RxODE user manual

Matthew Fidler

2020-12-15

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 5) related to Leibnitz notation.
- The event data, which can be:
  - a `NONMEM` or `deSolve` compatible data frame (Chapter 6), or
  - created with `et()` or `EventTable()` for easy simulation of events(Chapter ??)
  - 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 uncertanty can be used to simulate while solving.

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

This book was assembled on Tue Dec 15 18:42:29 2020 with RxODE version 1.0.1 automatically by github actions.

Chapter 2

Related R packages

2.1 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 RxODE maintainers has had positive interactions with all of the ODE-solving pharmacometric projects listed.

2.2 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.

2.2.1 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.

2.2.2 Non-CRAN libraries:

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

Chapter 3

Installation

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

```
install.packages("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:

```
## 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:

3.0.1 Windows

In windows you may simply use installr to install rtools:

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

Alternatively you can download and install rtools directly.

3.0.2 Mac OSX

To get the most speed you need OpenMP enabled and compile RxODE against that binary. Here is some discussion about this:

https://mac.r-project.org/openmp/

3.0.3 Linux

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

3.1 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

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 4

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 1.0.1 using 4 threads (see ?getRxThreads)
```

```
library(units)
```

```
#> udunits system database from /usr/share/xml/udunits
```

```
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 v0.23.4.
```

4.1 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.

```
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.

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

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

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

4.2 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:

```
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:

```
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 `%>%`

```
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 vignette.

4.3 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

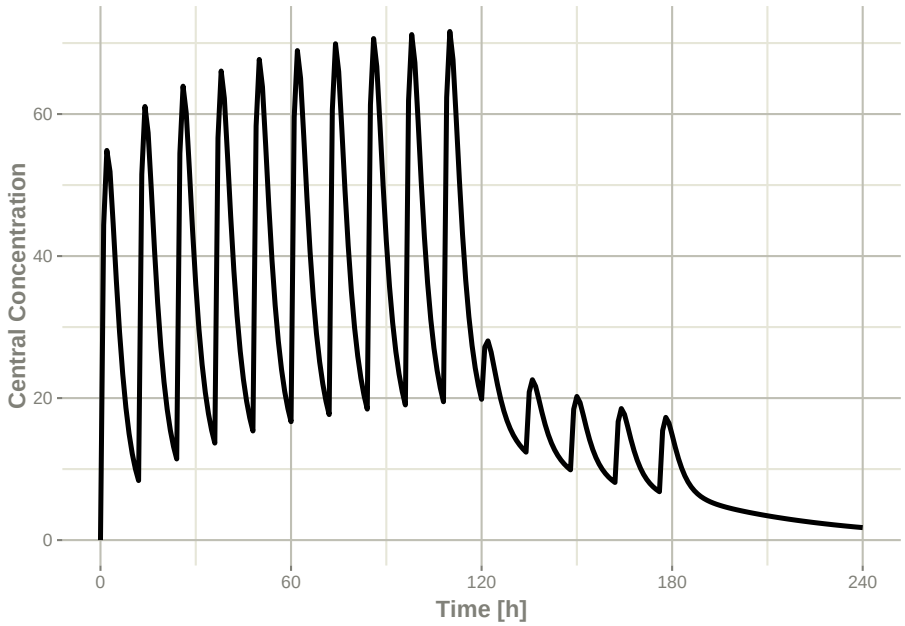
You can also solve this and create a RxODE data frame:

```
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
#> ____________________________________________________________
```

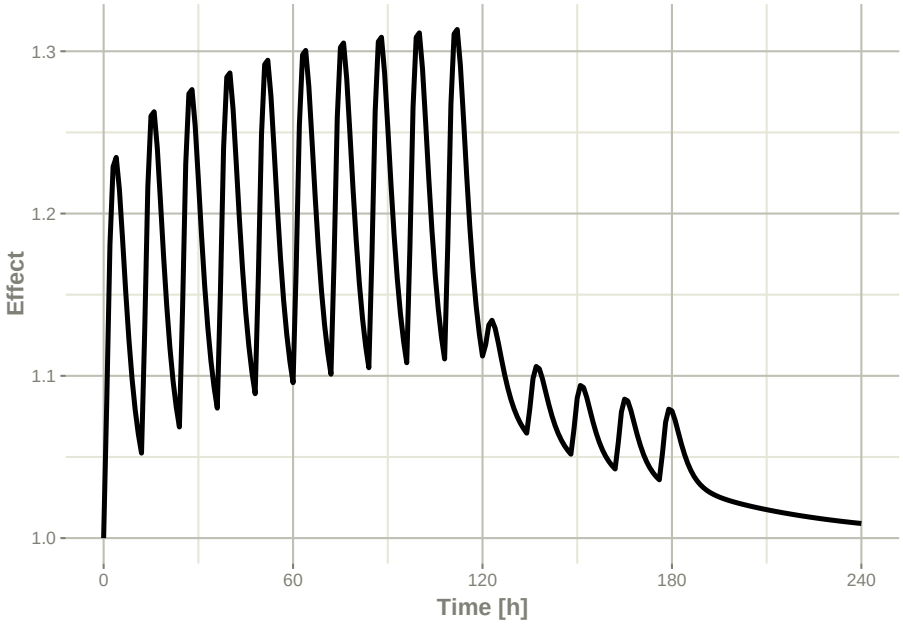
This returns a modified data frame. You can see the compartment values in the plot below:

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



Or,

```
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 5

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.

5.1 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;
```

5.2 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 vignette.
- 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.

5.2.1 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.

5.3 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.

5.4 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

5.4.1 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 1.0.1 model named rx_cb529f87d53ee398a0cae52b742263c7 model (ready).
#> DLL: /home/matt/.cache/R/RxODE/rx_cb529f87d53ee398a0cae52b742263c7__.rxd/rx_cb529f87d53ee398a0
#> 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.

5.4.2 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 1.0.1 model named rx_b36e6f19a6e700a81cf72fe0f38fab91 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`.

5.4.3 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 1.0.1 model named rx_19b9e41504acaacf9c477fb16c151209 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 6

RxODE events

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.

```
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
});
```

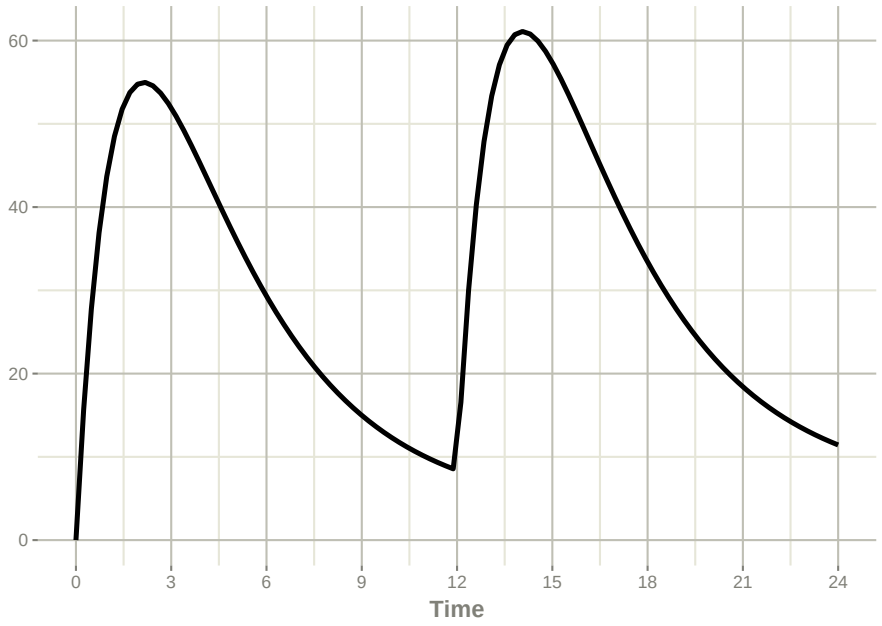
6.1 Bolus/Additive Doses

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

```
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
```

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



6.2 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

6.2.1 Constant Infusions (in terms of duration and rate)

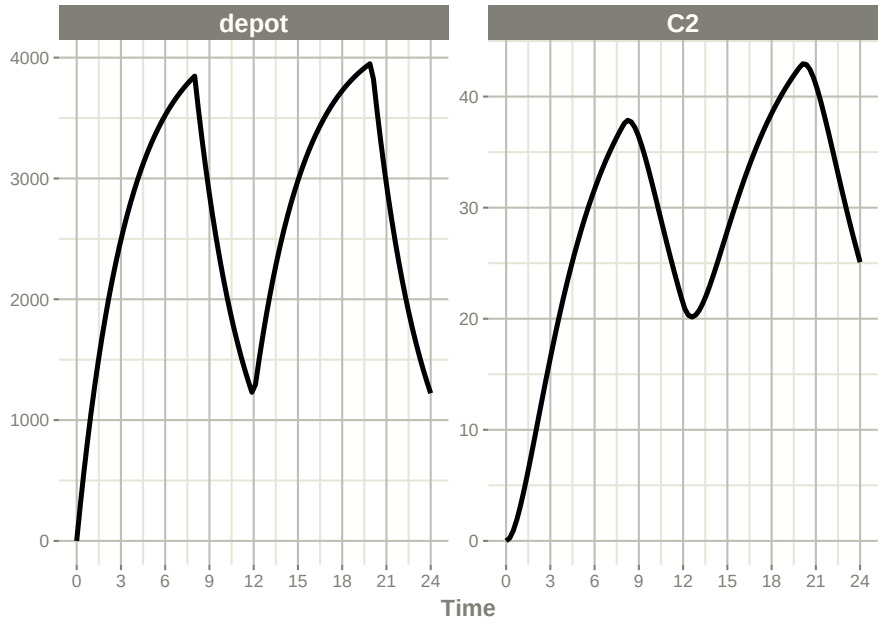
The next type of event is an infusion; There are two ways to specify an infusion; The first is the `dur` keyword.

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
```

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

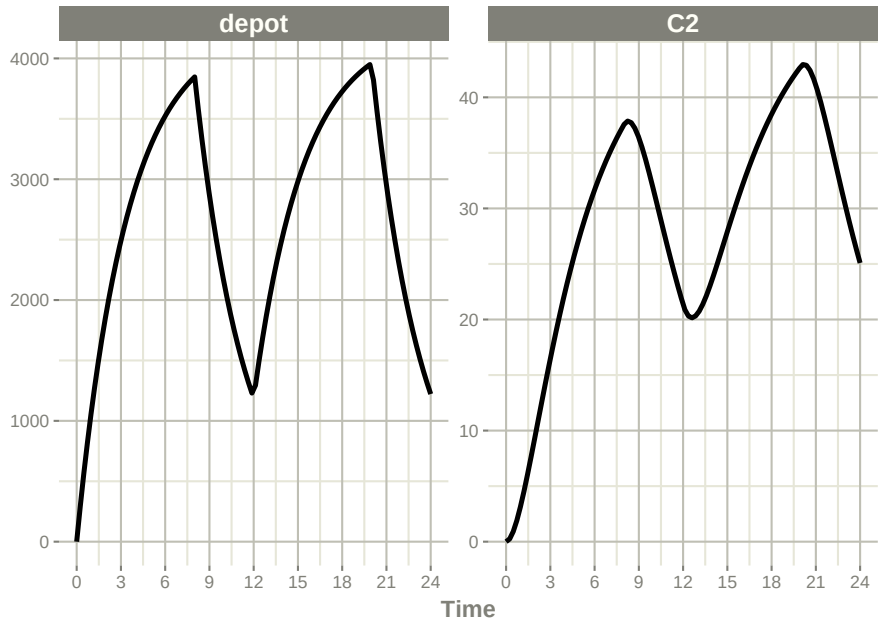


It can be also specified by the `rate` component:

```
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
```

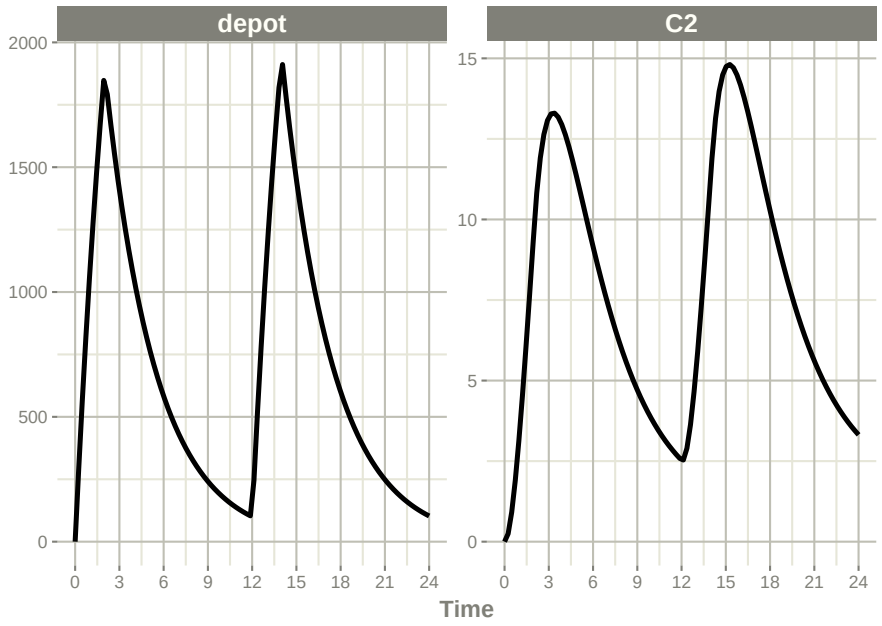
```
rxSolve(m1, ev) %>% plot(depot, C2) +
    xlab("Time")
```



These are the same with the exception of how bioavailability changes the infusion.

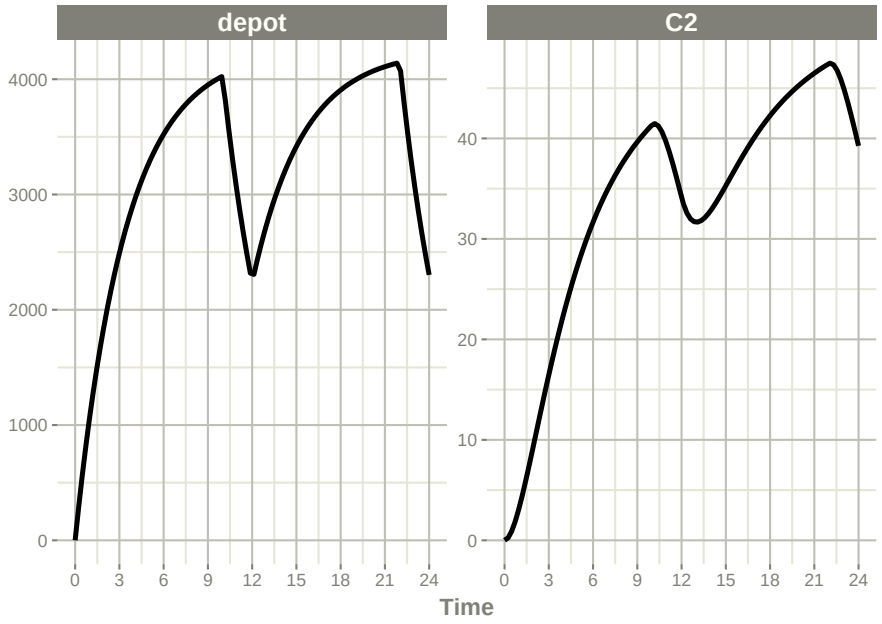
In the case of modeling `rate`, a bioavailability decrease, decreases the infusion duration, as in NONMEM. For example:

```
rxSolve(m1, ev, c(fdepot=0.25)) %>% plot(depot, C2) +
    xlab("Time")
```



Similarly increasing the bioavailability increases the infusion duration.

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



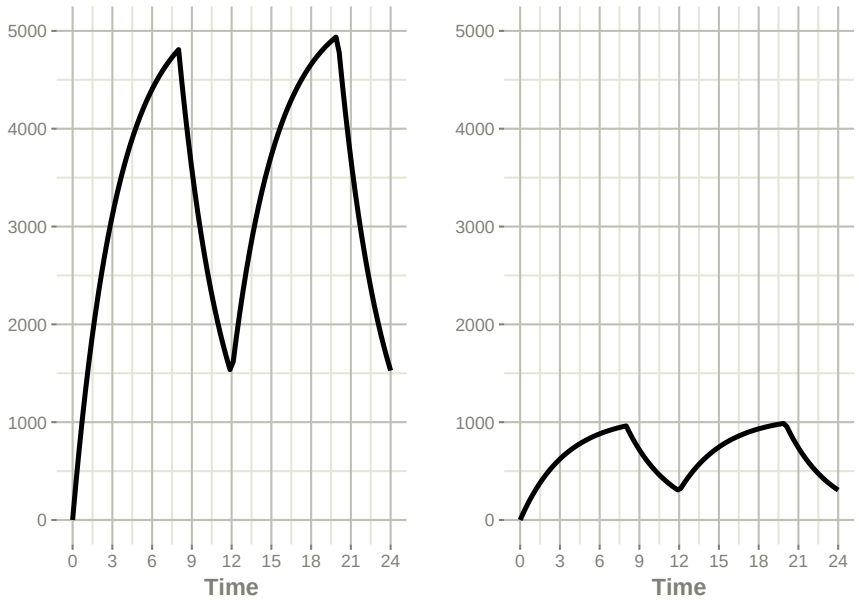
The rationale for this behavior is that the `rate` and `amt` are specified by the event table, so the only thing that can change with a bioavailability increase is the duration of the infusion.

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
```



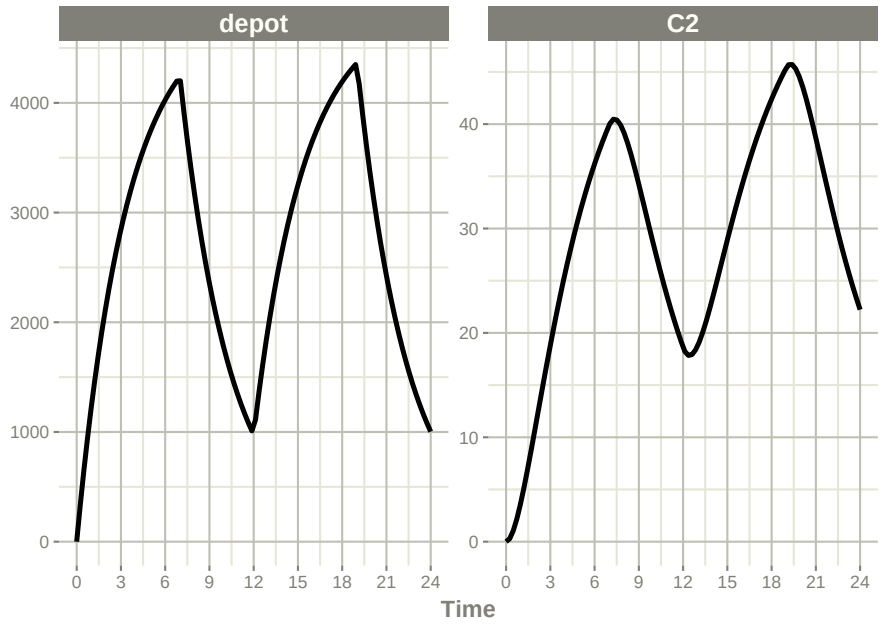
6.2.2 Modeled Rate and Duration of Infusion

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
```

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



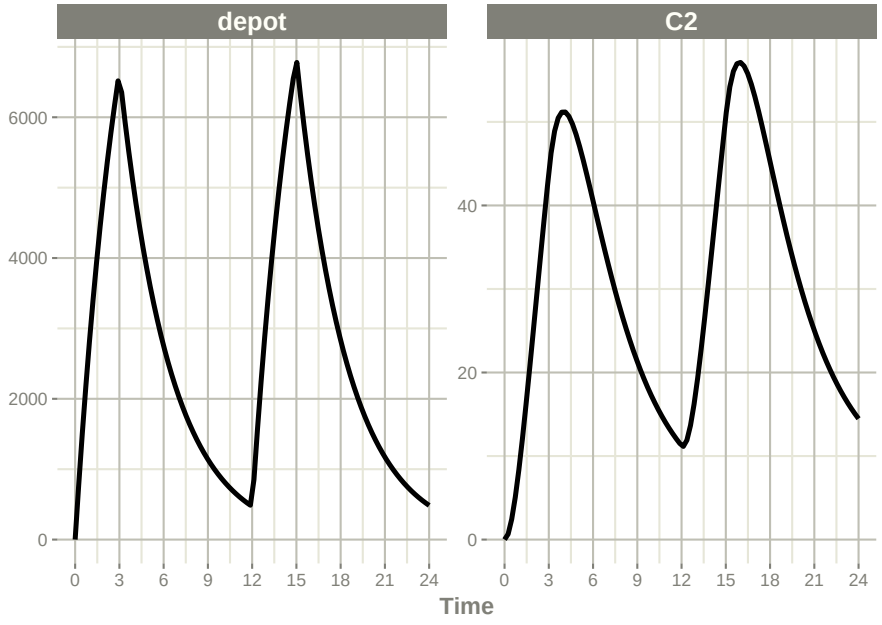
Similarly, you may also model rate. This is equivalent to NONMEM's `rate=-1` and is how RxODE's event table specifies the data item as well. You can also use `rate=model` as a mnemonic:

```
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")
```



6.3 Steady State

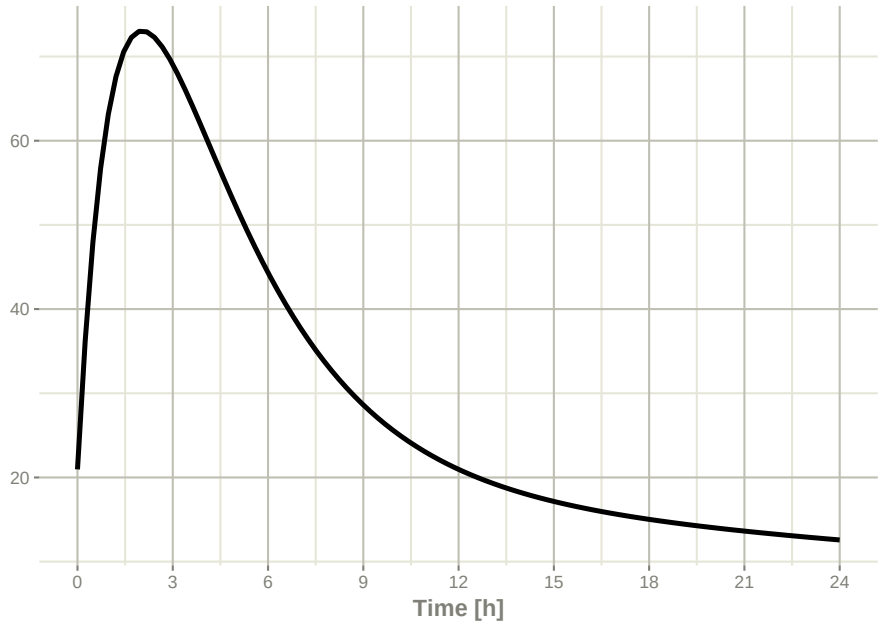
These doses are solved until a steady state is reached with a constant inter-dose interval.

```
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)
```



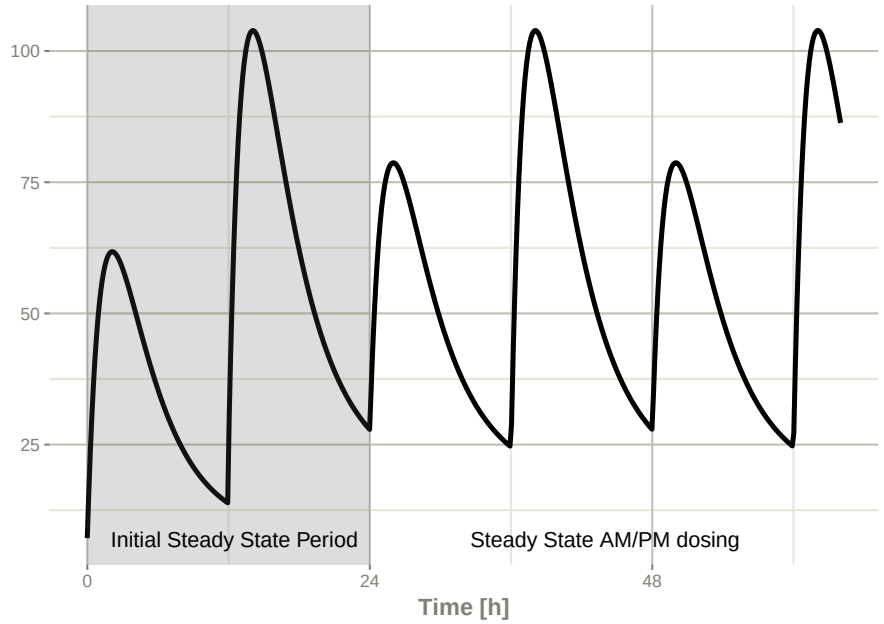
6.3.1 Steady state for complex dosing

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.

6.3.2 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:

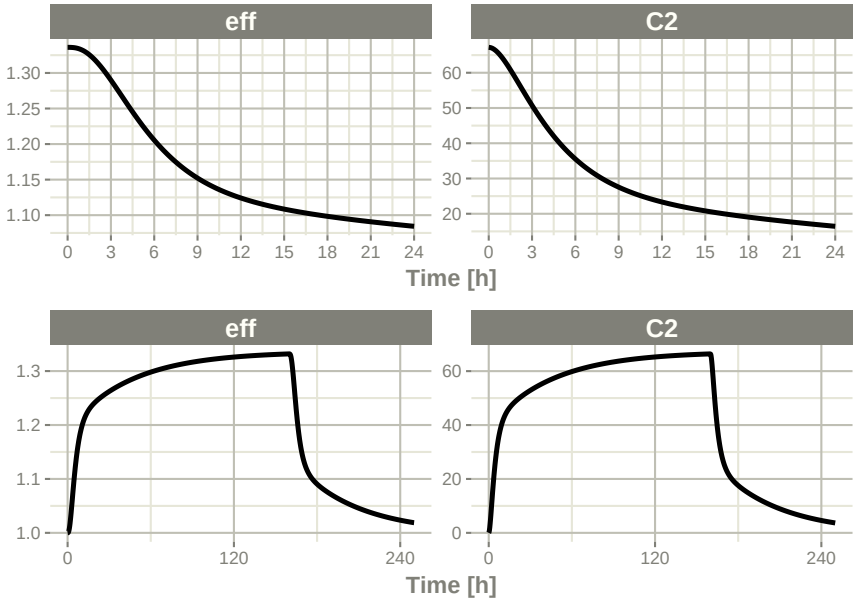
- No inter-dose interval `ii`=0
- A steady state dose, ie `ss`=1
- Either a positive rate (`rate`>0) or a estimated rate `rate`=-1.
- A zero dose, ie `amt`=0
- Once the steady-state constant infusion is achieved, the infusion is turned off when using this record, just like NONMEM.

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.

6.4 Reset Events

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

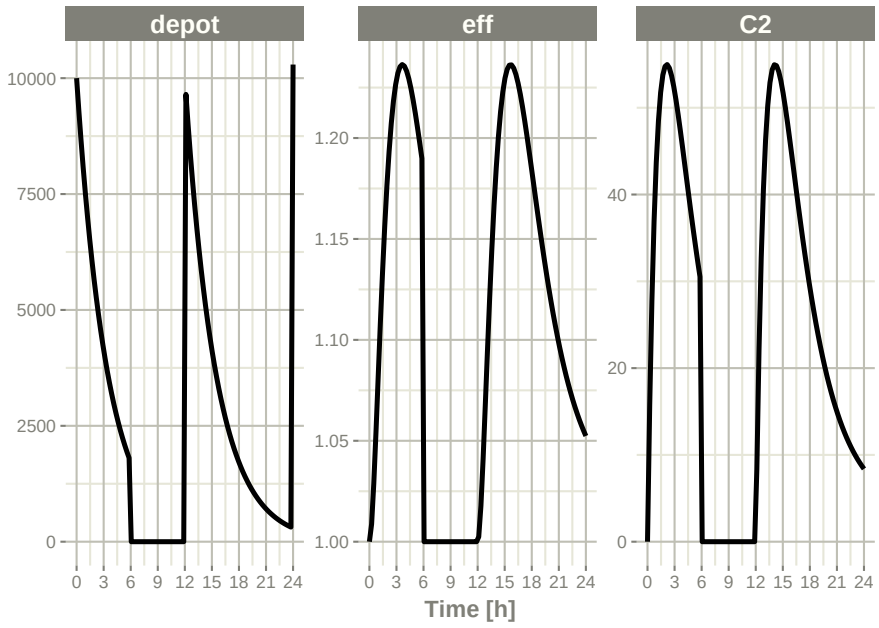
```
ev <- et(timeUnits="hr") %>%
    et(amt=10000, ii=12, addl=3) %>%
    et(time=6, evid=reset) %>%
    et(seq(0, 24, length.out=100))

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.

```
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.

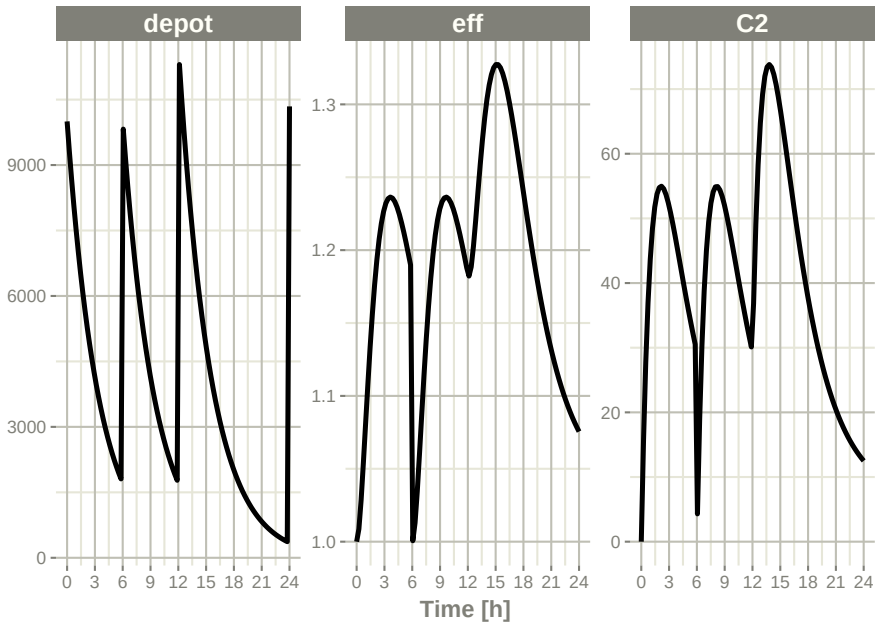
```
ev <- et(timeUnits="hr") %>%
    et(amt=10000, ii=12, addl=3) %>%
    et(time=6, amt=10000, evid=4) %>%
```

```
    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)
```



6.5 Turning off compartments

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

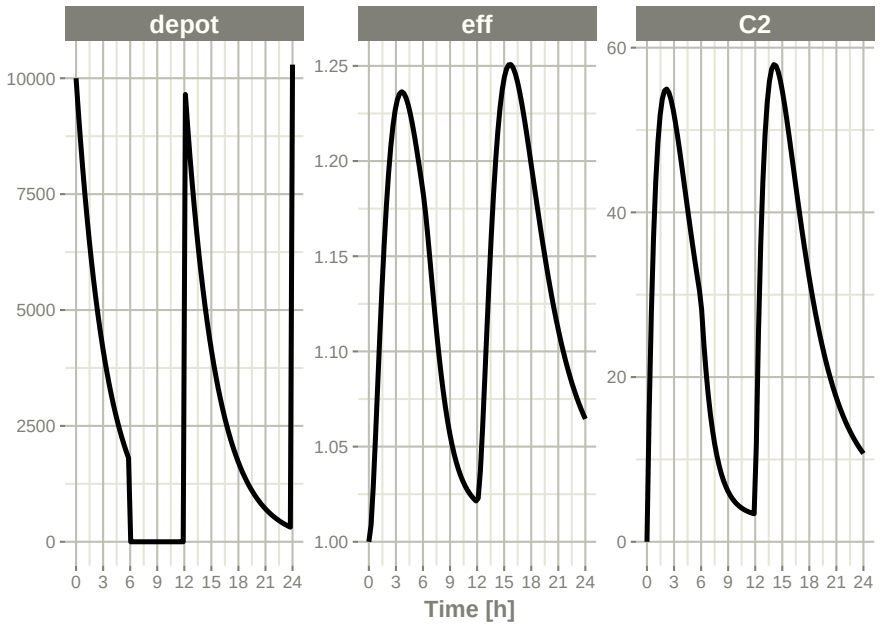
```
ev <- et(timeUnits="hr") %>%
    et(amt=10000, ii=12, addl=3) %>%
    et(time=6, cmt="-depot", evid=2) %>%
    et(seq(0, 24, length.out=100))

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)
```

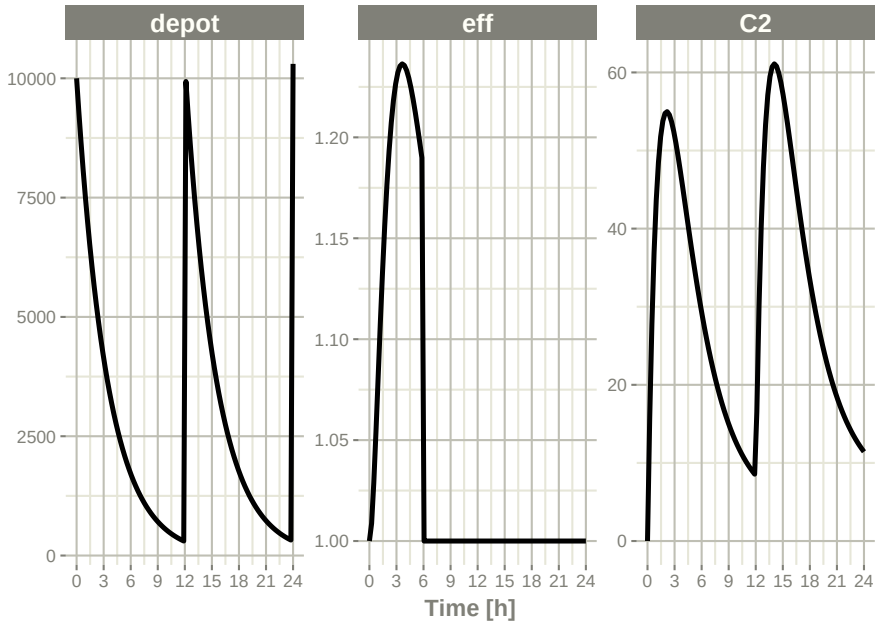


In this case, the depot is turned off, and the depot compartment concentrations are set to the initial values but the other compartment concentrations/levels are not reset. When another dose to the depot is administered the depot compartment is turned back on.

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.

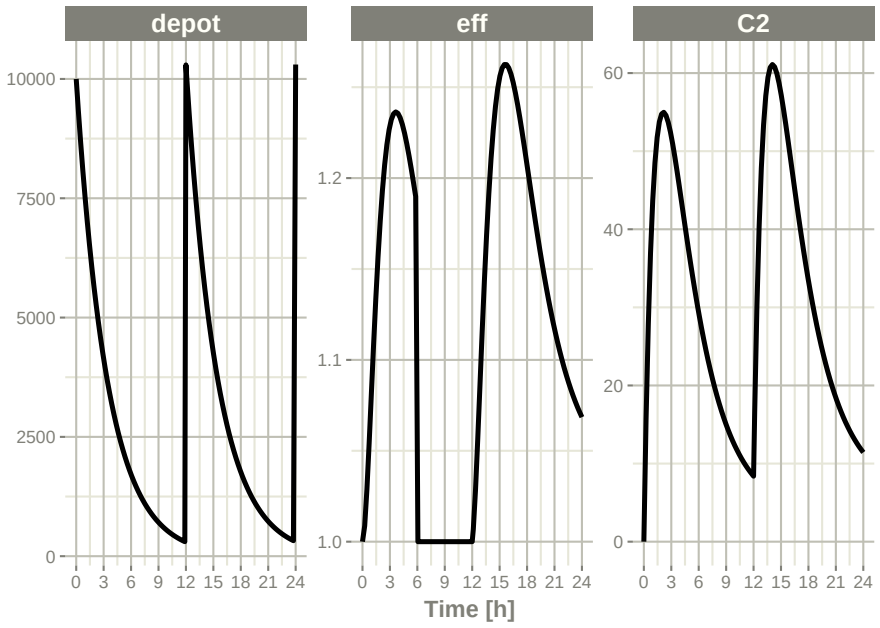
```
ev <- et(timeUnits="hr") %>%
    et(amt=10000, ii=12, addl=3) %>%
    et(time=6, cmt="-eff", evid=2) %>%
```

```
    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))

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



6.6 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)
	7 = turn off modeled rate		
	8 = turn on modeled duration		

100+ cmt	Infusion/Event Flag	<99 Cmt	SS flag & Turning of Compartment
	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:

6.6.1 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

6.6.2 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

6.6.3 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

6.6.4 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 `199`th compartment the `evid` for the bolus dose would be `109901`. An infusion dosing record for the `199`th compartment would be `119901`.

Chapter 7

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)
(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
})
```

7.1 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

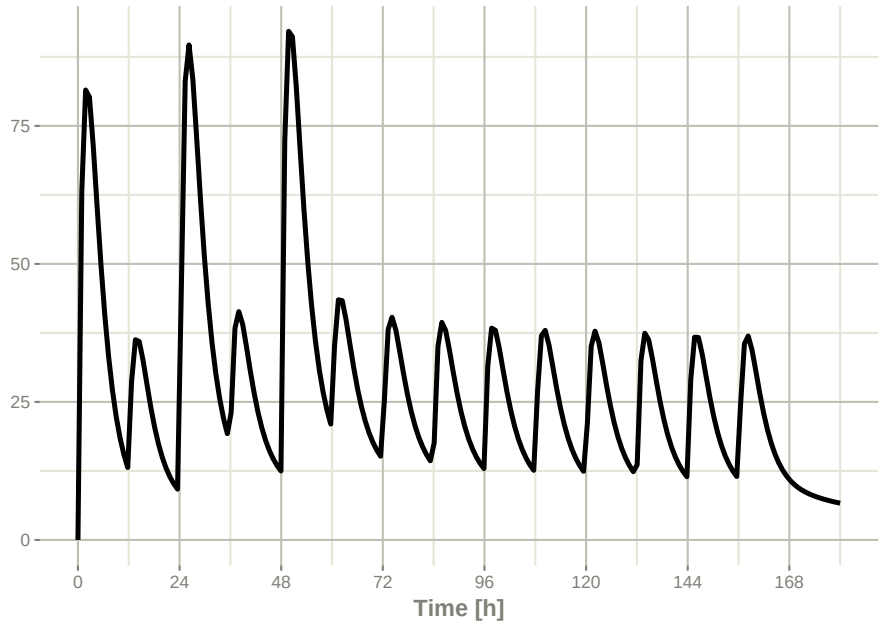
```
ev <- eventTable(amount.units="mg", time.units="hr")

## 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:

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



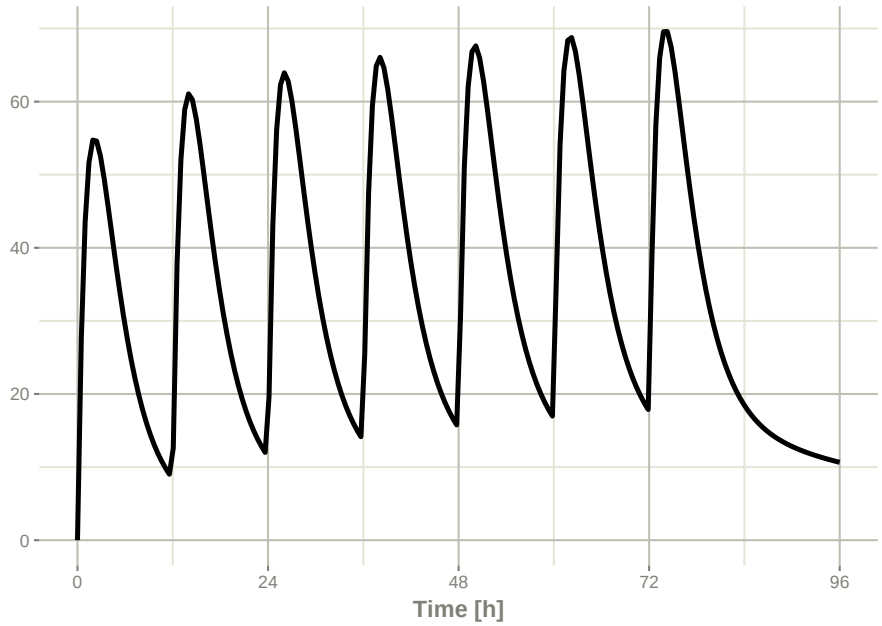
If you are more familiar with the NONMEM/RxODE event records, you can also specify dosing using `et` with the dose elements directly:

```
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.

7.2 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`

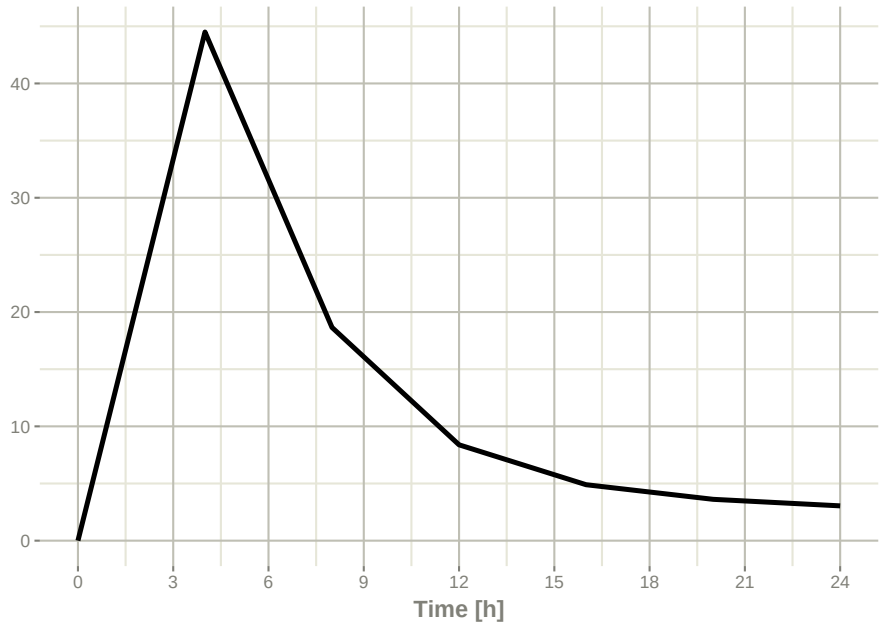
```
ev <- eventTable(amount.units="mg", time.units="hr")

## 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)
```



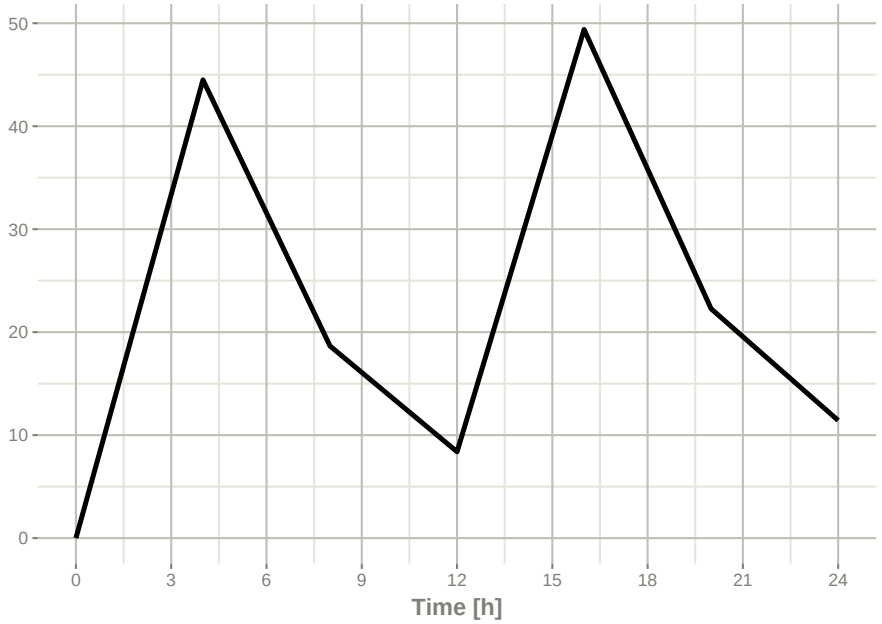
Or if you use `et` you can simply add them in a similar way to `add.sampling`:

```
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.

7.3 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
```

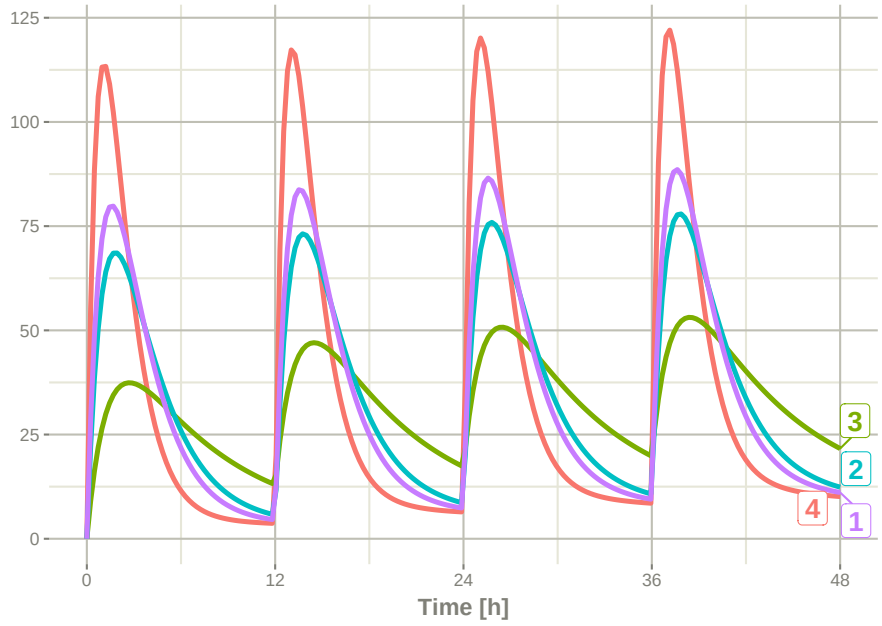
```
#> ---------------- EventTable with 804 records ---------------
#>    4 individuals
#>    4 dosing records (see x$get.dosing(); add with add.dosing or et)
#>    800 observation times (see x$get.sampling(); add with add.sampling or et)
#>    multiple doses in ‘addl‘ columns, expand with x$expand(); 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
```



7.4 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:

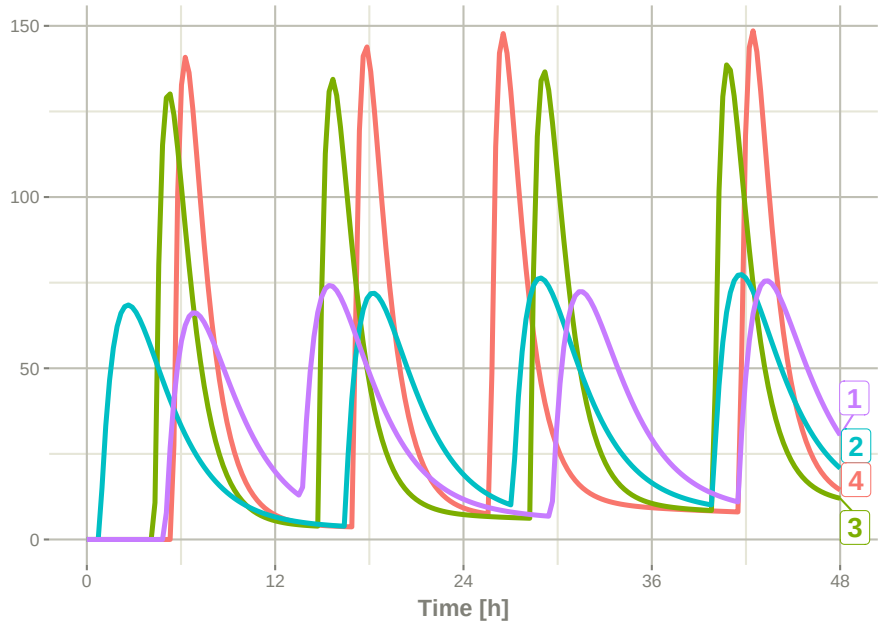
```
set.seed(42)
ev <- et(timeUnits="hr") %>%
    et(time=list(c(0,6)), amt=10000, until = set_units(2, days), ii=12) %>% # loading
    et(id=1:4)

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)
```

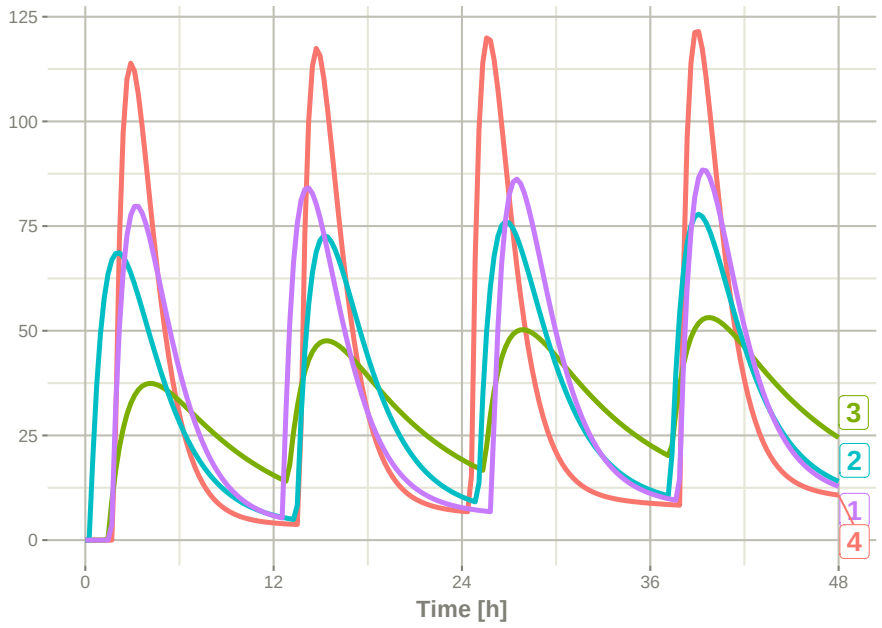
```
#> Warning: 'ID' missing in 'parameters' dataset
#> individual parameters are assumed to have the same order as the event dataset
```



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)
```

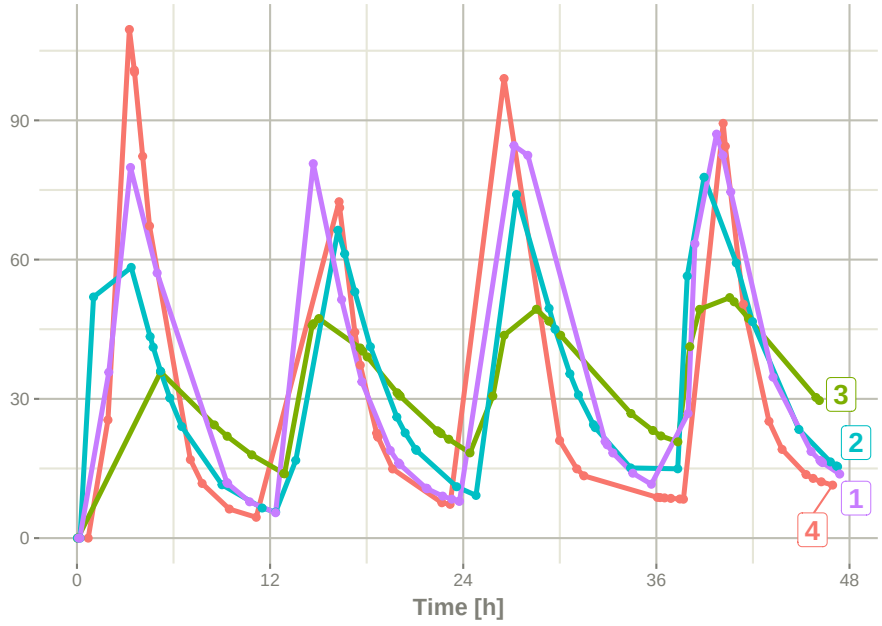
```
#> Warning: 'ID' missing in 'parameters' dataset
#> individual parameters are assumed to have the same order as the event dataset
```



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
    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((
```

```
#> 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.

7.5 Combining event tables

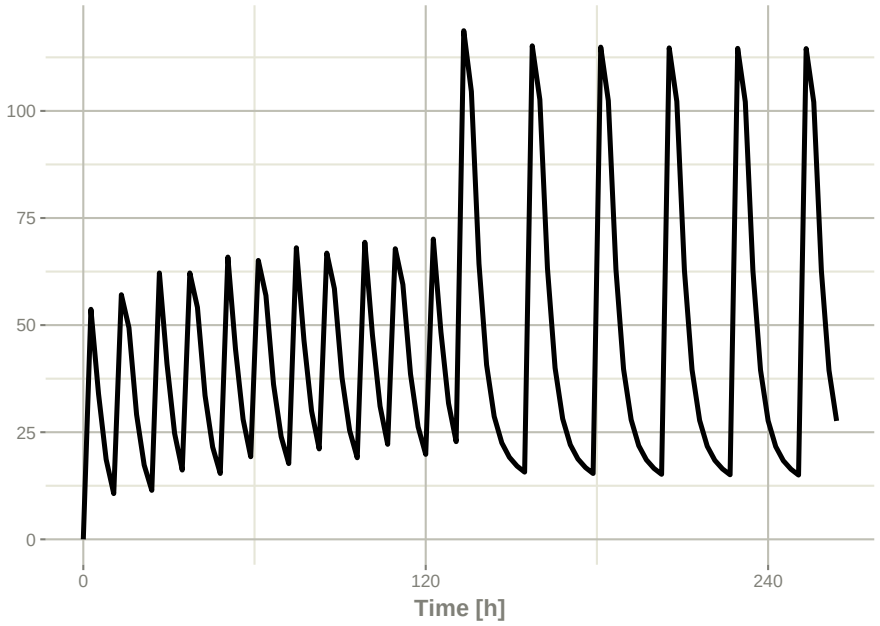
Since you can create dosing records and sampling records, you can create any complex dosing regimen you wish. In addition, RxODE allows you to combine event tables by `c`, `seq`, `rep`, and `rbind`.

7.6 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:

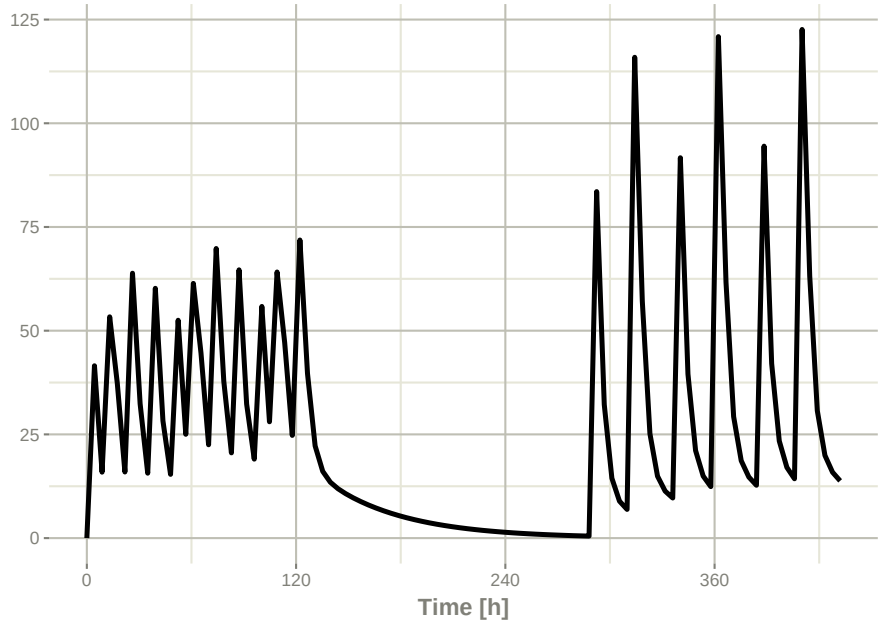
```
## bid for 5 days
bid <- et(timeUnits="hr") %>%
       et(amt=10000,ii=12,until=set_units(5, "days"))

## 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)
```



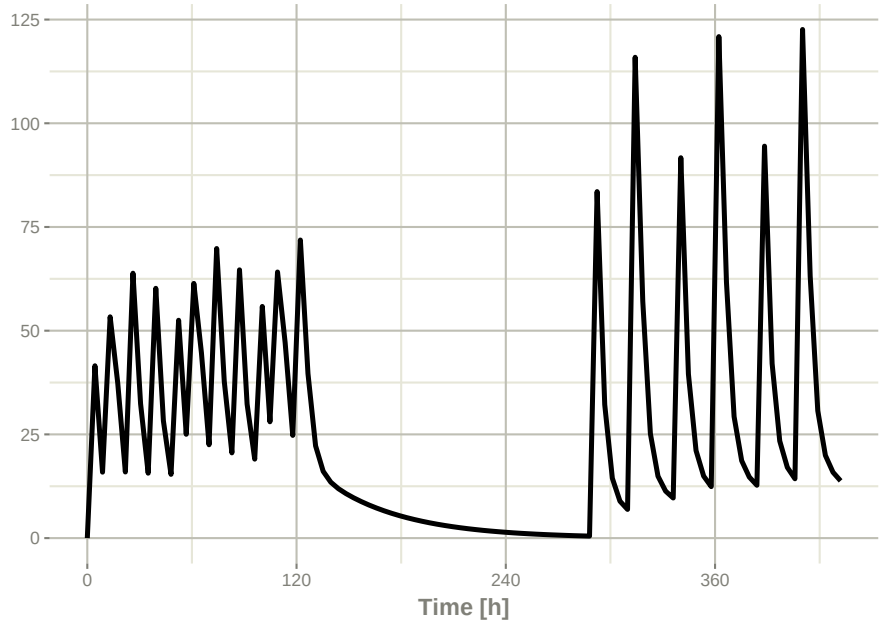
When sequencing events, you can also separate this sequence by a period of time; For example if you wanted to separate this by a week, you could easily do that with the following sequence of event tables:

```
## 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)
```

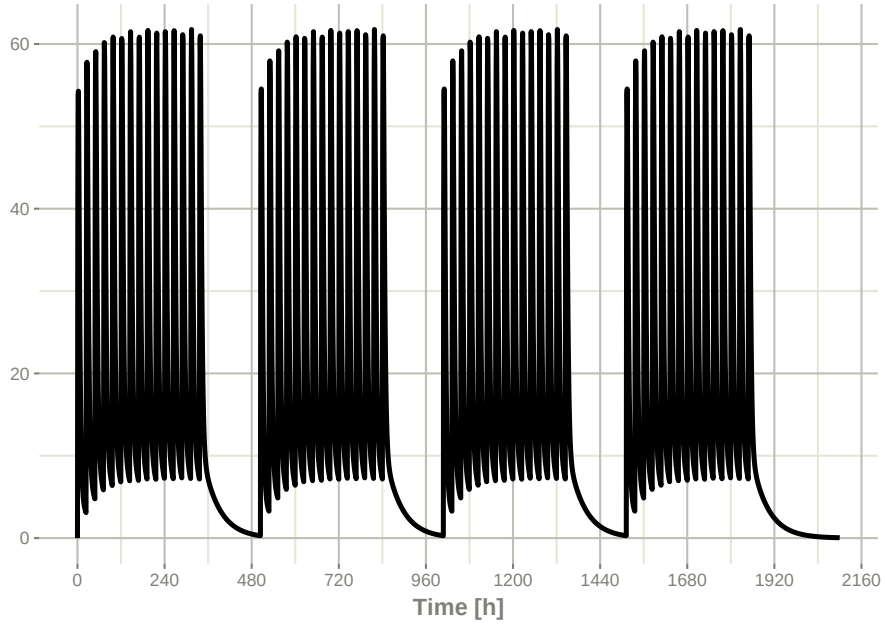


Also note, that RxODE assumes that the dosing is what you want to space the event tables by, and clears out any sampling records when you combine the event tables. If that is not true, you can also use the option `samples="use"`

7.7 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="de

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.

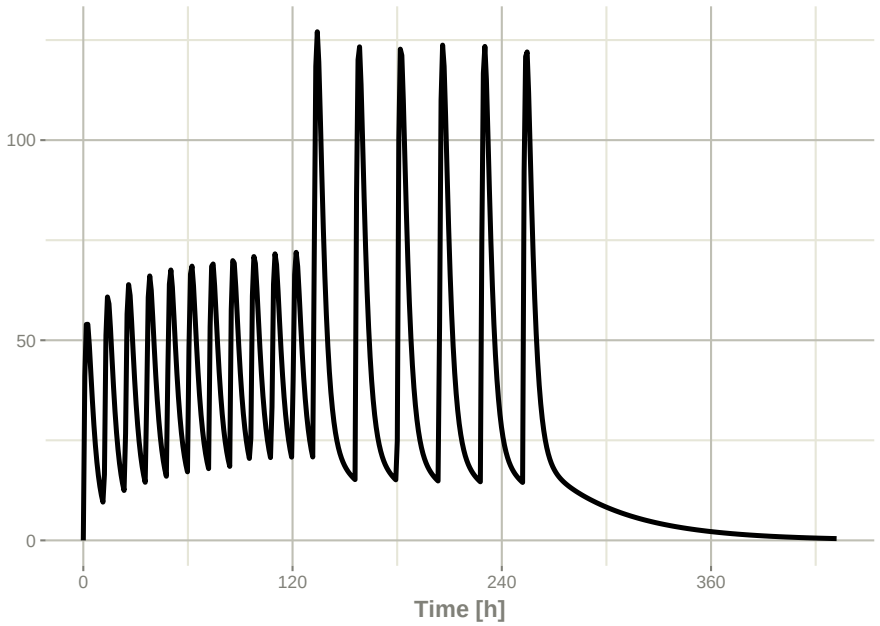
7.8 Combining event tables with rbind

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:

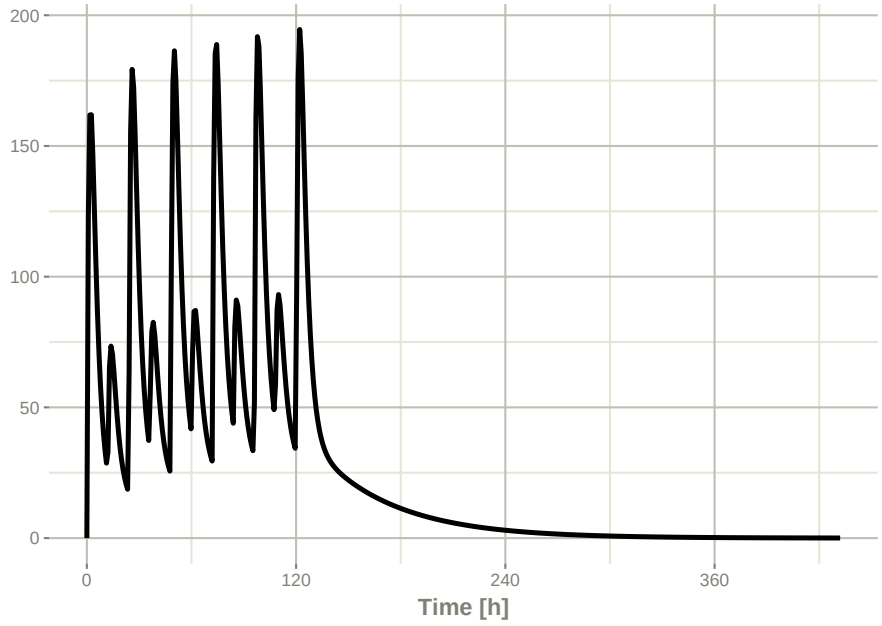
```
## bid for 5 days
bid <- et(timeUnits="hr") %>%
       et(amt=10000,ii=12,until=set_units(5, "days"))

## 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)
```



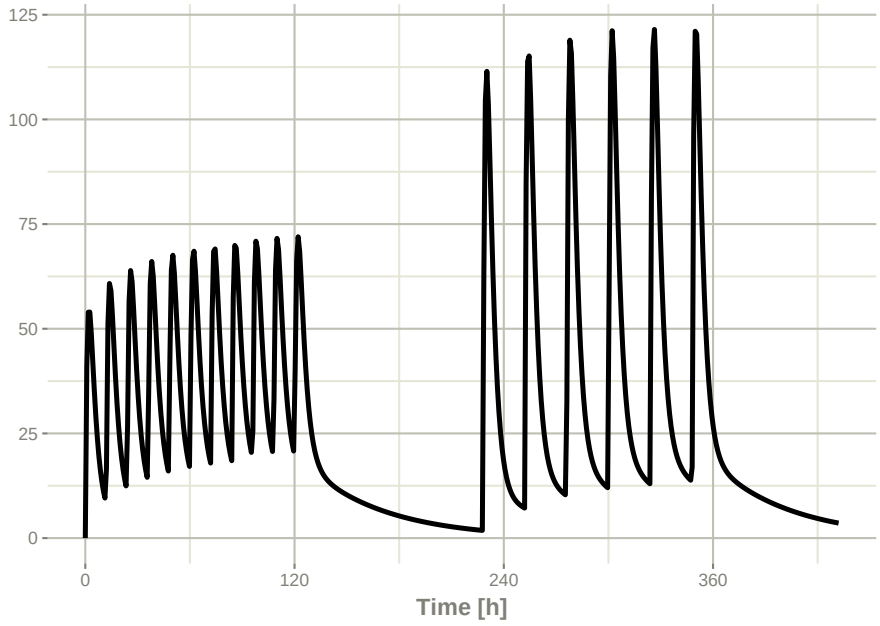
But if you bind them together with `rbind`

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

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



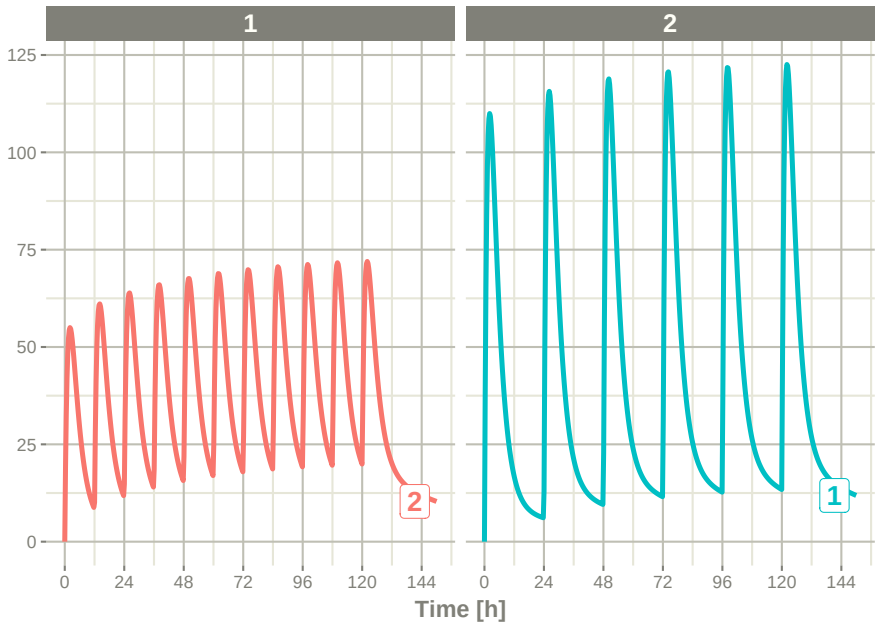
Still the waiting period applies (but does not consider the inter-dose interval)

```
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)
```


1
2
125
100
75
50
25
0
0
24
48
72
96
120
144
Time [h]
2
1

Chapter 8

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: ## 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.

8.1 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;

8.2 events

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

8.3 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);

8.4 iCov

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

8.5 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.

8.6 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.

8.7 transitAbs

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

8.8 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.

8.9 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.

8.10 maxsteps

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

8.11 hmin

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

8.12 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.

8.13 hmaxSd

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

8.14 hini

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

8.15 maxordn

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

8.16 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.

8.17 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).

8.18 hmxi

`hmxi` inverse of the maximum absolute value of H to be used. hmxi = 0.0 is allowed and corresponds to an infinite hmax (default). hmin and hmxi may 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.

8.19 ...

... 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.

8.20 cores

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

8.21 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.

8.22 addCov

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

8.23 matrix

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

8.24 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.

8.25 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.

8.26 sigmaDf

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

8.27 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.

8.28 sigmaIsChol

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

8.29 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.

8.30 nDisplayProgress

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

8.31 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.

8.32 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.

8.33 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).

8.34 theta

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

8.35 eta

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

8.36 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.

8.37 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.

8.38 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.

8.39 seed

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

8.40 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.

8.41 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 matrrix. This only works with a diagonal matrix.

8.42 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.

8.43 b

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

8.44 nsim

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

8.45 minSS

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

8.46 maxSS

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

8.47 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.

8.48 infSSstep

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

8.49 ssAtol

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

8.50 ssRtol

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

8.51 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.

8.52 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.

8.53 subsetNonmem

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

8.54 maxAtolRtolFactor

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

8.55 from

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

8.56 to

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

8.57 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.

8.58 by

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

8.59 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.

8.60 drop

`drop` Columns to drop from the output

8.61 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.

8.62 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.

8.63 warnDrop

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

8.64 safeZero

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

8.65 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)

8.66 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.

8.67 indLinPhiTol

`indLinPhiTol` the requested accuracy tolerance on exponential matrix.

8.68 indLinPhiM

`indLinPhiM` the maximum size for the Krylov basis

8.69 cacheEvent

`cacheEvent` is a boolean. If `TRUE` (default), events are cached in memory to speed up solving.

8.70 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

8.71 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.

8.72 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

8.73 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

8.74 linDiffCentral

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

8.75 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

8.76 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`.

8.77 maxwhile

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