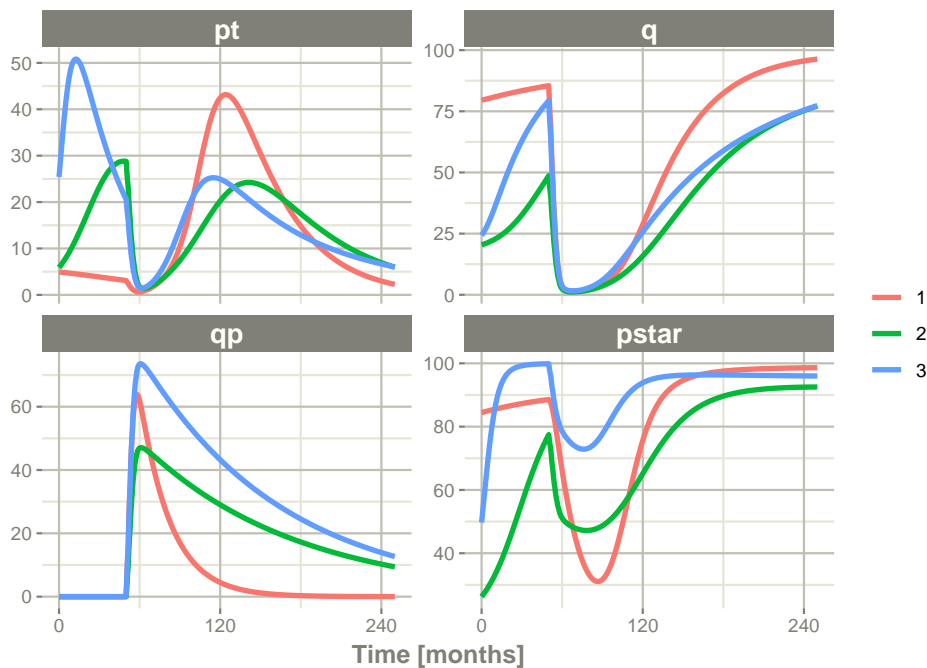

Note there are two different things that were added to this simulation: - nSub to specify how many subjects are in the model - omega to specify the between subject variability.

13.1.3.2   Simulation with unexplained variability

You can even add unexplained variability quite easily:

```
Ribba2012 %>% # Use RxODE
    et(time.units="months") %>% # Pipe to a new event table
    et(amt=1, time=50, until=58, ii=1.5) %>% # Add dosing every 1.5 months
     et(0, 250, by=0.5) %>% # Add some sampling times (not required)
    rxSolve(nSub=3, omega=omega, sigma=lotri(prop.err ~ 0.05^2)) %>% # Solve the simulation
    plot(pt, q, qp, pstar) # Plot it, plotting the variables of interest
```
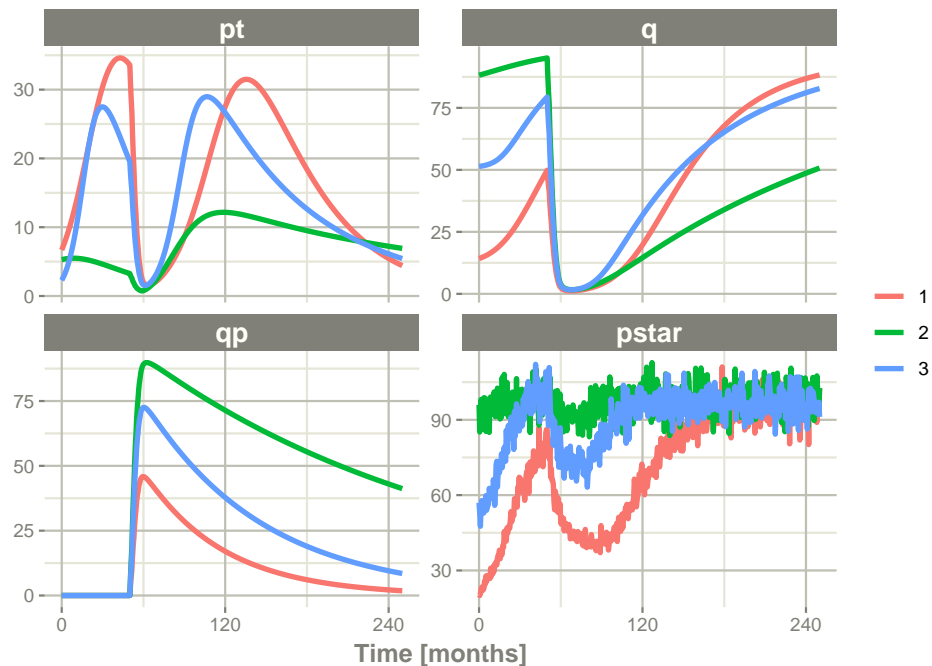
In this case we only added the `sigma` matrix to have unexplained variability on the `pstar` or total tumor tissue.

You can even simulate with uncertainty in the `theta` `omega` and `sigma` values if you wish.

13.1.3.3   Simulation with uncertainty in all the parameters (by matrices)

If we assume these parameters came from 95 subjects with 8 observations apiece, the degrees of freedom for the omega matrix would be 95, and the degrees of freedom of the `sigma` matrix would be 95*8=760 because 95 items informed the `omega` matrix, and 760 items informed the `sigma` matrix.

```
Ribba2012 %>% # Use RxODE
    et(time.units="months") %>% # Pipe to a new event table
    et(amt=1, time=50, until=58, ii=1.5) %>% # Add dosing every 1.5 months
    et(0, 250, by=0.5) %>% # Add some sampling times (not required)
    rxSolve(nSub=3, nStud=3, omega=omega, sigma=lotri(prop.err ~ 0.05^2),
            dfSub=760, dfObs=95) %>% # Solve the simulation
    plot(pt, q, qp, pstar) # Plot it, plotting the variables of interest
```
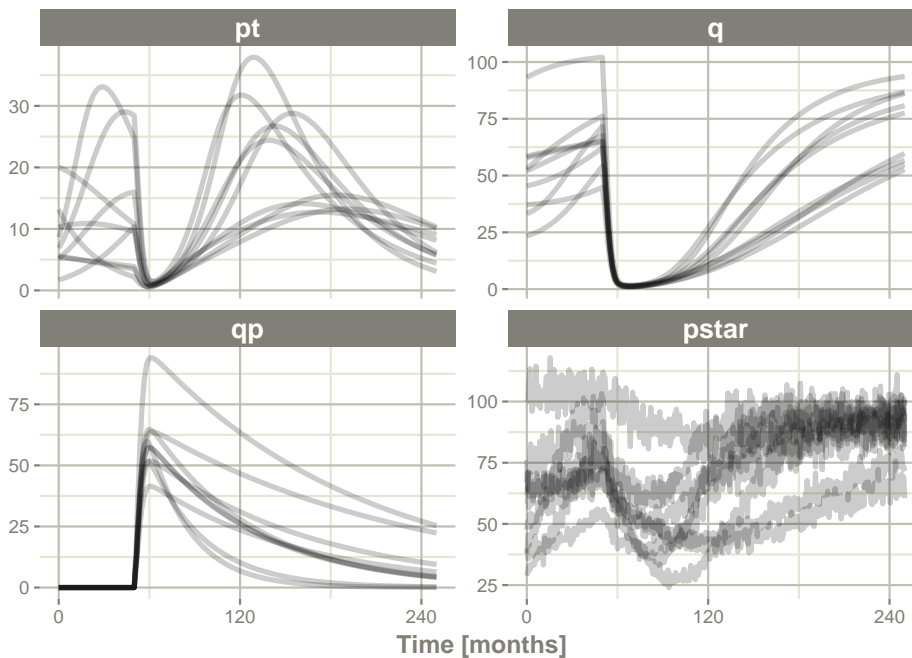
Often in simulations we have a full covariance matrix for the fixed effect parameters. In this case, we do not have the matrix, but it could be specified by `thetaMat`.

While we do not have a full covariance matrix, we can have information about the diagonal elements of the covariance matrix from the model paper. These can be converted as follows:

```
rseVar <- function(est, rse){
    return(est*rse/100)^2
}

thetaMat <- lotri(tpt0 ~ rseVar(7.13,25),
                  tq0 ~ rseVar(41.2,7),
                  tlambdap ~ rseVar(0.121, 16),
                  tkqpp ~ rseVar(0.0031, 35),
                  tdeltaqp ~ rseVar(0.00867, 21),
                  tgamma ~ rseVar(0.729, 37),
                  tkde ~ rseVar(0.24, 33)
                  );

thetaMat
```

```
#>            tpt0    tq0 tlambdap    tkqpp tdeltaqp  tgamma   tkde
#> tpt0     1.7825 0.000  0.00000 0.000000 0.0000000 0.00000 0.0000
#> tq0      0.0000 2.884  0.00000 0.000000 0.0000000 0.00000 0.0000
```

```
#> tlambdap 0.0000 0.000  0.01936 0.000000 0.0000000 0.00000 0.0000
#> tkqpp    0.0000 0.000  0.00000 0.001085 0.0000000 0.00000 0.0000
#> tdeltaqp 0.0000 0.000  0.00000 0.000000 0.0018207 0.00000 0.0000
#> tgamma   0.0000 0.000  0.00000 0.000000 0.0000000 0.26973 0.0000
#> tkde     0.0000 0.000  0.00000 0.000000 0.0000000 0.00000 0.0792
```

Now we have a `thetaMat` to represent the uncertainty in the `theta` matrix, as well as the other pieces in the simulation. Typically you can put this information into your simulation with the `thetaMat` matrix.

With such large variability in `theta` it is easy to sample a negative rate constant, which does not make sense. For example:

```
Ribba2012 %>% # Use RxODE
et(time.units="months") %>% # Pipe to a new event table
et(amt=1, time=50, until=58, ii=1.5) %>% # Add dosing every 1.5 months
et(0, 250, by=0.5) %>% # Add some sampling times (not required)
rxSolve(nSub=2, nStud=2, omega=omega, sigma=lotri(prop.err ~ 0.05^2),
thetaMat=thetaMat,
dfSub=760, dfObs=95) %>% # Solve the simulation
plot(pt, q, qp, pstar) # Plot it, plotting the variables of interest

#> unhandled error message: EE:[lsoda] 70000 steps taken before reaching tout
#> @(lsoda.c:750
#> Warning message:
#> In rxSolve_(object, .ctl, .nms, .xtra, params, events, inits, setupOnly = .setupOnly
#>  Some ID(s) could not solve the ODEs correctly; These values are replaced with NA.
```

To correct these problems you simply need to use a truncated multivariate normal and specify the reasonable ranges for the parameters. For `theta` this is specified by `thetaLower` and `thetaUpper`. Similar parameters are there for the other matrices: `omegaLower`, `omegaUpper`, `sigmaLower` and `sigmaUpper`. These may be named vectors, one numeric value, or a numeric vector matching the number of parameters specified in the `thetaMat` matrix.

In this case the simulation simply has to be modified to have `thetaLower=0` to make sure all rates are positive:
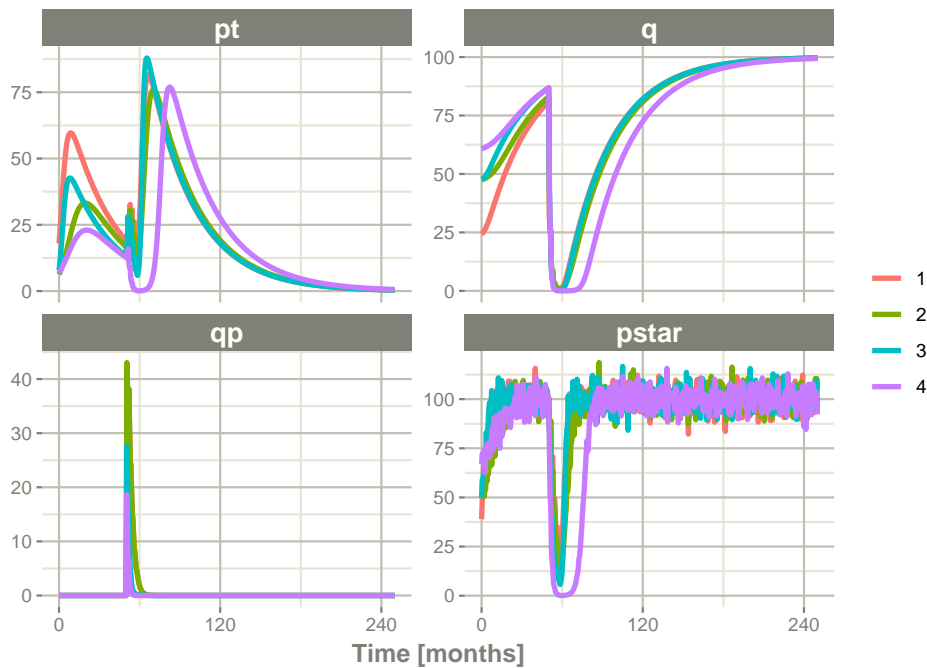
```
Ribba2012 %>% # Use RxODE
    et(time.units="months") %>% # Pipe to a new event table
    et(amt=1, time=50, until=58, ii=1.5) %>% # Add dosing every 1.5 months
    et(0, 250, by=0.5) %>% # Add some sampling times (not required)
    rxSolve(nSub=2, nStud=2, omega=omega, sigma=lotri(prop.err ~ 0.05^2),
            thetaMat=thetaMat,
            thetaLower=0, # Make sure the rates are reasonable
            dfSub=760, dfObs=95) %>% # Solve the simulation
    plot(pt, q, qp, pstar) # Plot it, plotting the variables of interest
```

13.1.4 Summarizing the simulation output

While it is easy to use `dplyr` and `data.table` to perform your own summary of
simulations, `RxODE` also provides this ability by the `confint` function.

```r
#### This takes a little more time; Most of the time is the summary
#### time.

sim0 <- Ribba2012 %>% # Use RxODE
    et(time.units="months") %>% # Pipe to a new event table
    et(amt=1, time=50, until=58, ii=1.5) %>% # Add dosing every 1.5 months
    et(0, 250, by=0.5) %>% # Add some sampling times (not required)
    rxSolve(nSub=10, nStud=10, omega=omega, sigma=lotri(prop.err ~ 0.05^2),
            thetaMat=thetaMat,
            thetaLower=0, # Make sure the rates are reasonable
            dfSub=760, dfObs=95) %>% # Solve the simulation
    confint(c("pt","q","qp","pstar"),level=0.90); # Create Simulation intervals
```

```
#> summarizing data...done
```

```
sim0 %>% plot() # Plot the simulation intervals
```
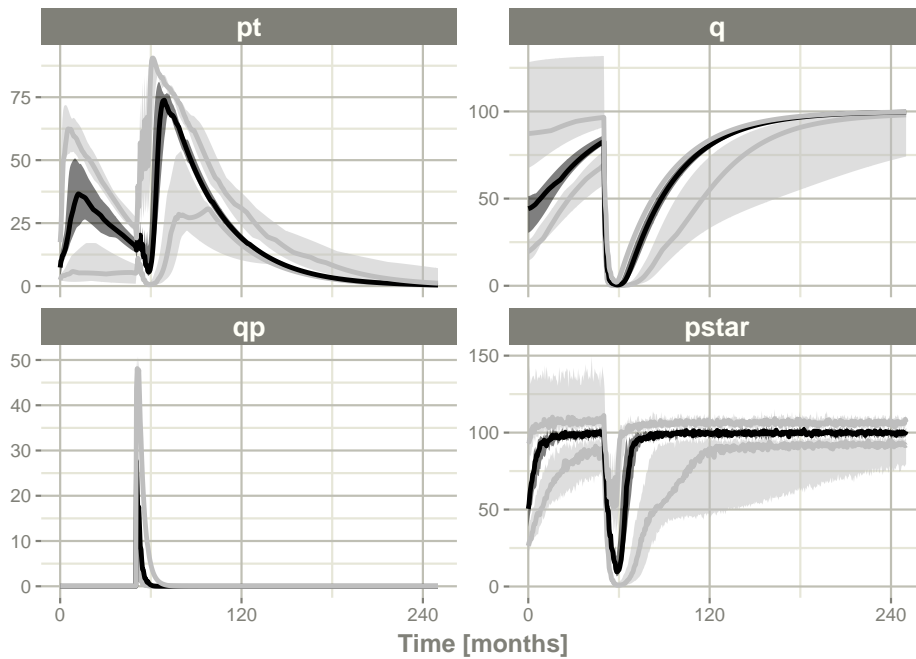

13.1.4.1 Simulating from a data-frame of parameters

While the simulation from matrices can be very useful and a fast way to simu-
late information, sometimes you may want to simulate more complex scenarios.
For instance, there may be some reason to believe that `tkde` needs to be above
`tlambdap`, therefore these need to be simulated more carefully. You can generate
the data frame in whatever way you want. The internal method of simulating the
new parameters is exported too.

```r
library(dplyr)
pars <- rxInits(Ribba2012);
pars <- pars[regexpr("(prop|eta)",names(pars)) == -1]
print(pars)
```

```
#>        k      tkde      tkpq     tkqpp  tlambdap   tgamma  tdeltaqp       tpt0
#> 1.00e+02 2.40e-01 2.95e-02 3.10e-03 1.21e-01 7.29e-01 8.67e-03 7.13e+00
#>      tq0
#> 4.12e+01
```

```
#### This is the exported method for simulation of Theta/Omega internally in RxODE
df <- rxSimThetaOmega(params=pars, omega=omega,dfSub=760,
                      thetaMat=thetaMat, thetaLower=0, nSub=60,nStud=60) %>%
    filter(tkde > tlambdap) %>% as.tbl()
#### You could also simulate more and bind them together to a data frame.
print(df)
```

```
#> # A tibble: 2,340 x 16
#>        k  tkde   tkpq tkqpp tlambdap tgamma tdeltaqp  tpt0    tq0 eta.pt0 eta.q0
#>    <dbl> <dbl>  <dbl> <dbl>    <dbl>  <dbl>    <dbl> <dbl>  <dbl>   <dbl>  <dbl>
#>  1   100  2.83 0.0295 0.239    0.683  0.861     1.25  7.67   42.0   0.559  0.136
#>  2   100  2.83 0.0295 0.239    0.683  0.861     1.25  7.67   42.0  0.0465 -0.581
#>  3   100  2.83 0.0295 0.239    0.683  0.861     1.25  7.67   42.0  -0.188 -0.180
#>  4   100  2.83 0.0295 0.239    0.683  0.861     1.25  7.67   42.0   0.321  0.614
#>  5   100  2.83 0.0295 0.239    0.683  0.861     1.25  7.67   42.0  0.0656 -0.232
#>  6   100  2.83 0.0295 0.239    0.683  0.861     1.25  7.67   42.0  0.0194  0.517
#>  7   100  2.83 0.0295 0.239    0.683  0.861     1.25  7.67   42.0  -0.218  0.260
#>  8   100  2.83 0.0295 0.239    0.683  0.861     1.25  7.67   42.0  -0.258 -0.761
#>  9   100  2.83 0.0295 0.239    0.683  0.861     1.25  7.67   42.0   -1.28  -1.34
#> 10   100  2.83 0.0295 0.239    0.683  0.861     1.25  7.67   42.0  -0.495  0.161
#> # ... with 2,330 more rows, and 5 more variables: eta.lambdap <dbl>,
#> #   eta.kqp <dbl>, eta.qpp <dbl>, eta.deltaqp <dbl>, eta.kde <dbl>
```

```
#### Quick check to make sure that all the parameters are OK.
all(df$tkde>df$tlambdap)
```

```
#> [1] TRUE
```

```
sim1 <- Ribba2012 %>% # Use RxODE
    et(time.units="months") %>% # Pipe to a new event table
    et(amt=1, time=50, until=58, ii=1.5) %>% # Add dosing every 1.5 months
    et(0, 250, by=0.5) %>% # Add some sampling times (not required)
    rxSolve(df)
#### Note this information looses information about which ID is in a
#### "study", so it summarizes the confidence intervals by dividing the
#### subjects into sqrt(#subjects) subjects and then summarizes the
#### confidence intervals
sim2 <- sim1 %>% confint(c("pt","q","qp","pstar"),level=0.90); # Create Simulation intervals
```

```
#> ! in order to put confidence bands around the intervals, you need at least 2500 simulations
```

```
#> summarizing data...done
```

```
save(sim2, file = file.path(system.file(package = "RxODE"), "pipeline-sim2.rds"), vers
sim2 %>% plot()
```
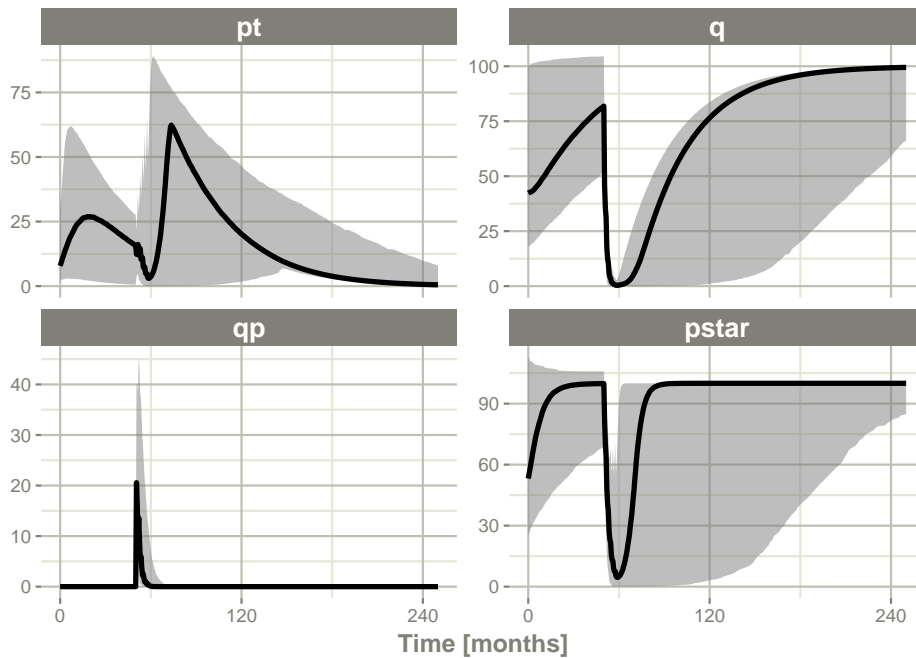

13.2   Speeding up RxODE

13.2.1   Increasing RxODE speed by multi-subject parallel solving

RxODE originally developed as an ODE solver that allowed an ODE solve for a single subject. This flexibility is still supported.

The original code from the RxODE tutorial is below:

```
library(RxODE)

library(microbenchmark)
library(ggplot2)

mod1 <- RxODE({
    C2 = centr/V2;
    C3 = peri/V3;
    d/dt(depot) = -KA*depot;
```

```
    d/dt(centr) = KA*depot - CL*C2 - Q*C2 + Q*C3;
    d/dt(peri) = Q*C2 - Q*C3;
    d/dt(eff) = Kin - Kout*(1-C2/(EC50+C2))*eff;
    eff(0) = 1
})

#### Create an event table

ev <- et() %>%
    et(amt=10000, addl=9,ii=12) %>%
    et(time=120, amt=20000, addl=4, ii=24) %>%
    et(0:240) ## Add Sampling

nsub <- 100 # 100 sub-problems
sigma <- matrix(c(0.09,0.08,0.08,0.25),2,2) # IIV covariance matrix
mv <- rxRmvn(n=nsub, rep(0,2), sigma) # Sample from covariance matrix
CL <- 7*exp(mv[,1])
V2 <- 40*exp(mv[,2])
params.all <- cbind(KA=0.3, CL=CL, V2=V2, Q=10, V3=300,
                     Kin=0.2, Kout=0.2, EC50=8)
```

13.2.1.1 For Loop

The slowest way to code this is to use a `for` loop. In this example we will enclose it
in a function to compare timing.

```
runFor <- function(){
    res <- NULL
    for (i in 1:nsub) {
        params <- params.all[i,]
        x <- mod1$solve(params, ev, cacheEvent=FALSE)
        ##Store results for effect compartment
        res <- cbind(res, x[, "eff"])
    }
    return(res)
}
```

13.2.1.2 Running with apply

In general for R, the `apply` types of functions perform better than a `for` loop, so
the tutorial also suggests this speed enhancement

```
runSapply <- function(){
    res <- apply(params.all, 1, function(theta)
        mod1$run(theta, ev, cacheEvent=FALSE)[, "eff"])
}
```

13.2.1.3   Run using a single-threaded solve

You can also have RxODE solve all the subject simultaneously without collecting
the results in R, using a single threaded solve.

The data output is slightly different here, but still gives the same information:

```
runSingleThread <- function(){
    solve(mod1, params.all, ev, cores=1, cacheEvent=FALSE)[,c("sim.id", "time", "eff")]
}
```

13.2.1.4   Run a 2 threaded solve

RxODE supports multi-threaded solves, so another option is to have 2 threads
(called `cores` in the solve options, you can see the options in `rxControl()` or
`rxSolve()`).

```
run2Thread <- function(){
    solve(mod1, params.all, ev, cores=2, cacheEvent=FALSE)[,c("sim.id", "time", "eff")]
}
```

13.2.1.5   Compare the times between all the methods

Now the moment of truth, the timings:

```
bench <- microbenchmark(runFor(), runSapply(), runSingleThread(),run2Thread())
print(bench)
```

```
#> Unit: milliseconds
#>                 expr       min        lq       mean     median         uq         max
#>             runFor() 151.03536 169.31284 206.37974 174.93747 179.15512 3028.12393
#>          runSapply() 149.50447 169.91367 191.32478 173.93002 178.55465 1829.41696
#>    runSingleThread()  21.88214  25.30863  25.71536  26.17450  26.63824   30.00154
#>         run2Thread()  14.96878  27.85664 115.22371  28.04107  29.67325  422.89574
#>  neval
#>    100
#>    100
```

```
#>     100
#>     100
```

```
autoplot(bench)
```

```
#> Coordinate system already present. Adding new coordinate system, which will replace the existi
```
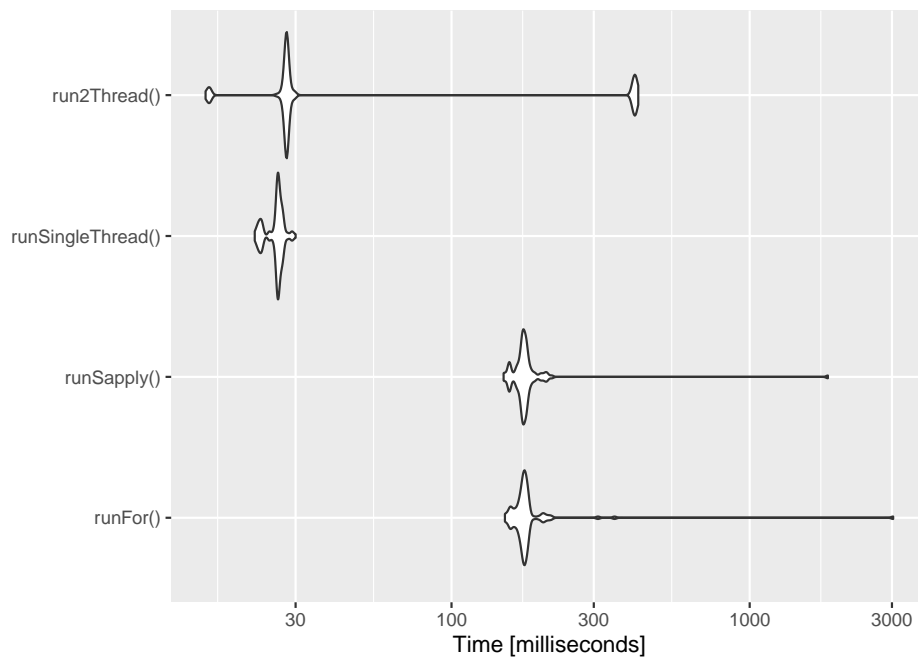

It is clear that the **largest** jump in performance when using the `solve` method and providing *all* the parameters to RxODE to solve without looping over each subject with either a `for` or a `sapply`. The number of cores/threads applied to the solve also plays a role in the solving.

We can explore the number of threads further with the following code:

```
runThread <- function(n){
    solve(mod1, params.all, ev, cores=n, cacheEvent=FALSE)[,c("sim.id", "time", "eff")]
}

bench <- eval(parse(text=sprintf("microbenchmark(%s)",
                                 paste(paste0("runThread(", seq(1, 2 * rxCores()),")"),
                                       collapse=","))))
print(bench)
```

```
#> Unit: milliseconds
```

```
#>          expr       min        lq      mean   median         uq        max neval
#>  runThread(1) 21.946265  23.05685  72.68991  25.77187   61.72614 987.8185    100
#>  runThread(2) 13.082558  13.71551  91.50768  27.75959   36.67690 566.9945    100
#>  runThread(3)  9.830327  17.72593  93.06805  26.66486  183.49878 439.2474    100
#>  runThread(4) 13.734207  15.56808  66.36228  22.99597   31.79808 354.2934    100
#>  runThread(5) 12.097867  13.88753  85.05241  21.17457  173.38444 323.9333    100
#>  runThread(6) 11.000963  12.76133 102.22429 143.02280  152.21046 310.7560    100
#>  runThread(7) 10.800733  19.17444 104.42476 130.28977  135.13572 290.9309    100
#>  runThread(8) 25.176448 112.37482 117.53097 119.79929  126.78507 280.8444    100
```

```
autoplot(bench)
```

```
#> Coordinate system already present. Adding new coordinate system, which will replace
```
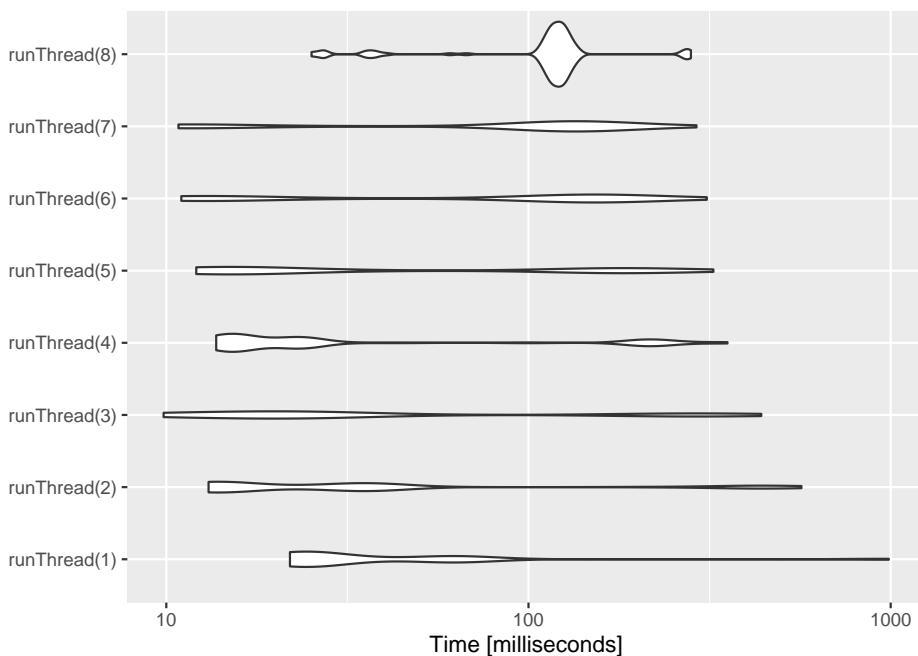

There can be a suite spot in speed vs number or cores. The system type (mac, linux, windows and/or processor), complexity of the ODE solving and the number of subjects may affect this arbitrary number of threads. 4 threads is a good number to use without any prior knowledge because most systems these days have at least 4 threads (or 2 processors with 4 threads).

13.2.2   A real life example

Before some of the parallel solving was implemented, the fastest way to run RxODE was with lapply. This is how Rik Schoemaker created the data-set for nlmixr com-

parisons, but reduced to run faster automatic building of the pkgdown website.

```r
library(RxODE)
library(data.table)
```

```
#>
#> Attaching package: 'data.table'

#> The following objects are masked from 'package:dplyr':
#>
#>     between, first, last
```

```r
#Define the RxODE model
  ode1 <- "
  d/dt(abs)    = -KA*abs;
  d/dt(centr)  =  KA*abs-(CL/V)*centr;
  C2=centr/V;
  "

#Create the RxODE simulation object
mod1 <- RxODE(model = ode1)

#Population parameter values on log-scale
  paramsl <- c(CL = log(4),
               V = log(70),
               KA = log(1))
#make 10,000 subjects to sample from:
  nsubg <- 300 # subjects per dose
  doses <- c(10, 30, 60, 120)
  nsub <- nsubg * length(doses)
#IIV of 30% for each parameter
  omega <- diag(c(0.09, 0.09, 0.09))# IIV covariance matrix
  sigma <- 0.2
#Sample from the multivariate normal
  set.seed(98176247)
  library(MASS)
```

```
#>
#> Attaching package: 'MASS'

#> The following object is masked from 'package:dplyr':
#>
#>     select
```

```
#> The following object is masked from 'package:patchwork':
#>
#>     area
```

```r
  mv <-
    mvrnorm(nsub, rep(0, dim(omega)[1]), omega) # Sample from covariance matrix
#Combine population parameters with IIV
  params.all <-
    data.table(
      "ID" = seq(1:nsub),
      "CL" = exp(paramsl['CL'] + mv[, 1]),
      "V" = exp(paramsl['V'] + mv[, 2]),
      "KA" = exp(paramsl['KA'] + mv[, 3])
    )
#set the doses (looping through the 4 doses)
params.all[, AMT := rep(100 * doses,nsubg)]

Startlapply <- Sys.time()

#Run the simulations using lapply for speed
  s = lapply(1:nsub, function(i) {
#selects the parameters associated with the subject to be simulated
    params <- params.all[i]
#creates an eventTable with 7 doses every 24 hours
    ev <- eventTable()
    ev$add.dosing(
      dose = params$AMT,
      nbr.doses = 1,
      dosing.to = 1,
      rate = NULL,
      start.time = 0
    )
#generates 4 random samples in a 24 hour period
    ev$add.sampling(c(0, sort(round(sample(runif(600, 0, 1440), 4) / 60, 2))))
#runs the RxODE simulation
    x <- as.data.table(mod1$run(params, ev))
#merges the parameters and ID number to the simulation output
    x[, names(params) := params]
  })

#runs the entire sequence of 100 subjects and binds the results to the object res
  res = as.data.table(do.call("rbind", s))

Stoplapply <- Sys.time()
```

```r
print(Stoplapply - Startlapply)
```

```
#> Time difference of 5.366516 secs
```

By applying some of the new parallel solving concepts you can simply run the same simulation both with less code and faster:

```r
rx <- RxODE({
    CL =  log(4)
    V = log(70)
    KA = log(1)
    CL = exp(CL + eta.CL)
    V = exp(V + eta.V)
    KA = exp(KA + eta.KA)
    d/dt(abs)    = -KA*abs;
    d/dt(centr)  =  KA*abs-(CL/V)*centr;
    C2=centr/V;
})

omega <- lotri(eta.CL ~ 0.09,
               eta.V ~ 0.09,
               eta.KA ~ 0.09)

doses <- c(10, 30, 60, 120)


startParallel <- Sys.time()
ev <- do.call("rbind",
        lapply(seq_along(doses), function(i){
            et() %>%
                et(amt=doses[i]) %>% # Add single dose
                et(0) %>% # Add 0 observation
#### Generate 4 samples in 24 hour period
                et(lapply(1:4, function(...){c(0, 24)})) %>%
                et(id=seq(1, nsubg) + (i - 1) * nsubg) %>%
#### Convert to data frame to skip sorting the data
#### When binding the data together
                as.data.frame
        }))
#### To better compare, use the same output, that is data.table
res <- rxSolve(rx, ev, omega=omega, returnType="data.table")
endParallel <- Sys.time()
print(endParallel - startParallel)
```

```
#> Time difference of 0.3108301 secs
```

You can see a striking time difference between the two methods; A few things to keep in mind:

- RxODE use the thread-safe sitmo `threefry` routines for simulation of `eta` values. Therefore the results are expected to be different (also the random samples are taken in a different order which would be different)

- This prior simulation was run in R 3.5, which has a different random number generator so the results in this simulation will be different from the actual nlmixr comparison when using the slower simulation.

- This speed comparison used `data.table`. `RxODE` uses `data.table` internally (when available) try to speed up sorting, so this would be different than installations where `data.table` is not installed. You can force RxODE to use `order()` when sorting by using `forderForceBase(TRUE)`. In this case there is little difference between the two, though in other examples `data.table`'s presence leads to a speed increase (and less likely it could lead to a slowdown).

13.2.2.1   Want more ways to run multi-subject simulations

The version since the tutorial has even more ways to run multi-subject simulations, including adding variability in sampling and dosing times with `et()` (see RxODE events for more information), ability to supply both an `omega` and `sigma` matrix as well as adding as a `thetaMat` to R to simulate with uncertainty in the `omega`, `sigma` and `theta` matrices; see RxODE simulation vignette.

13.3   Integrating RxODE models in your package

13.3.1   Using Pre-compiled models in your packages

If you have a package and would like to include pre-compiled RxODE models in your package it is easy to create the package. You simple make the package with the `rxPkg()` command.

```
library(RxODE);
#### Now Create a model
idr <- RxODE({
    C2 = centr/V2;
    C3 = peri/V3;
    d/dt(depot) =-KA*depot;
    d/dt(centr) = KA*depot - CL*C2 - Q*C2 + Q*C3;
    d/dt(peri)  =                     Q*C2 - Q*C3;
```

```
    d/dt(eff)  = Kin - Kout*(1-C2/(EC50+C2))*eff;
})
```

```
#### You can specify as many models as you want to add
```

```
rxPkg(idr, package="myPackage"); ## Add the idr model to your package
```

This will:

- Add the model to your package; You can use the package data as idr once the package loads

- Add the right package requirements to the DESCRIPTION file. You will want to update this to describe the package and modify authors, license etc.

- Create skeleton model documentation files you can add to for your package documentation. In this case it would be the file idr-doc.R in your R directory

- Create a configure and configure.win script that removes and regenerates the src directory based on whatever version of RxODE this is compiled against. This should be modified if you plan to have your own compiled code, though this is not suggested.

- You can write your own R code in your package that interacts with the RxODE object so you can distribute shiny apps and similar things in the package context.

Once this is present you can add more models to your package by rxUse(). Simply compile the RxODE model in your package then add the model with rxUse()

```
rxUse(model)
```

Now both model and idr are in the model library. This will also create model-doc.R in your R directory so you can document this model.

You can then use devtools methods to install/test your model

```
devtools::load_all() # Load all the functions in the package
devtools::document() # Create package documentation
devtools::install() # Install package
devtools::check() # Check the package
devtools::build() # build the package so you can submit it to places like CRAN
```

13.3.2   Using Models in a already present package

To illustrate, lets start with a blank package

```
library(RxODE)
library(usethis)
pkgPath  <- file.path(rxTempDir(),"MyRxModel")
create_package(pkgPath);
use_gpl3_license("Matt")
use_package("RxODE", "LinkingTo")
use_package("RxODE", "Depends") ##  library(RxODE) on load; Can use imports instead.
use_roxygen_md()
##use_readme_md()
library(RxODE);
#### Now Create a model
idr <- RxODE({
    C2 = centr/V2;
    C3 = peri/V3;
    d/dt(depot) =-KA*depot;
    d/dt(centr) = KA*depot - CL*C2 - Q*C2 + Q*C3;
    d/dt(peri)  =                    Q*C2 - Q*C3;
    d/dt(eff)  = Kin - Kout*(1-C2/(EC50+C2))*eff;
});

rxUse(idr); ## Add the idr model to your package
rxUse(); # Update the compiled RxODE sources for all of your packages
```

The `rxUse()` will: - Create RxODE sources and move them into the package's `src/` directory. If there is only R source in the package, it will also finish off the directory with an `library-init.c` which registers all the RxODE models in the package for use in R. - Create stub R documentation for each of the models your are including in your package. You will be able to see the R documentation when loading your package by the standard `?` interface.

You will still need to: - Export at least one function. If you do not have a function that you wish to export, you can add a re-export of `RxODE` using roxygen as follows:

```
##' @importFrom RxODE RxODE
##' @export
RxODE::RxODE
```

If you want to use `Suggests` instead of `Depends` in your package, you way want to export all of RxODE's normal routines

```
##' @importFrom RxODE RxODE
```

```
##' @export
RxODE::RxODE

##' @importFrom RxODE et
##' @export
RxODE::et

##' @importFrom RxODE etRep
##' @export
RxODE::etRep

##' @importFrom RxODE etSeq
##' @export
RxODE::etSeq

##' @importFrom RxODE as.et
##' @export
RxODE::as.et

##' @importFrom RxODE eventTable
##' @export
RxODE::eventTable

##' @importFrom RxODE add.dosing
##' @export
RxODE::add.dosing

##' @importFrom RxODE add.sampling
##' @export
RxODE::add.sampling

##' @importFrom RxODE rxSolve
##' @export
RxODE::rxSolve

##' @importFrom RxODE rxControl
##' @export
RxODE::rxControl

##' @importFrom RxODE rxClean
##' @export
RxODE::rxClean

##' @importFrom RxODE rxUse
##' @export
RxODE::rxUse
```

```
##' @importFrom RxODE rxShiny
##' @export
RxODE::rxShiny

##' @importFrom RxODE genShinyApp.template
##' @export
RxODE::genShinyApp.template

##' @importFrom RxODE cvPost
##' @export
RxODE::cvPost

### This is actually from `magrittr` but allows less imports
##' @importFrom RxODE %>%
##' @export
RxODE::`%>%`
```

- You also need to instruct R to load the model library models included in the model's dll. This is done by:

```
### In this case `rxModels` is the package name
##' @useDynLib rxModels, .registration=TRUE
```

If this is a R package with RxODE models and you do not intend to add any other compiled sources (recommended), you can add the following configure scripts

```
#!/bin/sh
### This should be used for both configure and configure.win
echo "unlink('src', recursive=TRUE);RxODE::rxUse()" > build.R
${R_HOME}/bin/Rscript build.R
rm build.R
```

Depending on the `check` you may need a dummy autoconf script,

```
#### dummy autoconf script
#### It is saved to configure.ac
```

If you want to integrate with other sources in your `Rcpp` or `C/Fortan` based packages, you need to include `rxModels-compiled.h` and: - Add the define macro `compiledModelCall` to the list of registered `.Call` functions. - Register C interface to allow model solving by `R_init0_rxModels_RxODE_models()` (again `rxModels` would be replaced by your package name).

Once this is complete, you can compile/document by the standard methods:

```
devtools::load_all()
devtools::document()
devtools::install()
```

If you load the package with a new version of RxODE, the models will be recompiled when they are used.

However, if you want the models recompiled for the most recent version of RxODE, you simply need to call `rxUse()` again in the project directory followed by the standard methods for install/create a package.

```
devtools::load_all()
devtools::document()
devtools::install()
```

Note you do not have to include the `RxODE` code required to generate the model to regenerate the RxODE c-code in the `src` directory. As with all RxODE objects, a `summary` will show one way to recreate the same model.

An example of compiled models package can be found in the rxModels repository.

13.4 Stiff ODEs with Jacobian Specification

13.4.0.1 Stiff ODEs with Jacobian Specification

Occasionally, you may come across a **stiff** differential equation, that is a differential equation that is numerically unstable and small variations in parameters cause different solutions to the ODEs. One way to tackle this is to choose a stiff-solver, or hybrid stiff solver (like the default LSODA). Typically this is enough. However exact Jacobian solutions may increase the stability of the ODE. (Note the Jacobian is the derivative of the ODE specification with respect to each variable). In RxODE you can specify the Jacobian with the `df(state)/dy(variable)=` statement. A classic ODE that has stiff properties under various conditions is the Van der Pol differential equations.

In RxODE these can be specified by the following:

```
library(RxODE)

Vtpol2 <- RxODE({
    d/dt(y)  = dy
    d/dt(dy) = mu*(1-y^2)*dy - y
##### Jacobian
    df(y)/dy(dy)  = 1
```

```
    df(dy)/dy(y)   = -2*dy*mu*y - 1
    df(dy)/dy(dy) = mu*(1-y^2)
##### Initial conditions
    y(0) = 2
    dy(0) = 0
##### mu
    mu = 1 ## nonstiff; 10 moderately stiff; 1000 stiff
})

et <- eventTable();
et$add.sampling(seq(0, 10, length.out=200));
et$add.dosing(20, start.time=0);

s1 <- Vtpol2 %>%  solve(et, method="lsoda")
print(s1)
```

```
#> _____ Solved RxODE object _____
#> -- Parameters ($params): --------------------------------------------------
#> mu
#>  1
#> -- Initial Conditions ($inits): -------------------------------------------
#>  y dy
#>  2  0
#> -- First part of data (object): -------------------------------------------
#> # A tibble: 200 x 3
#>     time     y      dy
#>    <dbl> <dbl>   <dbl>
#> 1 0        22     0
#> 2 0.0503  22.0 -0.0456
#> 3 0.101   22.0 -0.0456
#> 4 0.151   22.0 -0.0456
#> 5 0.201   22.0 -0.0456
#> 6 0.251   22.0 -0.0456
#> # ... with 194 more rows
#> _____
```

While this is not stiff at mu=1, mu=1000 is a stiff system

```
s2 <- Vtpol2 %>%  solve(c(mu=1000), et)
print(s2)
```

```
#> _____ Solved RxODE object _____
#> -- Parameters ($params): --------------------------------------------------
#>    mu
```

```
#> 1000
#> -- Initial Conditions ($inits): --------------------------------------------------
#>  y dy
#>  2  0
#> -- First part of data (object): --------------------------------------------------
#> # A tibble: 200 x 3
#>     time     y         dy
#>    <dbl> <dbl>      <dbl>
#> 1 0        22    0
#> 2 0.0503  22.0 -0.0000455
#> 3 0.101   22.0 -0.0000455
#> 4 0.151   22.0 -0.0000455
#> 5 0.201   22.0 -0.0000455
#> 6 0.251   22.0 -0.0000455
#> # ... with 194 more rows
#> ------------------------------------------------------------------------------------
```

While this is easy enough to do, it is a bit tedious. If you have RxODE setup appropriately, that is you have:

- **Python** installed in your system
- **sympy** installed in your system
- **SnakeCharmR** installed in R

You can use the computer algebra system sympy to calculate the Jacobian automatically.

This is done by the RxODE option `calcJac` option:

```
Vtpol <- RxODE({
    d/dt(y)  = dy
    d/dt(dy) = mu*(1-y^2)*dy - y
##### Initial conditions
    y(0) = 2
    dy(0) = 0
##### mu
    mu = 1 ## nonstiff; 10 moderately stiff; 1000 stiff
}, calcJac=TRUE)
```

To see the generated model, you can use `rxCat()`:

```
> rxCat(Vtpol)
d/dt(y)=dy;
d/dt(dy)=mu*(1-y^2)*dy-y;
y(0)=2;
```

```
dy(0)=0;
mu=1;
df(y)/dy(y)=0;
df(dy)/dy(y)=-2*dy*mu*y-1;
df(y)/dy(dy)=1;
df(dy)/dy(dy)=mu*(-Rx_pow_di(y,2)+1);
```