rxode2 user manual

Matthew Fidler, Melissa Hallow, Wenping Wang

2023-07-05

Contents

Chapter 1

Introduction

Welcome to the rxode2 user guide; **rxode2** is an R package for solving and simulating from ode-based models. These models are convert the rxode2 mini-language to C and create a compiled dll for fast solving. ODE solving using rxode2 has a few key parts:

- `rxode2()` which creates the C code for fast ODE solving based on a simple syntax (Chapter 6) related to Leibnitz notation.
- The event data, which can be:
  - a `NONMEM` or `deSolve` compatible data frame (Chapter 7), or
  - created with `et()` or `EventTable()` for easy simulation of events (Chapter 11)
  - The data frame can be augmented by adding time varying or adding individual covariates (`iCov=` as needed)
- `rxSolve()` which solves the system of equations using initial conditions and parameters to make predictions
  - With multiple subject data, this may be parallelized.
  - With single subject the output data frame is adaptive
  - Covariances and other metrics of uncertainty can be used to simulate while solving.

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

- `rxode2` github pkgdown page
- `rxode2` tutorial (accessible in tutorials in Rstudio 1.3+)
- `rxode2` github discussions
- There is an Chinese `rxode2` manual translated by Fu Yongchao which was translated based on the manual dated June 9, 2023.

This book was assembled on Wed Jul 5 18:27:20 2023 with rxode2 version 2.0.13.9000 automatically by github actions.

Chapter 2

Authors and Acknowledgments

2.1 Authors

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

2.2 Contributors

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

- Matt Dowle – forder primary author (version modified by Matthew Fidler to allow different type of threading and exclude grouping)
- Cleve Moler – LApack author
- David Cooley – Author of fast_factor which was modified and now is used RxODE to quickly create factors for IDs without sorting them like R does
- Drew Schmidt – Drew Schmidt author of edits for exponential matrix utility taken from R package expm
- Arun Srinivasan – forder secondary author (version modified by Matthew Fidler to allow different type of threading, indexing and exclude grouping)

2.3 RxODE acknowledgments:

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

Chapter 3

Related R packages

3.1 ODE solving

This is a brief comparison of pharmacometric ODE solving R packages to `rxode2`.

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 rxode2 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, `rxode2` has a R-like mini-language that is parsed into C code that solves the ODE system.

  Unlike `rxode2`, `mrgsolve` does not currently support symbolic manipulation of ODE systems, like automatic Jacobian calculation or forward sensitivity calculation (`rxode2` currently supports this and this is the basis of nlmixr2'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 `rxode2` defines ODE systems at a lower level. `rxode2`'s parsing of the mini-language comes from C, whereas `dMod`'s parsing comes from R.

  Like `rxode2`, `dMod` supports symbolic manipulation of ODE systems and calculates forward sensitivities and adjoint sensitivities of systems.

  Unlike `rxode2`, `dMod` is not thread-safe since `deSolve` is not yet thread-safe.

- 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 rxode2 maintainers has had positive interactions with all of the ODE-solving pharmacometric projects listed.

3.2 PK Solved systems

`rxode2` 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.

`rxode2` can mix ODEs and solved systems.

3.2.1 The following packages for solved PK systems are on CRAN

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

3.2.2 Non-CRAN libraries:

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

Chapter 4

Installation

You can install the released version of rxode2 from CRAN with:

```r
install.packages("rxode2")
```

The fastest way to install the development version of `rxode2` is to use the `r-universe` service. This service compiles binares of the development version for MacOS and for Windows so you don't have to wait for package compilation:

```r
install.packages(c("dparser", "rxode2ll", "rxode2parse",
                   "rxode2random", "rxode2et", "rxode2"),
                 repos=c(nlmixr2="https://nlmixr2.r-universe.dev",
                         CRAN="https://cloud.r-project.org"))
```

If this doesn't work you install the development version of rxode2 with

```r
devtools::install_github("nlmixr2/rxode2parse")
devtools::install_github("nlmixr2/rxode2random")
devtools::install_github("nlmixr2/rxode2et")
devtools::install_github("nlmixr2/rxode2ll")
devtools::install_github("nlmixr2/rxode2")
```

To build models with rxode2, you need a working c compiler. To use parallel threaded solving in rxode2, this c compiler needs to support open-mp.

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

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

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

4.0.1 Windows

In windows you may simply use installr to install rtools:

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

Alternatively you can download and install rtools directly.

4.0.2 Mac OSX

To get the most speed you need OpenMP enabled and compile rxode2 with that compiler. There are various options and the most up to date discussion about this is likely the data.table installation faq for MacOS. The last thing to keep in mind is that `rxode2` uses the code very similar to the original `lsoda` which requires the `gfortran` compiler to be setup as well as the `OpenMP` compilers.

If you are going to be using `rxode2` and `nlmixr` together and have an older mac computer, I would suggest trying the following:

```
library(symengine)
```

If this crashes your R session then the binary does not work with your Mac machine. To be able to run nlmixr, you will need to compile this package manually. I will proceed assuming you have `homebrew` installed on your system.

On your system terminal you will need to install the dependencies to compile `symengine`:

```
brew install cmake gmp mpfr libmpc
```

After installing the dependencies, you need to reinstall `symengine`:

```
install.packages("symengine", type="source")
library(symengine)
```

4.0.3 Linux

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

4.1 Development Version

Since the development version of rxode2 uses StanHeaders, you will need to make sure your compiler is setup to support C++14, as described in the rstan setup page. For R 4.0, I do not believe this requires modifying the windows toolchain any longer (so it is much easier to setup).

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

```r
# install.packages("devtools")
devtools::install_github("nlmixr2/rxode2parse")
devtools::install_github("nlmixr2/rxode2random")
devtools::install_github("nlmixr2/rxode2et")
devtools::install_github("nlmixr2/rxode2ll")
devtools::install_github("nlmixr2/rxode2")
```

Chapter 5

Getting Started

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

To load `rxode2` package and compile the model:

```r
library(rxode2)
```

```
#> rxode2 2.0.13.9000 using 8 threads (see ?getRxThreads)
#>   no cache: create with `rxCreateCache()`
```

```r
mod1 <- function() {
  ini({
    # central
    KA=2.94E-01
    CL=1.86E+01
    V2=4.02E+01
    # peripheral
    Q=1.05E+01
    V3=2.97E+02
    # effects
    Kin=1
    Kout=1
    EC50=200
  })
  model({
    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
    eff(0) <- 1
    d/dt(eff)   <- Kin - Kout*(1-C2/(EC50+C2))*eff
  })
}
```

Model parameters may be specified in the `ini({})` model block, initial conditions can be specified within the model with the `cmt(0)= X`, like in this model `eff(0) <- 1`.

You may also specify between subject variability initial conditions and residual error components just like nlmixr2. This allows a single interface for `nlmixr2/rxode2` models. Also note, the classic `rxode2` interface still works just like it did in the past (so don't worry about breaking code at this time).

In fact, you can get the classic `rxode2` model `$simulationModel` in the ui object:

```
mod1 <- mod1() # create the ui object (can also use `rxode2(mod1)`)
mod1

summary(mod1$simulationModel)
```

5.1 Specify Dosing and sampling in rxode2

`rxode2` 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 "et()" function. This has an interface that is similar to NONMEM event tables:

```
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 rxode2 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 rxode2 see the rxode2 events section.

5.2 Solving ODEs

The ODE can now be solved using `rxSolve`:

```
x <- mod1 %>% rxSolve(ev)
```

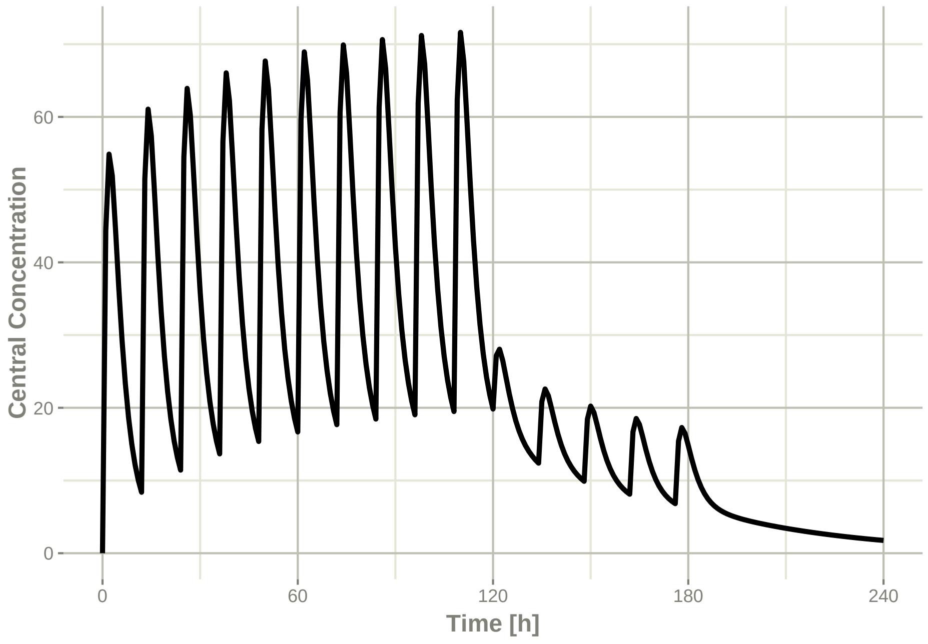
```
#> using C compiler: ‘gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0’
```

```
x
```

```
#> -- Solved rxode2 object --
#> -- Parameters (x$params): --
#>      KA      CL      V2       Q      V3     Kin    Kout    EC50
#>   0.294  18.600  40.200  10.500 297.000   1.000   1.000 200.000
#> -- Initial Conditions (x$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
#> # i 235 more rows
```

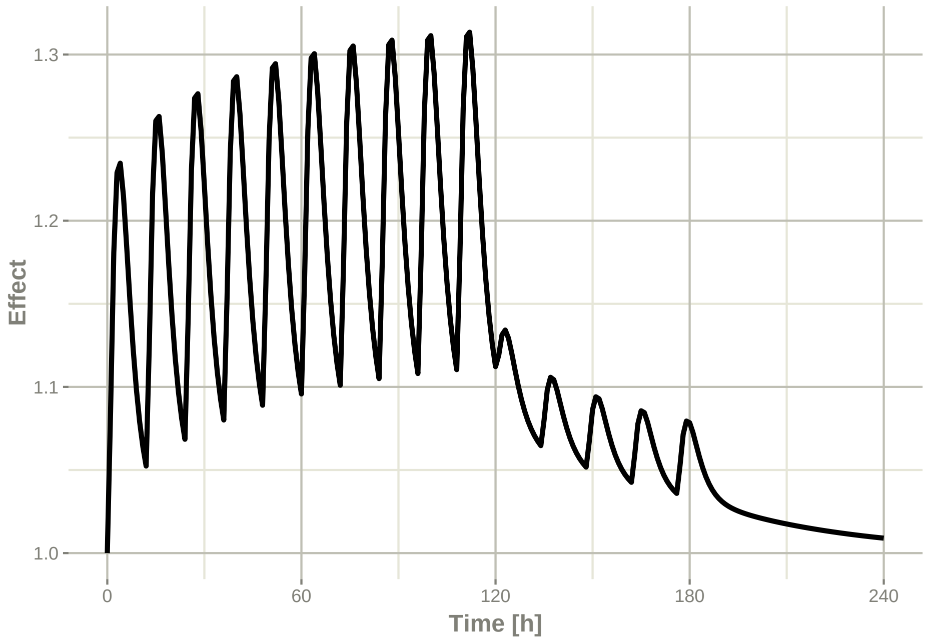
This returns a modified data frame. You can see the compartment values in the plot below:

```
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. rxode2 extracts `units` to label the plot (if they are present).

Chapter 6

rxode2 syntax

This briefly describes the syntax used to define models that `rxode2` will translate into R-callable compiled code. It also describes the communication of variables between `R` and the `rxode2` modeling specification.

6.1 Example

```
# An rxode2 model specification (this line is a comment).

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

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

# ODE describing the PK and PD

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

6.2 Syntax

An `rxode2` 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 rxode2 data specification is found in rxode2 events section.
- special **change point syntax, or model times**. These model times are specified by `mtime(var)=time`
- special **Jacobian-derivative** assignments, where the left hand specifies the change in the compartment ode with respect to a variable. For example, if `d/dt(y) = dy`, then a Jacobian for this compartment can be specified as `df(y)/dy(dy) = 1`. There may be some advantage to obtaining the solution or specifying the Jacobian for very stiff ODE systems. However, for the few stiff systems we tried with LSODA, this actually slightly slowed down the solving.

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

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

Special statements can be:

- **Compartment declaration statements**, which can change the default dosing compartment and the assumed compartment number(s) as well as add extra compartment names at the end (useful for multiple-endpoint nlmixr models); These are specified by `cmt(compartmentName)`
- **Parameter declaration statements**, which can make sure the input parameters are in a certain order instead of ordering the parameters by the order

they are parsed. This is useful for keeping the parameter order the same when using 2 different ODE models. These are specified by `param(par1, par2,...)`

An example model is shown below:

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

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

Expressions in assignment and `if` statements can be numeric or logical.

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 `rxode2` syntax is case-sensitive, i.e., `ABC` is different than `abc`, `Abc`, `ABc`, etc.

6.2.1 Identifiers

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

Identifiers in a model specification can refer to:

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

Currently, the `rxode2` 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 `rxode2` 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 rxode2 code:

- `cmt`
- `dvid`

- `addl`
- `ss`
- `rate`
- `id`

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

- `evid`
- `ii`

Sometimes rxode2 generates variables that are fed back to rxode2. Similarly, nlmixr generates some variables that are used in nlmixr estimation and simulation. These variables start with the either the `rx` or `nlmixr` prefixes. To avoid any problems, it is suggested to not use these variables starting with either the `rx` or `nlmixr` prefixes.

6.3 Logical Operators

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

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

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

6.4 cmt() changing compartment numbers for states

The compartment order can be changed with the `cmt()` syntax in the model. To understand what the `cmt()` can do you need to understand how `rxode2` numbers the compartments.

Below is an example of how rxode2 numbers compartments

6.4.1 How rxode2 numbers compartments

rxode2 automatically assigns compartment numbers when parsing. For example, with the Mavoglurant PBPK model the following model may be used:

```r
library(rxode2)
pbpk <- rxode2({
    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);
})
```

```
#> using C compiler: 'gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0'
```

If you look at the summary, you can see where rxode2 assigned the compartment number(s)

```
summary(pbpk)
```

```
#> rxode2 2.0.13.9000 model named rx_0d8d3c8f765b9ace54057fc27aa07891 model (ready).
#> DLL: /tmp/Rtmp58ugTb/rxode2/rx_0d8d3c8f765b9ace54057fc27aa07891__.rxd/rx_0d8d3c8f765b9ace54057
#> 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"
#> -- rxode2 Model Syntax --
#> rxode2({
#>     KbBR = exp(lKbBR)
#>     KbMU = exp(lKbMU)
#>     KbAD = exp(lKbAD)
#>     CLint = exp(lCLint + eta.LClint)
#>     KbBO = exp(lKbBO)
#>     KbRB = exp(lKbRB)
#>     CO = (187 * WT^0.81) * 60/1000
#>     QHT = 4 * CO/100
#>     QBR = 12 * CO/100
#>     QMU = 17 * CO/100
#>     QAD = 5 * CO/100
#>     QSK = 5 * CO/100
#>     QSP = 3 * CO/100
#>     QPA = 1 * CO/100
#>     QLI = 25.5 * CO/100
#>     QST = 1 * CO/100
#>     QGU = 14 * CO/100
#>     QHA = QLI - (QSP + QPA + QST + QGU)
#>     QBO = 5 * CO/100
#>     QKI = 19 * CO/100
#>     QRB = CO - (QHT + QBR + QMU + QAD + QSK + QLI + QBO + QKI)
#>     QLU = QHT + QBR + QMU + QAD + QSK + QLI + QBO + QKI + QRB
#>     VLU = (0.76 * WT/100)/1.051
#>     VHT = (0.47 * WT/100)/1.03
#>     VBR = (2 * WT/100)/1.036
#>     VMU = (40 * WT/100)/1.041
#>     VAD = (21.42 * WT/100)/0.916
#>     VSK = (3.71 * WT/100)/1.116
#>     VSP = (0.26 * WT/100)/1.054
#>     VPA = (0.14 * WT/100)/1.045
#>     VLI = (2.57 * WT/100)/1.04
#>     VST = (0.21 * WT/100)/1.05
#>     VGU = (1.44 * WT/100)/1.043
#>     VBO = (14.29 * WT/100)/1.99
#>     VKI = (0.44 * WT/100)/1.05
#>     VAB = (2.81 * WT/100)/1.04
#>     VVB = (5.62 * WT/100)/1.04
#>     VRB = (3.86 * WT/100)/1.04
#>     BP = 0.61
#>     fup = 0.028
#>     fub = fup/BP
#>     KbLU = exp(0.8334)
#>     KbHT = exp(1.1205)
#>     KbSK = exp(-0.5238)
#>     KbSP = exp(0.3224)
#>     KbPA = exp(0.3224)
#>     KbLI = exp(1.7604)
#>     KbST = exp(0.3224)
#>     KbGU = exp(1.2026)
#>     KbKI = exp(1.3171)
#>     S15 = VVB * BP/1000
#>     C15 = Venous_Blood/S15
#>     d/dt(Lungs) = QLU * (Venous_Blood/VVB - Lungs/KbLU/VLU)
#>     d/dt(Heart) = QHT * (Arterial_Blood/VAB - Heart/KbHT/VHT)
#>     d/dt(Brain) = QBR * (Arterial_Blood/VAB - Brain/KbBR/VBR)
#>     d/dt(Muscles) = QMU * (Arterial_Blood/VAB - Muscles/KbMU/VMU)
#>     d/dt(Adipose) = QAD * (Arterial_Blood/VAB - Adipose/KbAD/VAD)
#>     d/dt(Skin) = QSK * (Arterial_Blood/VAB - Skin/KbSK/VSK)
#>     d/dt(Spleen) = QSP * (Arterial_Blood/VAB - Spleen/KbSP/VSP)
#>     d/dt(Pancreas) = QPA * (Arterial_Blood/VAB - Pancreas/KbPA/VPA)
#>     d/dt(Liver) = QHA * Arterial_Blood/VAB + QSP * Spleen/KbSP/VSP +
#>         QPA * Pancreas/KbPA/VPA + QST * Stomach/KbST/VST + QGU *
#>         Gut/KbGU/VGU - CLint * fub * Liver/KbLI/VLI - QLI * Liver/KbLI/VLI
#>     d/dt(Stomach) = QST * (Arterial_Blood/VAB - Stomach/KbST/VST)
#>     d/dt(Gut) = QGU * (Arterial_Blood/VAB - Gut/KbGU/VGU)
#>     d/dt(Bones) = QBO * (Arterial_Blood/VAB - Bones/KbBO/VBO)
#>     d/dt(Kidneys) = QKI * (Arterial_Blood/VAB - Kidneys/KbKI/VKI)
#>     d/dt(Arterial_Blood) = QLU * (Lungs/KbLU/VLU - Arterial_Blood/VAB)
#>     d/dt(Venous_Blood) = QHT * Heart/KbHT/VHT + QBR * Brain/KbBR/VBR +
#>         QMU * Muscles/KbMU/VMU + QAD * Adipose/KbAD/VAD + QSK *
#>         Skin/KbSK/VSK + QLI * Liver/KbLI/VLI + QBO * Bones/KbBO/VBO +
#>         QKI * Kidneys/KbKI/VKI + QRB * Rest_of_Body/KbRB/VRB -
#>         QLU * Venous_Blood/VVB
#>     d/dt(Rest_of_Body) = QRB * (Arterial_Blood/VAB - Rest_of_Body/KbRB/VRB)
#> })
```

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

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

To add the compartments to the rxode2 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:

```r
pbpk2 <- rxode2({
  ## 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);
})
```

```
#> using C compiler: ‘gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0’
```

You can see this change in the simple printout

```
pbpk2
```

```
#> rxode2 2.0.13.9000 model named rx_dcbef58b165c0a4cbc2c9a323d7584cd model (ready).
#> x$state: Venous_Blood, Skin, Lungs, Heart, Brain, Muscles, Adipose, Spleen, Pancreas, Liver, S
#> x$params: lKbBR, lKbMU, lKbAD, lCLint, eta.LClint, lKbBO, lKbRB, WT, BP, fup
#> x$lhs: KbBR, KbMU, KbAD, CLint, KbBO, KbRB, CO, QHT, QBR, QMU, QAD, QSK, QSP, QPA, QLI, QST, Q
```

The first two compartments are `Venous_Blood` followed by `Skin`.

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

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

For example this is legal:

```
ode.1c.ka <- rxode2({
    C2 = center/V;
    d / dt(depot) = -KA * depot
    d/dt(center) = KA * depot - CL*C2
    cmt(eff);
})
```

```
#> using C compiler: ‘gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0’
```

```
print(ode.1c.ka)
```

```
#> rxode2 2.0.13.9000 model named rx_cd29660413c35ed2495a372cb7cc0487 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 <- rxode2({
    cmt(eff);
    C2 = center/V;
    d / dt(depot) = -KA * depot
    d/dt(center) = KA * depot - CL*C2
})
```

will give an error:

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

Chapter 7

rxode2 events

7.1 rxode2 event tables

In general, rxode2 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`.
- NONMEM-style events are supported (0: Observation, 1: Dose, 2: Other, 3: Reset, 4: Reset+Dose). 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`.

  - `evid=7` or transit compartment model/phantom event. This puts the dose in the `dose()` function and calculates time since last dose `tad()` but doesn't actually put the dose in the compartment. This allows the `transit()` function to easily apply to the compartment.

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;7=Transit
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; `rxode2` 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 `rxode2` 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 `rxode2` tutorial.

```
library(rxode2)
### Model from rxode2 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
});
```

```
#> using C compiler: ‘gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0’
```

7.2 Bolus/Additive Doses

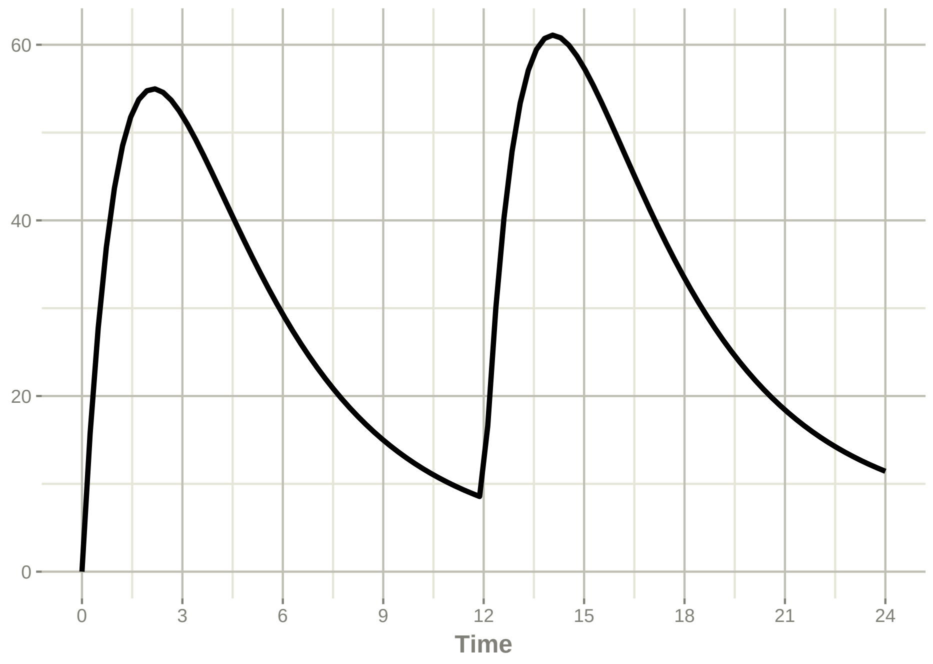
A bolus dose is the default type of dose in `rxode2` and only requires the `amt`/`dose`. Note that this uses the convenience function `et()` described in the rxode2 event tables

```
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        NA  NA    NA 0:Observation
#>  2 0     10000  12     2 1:Dose (Add)
#>  3 0.242    NA  NA    NA 0:Observation
#>  4 0.485    NA  NA    NA 0:Observation
#>  5 0.727    NA  NA    NA 0:Observation
#>  6 0.970    NA  NA    NA 0:Observation
#>  7 1.21     NA  NA    NA 0:Observation
#>  8 1.45     NA  NA    NA 0:Observation
#>  9 1.70     NA  NA    NA 0:Observation
#> 10 1.94     NA  NA    NA 0:Observation
#> # i 91 more rows
```

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



7.3 Infusion Doses

There are a few different type of infusions that `rxode2` supports:

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

7.3.1 Constant Infusion (in terms of duration and rate)

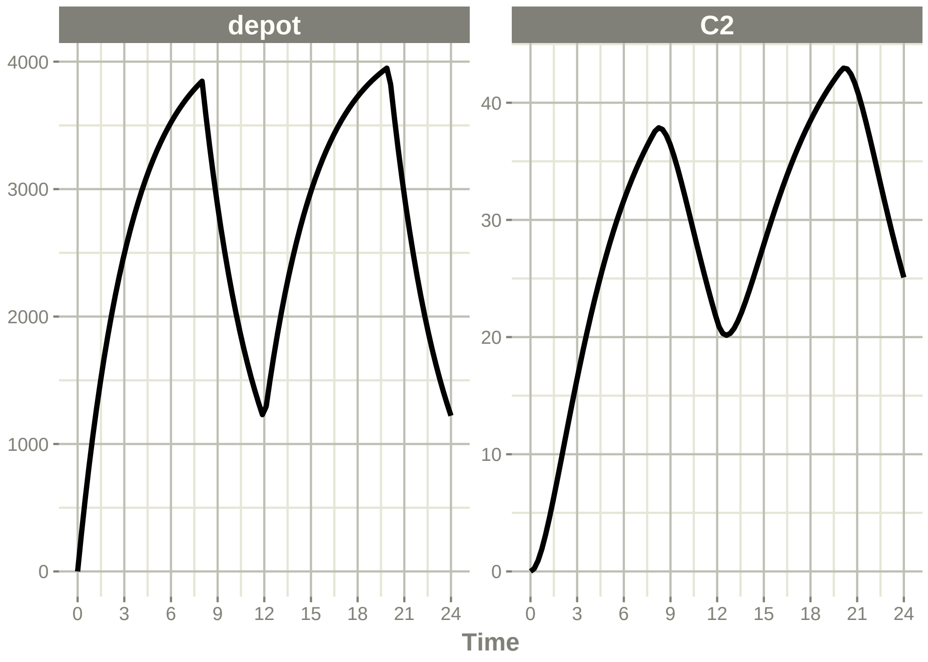
The next type of event is an infusion; There are two ways to specify an infusion; The first is the `dur` keyword.

An example of this is:

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

ev
```

```
#> -- EventTable with 101 records --
#> 1 dosing records (see x$get.dosing(); add with add.dosing
#> or et)
#> 100 observation times (see x$get.sampling(); add with
#> add.sampling or et)
#> multiple doses in `addl` columns, expand with x$expand();
#> or etExpand(x)
#> -- First part of x: --
#> # A tibble: 101 x 6
#>     time   amt  ii  addl evid          dur
#>      [h] <dbl> [h] <int> <evid>        [h]
#>  1 0        NA  NA    NA 0:Observation  NA
#>  2 0     10000  12     2 1:Dose (Add)    8
#>  3 0.242    NA  NA    NA 0:Observation  NA
#>  4 0.485    NA  NA    NA 0:Observation  NA
#>  5 0.727    NA  NA    NA 0:Observation  NA
#>  6 0.970    NA  NA    NA 0:Observation  NA
#>  7 1.21     NA  NA    NA 0:Observation  NA
#>  8 1.45     NA  NA    NA 0:Observation  NA
#>  9 1.70     NA  NA    NA 0:Observation  NA
#> 10 1.94     NA  NA    NA 0:Observation  NA
#> # i 91 more rows
```

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

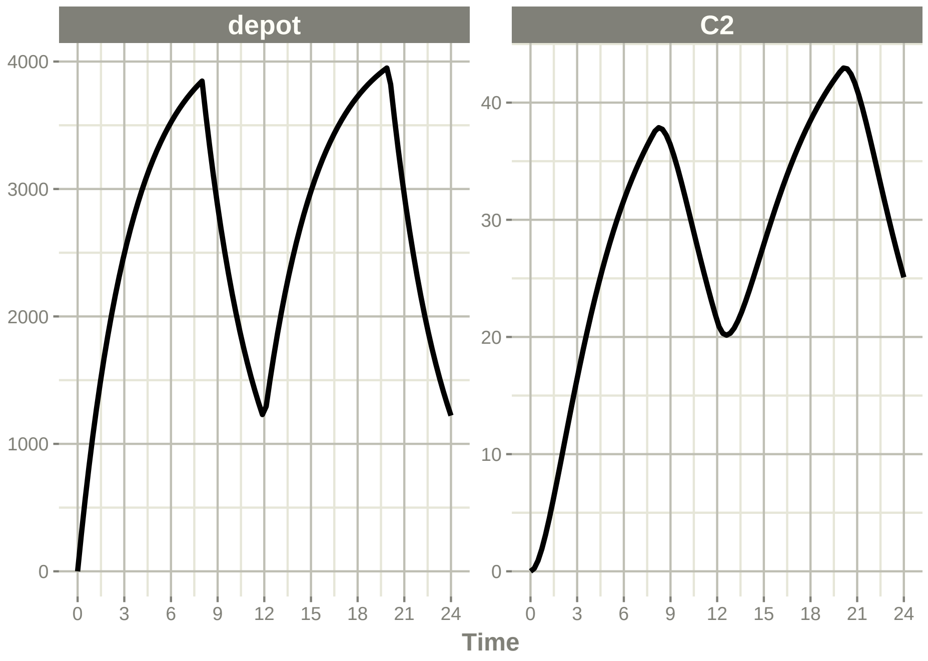


It can be also specified by the `rate` component:

```r
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        NA NA          NA    NA 0:Observation
#>  2 0     10000  1250       12     2 1:Dose (Add)
#>  3 0.242    NA NA          NA    NA 0:Observation
#>  4 0.485    NA NA          NA    NA 0:Observation
#>  5 0.727    NA NA          NA    NA 0:Observation
#>  6 0.970    NA NA          NA    NA 0:Observation
#>  7 1.21     NA NA          NA    NA 0:Observation
#>  8 1.45     NA NA          NA    NA 0:Observation
#>  9 1.70     NA NA          NA    NA 0:Observation
#> 10 1.94     NA NA          NA    NA 0:Observation
#> # i 91 more rows
```

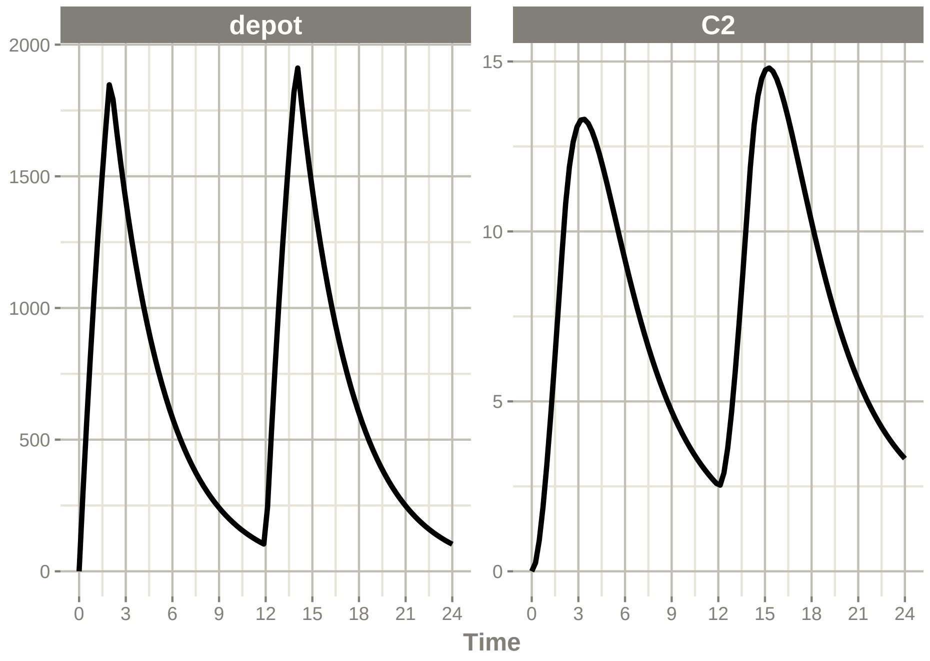
```r
rxSolve(m1, ev) %>% plot(depot, C2) +
    xlab("Time")
```



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

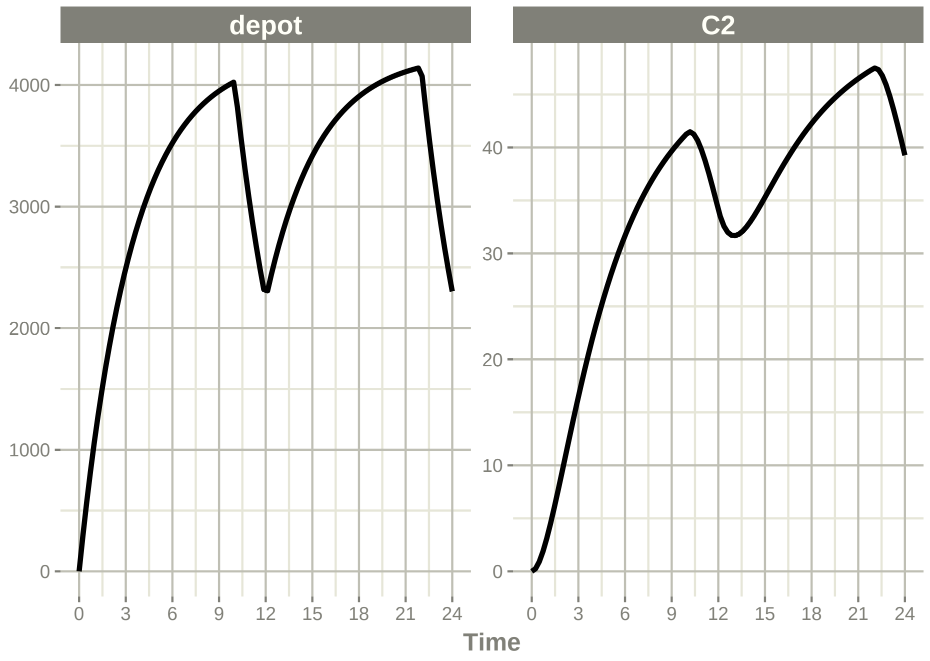
In the case of modeling `rate`, a bioavailability decrease, decreases the infusion duration, as in NONMEM. For example:

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



Similarly increasing the bioavailability increases the infusion duration.

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



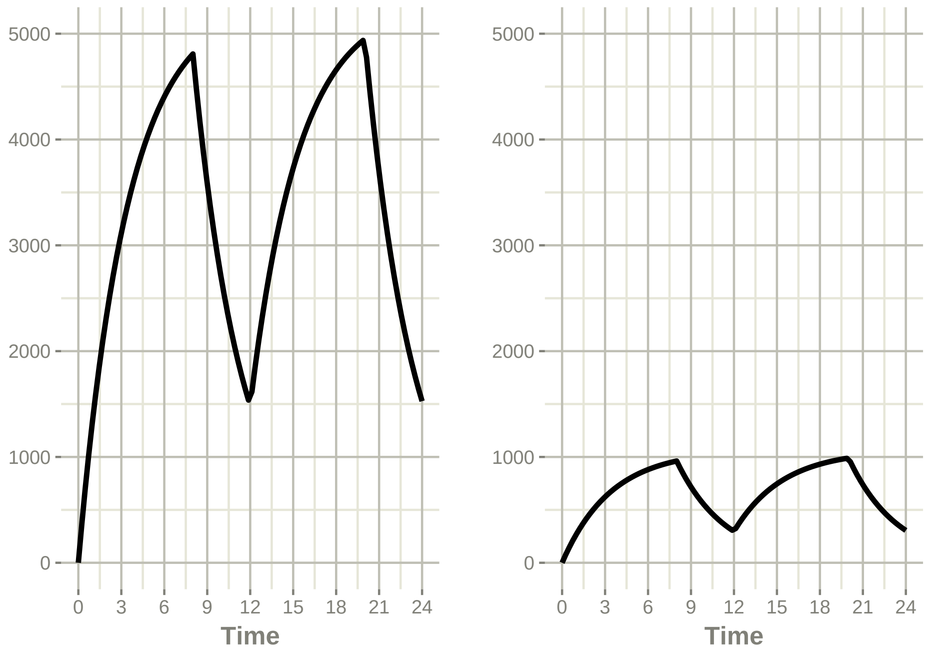
The rationale for this behavior is that the `rate` and `amt` are specified by the event table, so the only thing that can change with a bioavailability increase is the duration of the infusion.

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

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

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

```
library(ggplot2)
library(patchwork)

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

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

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



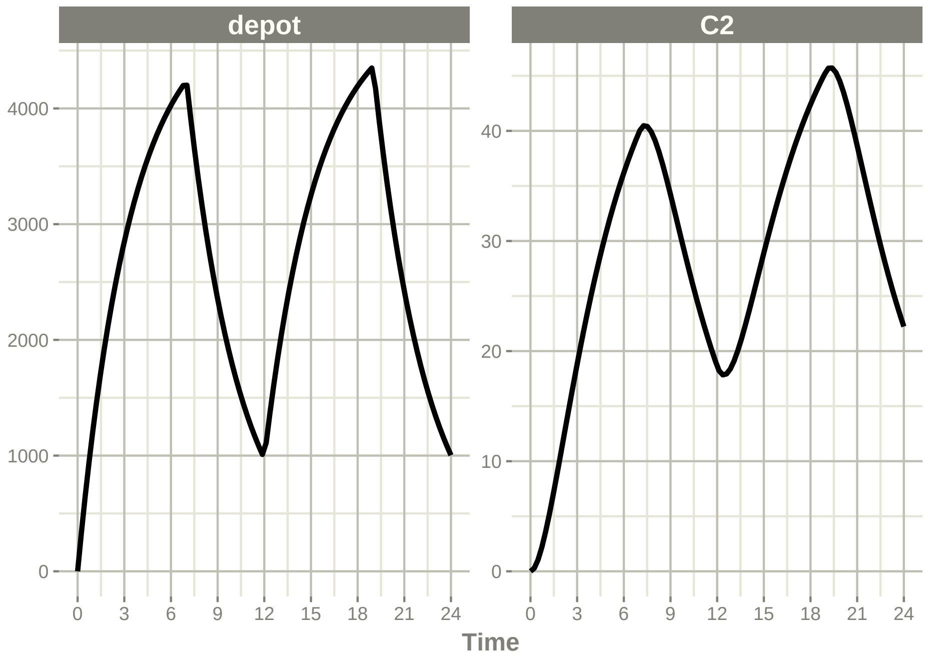
7.3.2 Modeled Rate and Duration of Infusion

You can model the duration, which is equivalent to NONMEM's `rate=-2`.

```
ev <- et(timeUnits="hr") %>%
    et(amt=10000, ii=12,until=24, rate=-2) %>%
    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        NA NA          NA    NA 0:Observation
#>  2 0     10000 -2:dur      12     2 1:Dose (Add)
#>  3 0.242    NA NA          NA    NA 0:Observation
#>  4 0.485    NA NA          NA    NA 0:Observation
#>  5 0.727    NA NA          NA    NA 0:Observation
#>  6 0.970    NA NA          NA    NA 0:Observation
#>  7 1.21     NA NA          NA    NA 0:Observation
#>  8 1.45     NA NA          NA    NA 0:Observation
#>  9 1.70     NA NA          NA    NA 0:Observation
#> 10 1.94     NA NA          NA    NA 0:Observation
#> # i 91 more rows
```

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

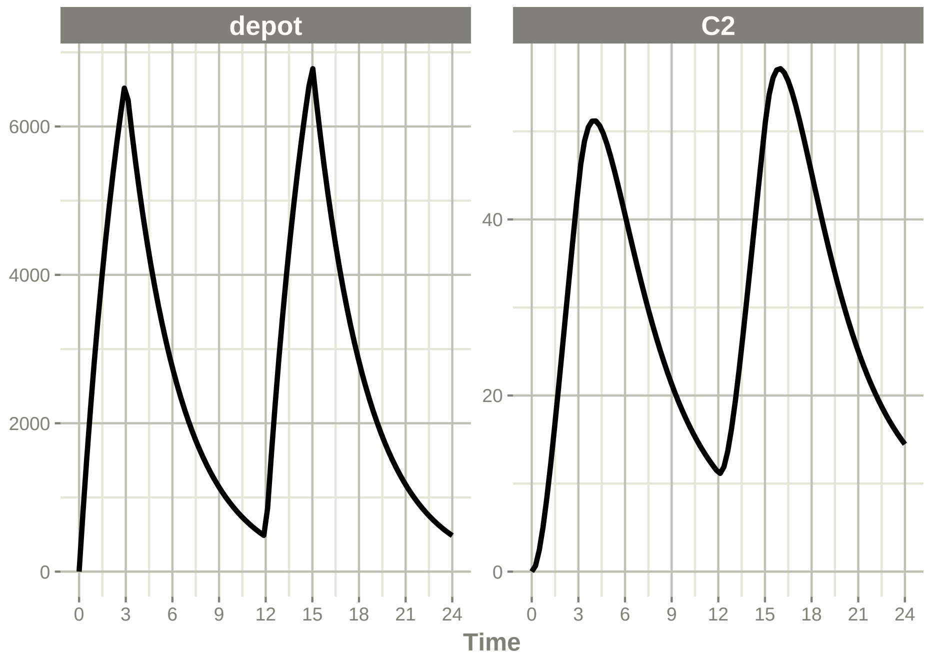


Similarly, you may also model rate. This is equivalent to NONMEM's `rate=-1` and is how `rxode2`'s event table specifies the data item as well.

```
ev <- et(timeUnits="hr") %>%
    et(amt=10000, ii=12,until=24, rate=-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)
#> 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        NA NA          NA    NA 0:Observation
#>   2 0     10000 -1:rate     12     2 1:Dose (Add)
#>   3 0.242    NA NA          NA    NA 0:Observation
#>   4 0.485    NA NA          NA    NA 0:Observation
#>   5 0.727    NA NA          NA    NA 0:Observation
#>   6 0.970    NA NA           NA    NA 0:Observation
#>   7 1.21     NA NA           NA    NA 0:Observation
#>   8 1.45     NA NA           NA    NA 0:Observation
#>   9 1.70     NA NA           NA    NA 0:Observation
#> 10 1.94     NA NA           NA    NA 0:Observation
#> # i 91 more rows
```

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



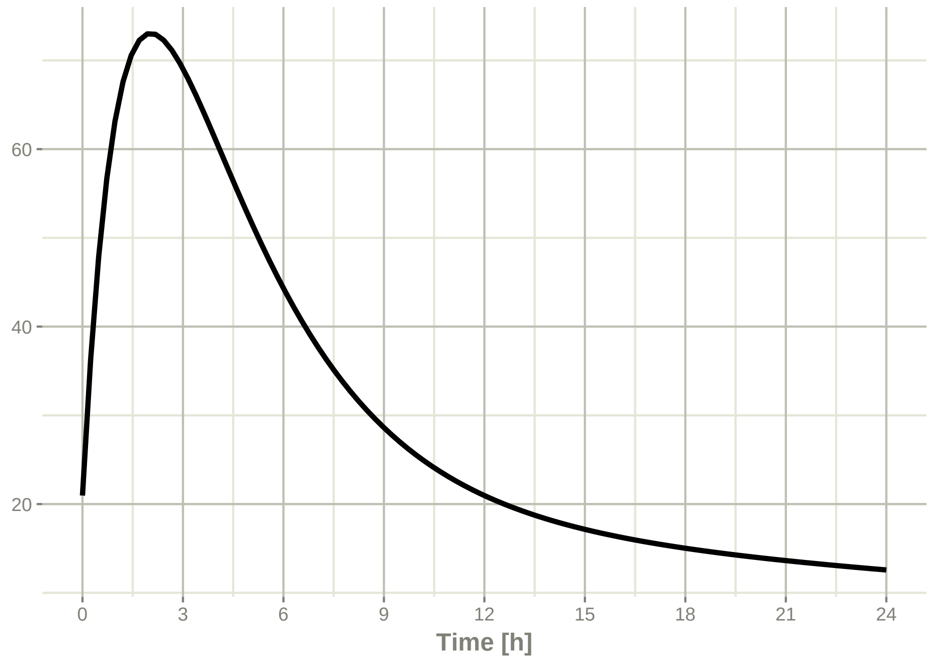
7.4 Steady State

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

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

ev
```

```
#> -- EventTable with 101 records --
#> 1 dosing records (see x$get.dosing(); add with add.dosing
#> or et)
#> 100 observation times (see x$get.sampling(); add with
#> add.sampling or et)
#> -- First part of x: --
#> # A tibble: 101 x 5
#>     time   amt  ii evid              ss
#>      [h] <dbl> [h] <evid>         <int>
#>  1 0        NA  NA 0:Observation    NA
#>  2 0     10000  12 1:Dose (Add)      1
#>  3 0.242    NA  NA 0:Observation    NA
#>  4 0.485    NA  NA 0:Observation    NA
#>  5 0.727    NA  NA 0:Observation    NA
#>  6 0.970    NA  NA 0:Observation    NA
#>  7 1.21     NA  NA 0:Observation    NA
#>  8 1.45     NA  NA 0:Observation    NA
#>  9 1.70     NA  NA 0:Observation    NA
#> 10 1.94     NA  NA 0:Observation    NA
#> # i 91 more rows
```

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

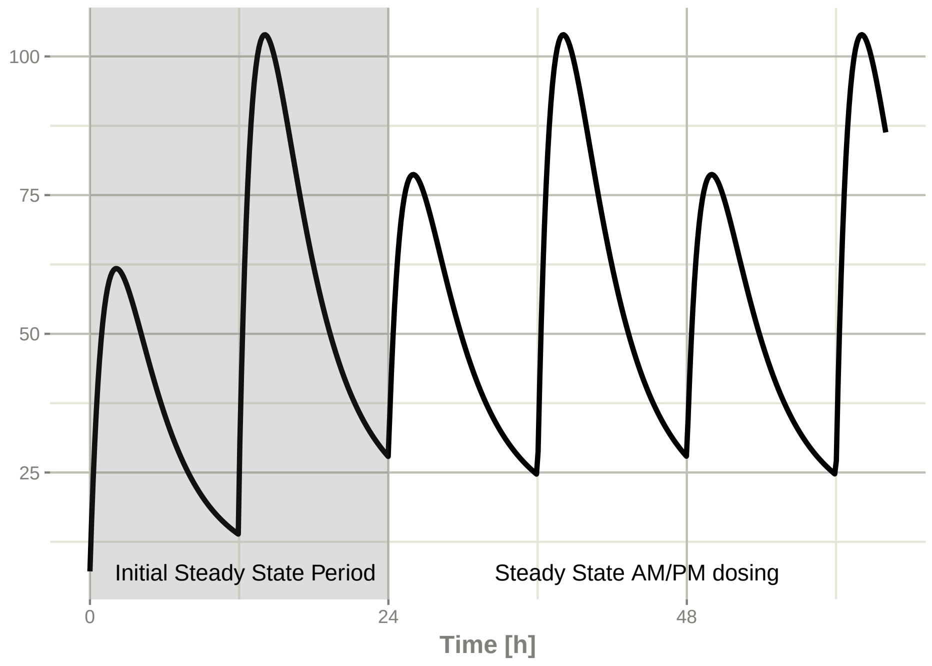


7.4.1 Steady state for complex dosing

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

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

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

library(ggplot2)

rxSolve(m1, ev,maxsteps=10000) %>% plot(C2) +
  annotate("rect", xmin=0, xmax=24, ymin=-Inf, ymax=Inf,
           alpha=0.2) +
  annotate("text", x=12.5, y=7,
           label="Initial Steady State Period") +
  annotate("text", x=44,   y=7,
           label="Steady State AM/PM dosing")
```



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

7.4.2 Steady state for constant infusion or zero order processes

The last type of steady state that `rxode2` supports is steady-state constant infusion rate. This can be specified the same way as NONMEM, that is:

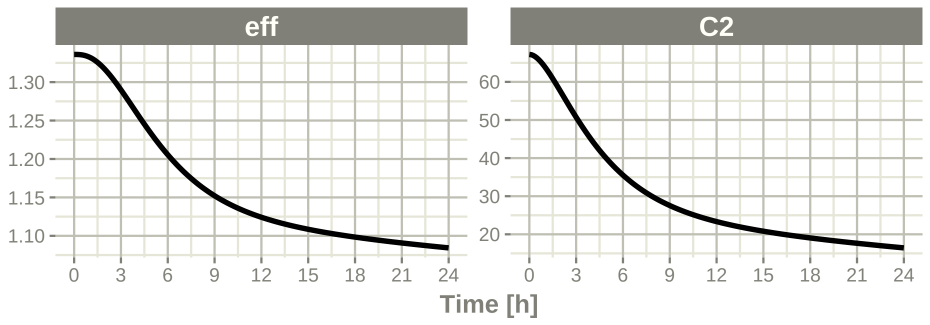
- No inter-dose interval `ii=0`
- A steady state dose, ie `ss=1`
- Either a positive rate (`rate>0`) or a estimated rate `rate=-1`.
- A zero dose, ie `amt=0`
- Once the steady-state constant infusion is achieved, the infusion is turned off when using this record, just like NONMEM.

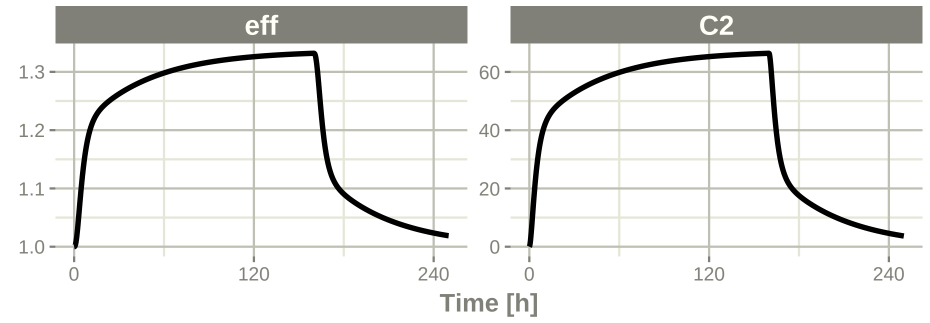
Note that `rate=-2` where we model the duration of infusion doesn't make much sense since we are solving the infusion until steady state. The duration is specified by the steady state solution.

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:

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

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


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

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

library(patchwork)

p1 / p2
```





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

7.5 Reset Events

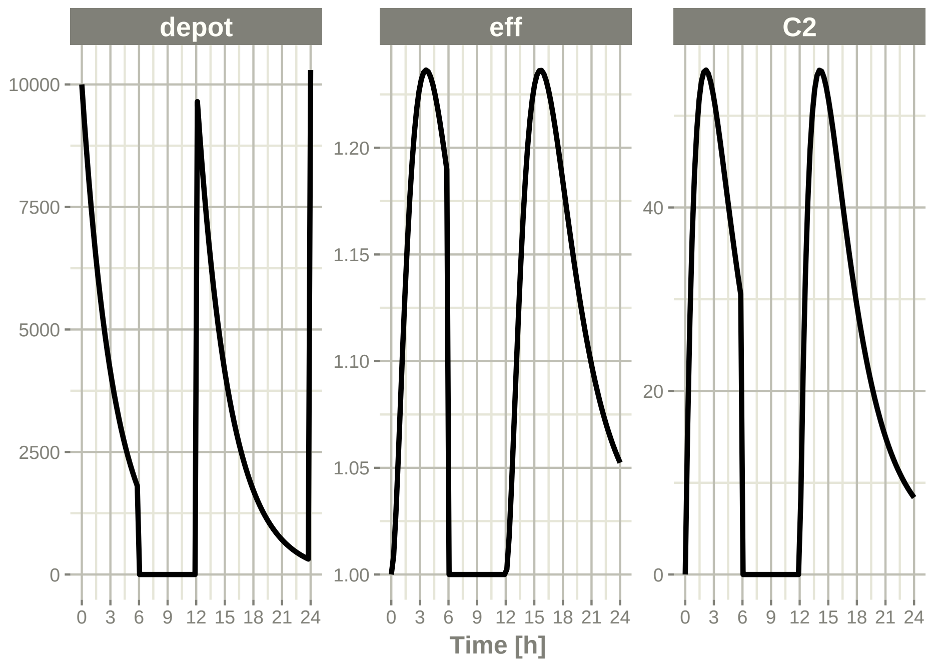
Reset events are implemented by `evid=3` or `evid=reset`, for reset and `evid=4` for reset and dose.

```
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         NA   NA     NA 0:Observation
#>   2 0      10000   12      3 1:Dose (Add)
#>   3 0.242     NA   NA     NA 0:Observation
#>   4 0.485     NA   NA     NA 0:Observation
#>   5 0.727     NA   NA     NA 0:Observation
#>   6 0.970     NA   NA     NA 0:Observation
#>   7 1.21      NA   NA     NA 0:Observation
#>   8 1.45      NA   NA     NA 0:Observation
#>   9 1.70      NA   NA     NA 0:Observation
#> 10 1.94      NA   NA     NA 0:Observation
#> # i 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)
```



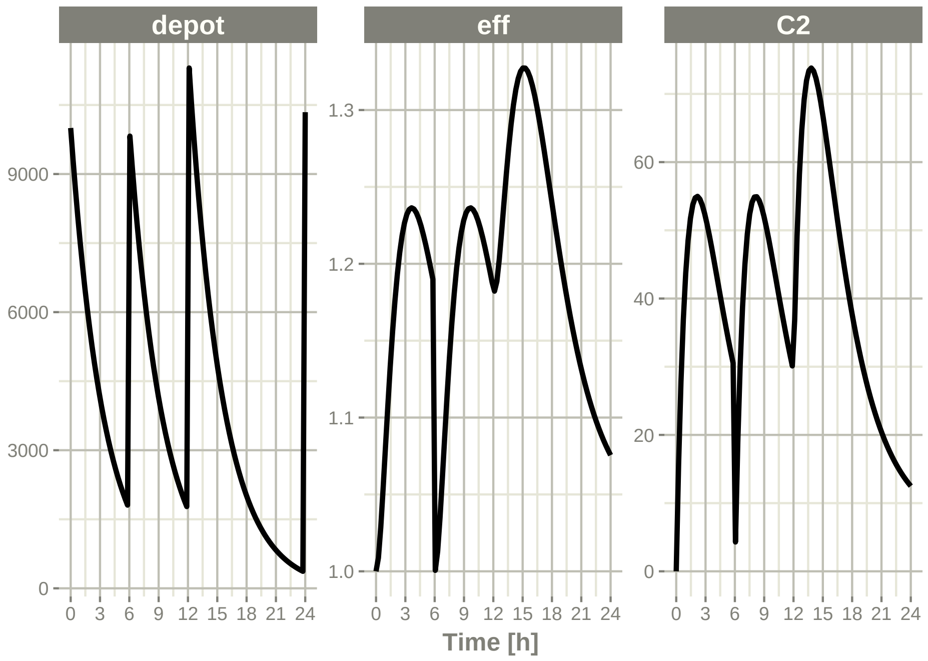
You can see all the compartments are reset to their initial values. The next dose start the dosing cycle over.

```
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        NA  NA    NA 0:Observation
#>  2 0     10000  12     3 1:Dose (Add)
#>  3 0.242    NA  NA    NA 0:Observation
#>  4 0.485    NA  NA    NA 0:Observation
#>  5 0.727    NA  NA    NA 0:Observation
#>  6 0.970    NA  NA    NA 0:Observation
#>  7 1.21     NA  NA    NA 0:Observation
#>  8 1.45     NA  NA    NA 0:Observation
#>  9 1.70     NA  NA    NA 0:Observation
#> 10 1.94     NA  NA    NA 0:Observation
#> # i 92 more rows
```

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

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



7.6 Turning off compartments

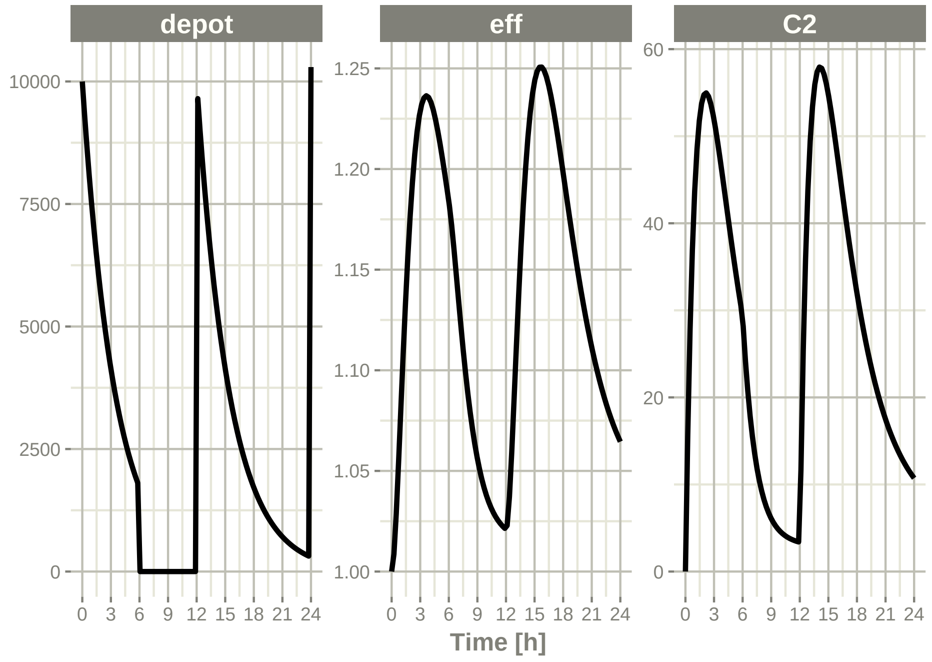
You may also turn off a compartment, which is similar to a reset event.

```r
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    (obs)         NA  NA    NA 0:Observation
#>   2 0    (default) 10000  12     3 1:Dose (Add)
#>  3 0.242 (obs)        NA  NA    NA 0:Observation
#>  4 0.485 (obs)        NA  NA    NA 0:Observation
#>  5 0.727 (obs)        NA  NA    NA 0:Observation
#>  6 0.970 (obs)        NA  NA    NA 0:Observation
#>  7 1.21  (obs)        NA  NA    NA 0:Observation
#>  8 1.45  (obs)        NA  NA    NA 0:Observation
#>  9 1.70  (obs)        NA  NA    NA 0:Observation
#> 10 1.94  (obs)        NA  NA    NA 0:Observation
#> # i 92 more rows
```

Solving shows what this does in the system:

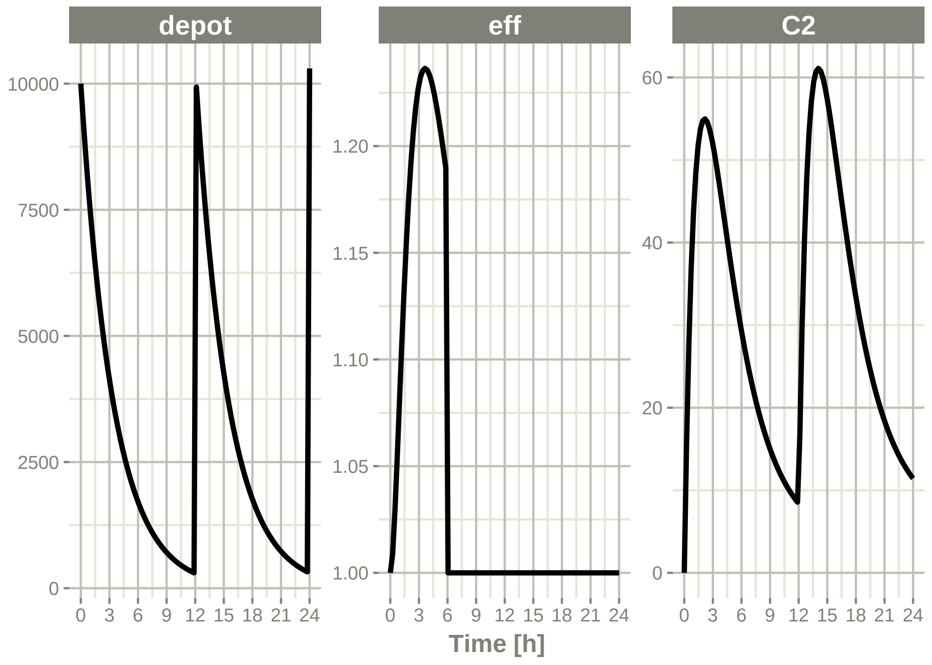
```
rxSolve(m1, ev) %>% plot(depot,C2, eff)
```



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.

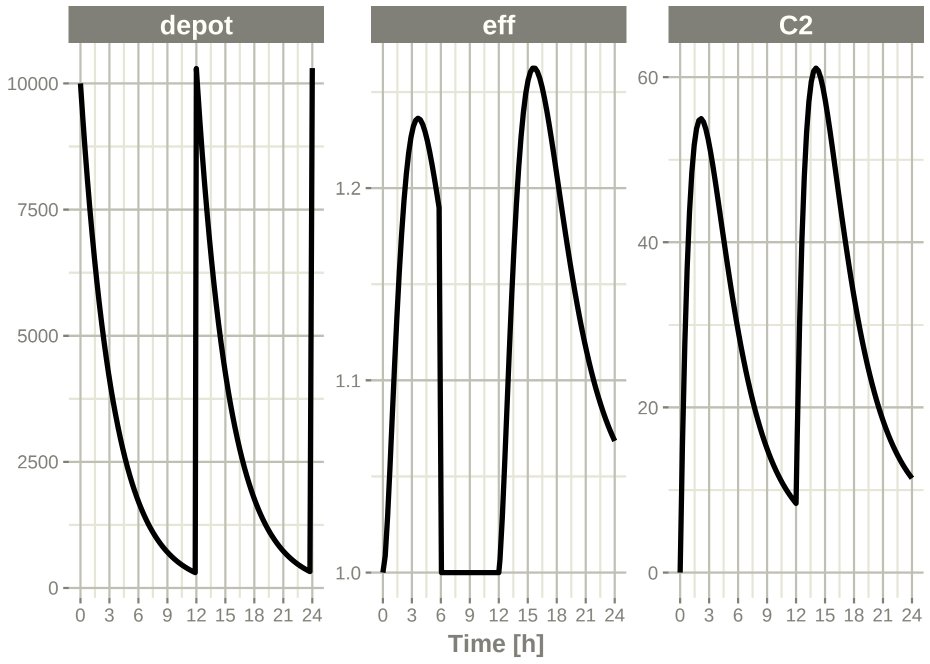
Note that a dose to a compartment only turns back on the compartment that was dosed. Hence if you turn off the effect compartment, it continues to be off after another dose to the depot.

```
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)
```



7.7 Classic rxode2 events

Originally RxODE supported compound event IDs; rxode2 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 rxode2 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:

7.7.1 Bolus Dose Examples

A 100 bolus dose to compartment #1 at time 0

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

A 100 bolus dose to compartment #99 at time 0

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

A 100 bolus dose to compartment #199 at time 0

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

7.7.2 Infusion Event Examples

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

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

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

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

Modeled rate with amount of 50

time	evid	amt
0	90101	50
0	70101	50
0.5	0	0
1	0	0

Modeled duration with amount of 50

time	evid	amt
0	80101	50

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

7.7.3 Steady State for classic RxODE EVID example

Steady state dose to cmt 1

time	evid	amt
0	110	50

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

time	evid	amt
0	110	50
12	120	100

7.7.4 Turning off a compartment with classic RxODE EVID

Turn off the first compartment at time 12

time	evid	amt
0	110	50
12	130	NA

Event coding in `rxode2` 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`.

7.8 Datasets for rxode2 & nlmixr

Data for input into `nlmixr` is the same type of data input for `rxode2`, and it is similar to data for NONMEM (most NONMEM-ready datasets can be used directly in `nlmixr`).

7.9 Columns Described by Type of Use

7.9.1 Subject Identification Columns

The subject identification column separates subjects for identification of random effects.

- `ID`: A subject identifier that may be an integer, character, or factor.

7.9.2 Observation Columns

Observation columns are used to indicate the dependent variable and how to use or measure it.

- `DV`: A numeric column with the measurement
- `CENS`: A numeric column for indication of censoring, such as below the limit of quantification for an assay.
- `LIMIT`: A numeric column for helping indicate the type of censoring, such as below the limit of quantification for an assay.
- `MDV`: An indicator for missing `DV` values
- `CMT`: The name or number of the compartment
- `DVID`: The dependent variable identifier
- `EVID` The event identifier

7.9.3 Dosing Columns

- `AMT`: The amount of the dose
- `CMT`: The name or number of the compartment
- `EVID`: The event identifier
- `ADDL`: The number of additional doses
- `RATE` or `DUR`: The rate or duration of a dose

7.9.4 Covariate Columns

7.10 Details for Specific Dataset Columns

The details below are sorted alphabetically by column name. For grouping by use, see the documentation above.

7.10.1 `AMT` Column

The `AMT` column defines the amount of a dose.

For observation rows, it should be `0` or `NA`.

For dosing rows, it is the amount of the dose administered to the `CMT`. If the dose has a zero-order rate (such as a constant infusion), the infusion may be setup using the `RATE` or `DUR` column.

7.10.2 `CENS/LIMIT` Columns

The `CENS` column is an indicator column indicating if censoring occurred. For pharmacokinetic modeling, censoring is typically when a sample is below the limit of quantification. Internally `rxode2` saves these values so that `nlmixr` can use them in likelihood calculations.

`CENS = 0` indicates that the value in `DV` is measured without censoring.

`CENS = 1` indicates that a value is left censored (or below the limit of quantitation) and that the value in `DV` is censoring/quantitation limit.

`CENS = -1` indicates that a value is right censored (or above limit of quantitation) and that the value in `DV` is censoring/quantitation limit.

The `LIMIT` is additional information about how censoring is handled with `nlmixr` and is stored in `rxode2`'s data structure as well. When a value is left censored, like below a limit of `1` you may also believe that the value is above a certain threshold, like zero. In this case, a limit of `0` indicates that the censored value is between `0` and `1`.

In short when:

`CENS = 0` a `LIMIT` is ignored because the observation is not censored

`CENS = 1` the value is censored between (`LIMIT`, `DV`)

`CENS = -1` the value is censored between (`DV`, `LIMIT`)

7.10.3 `CMT` Column

The `CMT` column indicates the compartment where an event occurs. When given as a character string or factor (the preferred method), it is matched by name in the model. When given as an integer, it is matched by the order that compartments appear in the model.

7.10.4 `DUR` Column

The `DUR` column defines the duration of an infusion. It is used to set the duration of a zero-order rate of infusion.

7.10.5 `DV` Column

The `DV` column indicates the current measurement in the current compartment (see `CMT`) with the current measurement identifier (see `DVID`) which may be missing (see `MDV`) or censored (see `CENS`).

7.10.6 `DVID` Column

TODO

7.10.7 `EVID` Column

The `EVID` column is the event identifier for a row of data.

For observation records, it will be `0`. For normal dosing records, it will be `1`. Many more `EVID` values are detailed in the rxode2 Event Types and Classic rxode2 Events vignettes.

7.10.8 `ID` Column

The `ID` column is a subject identifier. This column is used to separate one individual (usually a single person or animal) from another.

In the model, the `ID` column is used to separate individuals. The numerical integrator re-initializes with each new individual, and new values for all random effects are selected.

7.10.9 `RATE` Column

TODO

Chapter 8

Easily creating rxode2 events

An event table in rxode2 is a specialized data frame that acts as a container for all of rxode2's events and observation times.

To create an rxode2 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 rxode2 supports.

```r
library(rxode2)
library(units)
```

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

```r
(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

```r
(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 rxode2 tutorial:

```
## Model from rxode2 tutorial
m1 <-rxode2({
    KA=2.94E-01;
    CL=1.86E+01;
    V2=4.02E+01;
    Q=1.05E+01;
    V3=2.97E+02;
    Kin=1;
    Kout=1;
    EC50=200;
    ## Added modeled bioavaiblity, duration and rate
    fdepot = 1;
    durDepot = 8;
    rateDepot = 1250;
    C2 = centr/V2;
    C3 = peri/V3;
    d/dt(depot) =-KA*depot;
    f(depot) = fdepot
    dur(depot) = durDepot
    rate(depot) = rateDepot
    d/dt(centr) = KA*depot - CL*C2 - Q*C2 + Q*C3;
    d/dt(peri)  =                    Q*C2 - Q*C3;
    d/dt(eff)  = Kin - Kout*(1-C2/(EC50+C2))*eff;
    eff(0) = 1
})
```

8.1 Adding doses to the event table

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

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

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

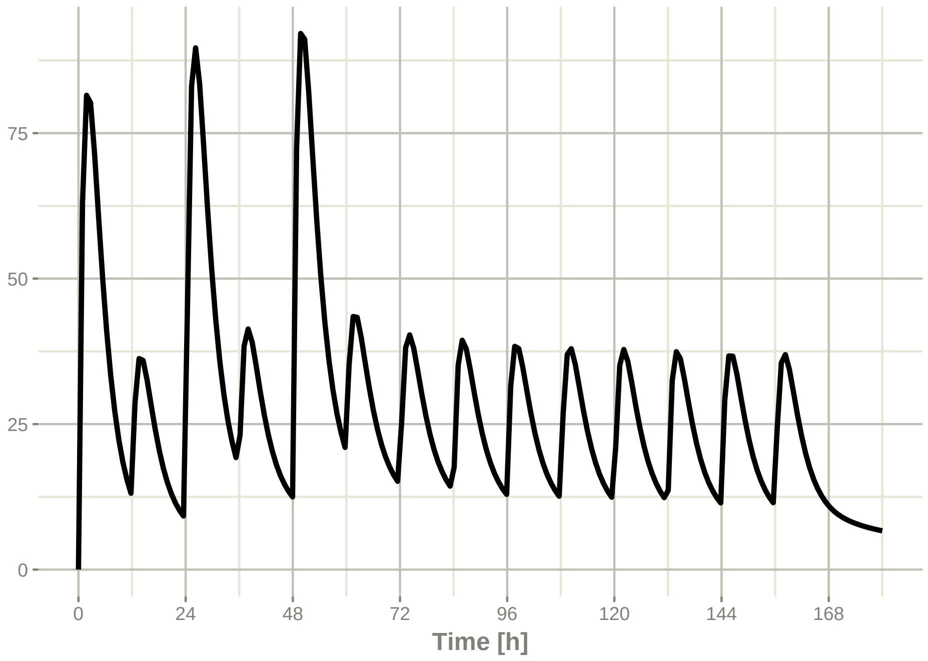
```r
ev <- eventTable(amount.units="mg", time.units="hr")

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

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

ev
```

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

Notice that the units were specified in the table. When specified, the units use the `units` package to keep track of the units and convert them if needed. Additionally, ggforce uses them to label the ggplot axes. The set_units and drop_units are useful to set and drop the rxode2 event table units.

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

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



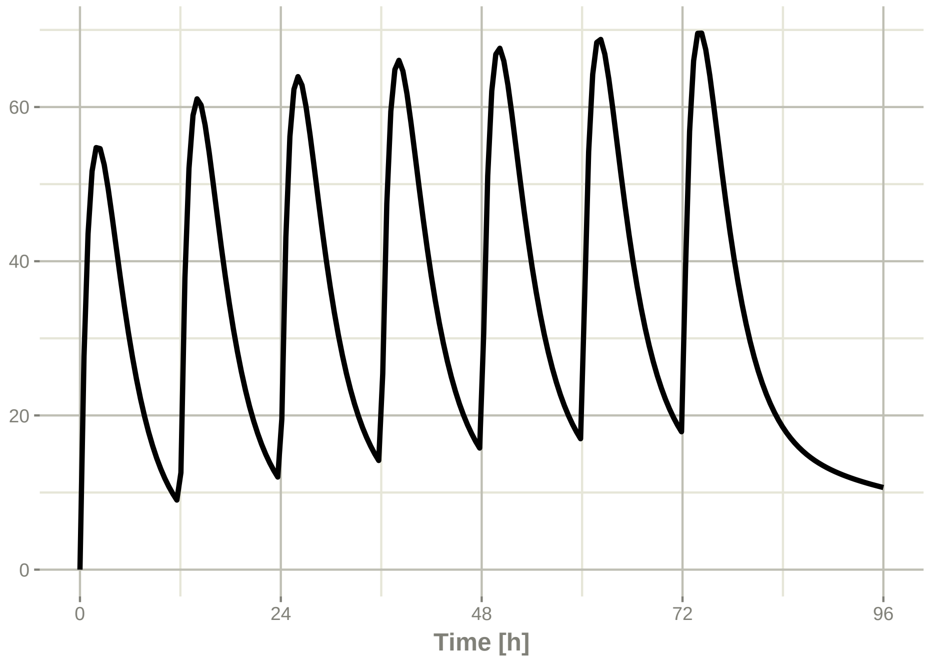
If you are more familiar with the NONMEM/rxode2 event records, you can also specify dosing using et with the dose elements directly:

```
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.

8.2 Adding sampling to an event table

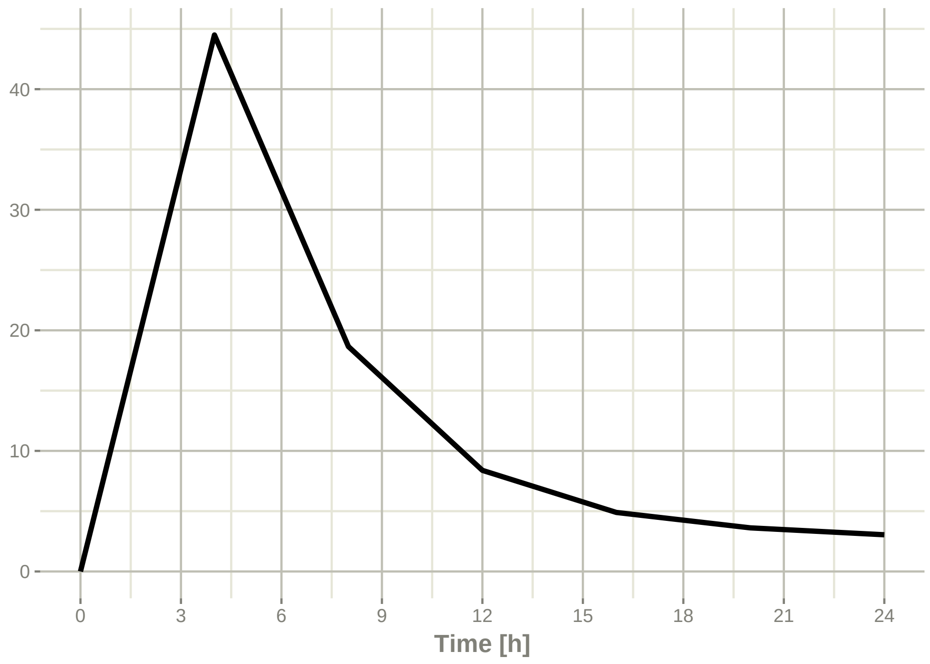
If you notice in the above examples, rxode2 generated some default sampling times since there was not any sampling times. If you wish more control over the sampling time, you should add the samples to the rxode2 event table by `add.sampling` or `et`

```
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)
```



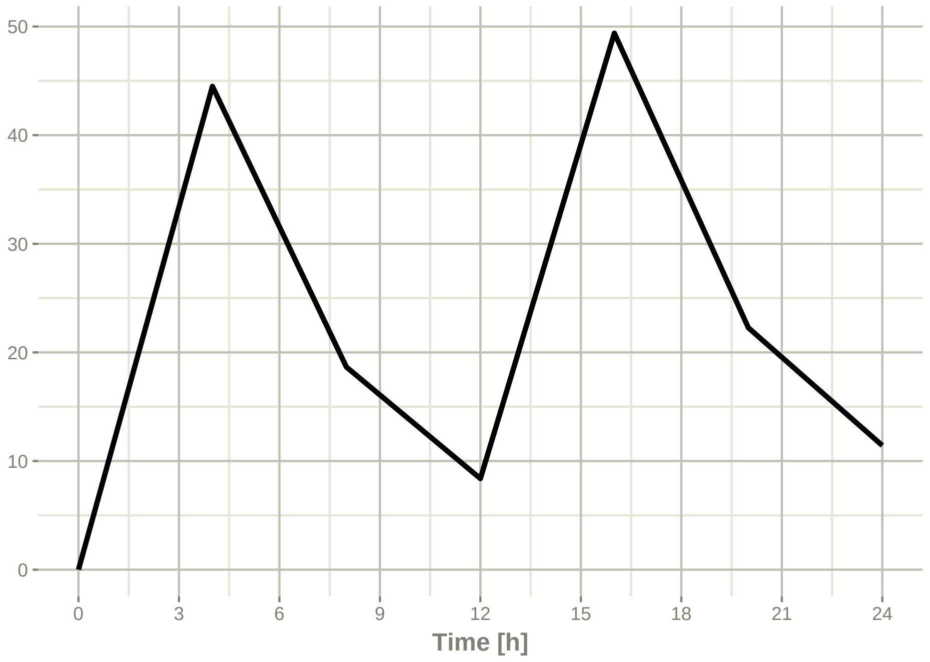
Or if you use `et` you can simply add them in a similar way to `add.sampling`:

```
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 rxode2 solve:

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



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

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

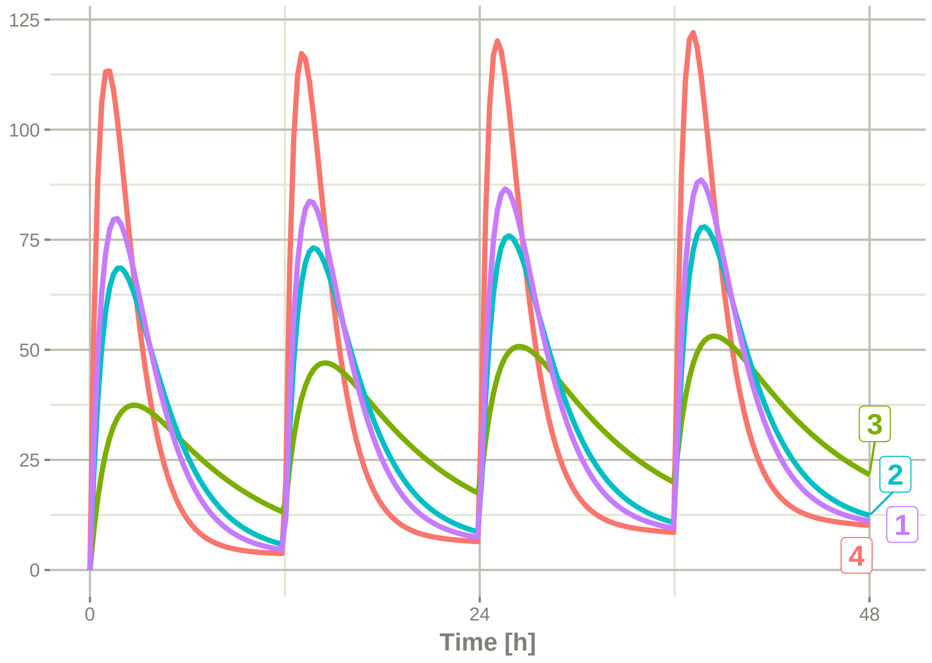
The only thing that is needed to expand an event table is a list of IDs that you want to expand;

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

ev
```

```
#> -- EventTable with 804 records --
#> 4 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        NA  NA    NA 0:Observation
#>  2      1 0     10000  12     6 1:Dose (Add)
#>  3      1 0.241    NA  NA    NA 0:Observation
#>  4      1 0.482    NA  NA    NA 0:Observation
#>  5      1 0.724    NA  NA    NA 0:Observation
#>  6      1 0.965    NA  NA    NA 0:Observation
#>  7      1 1.21     NA  NA    NA 0:Observation
#>  8      1 1.45     NA  NA    NA 0:Observation
#>  9      1 1.69     NA  NA    NA 0:Observation
#> 10      1 1.93     NA  NA    NA 0:Observation
#> # i 794 more rows
```

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

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



8.4 Add doses and samples within a sampling window

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

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

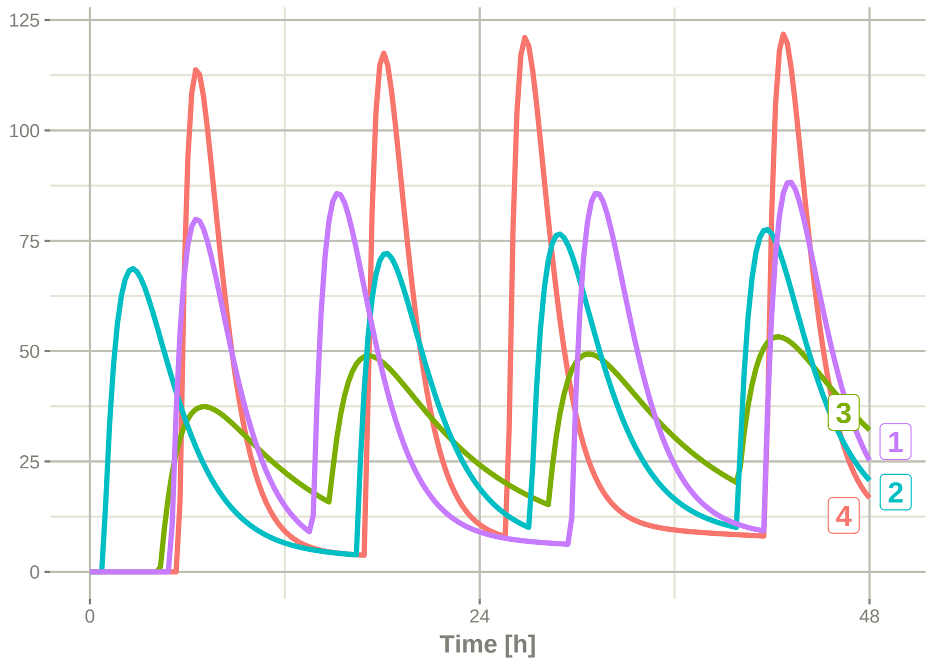
```
set.seed(42)
rxSetSeed(42)
ev <- et(timeUnits="hr") %>%
  et(time=list(c(0,6)), amt=10000, until = set_units(2, days),
     ii=12) %>% # loading doses
  et(id=1:4)

ev
```

```
#> -- EventTable with 16 records --
#> 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  5.49     6 10000 1:Dose (Add)
#>  2     1  12 17.0     18 10000 1:Dose (Add)
#>  3     1  24 25.7     30 10000 1:Dose (Add)
#>  4     1  36 41.6     42 10000 1:Dose (Add)
#>  5     2   0  4.31     6 10000 1:Dose (Add)
#>  6     2  12 14.7     18 10000 1:Dose (Add)
#>  7     2  24 28.2     30 10000 1:Dose (Add)
#>  8     2  36 39.9     42 10000 1:Dose (Add)
#>  9     3   0  0.808    6 10000 1:Dose (Add)
#> 10     3  12 16.4     18 10000 1:Dose (Add)
#> 11     3  24 27.1     30 10000 1:Dose (Add)
#> 12     3  36 39.9     42 10000 1:Dose (Add)
#> 13     4   0  4.98     6 10000 1:Dose (Add)
#> 14     4  12 13.7     18 10000 1:Dose (Add)
#> 15     4  24 29.6     30 10000 1:Dose (Add)
#> 16     4  36 41.5     42 10000 1:Dose (Add)
```

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

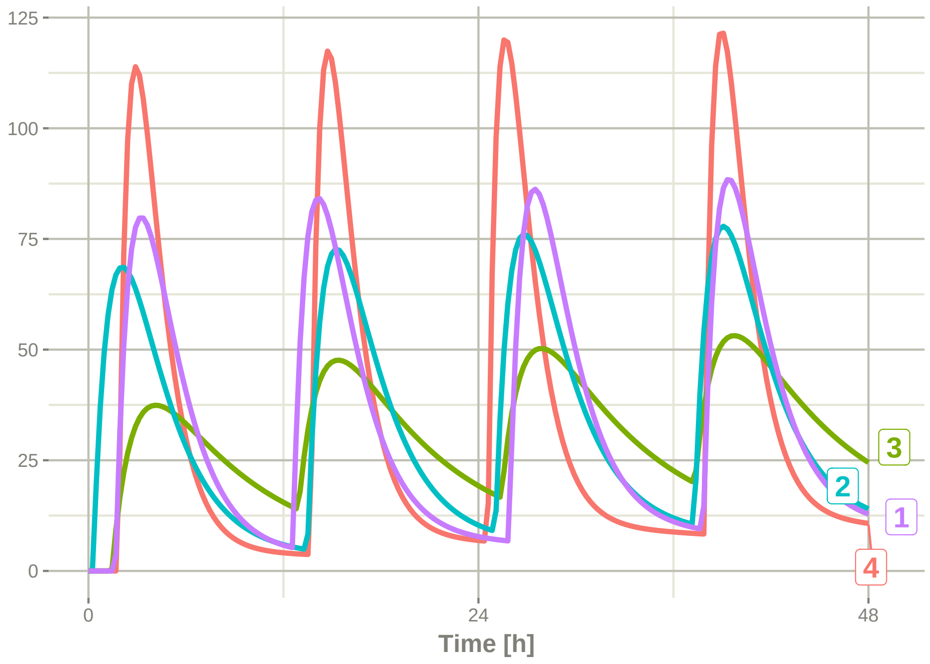
```
ev <- ev %>% et(seq(0,48,length.out=200))
```

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



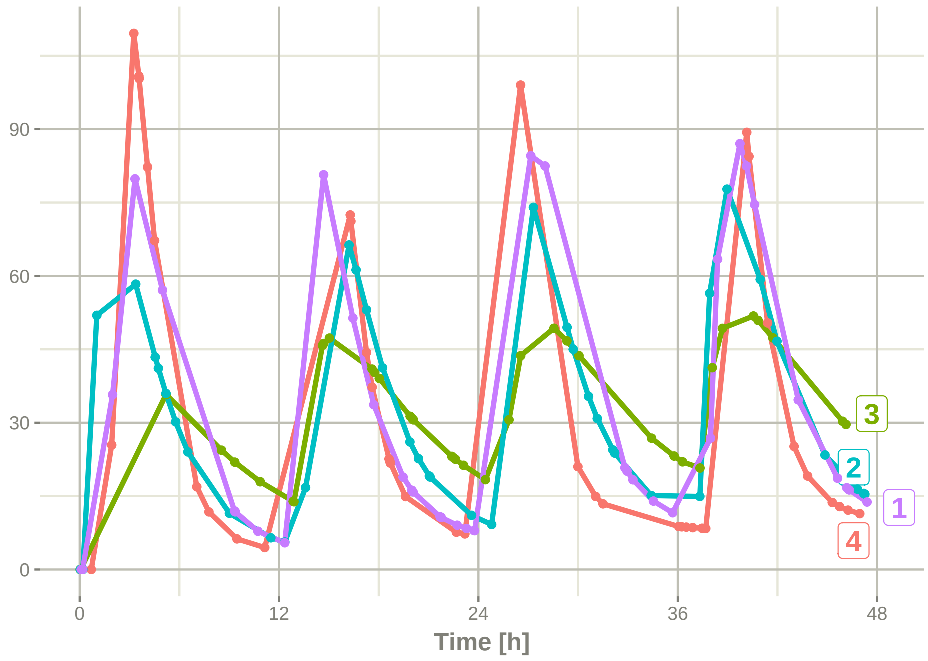
Of course in reality the dosing interval may only be 2 hours:

```r
set.seed(42)
rxSetSeed(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)
```



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.

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

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

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

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



This shows the flexibility in dosing and sampling that the rxode2 event tables allow.

8.5 Combining event tables

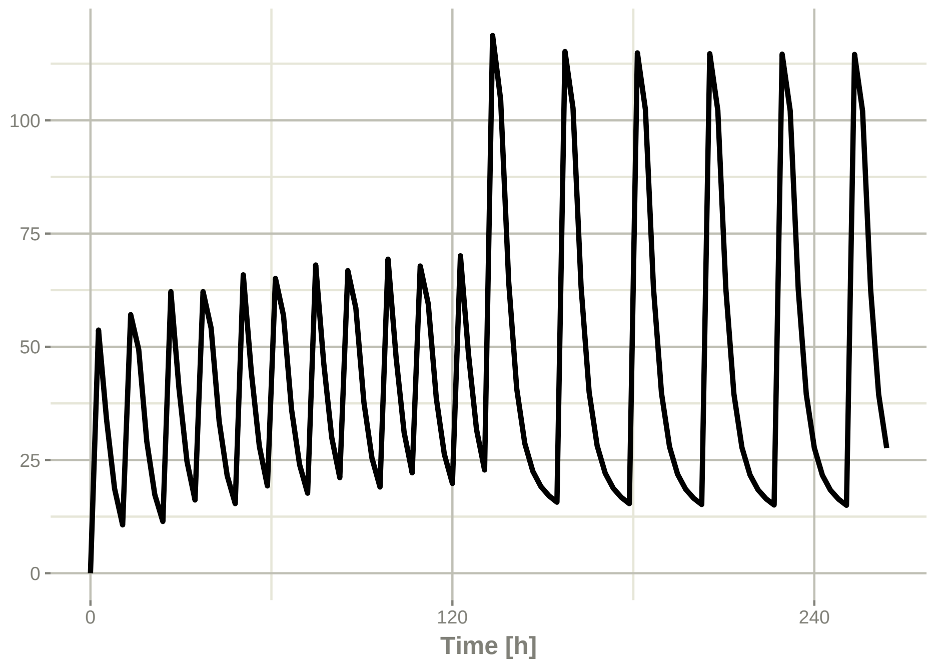
Since you can create dosing records and sampling records, you can create any complex dosing regimen you wish. In addition, rxode2 allows you to combine event tables by `c`, `seq`, `rep`, and `rbind`.

8.6 Sequencing event tables

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

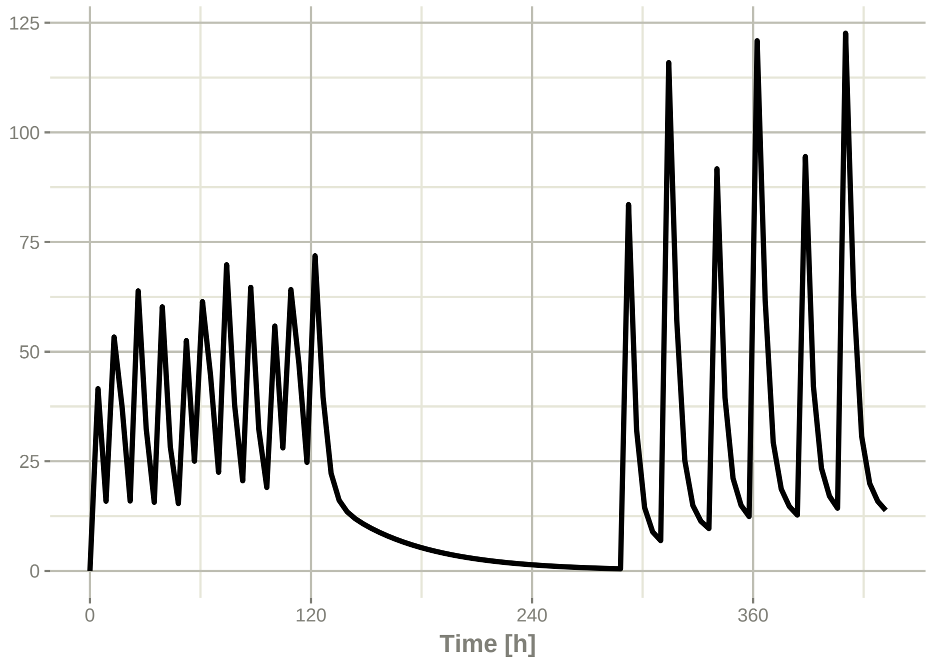
```r
## bid for 5 days
bid <- et(timeUnits="hr") %>%
       et(amt=10000,ii=12,until=set_units(5, "days"))

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

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

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



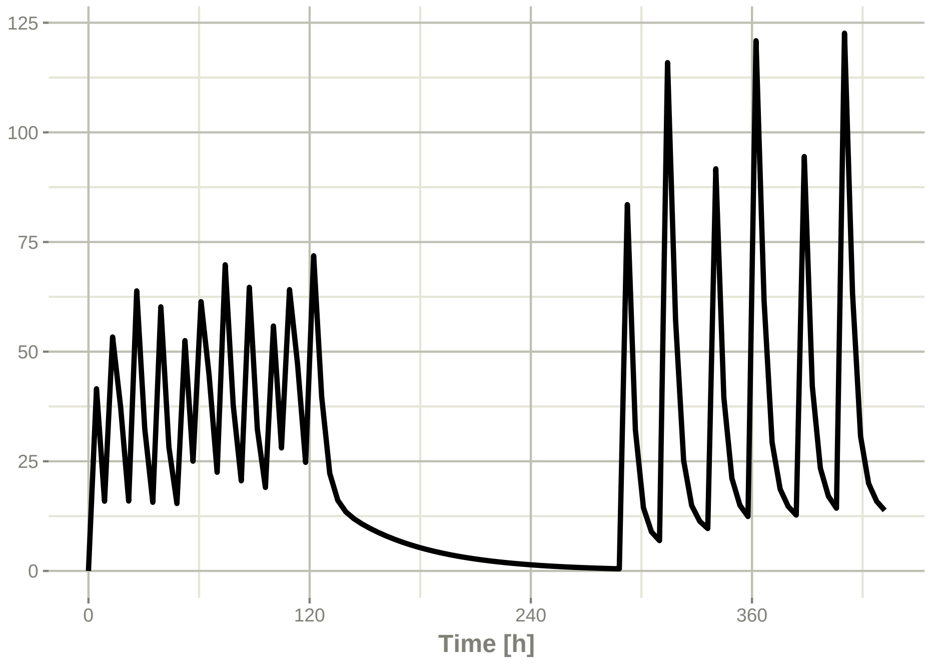
When sequencing events, you can also separate this sequence by a period of time; For example if you wanted to separate this by a week, you could easily do that with the following sequence of event tables:

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

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



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

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

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

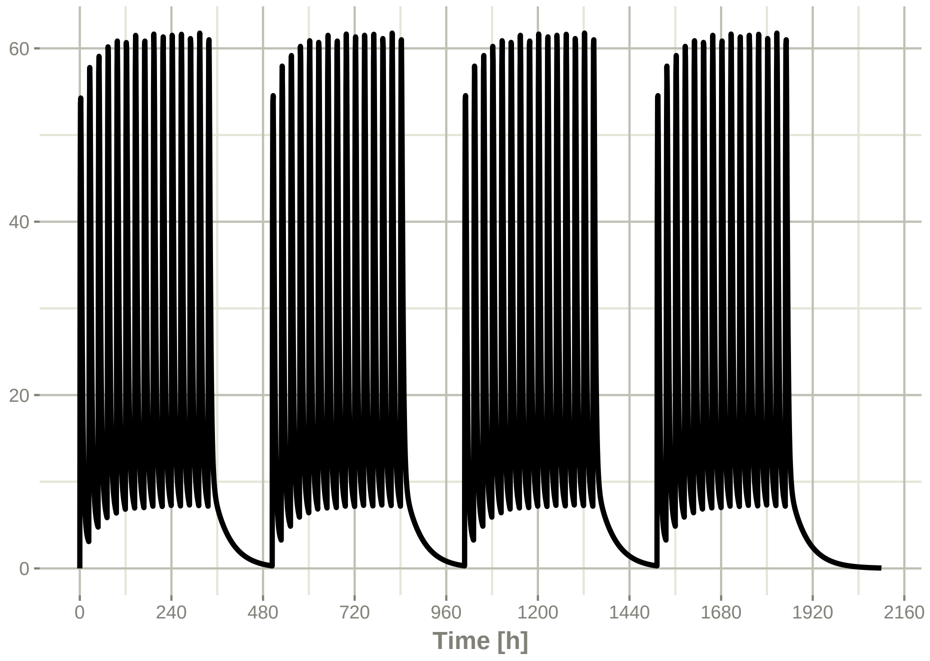


Also note, that rxode2 assumes that the dosing is what you want to space the event tables by, and clears out any sampling records when you combine the event tables. If that is not true, you can also use the option `samples="use"`

8.7 Repeating event tables

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

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

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

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



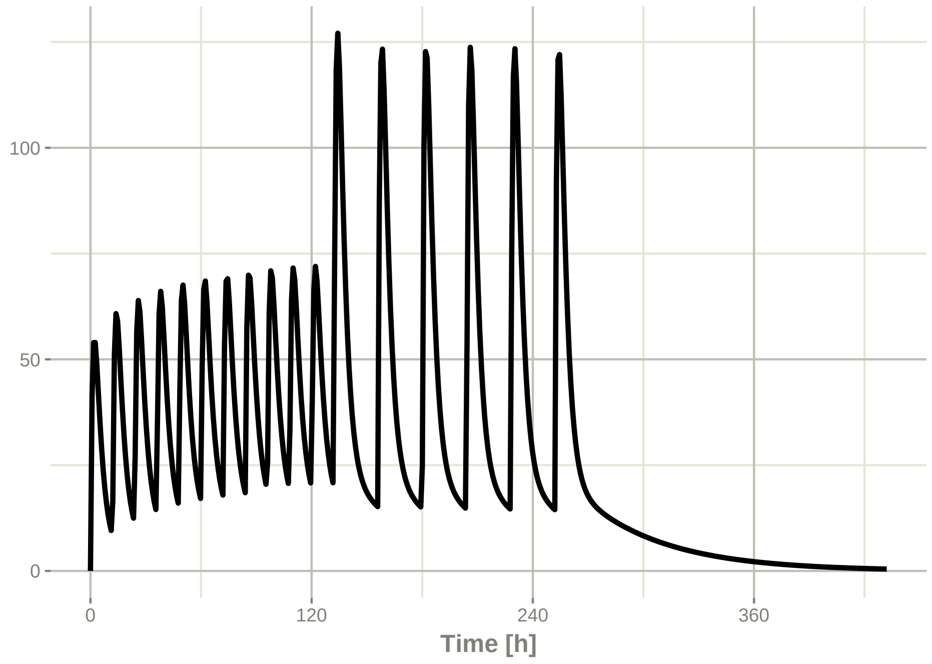
This is a simplified way to use a sequence of event tables. Therefore, many of the same options still apply; That is `samples` are cleared unless you use `samples="use"`, and the time between event tables is at least the inter-dose interval. You can adjust the timing by the `wait` option.

8.8 Combining event tables with rbind

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

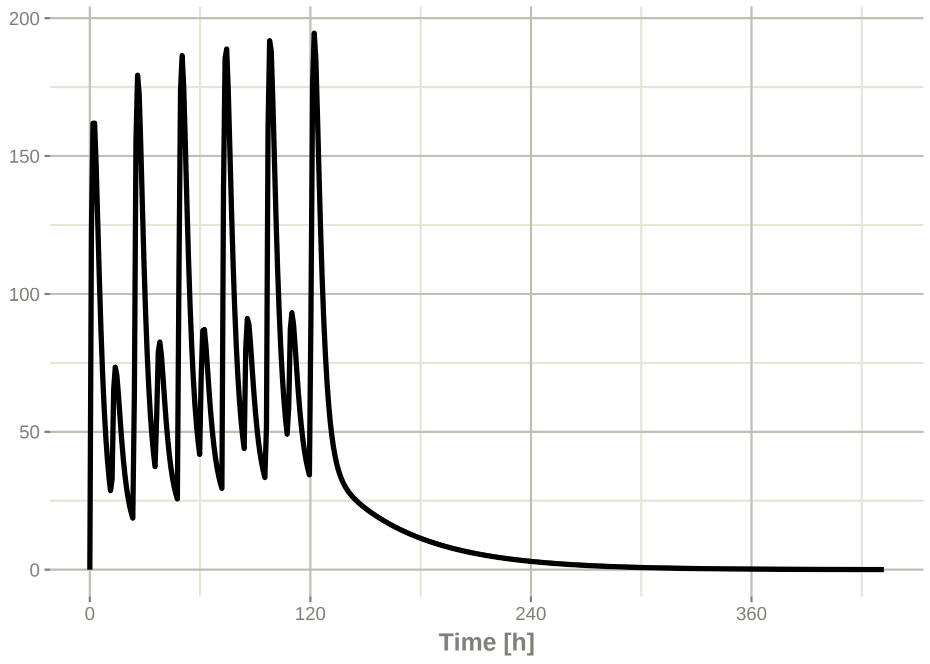
Using the previous `seq` you can clearly see the difference. Here was the sequence:

```
## 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)
```



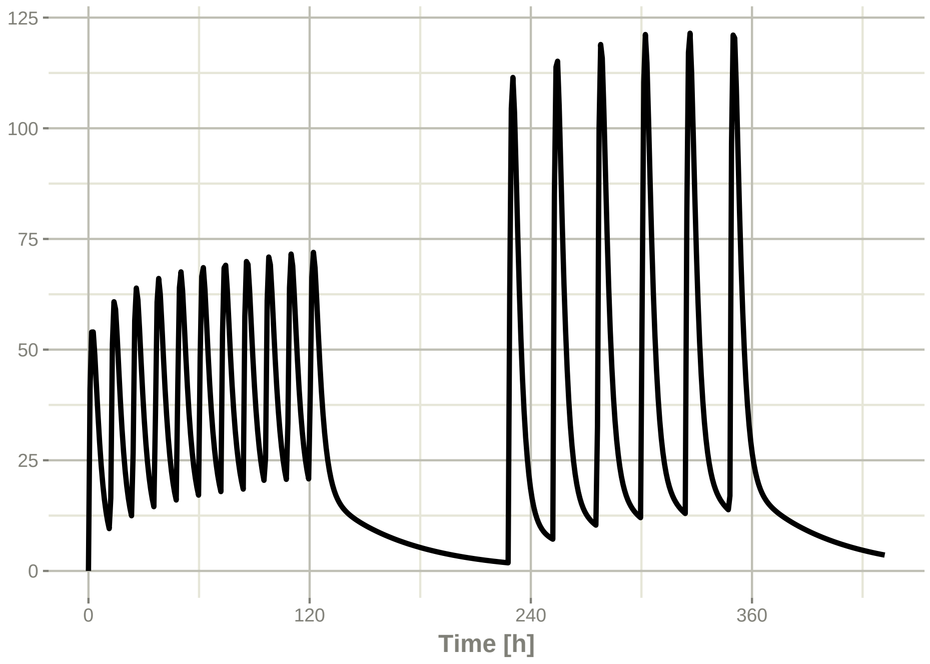
But if you bind them together with `rbind`

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

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



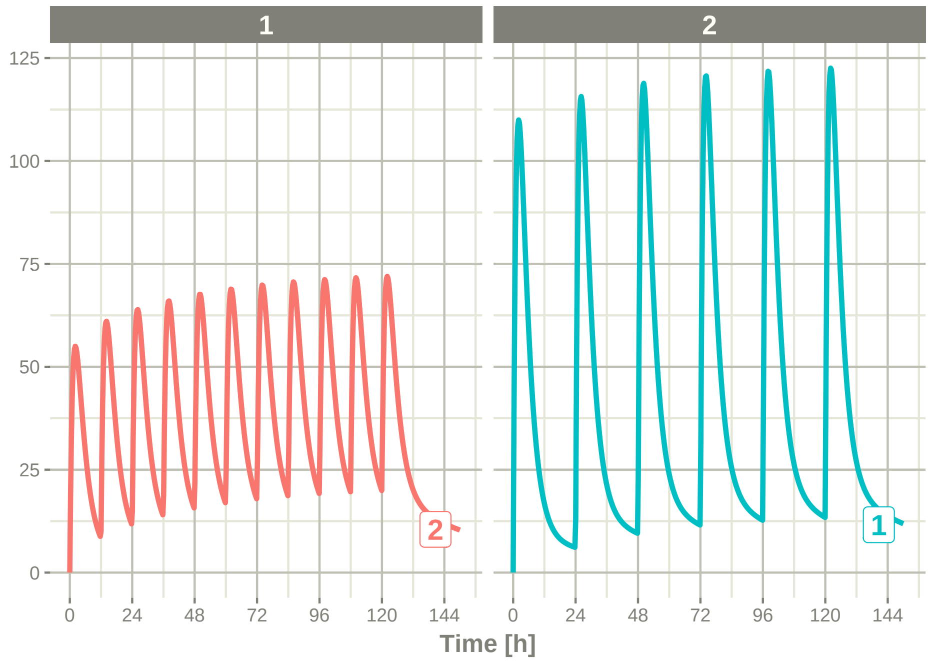
Still the waiting period applies (but does not consider the inter-dose interval)

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

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



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

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

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



8.9 Expanding events

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

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

ev
```

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

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

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

```
etExpand(ev)
```

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

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

```
print(ev)
```

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

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

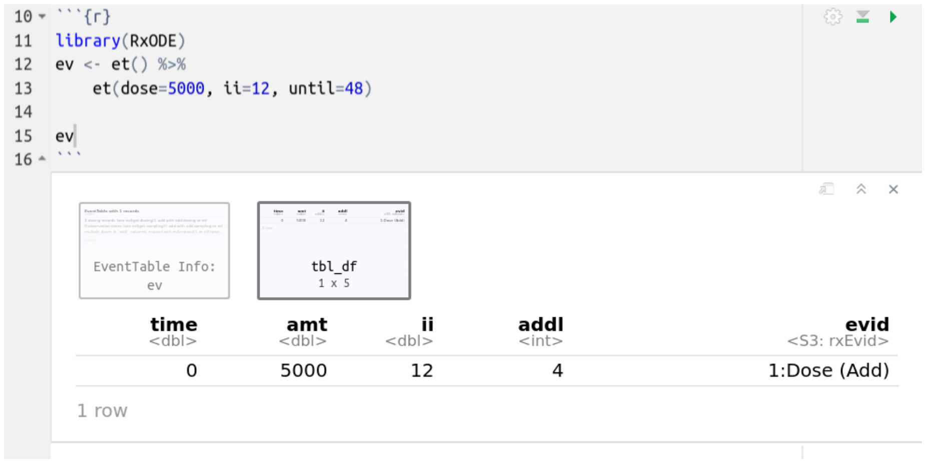
```
ev$expand()
ev
```

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

8.10 Event tables in Rstudio Notebooks

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

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

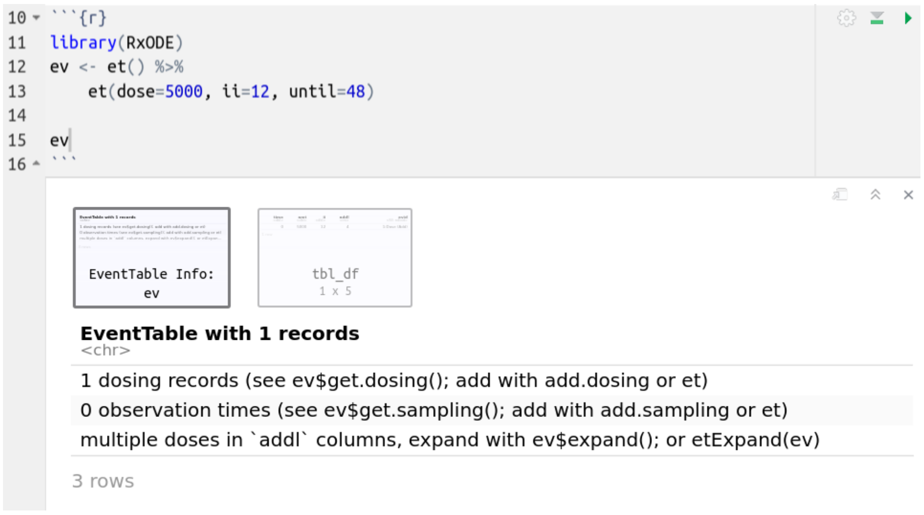


~~~
```{r}
library(RxODE)
ev <- et() %>%
    et(dose=5000, ii=12, until=48)

ev
```
~~~

time <dbl>	amt <dbl>	ii <dbl>	addl <int>	evid <S3: rxEvid>
0	5000	12	4	1:Dose (Add)

1 row

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



~~~r
```{r}
library(RxODE)
ev <- et() %>%
    et(dose=5000, ii=12, until=48)

ev
~~~
```

EventTable Info: ev
tbl_df
1 x 5

EventTable with 1 records
<chr>

```
1 dosing records (see ev$get.dosing(); add with add.dosing or et)
0 observation times (see ev$get.sampling(); add with add.sampling or et)
multiple doses in `addl` columns, expand with ev$expand(); or etExpand(ev)
```

3 rows
```

# Chapter 9

# Solving and solving options

In general, ODEs are solved using a combination of:

- A compiled model specification from `rxode2()`, specified with `object=`
- Input parameters, specified with `params=` (and could be blank)
- Input data or event table, specified with `events=`
- Initial conditions, specified by `inits=` (and possibly in the model itself by `state(0)=`)

The solving options are given in the sections below:

## 9.1 General Solving Options

### 9.1.1 object

`object` is a either a rxode2 family of objects, or a file-name with a rxode2 model specification, or a string with a rxode2 model specification.

### 9.1.2 params

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

### 9.1.3 events

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

### 9.1.4 inits

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

### 9.1.5 sigdig

`sigdig` Specifies the "significant digits" that the ode solving requests. When specified this controls the relative and absolute tolerances of the ODE solvers. By default the tolerance is `0.5*10^(-sigdig-2)` for regular ODEs. For the sensitivity equations the default is `0.5*10\^(-sigdig-1.5)` (sensitivity changes only applicable for liblsoda). This also ## lsoda/dop solving options

controls the `atol/rtol` of the steady state solutions. The `ssAtol/ssRtol` is `0.5*10\^(-sigdig)` and for the sensitivities `0.5*10\^(-sigdig+0.625).` By default this is unspecified (`NULL`) and uses the standard `atol/rtol`.

### 9.1.6 atol

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

### 9.1.7 rtol

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

### 9.1.8 atolSens

`atolSens` Sensitivity atol, can be different than atol with liblsoda. This allows a less accurate solve for gradients (if desired)

### 9.1.9 rtolSens

`rtolSens` Sensitivity rtol, can be different than rtol with liblsoda. This allows a less accurate solve for gradients (if desired)

### 9.1.10 maxsteps

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

### 9.1.11 hmin

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

### 9.1.12 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` rxode2 uses the maximum difference in times in your sampling and events. The value 0 is equivalent to infinite maximum absolute step size.

### 9.1.13 hmaxSd

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

### 9.1.14 hini

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

### 9.1.15 maxordn

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

### 9.1.16 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.

### 9.1.17 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).

### 9.1.18 hmxi

`hmxi` inverse of the maximum absolute value of `H` to are used. hmxi = 0.0 is allowed and corresponds to an infinite `hmax1 (default).`hminandhmxi`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"‘.

### 9.1.19 istateReset

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

## 9.2 Inductive Linerization Options

### 9.2.1 indLinMatExpType

`indLinMatExpType` This is them matrix exponential type that is use for rxode2. Currently the following are supported:

- `Al-Mohy` Uses the exponential matrix method of Al-Mohy Higham (2009)
- `arma` Use the exponential matrix from RcppArmadillo
- `expokit` Use the exponential matrix from Roger B. Sidje (1998)

### 9.2.2 indLinMatExpOrder

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

### 9.2.3 indLinPhiTol

`indLinPhiTol` the requested accuracy tolerance on exponential matrix.

### 9.2.4 indLinPhiM

`indLinPhiM` the maximum size for the Krylov basis

## 9.3 Steady State Solving Options

### 9.3.1 minSS

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

### 9.3.2 maxSS

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

### 9.3.3 strictSS

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

### 9.3.4 infSSstep

`infSSstep` Step size for determining if a constant infusion has reached steady state. By default this is large value, 12.

### 9.3.5 ssAtol

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

### 9.3.6 ssRtol

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

### 9.3.7 ssAtolSens

`ssAtolSens` Sensitivity absolute tolerance (atol) for calculating if steady state has been achieved for sensitivity compartments.

### 9.3.8 ssRtolSens

`ssRtolSens` Sensitivity relative tolerance (rtol) for calculating if steady state has been achieved for sensitivity compartments.

## 9.4 rxode2 numeric stability options

### 9.4.1 maxAtolRtolFactor

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

### 9.4.2 stateTrim

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

### 9.4.3 safeZero

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

### 9.4.4 sumType

`sumType` Sum type to use for `sum()` in rxode2 code blocks.

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

`fsum` uses the PreciseSum package's fsum function (most accurate)

`kahan` uses Kahan correction

`neumaier` uses Neumaier correction

`c` uses no correction: default/native summing

### 9.4.5 prodType

`prodType` Product to use for `prod()` in rxode2 blocks

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

`double` uses the standard double scale for multiplication.

### 9.4.6 maxwhile

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

## 9.5 Linear compartment model sensitivity options

### 9.5.1 sensType

`sensType` Sensitivity type for `linCmt()` model:

`advan` Use the direct advan solutions

`autodiff` Use the autodiff advan solutions

`forward` Use forward difference solutions

`central` Use central differences

### 9.5.2 linDiff

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

- `"lag"` Central compartment lag
- `"f"` Central compartment bioavailability
- `"rate"` Central compartment modeled rate
- `"dur"` Central compartment modeled duration
- `"lag2"` Depot compartment lag
- `"f2"` Depot compartment bioavailability
- `"rate2"` Depot compartment modeled rate
- `"dur2"` Depot compartment modeled duration

### 9.5.3 linDiffCentral

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

## 9.6 Covariate Solving Options

### 9.6.1 iCov

`iCov` A data frame of individual non-time varying covariates to combine with the `events` dataset by merge.

### 9.6.2 covsInterpolation

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

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

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

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

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

### 9.6.3 addCov

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

## 9.7 Simulation options

### 9.7.1 seed

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

### 9.7.2 nsim

`nsim` represents the number of simulations. For rxode2, if you supply single subject event tables (created with `[eventTable()]`)

### 9.7.3 thetaMat

`thetaMat` Named theta matrix.

### 9.7.4 thetaLower

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

### 9.7.5 thetaUpper

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

### 9.7.6 thetaDf

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

### 9.7.7 thetaIsChol

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

### 9.7.8 nStud

`nStud` Number virtual studies to characterize uncertainty in estimated parameters.

### 9.7.9 omega

`omega` Estimate of Covariance matrix. When omega is a list, assume it is a block matrix and convert it to a full matrix for simulations. When `omega` is `NA` and you are using it with a `rxode2` ui model, the between subject variability described by the `omega` matrix are set to zero.

### 9.7.10 omegaIsChol

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

### 9.7.11 omegaSeparation

`omegaSeparation` Omega separation strategy

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

- `"lkj"` simulates the correlation matrix from the `rLKJ1` matrix with the distribution parameter `eta` equal to the degrees of freedom `nu` by `(nu-1)/2`
- `"separation"` simulates from the identity inverse Wishart covariance matrix with `nu` degrees of freedom. This is then converted to a covariance matrix and augmented with the modeled standard deviations. While computationally more complex than the `"lkj"` prior, it performs better when the covariance matrix size is greater or equal to 10
- `"auto"` chooses `"lkj"` when the dimension of the matrix is less than 10 and `"separation"` when greater than equal to 10.

### 9.7.12 omegaXform

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

- `identity` This is when standard deviation values are directly modeled by the `params` and `thetaMat` matrix
- `variance` This is when the `params` and `thetaMat` simulates the variance that are directly modeled by the `thetaMat` matrix
- `log` This is when the `params` and `thetaMat` simulates `log(sd)`
- `nlmixrSqrt` This is when the `params` and `thetaMat` simulates the inverse cholesky decomposed matrix with the `x\^2` modeled along the diagonal. This only works with a diagonal matrix.

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

### 9.7.13 omegaLower

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

### 9.7.14 omegaUpper

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

### 9.7.15 omegaDf

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

### 9.7.16 nSub

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

### 9.7.17 dfSub

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

### 9.7.18 sigma

`sigma` Named sigma covariance or Cholesky decomposition of a covariance matrix. The names of the columns indicate parameters that are simulated. These are simulated for every observation in the solved system. When `sigma` is `NA` and you are using it with a `rxode2` ui model, the unexplained variability described by the `sigma` matrix are set to zero.

### 9.7.19 sigmaLower

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

### 9.7.20 sigmaUpper

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

### 9.7.21 sigmaXform

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

- `identity` This is when standard deviation values are directly modeled by the `params` and `thetaMat` matrix
- `variance` This is when the `params` and `thetaMat` simulates the variance that are directly modeled by the `thetaMat` matrix
- `log` This is when the `params` and `thetaMat` simulates `log(sd)`
- `nlmixrSqrt` This is when the `params` and `thetaMat` simulates the inverse cholesky decomposed matrix with the `x\^2` modeled along the diagonal. This only works with a diagonal matrix.
- `nlmixrLog` This is when the `params` and `thetaMat` simulates the inverse cholesky decomposed matrix with the `exp(x\^2)` along the diagonal. This only works with a diagonal matrix.
- `nlmixrIdentity` This is when the `params` and `thetaMat` simulates the inverse cholesky decomposed matrix. This only works with a diagonal matrix.

### 9.7.22 sigmaDf

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

### 9.7.23 sigmaIsChol

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

### 9.7.24 sigmaSeparation

`sigmaSeparation` separation strategy for sigma;

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

- `"lkj"` simulates the correlation matrix from the `rLKJ1` matrix with the distribution parameter `eta` equal to the degrees of freedom `nu` by `(nu-1)/2`
- `"separation"` simulates from the identity inverse Wishart covariance matrix with `nu` degrees of freedom. This is then converted to a covariance matrix and augmented with the modeled standard deviations. While computationally more complex than the `"lkj"` prior, it performs better when the covariance matrix size is greater or equal to 10

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

### 9.7.25 dfObs

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

### 9.7.26 resample

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

### 9.7.27 resampleID

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

## 9.8 rxode2 output options

### 9.8.1 returnType

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

- `"rxSolve"` (default) will return a reactive data frame that can change easily change different pieces of the solve and update the data frame. This is the currently standard solving method in rxode2, is used for `rxSolve(object, ...)`, `solve(object,...)`,
- `"data.frame"` – returns a plain, non-reactive data frame; Currently very slightly faster than `returnType="matrix"`
- `"matrix"` – returns a plain matrix with column names attached to the solved object. This is what is used `object$run` as well as `object$solve`
- `"data.table"` – returns a `data.table`; The `data.table` is created by reference (ie `setDt()`), which should be fast.
- `"tbl"` or `"tibble"` returns a tibble format.

### 9.8.2 addDosing

`addDosing` Boolean indicating if the solve should add rxode2 EVID and related columns. This will also include dosing information and estimates at the doses. Be default, rxode2 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 `rxode2` EVID events are returned. When `addDosing` is `TRUE` add the event information in NONMEM-style format; If `subsetNonmem=FALSE` rxode2 will also include extra event types (`EVID`) for ending infusion and modeled times:

- `EVID=-1` when the modeled rate infusions are turned off (matches `rate=-1`)
- `EVID=-2` When the modeled duration infusions are turned off (matches `rate=-2`)
- `EVID=-10` When the specified `rate` infusions are turned off (matches `rate>0`)
- `EVID=-20` When the specified `dur` infusions are turned off (matches `dur>0`)
- `EVID=101,102,103,...` Modeled time where 101 is the first model time, 102 is the second etc.

### 9.8.3 keep

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

### 9.8.4 drop

`drop` Columns to drop from the output

### 9.8.5 idFactor

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

### 9.8.6 subsetNonmem

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

### 9.8.7 scale

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

### 9.8.8 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 rxode2 event table.

### 9.8.9 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 rxode2 event table.

### 9.8.10 theta

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

### 9.8.11 eta

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

### 9.8.12 from

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

### 9.8.13 to

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

### 9.8.14 length.out

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

### 9.8.15 by

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

### 9.8.16 warnIdSort

`warnIdSort` Warn if the ID is not present and rxode2 assumes the order of the parameters/iCov are the same as the order of the parameters in the input dataset.

### 9.8.17 warnDrop

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

## 9.9 Internal rxode2 options

### 9.9.1 nDisplayProgress

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

### 9.9.2 simVariability

`simVariability` determines if the variability is simulated. When `NA` (default) this is determined by the solver.

### 9.9.3 ...

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

### 9.9.4 a

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

### 9.9.5 b

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

### 9.9.6 updateObject

`updateObject` This is an internally used flag to update the rxode2 solved object (when supplying an rxode2 solved object) as well as returning a new object. You probably should not modify it's `FALSE` default unless you are willing to have unexpected results.

## 9.10 Parallel/Threaded Solve

### 9.10.1 cores

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

### 9.10.2 nCoresRV

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

### 9.10.3 nLlikAlloc

`nLlikAlloc` The number of log likelihood endpoints that are used in the model. This allows independent log likelihood per endpoint in focei for nlmixr2. It likely shouldn't be set, though it won't hurt anything if you do (just may take up more memory for larger allocations).

### 9.10.4 useStdPow

`useStdPow` This uses C's `pow` for exponentiation instead of R's `R_pow` or `R_pow_di`. By default this is `FALSE`

### 9.10.5 naTimeHandle

`naTimeHandle` Determines what time of handling happens when the time becomes `NA`: current options are:

- `ignore` this ignores the `NA` time input and passes it through.
- `warn` (default) this will produce a warning at the end of the solve, but continues solving passing through the `NA` time
- `error` this will stop this solve if this is not a parallel solved ODE (otherwise stopping can crash R)

# Chapter 10

# rxode2 output

## 10.1 Using rxode2 data frames

### 10.1.1 Creating an interactive data frame

`rxode2` supports returning a solved object that is a modified data-frame. This is done by the `predict()`, `solve()`, or `rxSolve()` methods.

```
library(rxode2)
library(units)

### Setup example model
mod1 <- rxode2({
  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;
})
```

```
#> using C compiler: ‘gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0’
```

```
### Seup parameters and initial conditions

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

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

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


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

```
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#>      V2      V3      KA      CL       Q     Kin    Kout    EC50
#>  40.200 297.000   0.294  18.600  10.500   1.000   1.000 200.000
#> -- Initial Conditions ($inits): --
#> depot centr  peri   eff
#>     0     0     0     1
#> -- First part of data (object): --
#> # A tibble: 241 x 7
#>   time    C2    C3  depot centr  peri   eff
#>    [h] <dbl> <dbl>  <dbl> <dbl> <dbl> <dbl>
#> 1    0    0    0     10000     0     0    1
#> 2    1   44.4 0.920  7453. 1784.  273.  1.08
#> 3    2   54.9 2.67   5554. 2206.  794.  1.18
#> 4    3   51.9 4.46   4140. 2087. 1324.  1.23
#> 5    4   44.5 5.98   3085. 1789. 1776.  1.23
#> 6    5   36.5 7.18   2299. 1467. 2132.  1.21
#> # i 235 more rows
```

or

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

```
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#>      V2      V3      KA      CL       Q     Kin    Kout    EC50
#>  40.200 297.000   0.294  18.600  10.500   1.000   1.000 200.000
#> -- Initial Conditions ($inits): --
#> depot centr  peri   eff
#>     0     0     0     1
#> -- First part of data (object): --
#> # A tibble: 241 x 7
#>   time    C2    C3  depot centr  peri   eff
#>     [h] <dbl> <dbl>  <dbl> <dbl> <dbl> <dbl>
#> 1     0    0    0     10000     0     0    1
#> 2     1  44.4 0.920  7453. 1784.  273.  1.08
#> 3     2  54.9 2.67   5554. 2206.  794.  1.18
#> 4     3  51.9 4.46   4140. 2087. 1324.  1.23
#> 5     4  44.5 5.98   3085. 1789. 1776.  1.23
#> 6     5  36.5 7.18   2299. 1467. 2132.  1.21
#> # i 235 more rows
```

Or with mattigr

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

```
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#>      V2      V3      KA      CL       Q     Kin    Kout    EC50
#>  40.200 297.000   0.294  18.600  10.500   1.000   1.000 200.000
#> -- Initial Conditions ($inits): --
#> depot centr  peri   eff
#>     0     0     0     1
#> -- First part of data (object): --
#> # A tibble: 241 x 7
#>    time    C2    C3  depot centr  peri   eff
#>     [h] <dbl> <dbl>  <dbl> <dbl> <dbl> <dbl>
#> 1     0    0    0     10000     0     0    1
#> 2     1  44.4 0.920  7453. 1784.  273.  1.08
#> 3     2  54.9 2.67   5554. 2206.  794.  1.18
#> 4     3  51.9 4.46   4140. 2087. 1324.  1.23
#> 5     4  44.5 5.98   3085. 1789. 1776.  1.23
#> 6     5  36.5 7.18   2299. 1467. 2132.  1.21
#> # i 235 more rows
```

10.1.2 rxode2 solved object properties

10.1.3 Using the solved object as a simple data frame

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

```
library(dplyr)
```

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

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

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

```
#> Warning: `as.tbl()` was deprecated in dplyr 1.0.0.
#> i Please use `tibble::as_tibble()` instead.
#> Call `lifecycle::last_lifecycle_warnings()` to see where
#> this warning was generated.
```

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

```
#> Warning: `as.tbl()` was deprecated in dplyr 1.0.0.
#> i Please use `tibble::as_tibble()` instead.
#> Call `lifecycle::last_lifecycle_warnings()` to see where
#> this warning was generated.
```

```
x
```

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

10.2 Updating the data-set interactively

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

First we need to recreate the original solved system:

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

```
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#>      V2      V3      KA      CL       Q     Kin    Kout    EC50
#>  40.200 297.000   0.294  18.600  10.500   1.000   1.000 200.000
#> -- Initial Conditions ($inits): --
#> depot centr  peri   eff
#>     0     0     0     1
#> -- First part of data (object): --
#> # A tibble: 241 x 7
#>   time    C2    C3  depot centr  peri   eff
#>    [h] <dbl> <dbl>  <dbl> <dbl> <dbl> <dbl>
#> 1    0    0    0    10000     0     0   1
#> 2    1   44.4 0.920  7453. 1784.  273.  1.08
#> 3    2   54.9 2.67   5554. 2206.  794.  1.18
#> 4    3   51.9 4.46   4140. 2087. 1324.  1.23
#> 5    4   44.5 5.98   3085. 1789. 1776.  1.23
#> 6    5   36.5 7.18   2299. 1467. 2132.  1.21
#> # i 235 more rows
```

10.2.1 Modifying initial conditions

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

```
x$eff0
```

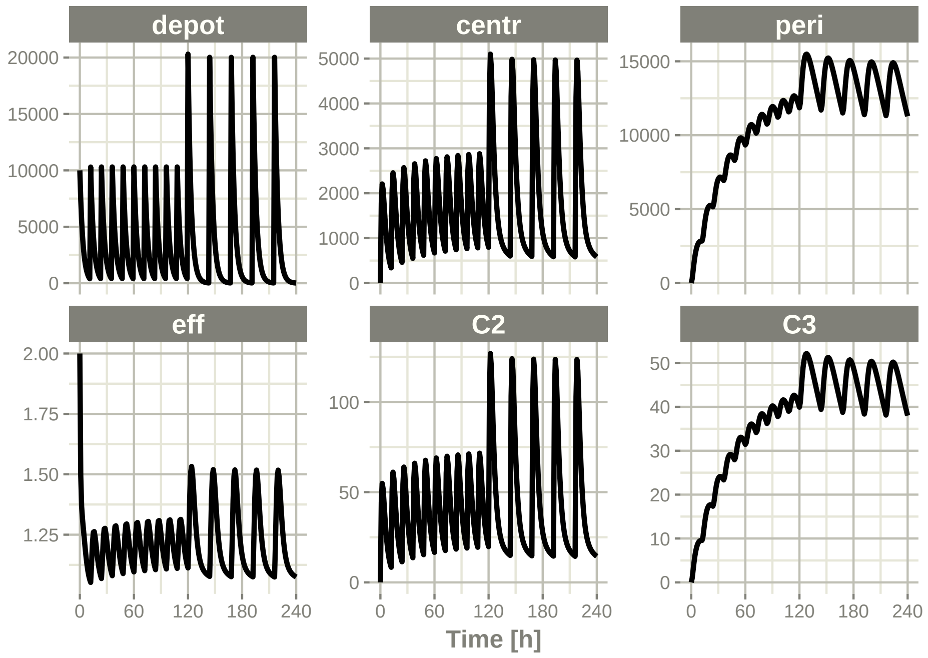
```
#> [1] 1
```

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

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

```
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#>      V2      V3      KA      CL       Q     Kin    Kout    EC50
#>  40.200 297.000   0.294  18.600  10.500   1.000   1.000 200.000
#> -- Initial Conditions ($inits): --
#> depot centr  peri   eff
#>     0     0     0     2
#> -- First part of data (object): --
#> # A tibble: 241 x 7
#>   time    C2    C3  depot centr  peri   eff
#>    [h] <dbl> <dbl>  <dbl> <dbl> <dbl> <dbl>
#> 1    0    0    0    10000     0     0   2
#> 2    1   44.4 0.920  7453. 1784.  273.  1.50
#> 3    2   54.9 2.67   5554. 2206.  794.  1.37
#> 4    3   51.9 4.46   4140. 2087. 1324.  1.31
#> 5    4   44.5 5.98   3085. 1789. 1776.  1.27
#> 6    5   36.5 7.18   2299. 1467. 2132.  1.23
#> # i 235 more rows
```

```
plot(x)
```



10.2.2 Modifying observation times for rxode2

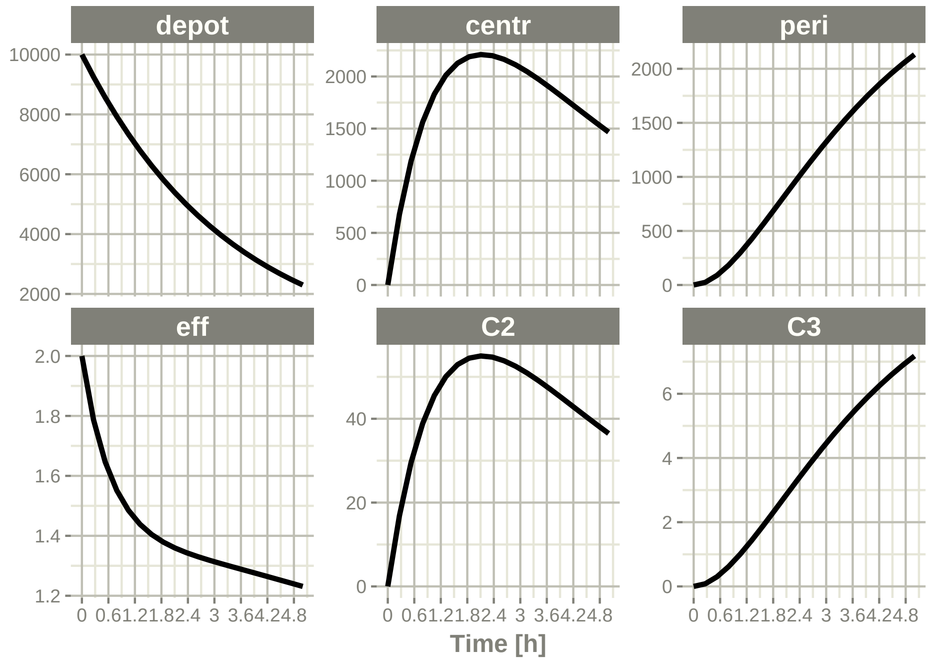
Notice that the initial effect is now 2.

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

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

```
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#>      V2      V3      KA      CL       Q     Kin    Kout    EC50
#>  40.200 297.000   0.294  18.600  10.500   1.000   1.000 200.000
#> -- Initial Conditions ($inits): --
#> depot centr  peri   eff
#>     0     0     0     2
#> -- First part of data (object): --
#> # A tibble: 20 x 7
#>    time    C2     C3  depot centr  peri   eff
#>     [h] <dbl>  <dbl>  <dbl> <dbl> <dbl> <dbl>
#> 1 0         0    0    10000     0     0    2
#> 2 0.263  16.8 0.0817  9255.  677.  24.3  1.79
#> 3 0.526  29.5 0.299   8566. 1187.  88.7  1.65
#> 4 0.789  38.9 0.615   7929. 1562. 183.   1.55
#> 5 1.05   45.5 1.00    7338. 1830. 298.   1.49
#> 6 1.32   50.1 1.44    6792. 2013. 427.   1.44
#> # i 14 more rows
```

```
plot(x)
```



10.2.3 Modifying simulation parameters

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

```
x$KA
```

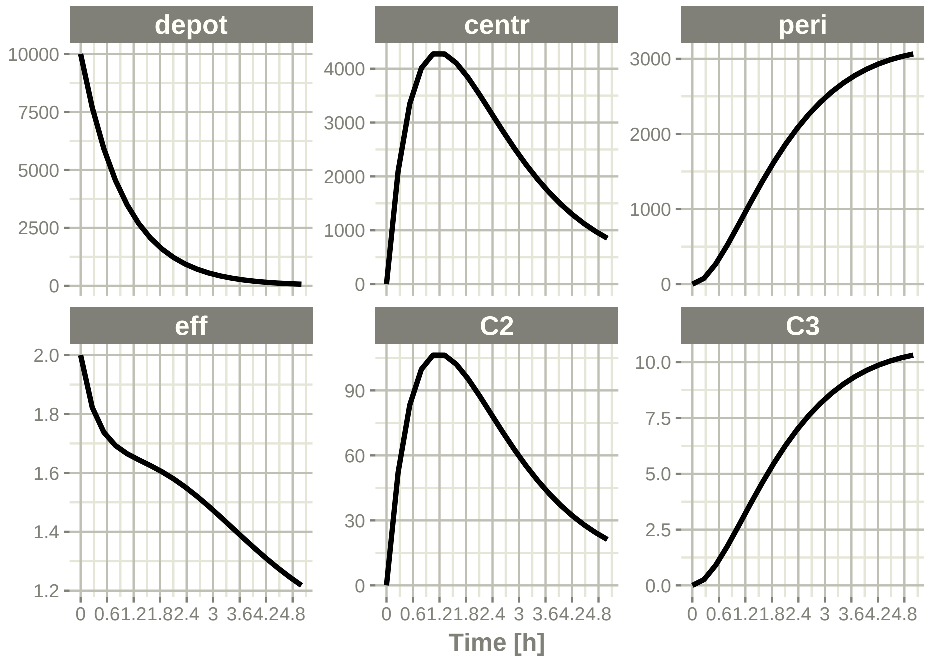
```
#> [1] 0.294
```

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

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

```
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#>    V2    V3    KA    CL     Q   Kin  Kout  EC50
#>  40.2 297.0   1.0  18.6  10.5   1.0   1.0 200.0
#> -- Initial Conditions ($inits): --
#> depot centr  peri   eff
#>     0     0     0     2
#> -- First part of data (object): --
#> # A tibble: 20 x 7
#>     time    C2    C3  depot centr   peri   eff
#>      [h] <dbl> <dbl>  <dbl> <dbl>  <dbl> <dbl>
#> 1 0         0   0     10000     0      0    2
#> 2 0.263  52.2 0.261  7686. 2098.   77.6  1.82
#> 3 0.526  83.3 0.900  5908. 3348.  267.   1.74
#> 4 0.789  99.8 1.75   4541. 4010.  519.   1.69
#> 5 1.05  106.  2.69   3490. 4273.  800.   1.67
#> 6 1.32  106.  3.66   2683. 4272. 1086.   1.64
#> # i 14 more rows
```

```
plot(x)
```



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

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

Chapter 11

Simulation

11.1 Single Subject solving

Originally, rxode2 was only created to solve ODEs for one individual. That is a single system without any changes in individual parameters.

Of course this is still supported, the classic examples are found in rxode2 intro.

This article discusses the differences between multiple subject and single subject solving. There are three differences:

- Single solving does not solve each ID in parallel
- Single solving lacks the `id` column in parameters(`$params`) as well as in the actual dataset.
- Single solving allows parameter exploration easier because each parameter can be modified. With multiple subject solves, you have to make sure to update each individual parameter.

The first obvious difference is in speed; With multiple subjects you can run each subject ID in parallel. For more information and examples of the speed gains with multiple subject solving see the Speeding up rxode2 vignette.

The next difference is the amount of information output in the final data.

Taking the 2 compartment indirect response model originally in the tutorial:

```r
library(rxode2)
mod1 <-rxode2({
    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
    C2 = centr/V2
    C3 = peri/V3
    d/dt(depot) =-KA*depot
    d/dt(centr) = KA*depot - CL*C2 - Q*C2 + Q*C3
    d/dt(peri)  =                    Q*C2 - Q*C3
    d/dt(eff)  = Kin - Kout*(1-C2/(EC50+C2))*eff
    eff(0) = 1
})
```

```
#> using C compiler: ‘gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0’
```

```
et <- et(amount.units='mg', time.units='hours') %>%
    et(dose=10000, addl=9, ii=12) %>%
    et(amt=20000, nbr.doses=5, start.time=120, dosing.interval=24) %>%
    et(0:240) # sampling
```

Now a simple solve

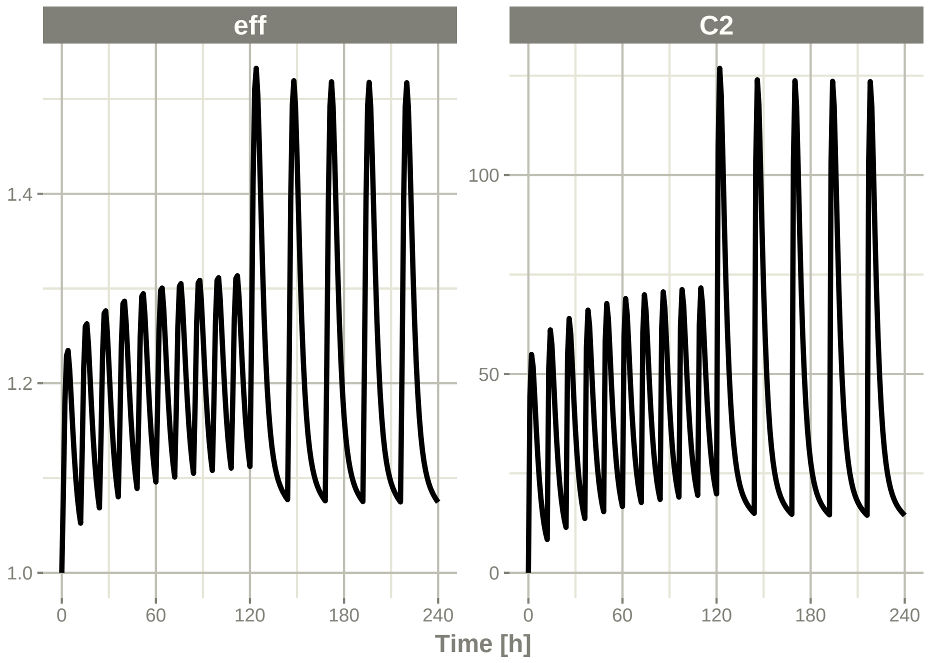
```
x <- rxSolve(mod1, et)
x
```

```
#> -- Solved rxode2 object --
#> -- Parameters (x$params): --
#>      KA      CL      V2       Q      V3     Kin    Kout    EC50
#>   0.294  18.600  40.200  10.500 297.000   1.000   1.000 200.000
#> -- Initial Conditions (x$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
#> # i 235 more rows
```

```
print(x)
```

```
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#>       KA      CL      V2       Q      V3     Kin    Kout    EC50
#>    0.294  18.600  40.200  10.500 297.000   1.000   1.000 200.000
#> -- Initial Conditions ($inits): --
#> 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
#> # i 235 more rows
```

```r
plot(x, C2, eff)
```

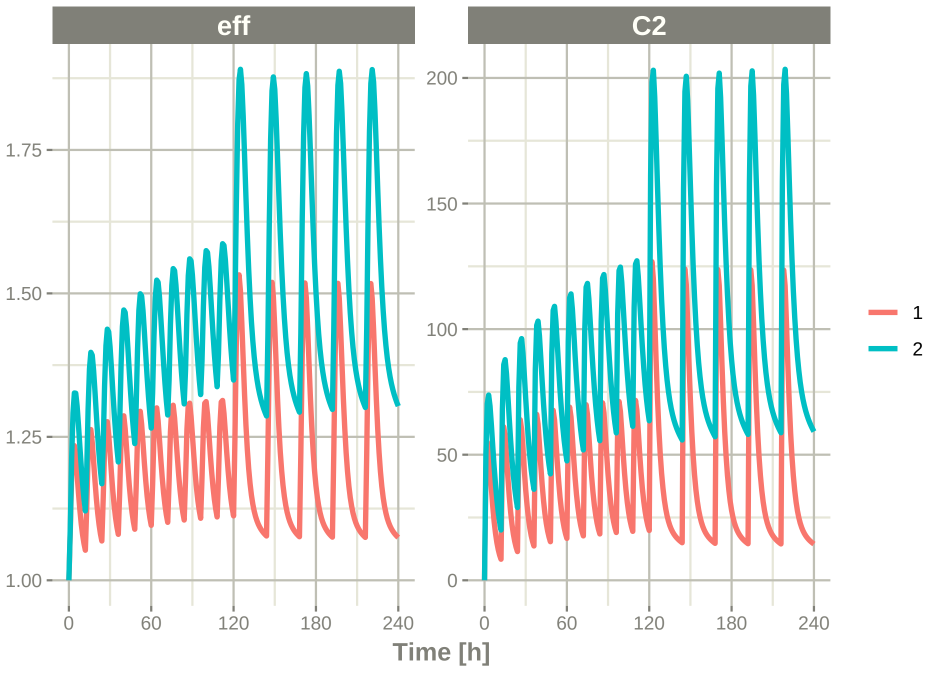


To better see the differences between the single solve, you can solve for 2 individuals

```r
x2 <- rxSolve(mod1, et %>% et(id=1:2), params=data.frame(CL=c(18.6, 7.6)))
print(x2)
```

```
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#> # A tibble: 2 x 9
#>   id        KA    CL    V2     Q    V3   Kin  Kout  EC50
#>   <fct> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 1     0.294  18.6  40.2  10.5   297     1     1   200
#> 2 2     0.294   7.6  40.2  10.5   297     1     1   200
#> -- Initial Conditions ($inits): --
#> depot centr  peri   eff
#>     0     0     0     1
#> -- First part of data (object): --
#> # A tibble: 482 x 8
#>      id time    C2    C3  depot centr  peri   eff
#>   <int>  [h] <dbl> <dbl>  <dbl> <dbl> <dbl> <dbl>
#> 1     1    0   0   0      10000     0     0   1
#> 2     1    1  44.4 0.920  7453. 1784.  273.  1.08
#> 3     1    2  54.9 2.67   5554. 2206.  794.  1.18
#> 4     1    3  51.9 4.46   4140. 2087. 1324.  1.23
#> 5     1    4  44.5 5.98   3085. 1789. 1776.  1.23
#> 6     1    5  36.5 7.18   2299. 1467. 2132.  1.21
#> # i 476 more rows
```

```
plot(x2, C2, eff)
```



By observing the two solves, you can see:

- A multiple subject solve contains the `id` column both in the data frame and then data frame of parameters for each subject.

The last feature that is not as obvious, modifying the individual parameters. For single subject data, you can modify the rxode2 data frame changing initial conditions and parameter values as if they were part of the data frame, as described in the rxode2 Data Frames.

For multiple subject solving, this feature still works, but requires care when supplying each individual's parameter value, otherwise you may change the solve and drop parameter for key individuals.

11.1.1 Summary of Single solve vs Multiple subject solving

Feature	Single Subject Solve	Multiple Subject Solve
Parallel	None	Each Subject
$params	data.frame with one parameter value	data.frame with one parameter per subject (w/ID column)
solved data	Can modify individual parameters with $ syntax	Have to modify all the parameters to update solved object

11.2 Population Simulations with rxode2

11.2.1 Simulation of Variability with rxode2

In pharmacometrics the nonlinear-mixed effect modeling software (like nlmixr) characterizes the between-subject variability. With this between subject variability you can simulate new subjects.

Assuming that you have a 2-compartment, indirect response model, you can set create an rxode2 model describing this system below:

11.2.1.1 Setting up the rxode2 model

```
library(rxode2)

set.seed(32)
rxSetSeed(32)

mod <- rxode2({
  eff(0) = 1
  C2 = centr/V2*(1+prop.err);
  C3 = peri/V3;
  CL =  TCl*exp(eta.Cl) ## This is coded as a variable in the model
  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;
})
```

```
#> using C compiler: 'gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0'
```

11.2.1.2 Adding the parameter estimates

The next step is to get the parameters into R so that you can start the simulation:

```
theta <- c(KA=2.94E-01, TCl=1.86E+01, V2=4.02E+01,  # central
           Q=1.05E+01, V3=2.97E+02,                # peripheral
           Kin=1, Kout=1, EC50=200, prop.err=0)      # effects
```

In this case, I use `lotri` to specify the omega since it uses similar lower-triangular matrix specification as nlmixr (also similar to NONMEM):

```
### the column names of the omega matrix need to match the parameters specified by rxo
omega <- lotri(eta.Cl ~ 0.4^2)
omega
```

```
#>        eta.Cl
#> eta.Cl   0.16
```

11.2.1.3 Simulating

The next step to simulate is to create the dosing regimen for overall simulation:

```
ev <- et(amount.units="mg", time.units="hours") %>%
  et(amt=10000, cmt="centr")
```

If you wish, you can also add sampling times (though now rxode2 can fill these in for you):

```
ev <- ev %>% et(0,48, length.out=100)
```

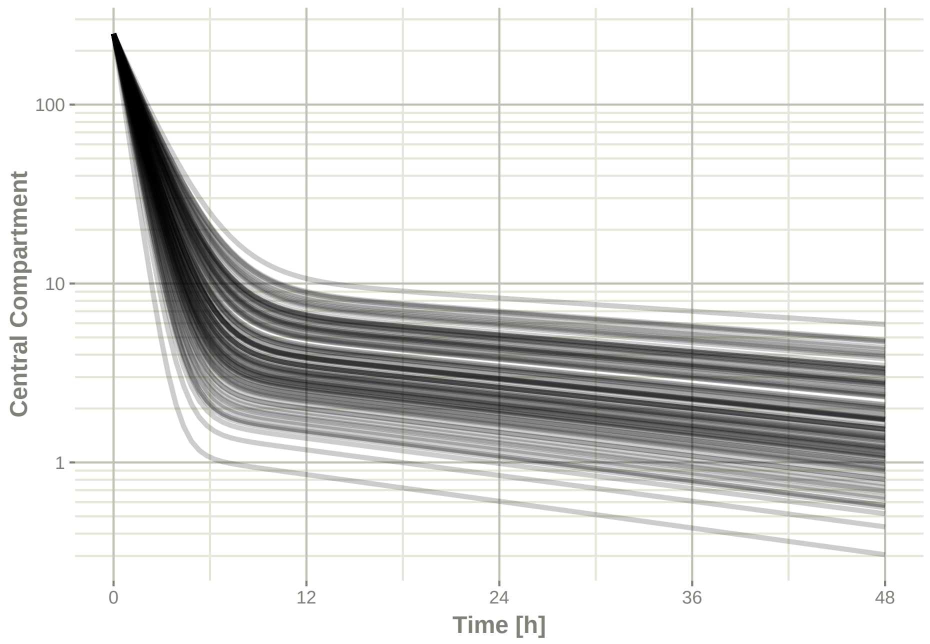
Note the `et` takes similar arguments as `seq` when adding sampling times. There are more methods to adding sampling times and events to make complex dosing regimens (See the event vignette). This includes ways to add variability to the both the sampling and dosing times).

Once this is complete you can simulate using the `rxSolve` routine:

```
sim  <- rxSolve(mod,theta,ev,omega=omega,nSub=100)
```

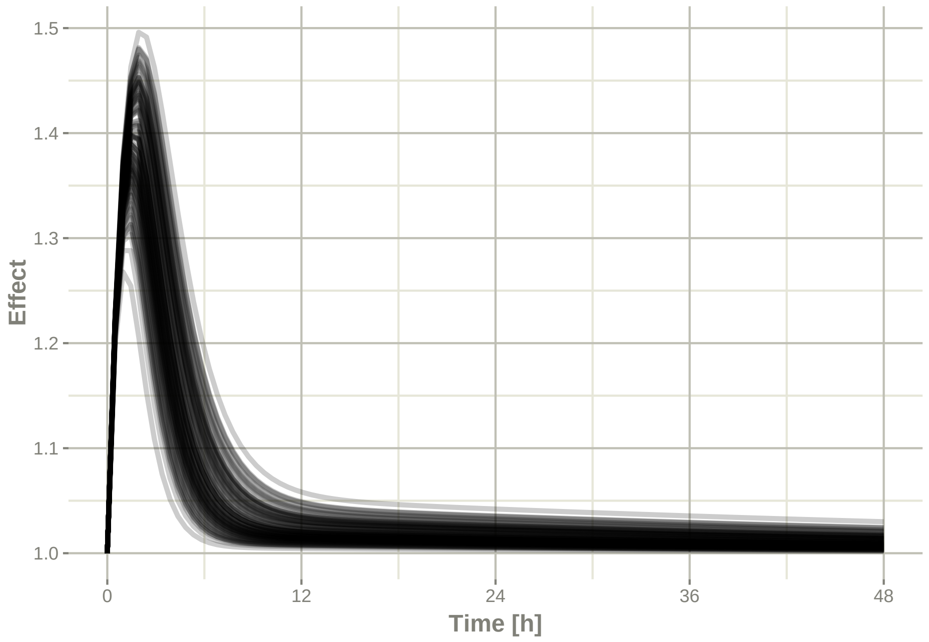
To quickly look and customize your simulation you use the default `plot` routine. Since this is an rxode2 object, it will create a `ggplot2` object that you can modify as you wish. The extra parameter to the `plot` tells `rxode2/R` what piece of information you are interested in plotting. In this case, we are interested in looking at the derived parameter `C2`:

```
library(ggplot2)
### The plots from rxode2 are ggplots so they can be modified with
### standard ggplot commands.
plot(sim, C2, log="y") +
    ylab("Central Compartment")
```



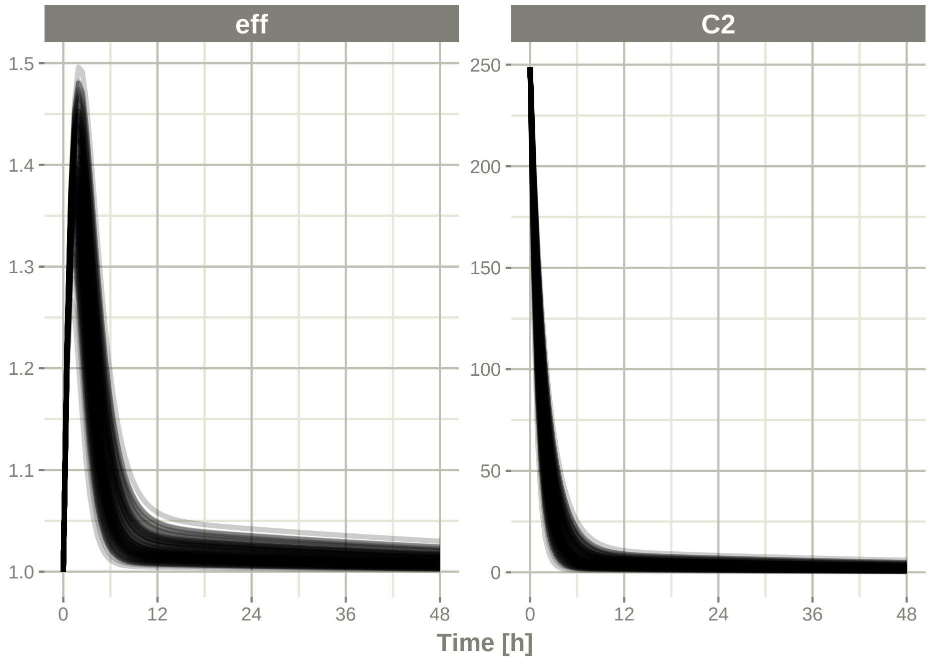
Of course this additional parameter could also be a state value, like `eff`:

```
### They also takes many of the standard plot arguments; See ?plot
plot(sim, eff, ylab="Effect")
```



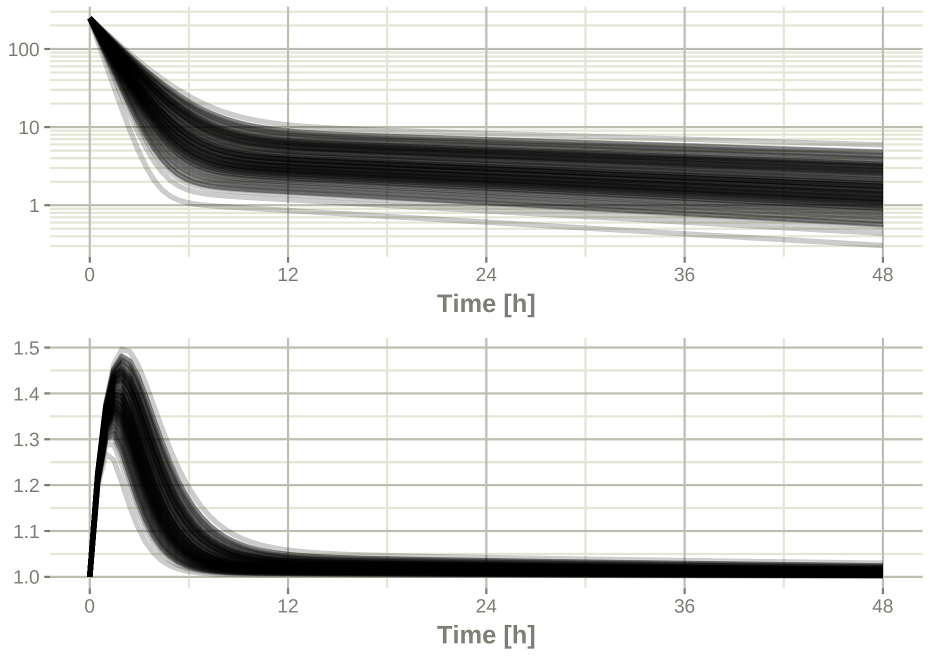
Or you could even look at the two side-by-side:

```
plot(sim, C2, eff)
```



Or stack them with `patchwork`

```r
library(patchwork)
plot(sim, C2, log="y") / plot(sim, eff)
```



11.2.1.5 Processing the data to create summary plots

Usually in pharmacometric simulations it is not enough to simply simulate the system. We have to do something easier to digest, like look at the central and extreme tendencies of the simulation.

Since the `rxode2` solve object is a type of data frame

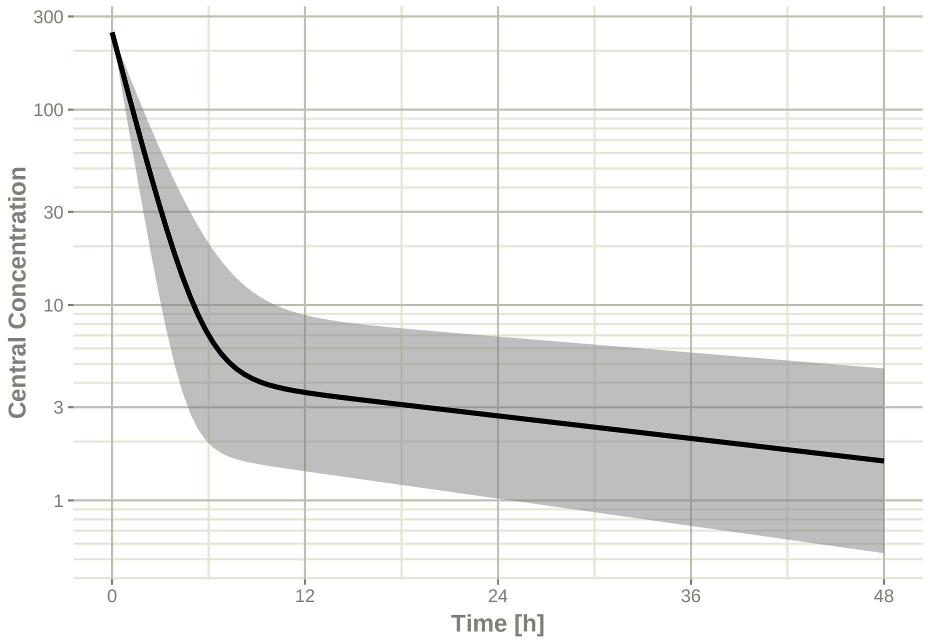
It is now straightforward to perform calculations and generate plots with the simulated data. You can

Below, the 5th, 50th, and 95th percentiles of the simulated data are plotted.

```r
confint(sim, "C2", level=0.95) %>%
    plot(ylab="Central Concentration", log="y")
```

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

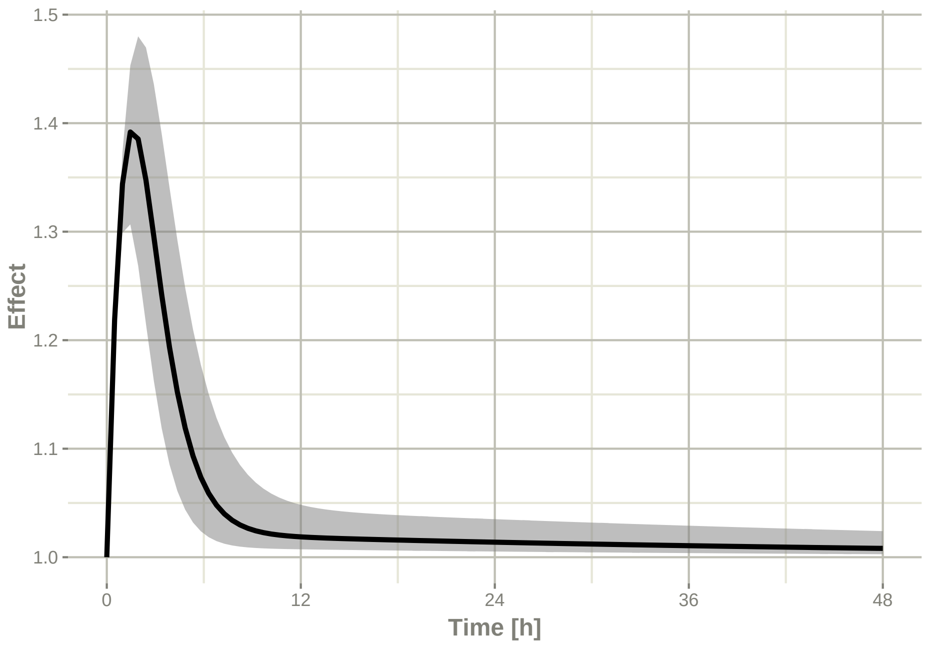
```
#> summarizing data...done
```



```r
confint(sim, "eff", level=0.95) %>%
    plot(ylab="Effect")
```

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

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



Note that you can see the parameters that were simulated for the example

```
head(sim$param)
```

```
#>   sim.id   V2 prop.err  V3  TCl      eta.Cl    KA    Q Kin Kout EC50
#> 1      1 40.2        0 297 18.6 -0.61812276 0.294 10.5   1    1  200
#> 2      2 40.2        0 297 18.6  0.08180225 0.294 10.5   1    1  200
#> 3      3 40.2        0 297 18.6  0.90546786 0.294 10.5   1    1  200
#> 4      4 40.2        0 297 18.6  0.47668266 0.294 10.5   1    1  200
#> 5      5 40.2        0 297 18.6  0.15939498 0.294 10.5   1    1  200
#> 6      6 40.2        0 297 18.6 -0.14083691 0.294 10.5   1    1  200
```

11.2.1.6 Simulation of unexplained variability (sigma)

In addition to conveniently simulating between subject variability, you can also easily simulate unexplained variability.

```
mod <- rxode2({
  eff(0) = 1
  C2 = centr/V2;
  C3 = peri/V3;
  CL =  TCl*exp(eta.Cl) ## This is coded as a variable in the model
  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;
  e = eff+eff.err
  cp = centr*(1+cp.err)
})
```

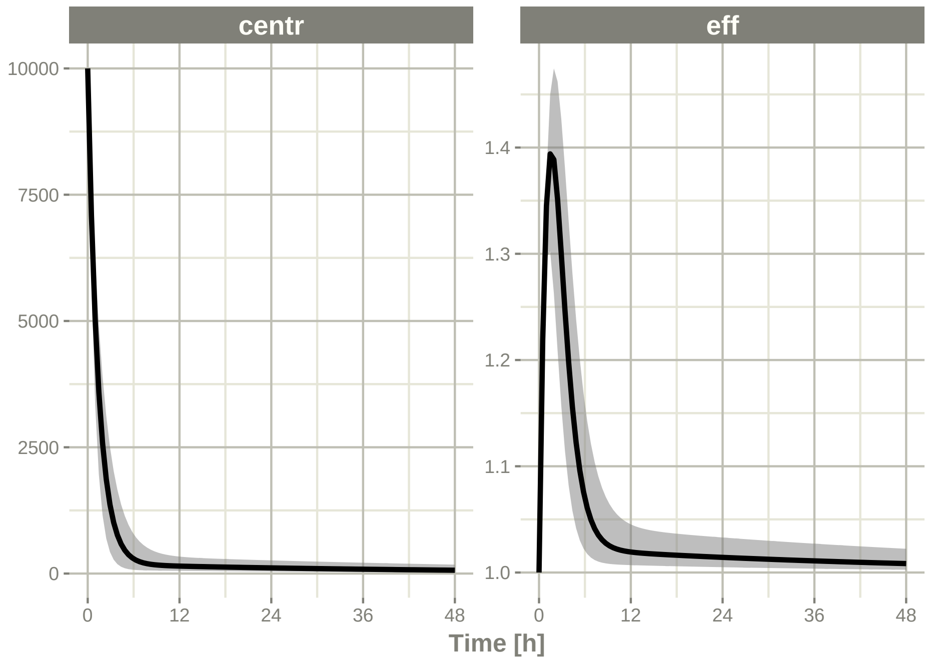
```
#> using C compiler: ‘gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0’
```

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

sigma <- lotri(eff.err ~ 0.1, cp.err ~ 0.1)


sim  <- rxSolve(mod, theta, ev, omega=omega, nSub=100, sigma=sigma)
s <- confint(sim, c("eff", "centr"));
```

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

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

```
plot(s)
```



11.2.1.7 Simulation of Individuals

Sometimes you may want to match the dosing and observations of individuals in a clinical trial. To do this you will have to create a data.frame using the `rxode2` event specification as well as an ID column to indicate an individual. The rxode2 event vignette talks more about how these datasets should be created.

```
library(dplyr)
ev1 <- eventTable(amount.units="mg", time.units="hours") %>%
    add.dosing(dose=10000, nbr.doses=1, dosing.to=2) %>%
    add.sampling(seq(0,48,length.out=10));

ev2 <- eventTable(amount.units="mg", time.units="hours") %>%
    add.dosing(dose=5000, nbr.doses=1, dosing.to=2) %>%
    add.sampling(seq(0,48,length.out=8));

dat <- rbind(data.frame(ID=1, ev1$get.EventTable()),
             data.frame(ID=2, ev2$get.EventTable()))


### Note the number of subject is not needed since it is determined by the data
sim  <- rxSolve(mod, theta, dat, omega=omega, sigma=sigma)

sim %>% select(id, time, e, cp)
```

```
#>    id          time         e          cp
#> 1   1  0.000000 [h] 0.7116275 11416.75636
#> 2   1  5.333333 [h] 1.1370890    44.85787
#> 3   1 10.666667 [h] 1.2158968    54.69894
#> 4   1 16.000000 [h] 0.7610539    52.82444
#> 5   1 21.333333 [h] 1.3294123    44.62836
#> 6   1 26.666667 [h] 0.9519536    15.41359
#> 7   1 32.000000 [h] 1.3996882    28.47164
#> 8   1 37.333333 [h] 1.2179631    34.45621
#> 9   1 42.666667 [h] 0.9950682    21.10386
#> 10  1 48.000000 [h] 0.7574426    19.63199
#> 11  2  0.000000 [h] 1.0447510  5557.85159
#> 12  2  6.857143 [h] 1.1685770    54.86144
#> 13  2 13.714286 [h] 0.9943757    82.42555
#> 14  2 20.571429 [h] 1.2658501    63.32900
#> 15  2 27.428571 [h] 0.5209770    48.77855
#> 16  2 34.285714 [h] 1.2732043    48.31506
#> 17  2 41.142857 [h] 0.7184654    49.78875
#> 18  2 48.000000 [h] 0.5720583    42.85381
```

11.3 Simulation of Clinical Trials

By either using a simple single event table, or data from a clinical trial as described above, a complete clinical trial simulation can be performed.

Typically in clinical trial simulations you want to account for the uncertainty in the fixed parameter estimates, and even the uncertainty in both your between subject variability as well as the unexplained variability.

rxode2 allows you to account for these uncertainties by simulating multiple virtual “studies,” specified by the parameter nStud. Each of these studies samples a realization of fixed effect parameters and covariance matrices for the between subject variability(omega) and unexplained variabilities (sigma). Depending on the information you have from the models, there are a few strategies for simulating a realization of the omega and sigma matrices.

The first strategy occurs when either there is not any standard errors for standard deviations (or related parameters), or there is a modeled correlation in the model you are simulating from. In that case the suggested strategy is to use the inverse Wishart (parameterized to scale to the conjugate prior)/scaled inverse chi distribution. this approach uses a single parameter to inform the variability of the covariance matrix sampled (the degrees of freedom).

The second strategy occurs if you have standard errors on the variance/standard deviation with no modeled correlations in the covariance matrix. In this approach you perform separate simulations for the standard deviations and the correlation matrix. First you simulate the variance/standard deviation components in the thetaMat multivariate normal simulation. After simulation and transformation to standard deviations, a correlation matrix is simulated using the degrees of freedom of your covariance matrix. Combining the simulated standard deviation with the simulated correlation matrix will give a simulated covariance matrix. For smaller dimension covariance matrices (dimension < 10x10) it is recommended you use the lkj distribution to simulate the correlation matrix. For higher dimension covariance matrices it is suggested you use the inverse wishart distribution (transformed to a correlation matrix) for the simulations.

The covariance/variance prior is simulated from rxode2s cvPost() function.

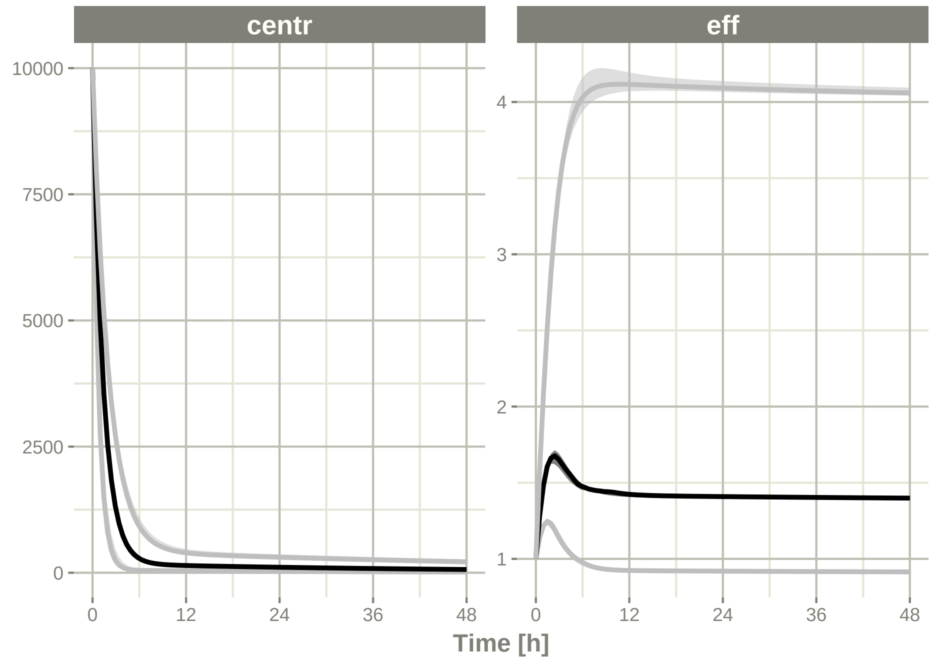
11.3.1 Simulation from inverse Wishart correlations

An example of this simulation is below:

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

sim  <- rxSolve(mod, theta, ev, omega=omega, nSub=100, sigma=sigma, thetaMat=tMat, nSt
                dfSub=10, dfObs=100)

s <-sim %>% confint(c("centr", "eff"))
```

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

```r
plot(s)
```



If you wish you can see what `omega` and `sigma` was used for each virtual study by accessing them in the solved data object with `$omega.list` and `$sigma.list`:

```r
head(sim$omega.list)
```

```
#> [[1]]
#>            eta.Cl
#> eta.Cl 0.1676778
#>
#> [[2]]
#>            eta.Cl
#> eta.Cl 0.2917085
#>
#> [[3]]
#>            eta.Cl
#> eta.Cl 0.1776813
#>
#> [[4]]
#>            eta.Cl
#> eta.Cl 0.1578682
#>
#> [[5]]
#>            eta.Cl
#> eta.Cl 0.1845614
#> 
#> [[6]]
#>           eta.Cl
#> eta.Cl 0.3282268
```

```
head(sim$sigma.list)
```

```
#> [[1]]
#>             eff.err      cp.err
#> eff.err 0.112416983 0.004197039
#> cp.err  0.004197039 0.097293971
#> 
#> [[2]]
#>              eff.err       cp.err
#> eff.err  0.084311199 -0.006277998
#> cp.err  -0.006277998  0.122140938
#> 
#> [[3]]
#>            eff.err     cp.err
#> eff.err 0.09834771 0.01060251
#> cp.err  0.01060251 0.10024751
#> 
#> [[4]]
#>             eff.err      cp.err
#> eff.err 0.125556975 0.007690868
#> cp.err  0.007690868 0.090991261
#> 
#> [[5]]
#>            eff.err     cp.err
#> eff.err  0.1116261 -0.0184748
#> cp.err  -0.0184748  0.1320288
#> 
#> [[6]]
#>             eff.err      cp.err
#> eff.err 0.093539238 0.007270049
#> cp.err  0.007270049 0.098648424
```

You can also see the parameter realizations from the `$params` data frame.

11.3.2 Simulate using variance/standard deviation standard errors

Lets assume we wish to simulate from the nonmem run included in xpose

First we setup the model:

```r
rx1 <- rxode2({
  cl <- tcl*(1+crcl.cl*(CLCR-65)) * exp(eta.cl)
  v <- tv * WT * exp(eta.v)
  ka <- tka * exp(eta.ka)
  ipred <- linCmt()
  obs <- ipred * (1 + prop.sd) + add.sd
})
```

```
#> using C compiler: ‘gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0’
```

Next we input the estimated parameters:

```r
theta <- c(tcl=2.63E+01, tv=1.35E+00, tka=4.20E+00, tlag=2.08E-01,
           prop.sd=2.05E-01, add.sd=1.06E-02, crcl.cl=7.17E-03,
           ## Note that since we are using the separation strategy the ETA variances are here too
           eta.cl=7.30E-02,  eta.v=3.80E-02, eta.ka=1.91E+00)
```

And also their covariances; To me, the easiest way to create a named covariance matrix is to use `lotri()`:

```r
thetaMat <- lotri(
    tcl + tv + tka + tlag + prop.sd + add.sd + crcl.cl + eta.cl + eta.v + eta.ka ~
        c(7.95E-01,
          2.05E-02, 1.92E-03,
          7.22E-02, -8.30E-03, 6.55E-01,
          -3.45E-03, -6.42E-05, 3.22E-03, 2.47E-04,
          8.71E-04, 2.53E-04, -4.71E-03, -5.79E-05, 5.04E-04,
          6.30E-04, -3.17E-06, -6.52E-04, -1.53E-05, -3.14E-05, 1.34E-05,
          -3.30E-04, 5.46E-06, -3.15E-04, 2.46E-06, 3.15E-06, -1.58E-06, 2.88E-06,
          -1.29E-03, -7.97E-05, 1.68E-03, -2.75E-05, -8.26E-05, 1.13E-05, -1.66E-06, 1.58E-04,
          -1.23E-03, -1.27E-05, -1.33E-03, -1.47E-05, -1.03E-04, 1.02E-05, 1.67E-06, 6.68E-05, 1.
          7.69E-02, -7.23E-03, 3.74E-01, 1.79E-03, -2.85E-03, 1.18E-05, -2.54E-04, 1.61E-03, -9.0

evw <- et(amount.units="mg", time.units="hours") %>%
    et(amt=100) %>%
    ## For this problem we will simulate with sampling windows
    et(list(c(0, 0.5),
       c(0.5, 1),
       c(1, 3),
       c(3, 6),
       c(6, 12))) %>%
    et(id=1:1000)

### From the run we know that:
###   total number of observations is: 476
###    Total number of individuals:     74
sim  <- rxSolve(rx1, theta, evw,  nSub=100, nStud=10,
                thetaMat=thetaMat,
                ## Match boundaries of problem
                thetaLower=0,
                sigma=c("prop.sd", "add.sd"), ## Sigmas are standard deviations
                sigmaXform="identity", # default sigma xform="identity"
                omega=c("eta.cl", "eta.v", "eta.ka"), ## etas are variances
                omegaXform="variance", # default omega xform="variance"
                iCov=data.frame(WT=rnorm(1000, 70, 15), CLCR=rnorm(1000, 65, 25)),
                dfSub=74, dfObs=476);
```

```
#> i thetaMat has too many items, ignored: 'tlag'
```

```
print(sim)
```

```
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#> # A tibble: 10,000 x 9
#>    sim.id id      tcl crcl.cl eta.cl    tv   eta.v   tka    eta.ka
#>     <int> <fct> <dbl>   <dbl>  <dbl> <dbl>   <dbl> <dbl>     <dbl>
#>  1      1 1      26.4  0.0528 -0.272  1.43 -0.620   4.68 -0.786
#>  2      1 2      26.4  0.0528  0.292  1.43  0.822   4.68  0.258
#>  3      1 3      26.4  0.0528  0.168  1.43  0.0419  4.68 -0.000311
#>  4      1 4      26.4  0.0528  0.554  1.43  0.600   4.68 -0.368
#>  5      1 5      26.4  0.0528 -0.416  1.43 -0.965   4.68 -0.168
#>  6      1 6      26.4  0.0528  0.228  1.43 -0.603   4.68 -0.0309
#>  7      1 7      26.4  0.0528 -0.141  1.43 -0.499   4.68 -0.588
#>  8      1 8      26.4  0.0528  0.282  1.43 -1.18    4.68 -0.834
#>  9      1 9      26.4  0.0528  0.168  1.43 -0.266   4.68 -0.881
#> 10      1 10     26.4  0.0528  0.715  1.43  0.0128  4.68  0.759
#> # i 9,990 more rows
#> -- Initial Conditions ($inits): --
#> named numeric(0)
#>
#> Simulation with uncertainty in:
#> * parameters ($thetaMat for changes)
#> * omega matrix ($omegaList)
#> * sigma matrix ($sigmaList)
#>
#> -- First part of data (object): --
#> # A tibble: 50,000 x 10
#>   sim.id    id   time    cl     v    ka  ipred    obs    WT  CLCR
#>    <int> <int>    [h] <dbl> <dbl> <dbl>  <dbl>  <dbl> <dbl> <dbl>
#> 1      1     1 0.0155  24.6  47.8  2.13 0.0678  2.60   62.2  69.3
#> 2      1     1 0.749   24.6  47.8  2.13 1.32    0.264  62.2  69.3
#> 3      1     1 1.02    24.6  47.8  2.13 1.32    3.60   62.2  69.3
#> 4      1     1 3.41    24.6  47.8  2.13 0.474  -0.561  62.2  69.3
#> 5      1     1 7.81    24.6   47.8   2.13 0.0491 -0.393  62.2  69.3
#> 6      1     2 0.0833 111.   116.    6.05 0.327   2.50   35.7 105.
#> # i 49,994 more rows
```

```
### Notice that the simulation time-points change for the individual

### If you want the same sampling time-points you can do that as well:
evw <- et(amount.units="mg", time.units="hours") %>%
    et(amt=100) %>%
    et(0, 24, length.out=50) %>%
    et(id=1:100)

sim  <- rxSolve(rx1, theta, evw,  nSub=100, nStud=10,
                thetaMat=thetaMat,
                ## Match boundaries of problem
                thetaLower=0,
                sigma=c("prop.sd", "add.sd"), ## Sigmas are standard deviations
                sigmaXform="identity", # default sigma xform="identity"
                omega=c("eta.cl", "eta.v", "eta.ka"), ## etas are variances
                omegaXform="variance", # default omega xform="variance"
                iCov=data.frame(WT=rnorm(100, 70, 15), CLCR=rnorm(100, 65, 25)),
                dfSub=74, dfObs=476,
                resample=TRUE)
```

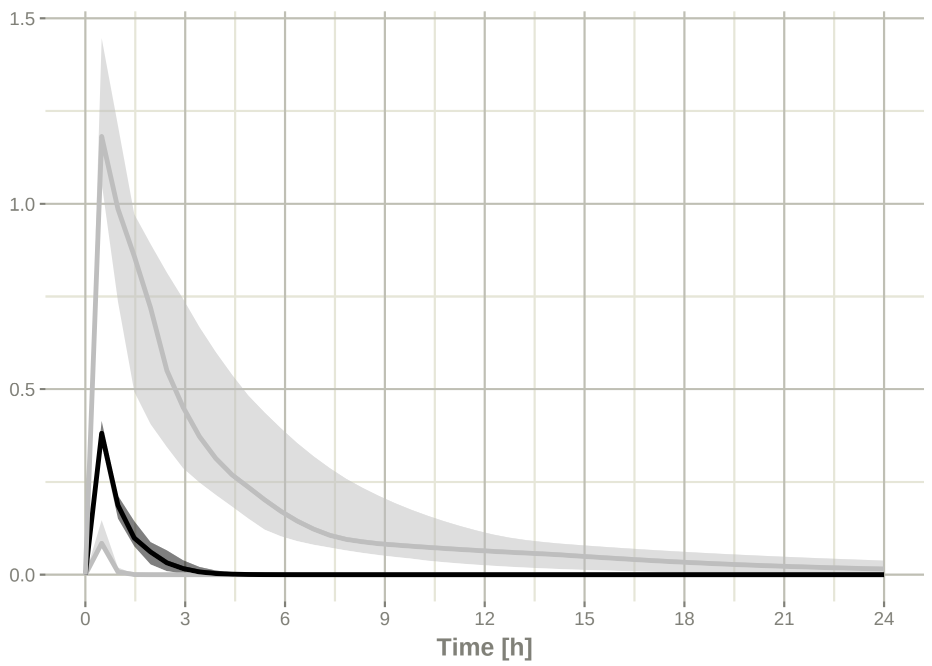
```
#> i thetaMat has too many items, ignored: 'tlag'
```

```
s <-sim %>% confint(c("ipred"))
```

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

```
#> done
```

```
plot(s)
```



11.3.3 Simulate without uncertainty in `omega` or `sigma` parameters

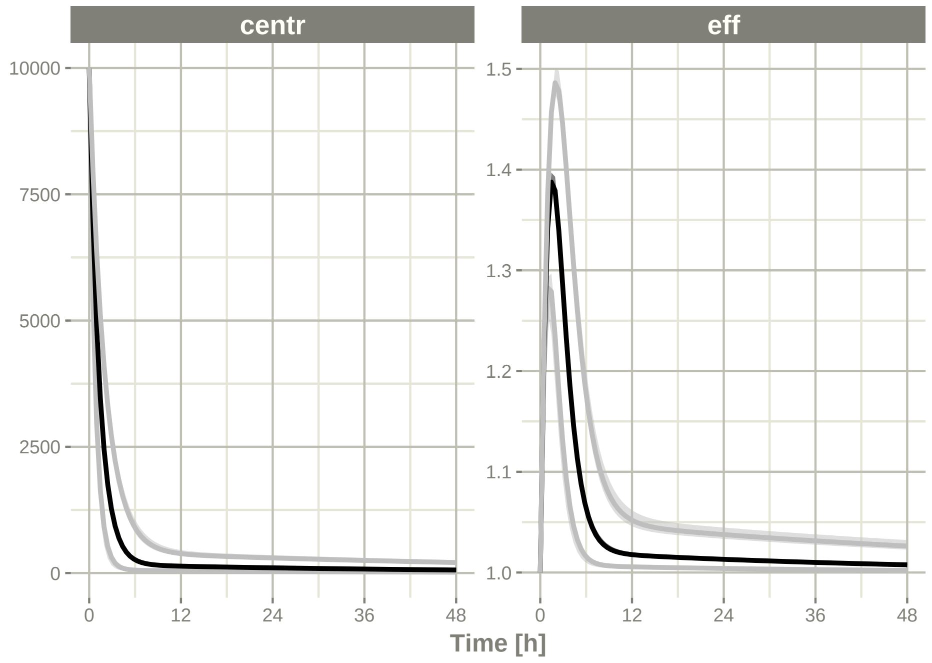
If you do not wish to sample from the prior distributions of either the `omega` or `sigma` matrices, you can turn off this feature by specifying the `simVariability = FALSE` option when solving:

```
mod <- rxode2({
  eff(0) = 1
  C2 = centr/V2;
  C3 = peri/V3;
  CL =  TCl*exp(eta.Cl) ## This is coded as a variable in the model
  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;
  e = eff+eff.err
  cp = centr*(1+cp.err)
})

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

```r
sigma <- lotri(eff.err ~ 0.1, cp.err ~ 0.1)


sim  <- rxSolve(mod, theta, ev, omega=omega, nSub=100, sigma=sigma,
                thetaMat=tMat, nStud=10,
                simVariability=FALSE)

s <-sim %>% confint(c("centr", "eff"))
```

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

```r
plot(s)
```



Note since realizations of `omega` and `sigma` were not simulated, `$omega.list` and `$sigma.list` both return `NULL`.

11.3.3.0.1 rxode2 multi-threaded solving and simulation rxode2 now supports multi-threaded solving on OpenMP supported compilers, including linux and windows. Mac OSX can also be supported By default it uses all your available cores for solving as determined by `rxCores()`. This may be overkill depending on your system, at a certain point the speed of solving is limited by things other than computing power.

You can also speed up simulation by using the multi-cores to generate random deviates with the threefry simulation engine. This is controlled by the `nCoresRV` parameter. For example:

```
sim  <- rxSolve(mod, theta, ev, omega=omega, nSub=100, sigma=sigma, thetaMat=tMat, nStu
                nCoresRV=2)

s <-sim %>% confint(c("eff", "centr"))
```

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

The default for this is 1 core since the result depends on the number of cores and the random seed you use in your simulation as well as the work-load each thread is sharing/architecture. However, you can always speed up this process with more cores if you are sure your collaborators have the same number of cores available to them and have OpenMP thread-capable compile.

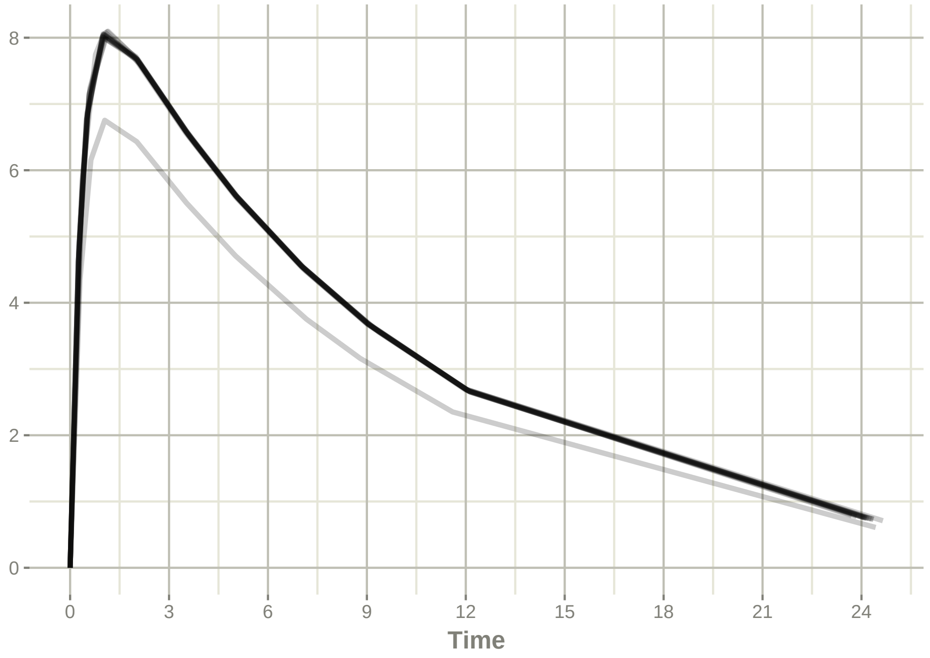
11.4 Using prior data for solving

rxode2 can use a single subject or multiple subjects with a single event table to solve ODEs. Additionally, rxode2 can use an arbitrary data frame with individualized events. For example when using `nlmixr`, you could use the `rxode2/vignettes/theo_sd` data frame

```
library(rxode2)
### Load data from nlmixr
d <- qs::qread("rxode2/vignettes/theo_sd.qs")

### Create rxode2 model
theo <- rxode2({
    tka ~ 0.45 # Log Ka
    tcl ~ 1 # Log Cl
    tv ~ 3.45    # Log V
    eta.ka ~ 0.6
    eta.cl ~ 0.3
    eta.v ~ 0.1
    ka <- exp(tka + eta.ka)
    cl <- exp(tcl + eta.cl)
    v <- exp(tv + eta.v)
    d/dt(depot) = -ka * depot
    d/dt(center) = ka * depot - cl / v * center
    cp = center / v
})
```

```
#> using C compiler: 'gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0'
```

```
### Create parameter dataset
library(dplyr)
parsDf <- tribble(
  ~ eta.ka, ~ eta.cl, ~ eta.v,
  0.105, -0.487, -0.080,
  0.221, 0.144, 0.021,
  0.368, 0.031, 0.058,
 -0.277, -0.015, -0.007,
 -0.046, -0.155, -0.142,
 -0.382, 0.367, 0.203,
 -0.791, 0.160, 0.047,
 -0.181, 0.168, 0.096,
  1.420, 0.042, 0.012,
 -0.738, -0.391, -0.170,
  0.790, 0.281, 0.146,
 -0.527, -0.126, -0.198) %>%
    mutate(tka = 0.451, tcl = 1.017, tv = 3.449)

### Now solve the dataset
solveData <- rxSolve(theo, parsDf, d)

plot(solveData, cp)
```



```
print(solveData)
```

```
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#> # A tibble: 12 x 1
#>    id
#>    <fct>
#>  1 1
#>  2 2
#>  3 3
#>  4 4
#>  5 5
#>  6 6
#>  7 7
#>  8 8
#>  9 9
#> 10 10
#> 11 11
#> 12 12
#> -- Initial Conditions ($inits): --
#>  depot center
#>      0      0
#> -- First part of data (object): --
#> # A tibble: 132 x 8
#>      id  time    ka    cl     v    cp     depot center
#>   <int> <dbl> <dbl> <dbl> <dbl> <dbl>     <dbl>  <dbl>
#> 1     1  0     2.86  3.67  34.8  0    320.          0
#> 2     1  0.25  2.86  3.67  34.8  4.62 157.        161.
#> 3     1  0.57  2.86  3.67  34.8  7.12  62.8       248.
#> 4     1  1.12  2.86  3.67  34.8  8.09  13.0       282.
#> 5     1  2.02  2.86  3.67  34.8  7.68   0.996     267.
#> 6     1  3.82  2.86  3.67  34.8  6.38   0.00581   222.
#> # i 126 more rows
```

```
### Of course the fasest way to solve if you don't care about the rxode2 extra paramet

solveData <- rxSolve(theo, parsDf, d, returnType="data.frame")

### solved data
dplyr::as.tbl(solveData)
```

```
#> Warning: `as.tbl()` was deprecated in dplyr 1.0.0.
#> i Please use `tibble::as_tibble()` instead.
#> Call `lifecycle::last_lifecycle_warnings()` to see where
#> this warning was generated.
```

```
#> # A tibble: 132 x 8
#>       id  time    ka    cl     v    cp   depot center
#>    <int> <dbl> <dbl> <dbl> <dbl> <dbl>   <dbl>  <dbl>
#>  1     1  0     2.86  3.67  34.8  0    3.20e+2    0
#>  2     1  0.25  2.86  3.67  34.8  4.62 1.57e+2  161.
#>  3     1  0.57  2.86  3.67  34.8  7.12 6.28e+1  248.
#>  4     1  1.12  2.86  3.67  34.8  8.09 1.30e+1  282.
#>  5     1  2.02  2.86  3.67  34.8  7.68 9.96e-1  267.
#>  6     1  3.82  2.86  3.67  34.8  6.38 5.81e-3  222.
#>  7     1  5.1   2.86  3.67  34.8  5.58 1.50e-4  194.
#>  8     1  7.03  2.86  3.67  34.8  4.55 6.02e-7  158.
#>  9     1  9.05  2.86  3.67  34.8  3.68 1.77e-9  128.
#> 10     1 12.1   2.86  3.67  34.8  2.66 9.43e-9   92.6
#> # i 122 more rows
```

```
data.table::data.table(solveData)
```

```
#>       id  time       ka       cl        v        cp         depot    center
#>   1:  1  0.00 2.857651 3.669297 34.81332 0.0000000  3.199920e+02   0.00000
#>   2:  1  0.25 2.857651 3.669297 34.81332 4.6240421  1.566295e+02 160.97825
#>   3:  1  0.57 2.857651 3.669297 34.81332 7.1151647  6.276731e+01 247.70249
#>   4:  1  1.12 2.857651 3.669297 34.81332 8.0922106  1.303613e+01 281.71670
#>   5:  1  2.02 2.857651 3.669297 34.81332 7.6837844  9.958446e-01 267.49803
#>  ---
#> 128: 12  5.07 2.857651 3.669297 34.81332 5.6044213  1.636210e-04 195.10850
#> 129: 12  7.07 2.857651 3.669297 34.81332 4.5392337  5.385697e-07 158.02579
#> 130: 12  9.03 2.857651 3.669297 34.81332 3.6920276  1.882087e-09 128.53173
#> 131: 12 12.05 2.857651 3.669297 34.81332 2.6855080  8.461424e-09  93.49144
#> 132: 12 24.15 2.857651 3.669297 34.81332 0.7501667 -4.775222e-10  26.11579
```

Chapter 12

Examples

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

12.1 Prediction only models

Prediction only models are simple to create. You use the rxode2 syntax without any ODE systems in them. A very simple example is a one-compartment model.

```
library(rxode2)
mod <- rxode2({
    ipre <- 10 * exp(-ke * t);
})
```

```
#> using C compiler: ‘gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0’
```

```
mod
```

```
#> rxode2 2.0.13.9000 model named rx_7d320f9008008e96872e52afa002bd5e model (ready).
#> x$params: ke
#> x$lhs: ipre
```

Solving the rxode2 models are the same as saving the simple ODE system, but faster of course.

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

```
#> -- Solved rxode2 object --
#> -- Parameters (x$params): --
#>  ke
#> 0.5
#> -- Initial Conditions (x$inits): --
#> named numeric(0)
#> -- First part of data (object): --
#> # A tibble: 50 x 2
#>    time  ipre
#>   <dbl> <dbl>
#> 1 0     10
#> 2 0.490  7.83
#> 3 0.980  6.13
#> 4 1.47   4.80
#> 5 1.96   3.75
#> 6 2.45   2.94
#> # i 44 more rows
```

12.2 Solved compartment models

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

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

```
#>
#> Attaching package: 'kableExtra'
#> The following object is masked from 'package:dplyr':
#>
#>     group_rows
```

Table 12.1: Clearance Based linCmt() parameterizations

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

Table 12.1: Clearance Based linCmt() parameterizations *(continued)*

par1	par2	par3	par4	par5	par6	par7	ncmt
ka	cl	q2	v	vp2			2
cl	q2	v	vp2				2
ka	cl	q2	vc	vp2			2
cl	q2	vc	vp2				2
ka	cl	cld	cld2	v	vp	vp2	3
cl	cld	cld2	v	vp	vp2		3
ka	cl	cld	cld2	vc	vp	vp2	3
cl	cld	cld2	vc	vp	vp2		3
ka	cl	cld2	v	vp2			2
cl	cld2	v	vp2				2
ka	cl	cld2	vc	vp2			2
cl	cld2	vc	vp2				2
ka	cl	q	v	vp2			2
cl	q	v	vp2				2
ka	cl	q	vc	vp2			2
cl	q	vc	vp2				2
ka	cl	q1	v	vp2			2
cl	q1	v	vp2				2
ka	cl	q1	vc	vp2			2
cl	q1	vc	vp2				2
ka	cl	cld	v	vp2			2
cl	cld	v	vp2				2
ka	cl	cld	vc	vp2			2
cl	cld	vc	vp2				2
ka	cl	q	q2	v	v2	v3	3
cl	q	q2	v	v2	v3		3
ka	cl	q	q2	v2	v3	vc	3
cl	q	q2	v2	v3	vc		3
ka	cl	q	q2	v1	v2	v3	3
cl	q	q2	v1	v2	v3		3
ka	cl	q1	q2	v	v2	v3	3
cl	q1	q2	v	v2	v3		3
ka	cl	q1	q2	v2	v3	vc	3
cl	q1	q2	v2	v3	vc		3
ka	cl	q1	q2	v1	v2	v3	3
cl	q1	q2	v1	v2	v3		3
ka	cl	q2	v2	v3			2
cl	q2	v2	v3				2
ka	cl	q2	v	v3			2

Table 12.1: Clearance Based linCmt() parameterizations *(continued)*

par1	par2	par3	par4	par5	par6	par7	ncmt
cl	q2	v	v3				2
ka	cl	q2	v3	vc			2
cl	q2	v3	vc				2
ka	cl	cld	cld2	v	v2	v3	3
cl	cld	cld2	v	v2	v3		3
ka	cl	cld	cld2	v2	v3	vc	3
cl	cld	cld2	v2	v3	vc		3
ka	cl	cld	cld2	v1	v2	v3	3
cl	cld	cld2	v1	v2	v3		3
ka	cl	cld2	v2	v3			2
cl	cld2	v2	v3				2
ka	cl	cld2	v	v3			2
cl	cld2	v	v3				2
ka	cl	cld2	v3	vc			2
cl	cld2	v3	vc				2
ka	cl	q	v2	v3			2
cl	q	v2	v3				2
ka	cl	q1	v2	v3			2
cl	q1	v2	v3				2
ka	cl	cld	v2	v3			2
cl	cld	v2	v3				2
ka	cl	q	v	v3			2
cl	q	v	v3				2
ka	cl	q	v3	vc			2
cl	q	v3	vc				2
ka	cl	q1	v	v3			2
cl	q1	v	v3				2
ka	cl	q1	v3	vc			2
cl	q1	v3	vc				2
ka	cl	cld	v	v3			2
cl	cld	v	v3				2
ka	cl	cld	v3	vc			2
cl	cld	v3	vc				2
ka	cl	q	q2	v	vt	vt2	3
cl	q	q2	v	vt	vt2		3
ka	cl	q	q2	vc	vt	vt2	3
cl	q	q2	vc	vt	vt2		3
ka	cl	q1	q2	v	vt	vt2	3
cl	q1	q2	v	vt	vt2		3

Table 12.1: Clearance Based linCmt() parameterizations *(continued)*

par1	par2	par3	par4	par5	par6	par7	ncmt
ka	cl	q1	q2	vc	vt	vt2	3
cl	q1	q2	vc	vt	vt2		3
ka	cl	q2	v	vt2			2
cl	q2	v	vt2				2
ka	cl	q2	vc	vt2			2
cl	q2	vc	vt2				2
ka	cl	cld	cld2	v	vt	vt2	3
cl	cld	cld2	v	vt	vt2		3
ka	cl	cld	cld2	vc	vt	vt2	3
cl	cld	cld2	vc	vt	vt2		3
ka	cl	cld2	v	vt2			2
cl	cld2	v	vt2				2
ka	cl	cld2	vc	vt2			2
cl	cld2	vc	vt2				2
ka	cl	q	v	vt2			2
cl	q	v	vt2				2
ka	cl	q	vc	vt2			2
cl	q	vc	vt2				2
ka	cl	q1	v	vt2			2
cl	q1	v	vt2				2
ka	cl	q1	vc	vt2			2
cl	q1	vc	vt2				2
ka	cl	cld	v	vt2			2
cl	cld	v	vt2				2
ka	cl	cld	vc	vt2			2
cl	cld	vc	vt2				2
ka	cl	q2	v	v2			2
cl	q2	v	v2				2
ka	cl	q2	v2	vc			2
cl	q2	v2	vc				2
ka	cl	q2	v1	v2			2
cl	q2	v1	v2				2
ka	cl	q2	v	vp			2
cl	q2	v	vp				2
ka	cl	q2	vc	vp			2
cl	q2	vc	vp				2
ka	cl	q2	v	vt			2
cl	q2	v	vt				2
ka	cl	q2	vc	vt			2

Table 12.1: Clearance Based linCmt() parameterizations *(continued)*

par1	par2	par3	par4	par5	par6	par7	ncmt
cl	q2	vc	vt				2
ka	cl	q2	v	vss			2
cl	q2	v	vss				2
ka	cl	q2	vc	vss			2
cl	q2	vc	vss				2
ka	cl	cld2	v	v2			2
cl	cld2	v	v2				2
ka	cl	cld2	v2	vc			2
cl	cld2	v2	vc				2
ka	cl	cld2	v1	v2			2
cl	cld2	v1	v2				2
ka	cl	cld2	v	vp			2
cl	cld2	v	vp				2
ka	cl	cld2	vc	vp			2
cl	cld2	vc	vp				2
ka	cl	cld2	v	vt			2
cl	cld2	v	vt				2
ka	cl	cld2	vc	vt			2
cl	cld2	vc	vt				2
ka	cl	cld2	v	vss			2
cl	cld2	v	vss				2
ka	cl	cld2	vc	vss			2
cl	cld2	vc	vss				2
ka	cl	q	v	v2			2
cl	q	v	v2				2
ka	cl	q	v2	vc			2
cl	q	v2	vc				2
ka	cl	q	v1	v2			2
cl	q	v1	v2				2
ka	cl	q1	v	v2			2
cl	q1	v	v2				2
ka	cl	q1	v2	vc			2
cl	q1	v2	vc				2
ka	cl	q1	v1	v2			2
cl	q1	v1	v2				2
ka	cl	cld	v	v2			2
cl	cld	v	v2				2
ka	cl	cld	v2	vc			2
cl	cld	v2	vc				2

Table 12.1: Clearance Based linCmt() parameterizations *(continued)*

par1	par2	par3	par4	par5	par6	par7	ncmt
ka	cl	cld	v1	v2			2
cl	cld	v1	v2				2
ka	cl	v2					1
cl	v2						1
ka	cl	q	v	vp			2
cl	q	v	vp				2
ka	cl	q	vc	vp			2
cl	q	vc	vp				2
ka	cl	q1	v	vp			2
cl	q1	v	vp				2
ka	cl	q1	vc	vp			2
cl	q1	vc	vp				2
ka	cl	cld	v	vp			2
cl	cld	v	vp				2
ka	cl	cld	vc	vp			2
cl	cld	vc	vp				2
ka	cl	q	v	vt			2
cl	q	v	vt				2
ka	cl	q	vc	vt			2
cl	q	vc	vt				2
ka	cl	q1	v	vt			2
cl	q1	v	vt				2
ka	cl	q1	vc	vt			2
cl	q1	vc	vt				2
ka	cl	cld	v	vt			2
cl	cld	v	vt				2
ka	cl	cld	vc	vt			2
cl	cld	vc	vt				2
ka	cl	q	v	vss			2
cl	q	v	vss				2
ka	cl	q	vc	vss			2
cl	q	vc	vss				2
ka	cl	q1	v	vss			2
cl	q1	v	vss				2
ka	cl	q1	vc	vss			2
cl	q1	vc	vss				2
ka	cl	cld	v	vss			2
cl	cld	v	vss				2
ka	cl	cld	vc	vss			2

Table 12.1: Clearance Based linCmt() parameterizations *(continued)*

par1	par2	par3	par4	par5	par6	par7	ncmt
cl	cld	vc	vss				2
ka	cl	v					1
cl	v						1
ka	cl	vc					1
cl	vc						1
ka	cl	v1					1
cl	v1						1

Another popular parameterization is in terms of micro-constants. rxode2 assumes compartment `1` is the central compartment. The elimination constant would be specified by `K`, `Ke` or `Kel`. Some example translations are below:

Table 12.2: Kel Based linCmt() parameterizations

par1	par2	par3	par4	par5	par6	par7	ncmt
ka	v	k	k12	k21	k13	k31	3
v	k	k12	k21	k13	k31		3
ka	vc	k	k12	k21	k13	k31	3
vc	k	k12	k21	k13	k31		3
ka	v1	k	k12	k21	k13	k31	3
v1	k	k12	k21	k13	k31		3
ka	v	ke	k12	k21	k13	k31	3
v	ke	k12	k21	k13	k31		3
ka	vc	ke	k12	k21	k13	k31	3
vc	ke	k12	k21	k13	k31		3
ka	v1	ke	k12	k21	k13	k31	3
v1	ke	k12	k21	k13	k31		3
ka	v	kel	k12	k21	k13	k31	3
v	kel	k12	k21	k13	k31		3
ka	vc	kel	k12	k21	k13	k31	3
vc	kel	k12	k21	k13	k31		3
ka	v1	kel	k12	k21	k13	k31	3
v1	kel	k12	k21	k13	k31		3
ka	v	k	k12	k21			2
v	k	k12	k21				2
ka	vc	k	k12	k21			2
vc	k	k12	k21				2
ka	v1	k	k12	k21			2

Table 12.2: Kel Based linCmt() parameterizations *(continued)*

par1	par2	par3	par4	par5	par6	par7	ncmt
v1	k	k12	k21				2
ka	v	ke	k12	k21			2
v	ke	k12	k21				2
ka	vc	ke	k12	k21			2
vc	ke	k12	k21				2
ka	v1	ke	k12	k21			2
v1	ke	k12	k21				2
ka	v	kel	k12	k21			2
v	kel	k12	k21				2
ka	vc	kel	k12	k21			2
vc	kel	k12	k21				2
ka	v1	kel	k12	k21			2
v1	kel	k12	k21				2
ka	v	k					1
v	k						1
ka	vc	k					1
vc	k						1
ka	v1	k					1
v1	k						1
ka	v	ke					1
v	ke						1
ka	vc	ke					1
vc	ke						1
ka	v1	ke					1
v1	ke						1
ka	v	kel					1
v	kel						1
ka	vc	kel					1
vc	kel						1
ka	v1	kel					1
v1	kel						1

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

Table 12.3: alpha Based linCmt() parameterizations

par1	par2	par3	par4	par5	par6	par7	ncmt
ka	v	alpha	beta	aob			1
v	alpha	beta	aob				1
ka	vc	alpha	beta	aob			1
vc	alpha	beta	aob				1
ka	v1	alpha	beta	aob			1
v1	alpha	beta	aob				1
ka	v	alpha	beta	k21			1
v	alpha	beta	k21				1
ka	vc	alpha	beta	k21			1
vc	alpha	beta	k21				1
ka	v1	alpha	beta	k21			1
v1	alpha	beta	k21				1
ka	v	alpha					2
v	alpha						2
ka	vc	alpha					2
vc	alpha						2
ka	v1	alpha					2
v1	alpha						2
ka	a	alpha	b	beta	c	gamma	3
a	alpha	b	beta	c	gamma		3
ka	a	alpha	b	beta			2
a	alpha	b	beta				2
ka	a	alpha					1
a	alpha						1

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

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

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

These compartments take the same sort of events that a ODE model can take, and are discussed in the rxode2 events vignette.

```
mod <- rxode2({
    ke <- 0.5
    V <- 1
    ipre <- linCmt();
})
```

```
#> using C compiler: 'gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0'
```

```
mod
```

```
#> rxode2 2.0.13.9000 model named rx_59f045bee61f788b86cebce40567ff2d model (ready).
#> x$stateExtra: central
#> x$params: ke, V
#> x$lhs: ipre
```

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

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

```
#> -- Solved rxode2 object --
#> -- Parameters (x$params): --
#>  ke   V
#> 0.5 1.0
#> -- Initial Conditions (x$inits): --
#> named numeric(0)
#> -- First part of data (object): --
#> # A tibble: 50 x 2
#>    time  ipre
#>   <dbl> <dbl>
#> 1 0     10
#> 2 0.490  7.83
#> 3 0.980  6.13
#> 4 1.47   4.80
#> 5 1.96   3.75
#> 6 2.45   2.94
#> # i 44 more rows
```

12.3 Mixing Solved Systems and ODEs

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

```
library(rxode2)
## Setup example model
mod1 <-rxode2({
    C2 = centr/V2;
    C3 = peri/V3;
    d/dt(depot) =-KA*depot;
    d/dt(centr) = KA*depot - CL*C2 - Q*C2 + Q*C3;
    d/dt(peri)  =                    Q*C2 - Q*C3;
    d/dt(eff)  = Kin - Kout*(1-C2/(EC50+C2))*eff;
});

## Seup parameters and initial conditions

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

inits <- c(eff=1);

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

## Setup a mixed solved/ode system:
mod2 <- rxode2({
    ## the order of variables do not matter, the type of compartmental
    ## model is determined by the parameters specified.
    C2   = linCmt(KA, CL, V2, Q, V3);
    eff(0) = 1  ## This specifies that the effect compartment starts at 1.
    d/dt(eff) =  Kin - Kout*(1-C2/(EC50+C2))*eff;
})
```

```
#> using C compiler: ‘gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0’
```

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

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

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

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

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

This can be solved with the following command:

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

```
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#>      CL      V2       Q      V3      KA     Kin    Kout    EC50
#>  18.600  40.200  10.500 297.000   0.294   1.000   1.000 200.000
#> -- Initial Conditions ($inits): --
#> eff
#>   1
#> -- First part of data (object): --
#> # A tibble: 241 x 3
#>   time    C2   eff
#>    [h] <dbl> <dbl>
#> 1    0 249.   1
#> 2    1 121.   1.35
#> 3    2  60.3  1.38
#> 4    3  31.0  1.28
#> 5    4  17.0  1.18
#> 6    5  10.2  1.11
#> # i 235 more rows
```

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

This can be solved for different initial conditions easily:

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

```
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#>      CL      V2       Q      V3      KA     Kin    Kout    EC50
#>  18.600  40.200  10.500 297.000   0.294   1.000   1.000 200.000
#> -- Initial Conditions ($inits): --
#> eff
#>   2
#> -- First part of data (object): --
#> # A tibble: 241 x 3
#>   time    C2   eff
#>    [h] <dbl> <dbl>
#> 1    0 249.   2
#> 2    1 121.   1.93
#> 3    2  60.3  1.67
#> 4    3  31.0  1.41
#> 5    4  17.0  1.23
#> 6    5  10.2  1.13
#> # i 235 more rows
```

The rxode2 detective also does not require you to specify the variables in the `linCmt()` function if they are already defined in the block. Therefore, the following function will also work to solve the same system.

```
mod3 <- rxode2({
    KA=2.94E-01;
    CL=1.86E+01;
    V2=4.02E+01;
    Q=1.05E+01;
    V3=2.97E+02;
    Kin=1;
    Kout=1;
    EC50=200;
    ## The linCmt() picks up the variables from above
    C2   = linCmt();
    eff(0) = 1  ## This specifies that the effect compartment starts at 1.
    d/dt(eff) =  Kin - Kout*(1-C2/(EC50+C2))*eff;
})
```

```
#> using C compiler: ‘gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0’
```

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

```
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#>      KA      CL      V2       Q      V3     Kin    Kout    EC50
#>   0.294  18.600  40.200  10.500 297.000   1.000   1.000 200.000
#> -- Initial Conditions ($inits): --
#> eff
#>   1
#> -- First part of data (object): --
#> # A tibble: 241 x 3
#>   time    C2   eff
#>    [h] <dbl> <dbl>
#> 1    0 249.   1
#> 2    1 121.   1.35
#> 3    2  60.3  1.38
#> 4     3  31.0  1.28
#> 5     4  17.0  1.18
#> 6     5  10.2  1.11
#> # i 235 more rows
```

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

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

```
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#>     KA     CL     V2      Q     V3    Kin  Kout   EC50
#>   10.0   18.6   40.2   10.5 297.0    1.0   1.0  200.0
#> -- Initial Conditions ($inits): --
#> eff
#>   1
#> -- First part of data (object): --
#> # A tibble: 241 x 3
#>    time    C2   eff
#>     [h] <dbl> <dbl>
#> 1     0 249.   1
#> 2     1 121.   1.35
#> 3     2  60.3  1.38
#> 4     3  31.0  1.28
#> 5     4  17.0  1.18
#> 6     5  10.2  1.11
#> # i 235 more rows
```

12.4 Weight based dosing

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

There are 3 stages for weight-based dosing simulations: - Create rxode2 model - Simulate Covariates - Create event table with weight-based dosing (merged back to covariates)

12.4.1 Creating a 2-compartment model in rxode2

```
library(rxode2)

## Note the time covariate is not included in the simulation
m1 <- rxode2({
  CL ~ (1-0.2*SEX)*(0.807+0.00514*(CRCL-91.2))*exp(eta.cl)
  V1 ~ 4.8*exp(eta.v1)
  Q ~ (3.46+0.0593*(WT-75.1))*exp(eta.q);
  V2 ~ 1.93*(3.13+0.0458*(WT-75.1))*exp(eta.v2)
  A1 ~ centr;
  A2 ~ peri;
  d/dt(centr) ~ - A1*(CL/V1 + Q/V1) + A2*Q/V2;
  d/dt(peri) ~ A1*Q/V1 - A2*Q/V2;
  DV = centr / V1 * (1 + prop.err)
})
```

```
#> using C compiler: ‘gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0’
```

12.4.2 Simulating Covariates

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

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

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

12.4.3 Creating weight based event table

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

e <- et() %>%
    ## Specify the id and weight based dosing from covariate data.frame
    ## This requires rxode2 XXX
    et(id=cov.df$id, amt=6*cov.df$WT, rate=6 * cov.df$WT) %>%
    ## Sampling is added for each ID
    et(s) %>%
    as.data.frame %>%
    ## Merge the event table with the covarite information
    merge(cov.df, by="id") %>%
    as_tibble


e
```

```
#> # A tibble: 900 x 12
#>       id   low  time  high cmt         amt  rate  evid   AGE   SEX    WT  CRCL
#>    <int> <dbl> <dbl> <dbl> <chr>     <dbl> <dbl> <int> <dbl> <dbl> <dbl> <dbl>
#>  1     1 0     0     0     (obs)        NA    NA     0    66     1  49.4    83
#>  2     1 NA    0     NA    (default)  296.  296.     1    66     1  49.4    83
#>  3     1 0.225 0.246 0.275 (obs)        NA    NA     0    66     1  49.4    83
#>  4     1 0.45  0.516 0.55  (obs)        NA    NA     0    66     1  49.4    83
#>  5     1 0.675 0.729 0.825 (obs)        NA    NA     0    66     1  49.4    83
#>  6     1 0.9   0.921 1.1   (obs)        NA    NA     0    66     1  49.4    83
#>  7     1 1.35  1.42  1.65  (obs)        NA    NA     0    66     1  49.4    83
#>  8     1 1.8   1.82  2.2   (obs)        NA    NA     0    66     1  49.4    83
#>  9     1 2.7   2.97  3.3   (obs)        NA    NA     0    66     1  49.4    83
#> 10     1 3.6   3.87  4.4   (obs)        NA    NA     0    66     1  49.4    83
#> # i 890 more rows
```

12.4.4 Solving Daptomycin simulation

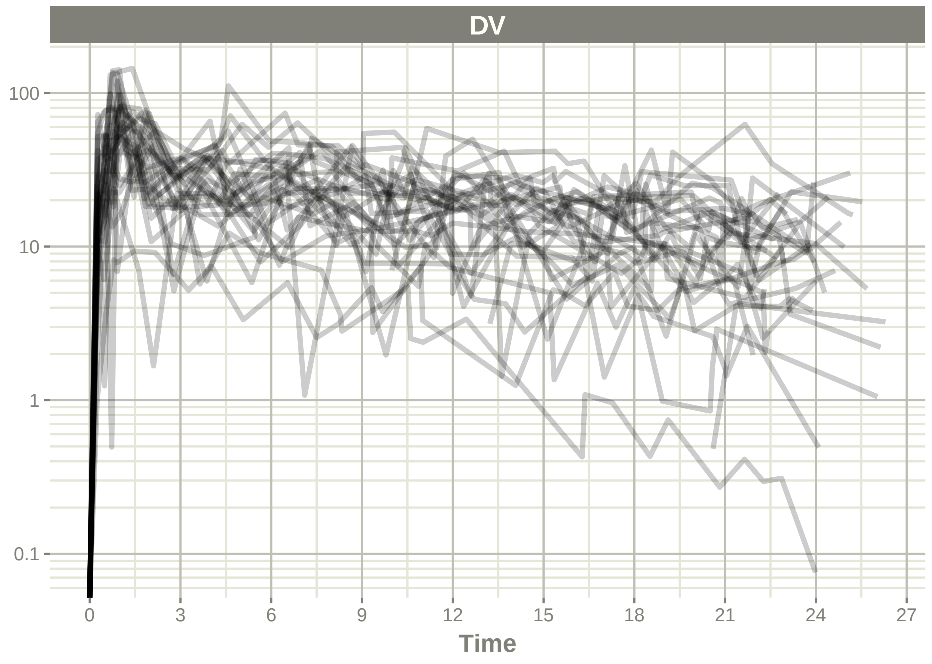
```
data <- rxSolve(m1, e,
             ## Lotri uses lower-triangular matrix rep. for named matrix
             omega=lotri(eta.cl ~ .306,
                         eta.q ~0.0652,
                         eta.v1 ~.567,
                         eta.v2 ~ .191),
             sigma=lotri(prop.err ~ 0.15),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

print(data)
```

```
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#> # A tibble: 30 x 5
#>    id      eta.cl  eta.v1    eta.q  eta.v2
#>    <fct>    <dbl>   <dbl>    <dbl>   <dbl>
#>  1 1     -0.852    0.158   0.00529  0.0100
#>  2 2     -0.444   -1.09   -0.374   -0.296
#>  3 3      0.930    0.553   0.0127  -0.446
#>  4 4     -0.813    0.0732 -0.230    0.211
#>  5 5      0.219    0.175  -0.0883  -0.122
#>  6 6     -0.875   -0.0607  0.0823   0.541
#>  7 7     -0.463    0.611   0.613   -0.630
#>  8 8      0.0513   0.167  -0.0800  -0.892
#>  9 9      0.220   -1.02    0.312    0.481
#> 10 10     0.00477  0.547  -0.216   -0.482
#> # i 20 more rows
#> -- Initial Conditions ($inits): --
#> centr  peri
#>     0     0
#> -- First part of data (object): --
#> # A tibble: 900 x 10
#>      id  evid   cmt   amt  rate  time    DV   SEX    WT  CRCL
#>   <int> <int> <int> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1     1     1     1  296.  296. 0         0     1  49.4    83
#> 2     1     0    NA    NA    NA 0         0     1  49.4    83
#> 3     1     0    NA    NA    NA 0.246  20.0     1  49.4    83
#> 4     1     0    NA    NA    NA 0.516  25.3     1  49.4    83
#> 5     1     0    NA    NA    NA 0.729  19.4     1  49.4    83
#> 6     1     0    NA    NA    NA 0.921  38.3     1  49.4    83
#> # i 894 more rows
```

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

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

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



12.4.5 Daptomycin Reference

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

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

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

12.5 Inter-occasion and other nesting examples

More than one level of nesting is possible in rxode2; In this example we will be using the following uncertainties and sources of variability:

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

Level	Variable	Matrix specified	Integrated Matrix
Unexplained Effect	`add.sd`	`sigma`	`sigma`

12.5.1 Event table

This event table contains nesting variables:

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

```r
library(rxode2)
library(dplyr)

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

12.5.2 rxode2 model

This creates the `rxode2` model with multi-level nesting. Note the variables `inv.Cl`, `inv.Ka`, `eta.Cl` etc; You only need one variable for each level of nesting.

```r
mod <- rxode2({
  ## Clearance with individuals
  eff(0) = 1
  C2 = centr/V2*(1+prop.sd);
  C3 = peri/V3;
  CL =  TCl*exp(eta.Cl + eye.Cl + iov.Cl + inv.Cl)
  KA = TKA * exp(eta.Ka + eye.Ka + iov.Cl + inv.Ka)
  d/dt(depot) =-KA*depot;
  d/dt(centr) = KA*depot - CL*C2 - Q*C2 + Q*C3;
  d/dt(peri)  =                    Q*C2 - Q*C3;
  d/dt(eff)  = Kin - Kout*(1-C2/(EC50+C2))*eff;
  ef0 = eff + add.sd
})
```

```
#> using C compiler: ‘gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0’
```

12.5.3 Uncertainty in Model parameters

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

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

tMat
```

```
#>                TKA         TCl           V2             Q           V3
#> TKA   1.408151e-01  0.08277499  0.0180178917 -0.0470325576  0.029172564
#> TCl   8.277499e-02  0.18104452 -0.0532724661 -0.0421074920  0.068093695
#> V2    1.801789e-02 -0.05327247  0.0581816756  0.0001167516  0.006496495
#> Q    -4.703256e-02 -0.04210749  0.0001167516  0.1549374667  0.020764042
#> V3    2.917256e-02  0.06809370  0.0064964951  0.0207640421  0.118986685
#> Kin  -3.445136e-02  0.01464937 -0.0426405263  0.1503174753 -0.039702872
#> Kout -2.904363e-02 -0.04914350  0.0324790929  0.0069332072  0.030349396
#> EC50 -4.017336e-05  0.02850637 -0.0326094799 -0.0489119232 -0.029606732
#>               Kin         Kout          EC50
#> TKA  -0.034451357 -0.029043632 -4.017336e-05
#> TCl   0.014649373 -0.049143503  2.850637e-02
#> V2   -0.042640526  0.032479093 -3.260948e-02
#> Q     0.150317475  0.006933207 -4.891192e-02
#> V3   -0.039702872  0.030349396 -2.960673e-02
#> Kin   0.299597107 -0.074421154 -6.528526e-03
#> Kout -0.074421154  0.061039604 -2.800741e-02
#> EC50 -0.006528526 -0.028007407  4.167429e-02
```

12.5.4 Nesting Variability

To specify multiple levels of nesting, you can specify it as a nested `lotri` matrix; When using this approach you use the condition operator `|` to specify what variable nesting occurs on; For the Bayesian simulation we need to specify how much information we have for each parameter; For `rxode2` this is the `nu` parameter.

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

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

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

```
#> $id
#>        eta.Cl eta.Ka
#> eta.Cl    0.1    0.0
#> eta.Ka    0.0    0.1
#>
#> $eye
#>        eye.Cl eye.Ka
#> eye.Cl   0.05   0.00
#> eye.Ka   0.00   0.05
#>
#> $occ
#>        iov.Cl iov.Ka
#> iov.Cl   0.01   0.00
#> iov.Ka   0.00   0.01
#>
#> $inv
#>        inv.Cl inv.Ka
#> inv.Cl   0.02   0.00
#> inv.Ka   0.00   0.02
#>
#> Properties: nu
```

12.5.5 Unexplained variability

The last piece of variability to specify is the unexplained variability

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

12.5.6 Solving the problem

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

```
#> using C compiler: ‘gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0’
```

```
#> unhandled error message: EE:[lsoda] 70000 steps taken before reaching tout
#>  @(lsoda.c:750
```

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

```
print(s)
```

```
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#> # A tibble: 8,000 x 24
#>    sim.id id    `inv.Cl(inv==1)` `inv.Cl(inv==2)` `inv.Ka(inv==1)`
#>     <int> <fct>            <dbl>            <dbl>            <dbl>
#>  1      1 1                0.251           0.0765          -0.0741
#>  2      1 2                0.251           0.0765          -0.0741
#>  3      1 3                0.251           0.0765          -0.0741
#>  4      1 4                0.251           0.0765          -0.0741
#>  5      1 5                0.251           0.0765          -0.0741
#>  6      1 6                0.251           0.0765          -0.0741
#>  7      1 7                0.251           0.0765          -0.0741
#>  8      1 8                0.251           0.0765          -0.0741
#>  9      1 9                0.251           0.0765          -0.0741
#> 10      1 10               0.251           0.0765          -0.0741
#> # i 7,990 more rows
#> # i 19 more variables: `inv.Ka(inv==2)` <dbl>, `eye.Cl(eye==1)` <dbl>,
#> #   `eye.Cl(eye==2)` <dbl>, `eye.Ka(eye==1)` <dbl>, `eye.Ka(eye==2)` <dbl>,
#> #   `iov.Cl(occ==1)` <dbl>, `iov.Cl(occ==2)` <dbl>, `iov.Ka(occ==1)` <dbl>,
#> #   `iov.Ka(occ==2)` <dbl>, V2 <dbl>, V3 <dbl>, TCl <dbl>, eta.Cl <dbl>,
#> #   TKA <dbl>, eta.Ka <dbl>, Q <dbl>, Kin <dbl>, Kout <dbl>, EC50 <dbl>
#> -- Initial Conditions ($inits): --
#> depot centr  peri   eff
#>     0     0     0     1
#>
#> Simulation with uncertainty in:
#> * parameters ($thetaMat for changes)
#> * omega matrix ($omegaList)
#>
#> -- First part of data (object): --
#> # A tibble: 976,000 x 21
#>   sim.id    id time inv.Cl  inv.Ka eye.Cl  eye.Ka   iov.Cl iov.Ka    C2     C3
#>    <int> <int>  [h]  <dbl>   <dbl>  <dbl>   <dbl>    <dbl>  <dbl> <dbl>  <dbl>
#> 1      1     1 0     0.251 -0.0741 0.0127 -0.156  -0.00963 0.0813  0     0
#> 2      1     1 0.1   0.251 -0.0741 0.122  -0.0143 -0.00963 0.0813 13.5   0.0159
#> 3      1     1  4    0.251 -0.0741 0.0127 -0.156  -0.00963 0.0813  2.36  3.80
#> 4      1     1  4.1  0.251 -0.0741 0.122  -0.0143 -0.00963 0.0813  8.16  3.84
#> 5      1     1  8    0.251 -0.0741 0.0127 -0.156  -0.00963 0.0813  0.392 4.26
#> 6      1     1  8.1  0.251 -0.0741 0.122  -0.0143 -0.00963 0.0813  0.728 4.25
#> # i 975,994 more rows
#> # i 10 more variables: CL <dbl>, KA <dbl>, ef0 <dbl>, depot <dbl>, centr <dbl>,
#> #   peri <dbl>, eff <dbl>, occ <fct>, eye <fct>, inv <fct>
```

There are multiple investigators in a study; Each investigator has a number of individuals enrolled at their site. `rxode2` automatically determines the number of investigators and then will simulate an effect for each investigator. With the output, `inv.Cl(inv==1)` will be the `inv.Cl` for investigator 1, `inv.Cl(inv==2)` will be the `inv.Cl` for investigator 2, etc.

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

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

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

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

```
#>   sim.id id inv.Cl(inv==1) inv.Cl(inv==2) inv.Ka(inv==1) inv.Ka(inv==2)
#> 1      1  1      0.2511322     0.07650072    -0.07407926     0.02202168
#> 2      1  2      0.2511322     0.07650072    -0.07407926     0.02202168
#> 3      1  3      0.2511322     0.07650072    -0.07407926     0.02202168
#> 4      1  4      0.2511322     0.07650072    -0.07407926     0.02202168
#> 5      1  5      0.2511322     0.07650072    -0.07407926     0.02202168
#> 6      1  6      0.2511322     0.07650072    -0.07407926     0.02202168
#>   eye.Cl(eye==1) eye.Cl(eye==2) eye.Ka(eye==1) eye.Ka(eye==2) iov.Cl(occ==1)
#> 1     0.01270627     0.12170298    -0.15645931    -0.01431089   -0.009629471
#> 2    -0.17730664     0.03257883     0.32307021     0.02920050   -0.174327268
#> 3     0.11662407     0.23070953     0.14031942     0.60774555    0.108968491
#> 4     0.15094002    -0.06667588     0.14962819     0.06226514   -0.043479003
#> 5    -0.03450830     0.18531423    -0.52570577    -0.14201201   -0.062005657
#> 6    -0.35488059     0.58910822    -0.05440263     0.01390173   -0.032395527
#>   iov.Cl(occ==2) iov.Ka(occ==1) iov.Ka(occ==2)       V2       V3      TCl
#> 1     0.01053195    0.081266104    -0.07738340 39.89608 296.4477 18.66578
#> 2     0.14111236    0.012959501     0.14003903 39.89608 296.4477 18.66578
#> 3    -0.08081801    0.064904438    -0.05309035 39.89608 296.4477 18.66578
#> 4    -0.02781613    0.021622026    -0.17969560 39.89608 296.4477 18.66578
#> 5     0.14666663   -0.032748073    -0.02417555 39.89608 296.4477 18.66578
#> 6    -0.02895152    0.002390472     0.05844429 39.89608 296.4477 18.66578
#>       eta.Cl      TKA      eta.Ka         Q        Kin      Kout     EC50
#> 1  0.6985427 0.346799  0.42017137 9.659552 0.03348837 0.9102127 200.427
#> 2 -0.1405521 0.346799 -0.17769437 9.659552 0.03348837 0.9102127 200.427
#> 3  0.2575912 0.346799  0.22284272 9.659552 0.03348837 0.9102127 200.427
#> 4  0.4722423 0.346799  0.49685712 9.659552 0.03348837 0.9102127 200.427
#> 5 -0.1505078 0.346799 -0.43312621 9.659552 0.03348837 0.9102127 200.427
#> 6 -0.1271020 0.346799  0.08195574 9.659552 0.03348837 0.9102127 200.427
```

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

```
#>   sim.id id inv.Cl(inv==1) inv.Cl(inv==2) inv.Ka(inv==1) inv.Ka(inv==2)
#> 1      2  1      0.1093052     -0.0673165      0.1186236    -0.08942504
#> 2      2  2      0.1093052     -0.0673165      0.1186236    -0.08942504
#> 3      2  3      0.1093052     -0.0673165      0.1186236    -0.08942504
#> 4      2  4      0.1093052     -0.0673165      0.1186236    -0.08942504
#> 5      2  5      0.1093052     -0.0673165      0.1186236    -0.08942504
#> 6      2  6      0.1093052     -0.0673165      0.1186236    -0.08942504
#>   eye.Cl(eye==1) eye.Cl(eye==2) eye.Ka(eye==1) eye.Ka(eye==2) iov.Cl(occ==1)
#> 1     0.35152729    0.006594725    -0.06268297     0.29815635    -0.00881754
#> 2     0.10269092    0.146594486    -0.20490974     0.28538009    -0.07747854
#> 3    -0.63206970    0.346871624     0.06642099     0.35270304    -0.06741916
#> 4    -0.08415895    0.235107634     0.12953250    -0.17707732     0.06017794
#> 5     0.33726812    0.072669926    -0.05740940    -0.08718347    -0.09088558
#> 6    -0.26619889   -0.007725891    -0.18858394    -0.15278522     0.14865110
#>   iov.Cl(occ==2) iov.Ka(occ==1) iov.Ka(occ==2)       V2       V3      TCl
#> 1    0.015905350    -0.02096693    -0.12871823 39.95281 297.0105 18.48517
#> 2    0.021713327     0.10111395    -0.04911822 39.95281 297.0105 18.48517
#> 3   -0.027550757    -0.08063562     0.07982127 39.95281 297.0105 18.48517
#> 4   -0.019521504     0.06468350     0.08335715 39.95281 297.0105 18.48517
#> 5   -0.003415422    -0.03601159     0.04816064 39.95281 297.0105 18.48517
#> 6   -0.053526809    -0.16281362     0.01215932 39.95281 297.0105 18.48517
#>        eta.Cl        TKA      eta.Ka        Q      Kin      Kout     EC50
#> 1 -0.35476449 -0.1722565  0.21079536 10.89209 1.256925 0.9366159 199.9913
#> 2 -0.17388248 -0.1722565  0.62943162 10.89209 1.256925 0.9366159 199.9913
#> 3  0.90160059 -0.1722565  0.08201732 10.89209 1.256925 0.9366159 199.9913
#> 4  0.10391092 -0.1722565  0.38871734 10.89209 1.256925 0.9366159 199.9913
#> 5  0.28247914 -0.1722565 -0.14271404 10.89209 1.256925 0.9366159 199.9913
#> 6 -0.02269591 -0.1722565  0.29263333 10.89209 1.256925 0.9366159 199.9913
```

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

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

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

12.6 Transit compartment models

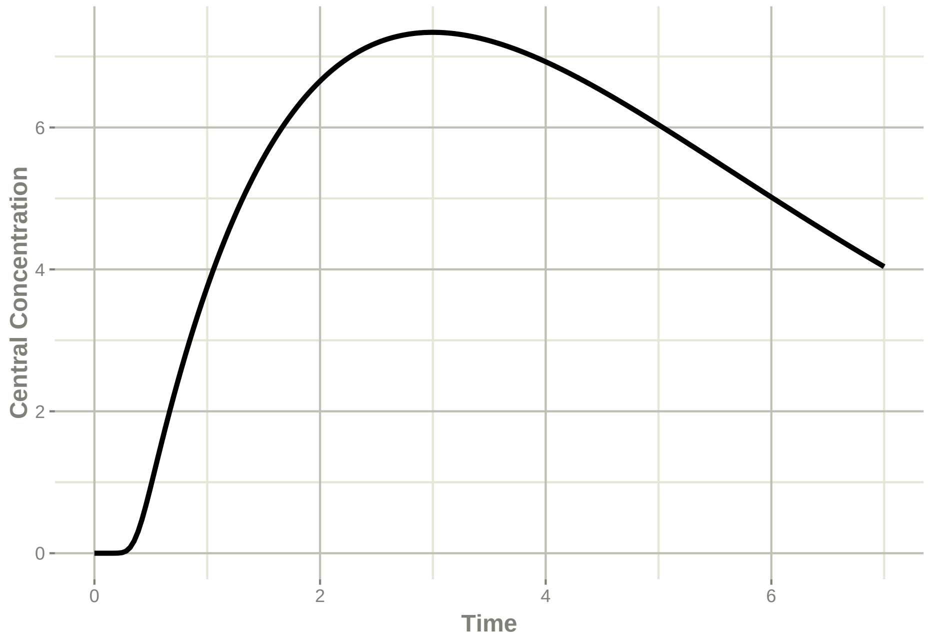
Savic 2008 first introduced the idea of transit compartments being a mechanistic explanation of a a lag-time type phenomena. rxode2 has special handling of these models:

You can specify this in a similar manner as the original paper:

```r
library(rxode2)
mod <- rxode2({
    ## Table 3 from Savic 2007
    cl = 17.2 # (L/hr)
    vc = 45.1 # L
    ka = 0.38 # 1/hr
    mtt = 0.37 # hr
    bio=1
    n = 20.1
    k = cl/vc
    ktr = (n+1)/mtt
    ## note that lgammafn is the same as lgamma in R.
    d/dt(depot) = exp(log(bio*podo(depot))+log(ktr)+n*log(ktr*tad(depot))-
                      ktr*tad(depot)-lgammafn(n+1))-ka*depot
    d/dt(cen) = ka*depot-k*cen
})
```

```
#> using C compiler: ‘gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0’
```

```r
et <- eventTable()
et$add.sampling(seq(0, 7, length.out=200))
et$add.dosing(20, start.time=0, evid=7)

transit <- rxSolve(mod, et)

plot(transit, cen, ylab="Central Concentration")
```



Another option is to specify the transit compartment function `transit` syntax. This specifies the parameters `transit(number of transit compartments, mean transit time, bioavailability)`. The bioavailability term is optional.

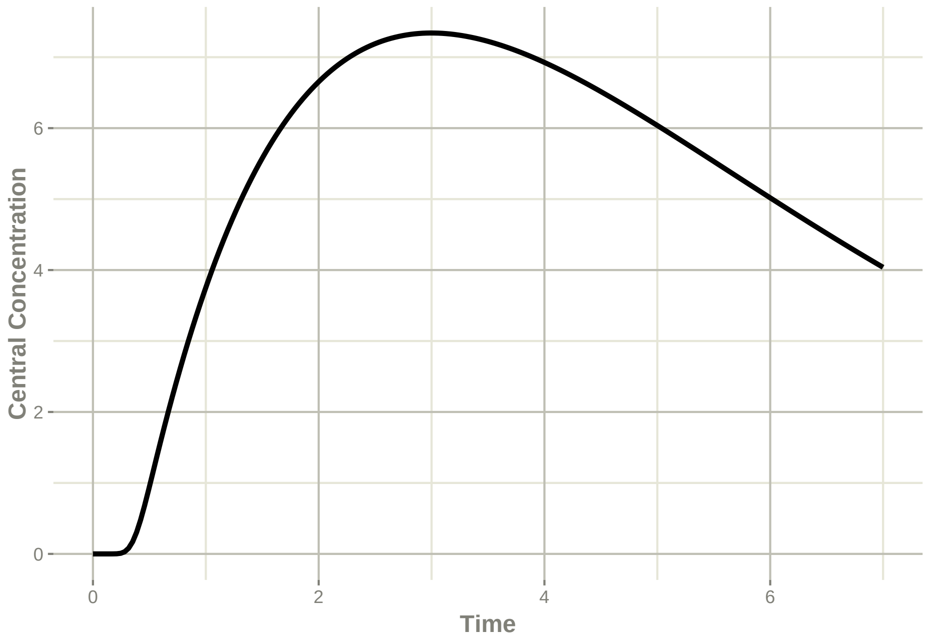
The same model can be specified by:

```
mod <- rxode2({
    ## Table 3 from Savic 2007
    cl = 17.2 # (L/hr)
    vc = 45.1 # L
    ka = 0.38 # 1/hr
    mtt = 0.37 # hr
    bio=1
    n = 20.1
    k = cl/vc
    ktr = (n+1)/mtt
    d/dt(depot) = transit(n,mtt,bio)-ka*depot
    d/dt(cen) = ka*depot-k*cen
})
```

```
#> using C compiler: ‘gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0’
```

```
et <- eventTable();
et$add.sampling(seq(0, 7, length.out=200));
et$add.dosing(20, start.time=0, evid=7);

transit <- rxSolve(mod, et)
```

```r
plot(transit, cen, ylab="Central Concentration")
```



A couple of things to keep in mind when using this approach:

- This approach implicitly assumes that the absorption through the transit compartment is completed before the next dose begins
- Different types of doses (ie bolus/infusion) to the compartment affect the time after dose calculation (`tad`) which is used in the transit compartment calculation. These (therefore) are not currently supported. The most stable way is to use `tad(cmt)` and `podo(cmt)`, this way doses to other compartments do not affect the transit compartment calculation.
- Internally, the `transit` syntax uses either the currently defined cmt `d/dt(cmt)=transit(...)`, or `cmt`. If the transit compartment is used outside of a `d/dt()` (not recommended), the `cmt` that is used is the last `d/dt(cmt)` defined it the model. This also means compartments do not affect one another (ie a oral, transit compartment drug dosed immediately with an IV infusion)

Chapter 13

Advanced & Miscellaneous Topics

This covers advanced or miscellaneous topics in `rxode2`

13.1 Covariates in rxode2

13.1.1 Individual Covariates

If there is an individual covariate you wish to solve for you may specify it by the `iCov` dataset:

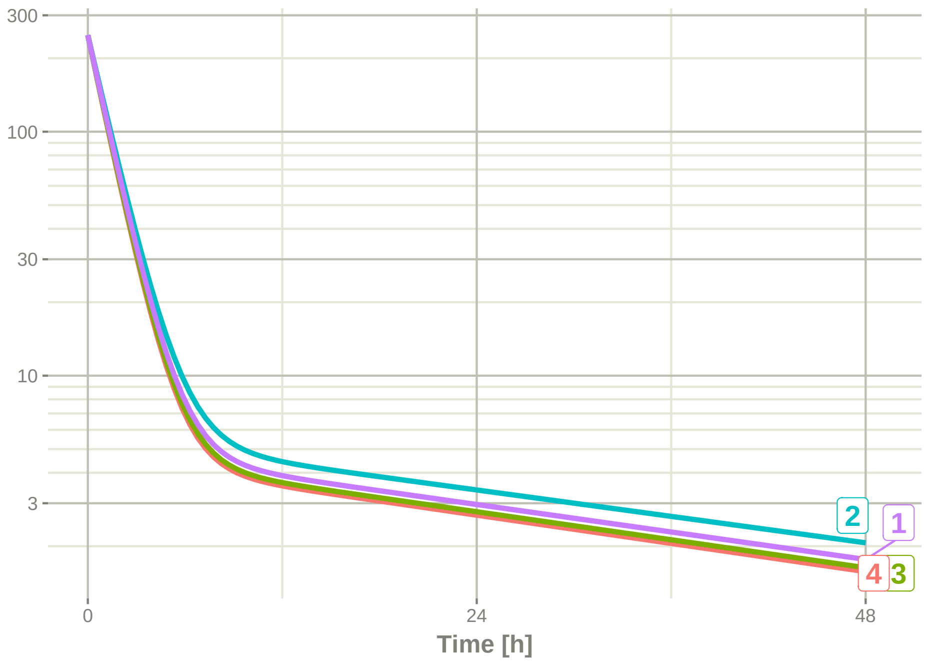
```
library(rxode2)
library(units)
library(xgxr)

mod3 <- rxode2({
    KA=2.94E-01;
#### Clearance with individuals
    CL=1.86E+01 * (WT / 70) ^ 0.75;
    V2=4.02E+01;
    Q=1.05E+01;
    V3=2.97E+02;
    Kin=1;
    Kout=1;
    EC50=200;
#### The linCmt() picks up the variables from above
    C2   = linCmt();
    Tz= 8
    amp=0.1
    eff(0) = 1  ## This specifies that the effect compartment starts at 1.
    d/dt(eff) =  Kin - Kout*(1-C2/(EC50+C2))*eff;
})
```

```
#> using C compiler: ‘gcc (Ubuntu 11.3.0-1ubuntu1~22.04.1) 11.3.0’
```

```
ev <- et(amount.units="mg", time.units="hours") %>%
    et(amt=10000, cmt=1) %>%
    et(0,48,length.out=100) %>%
    et(id=1:4);

set.seed(10)
rxSetSeed(10)
#### Now use iCov to simulate a 4-id sample
r1 <- solve(mod3, ev,
### Create individual covariate data-frame
            iCov=data.frame(id=1:4, WT=rnorm(4, 70, 10)))
print(r1)
```

```
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#>      KA      V2       Q      V3     Kin    Kout    EC50      Tz     amp
#>   0.294  40.200  10.500 297.000   1.000   1.000 200.000   8.000   0.100
#> -- Initial Conditions ($inits): --
#> eff
#>   1
#> -- First part of data (object): --
#> # A tibble: 400 x 6
#>      id  time    CL    C2   eff    WT
#>   <int>   [h] <dbl> <dbl> <dbl> <dbl>
#> 1     1 0      18.6 249.      1  70.2
#> 2     1 0.485  18.6 175.      1  70.2
#> 3     1 0.970  18.6 124.      1  70.2
#> 4     1 1.45   18.6  87.9     1  70.2
#> 5     1 1.94   18.6  62.7     1  70.2
#> 6     1 2.42   18.6  45.1     1  70.2
#> # i 394 more rows
```

```
plot(r1, C2, log="y")
```



13.1.2 Time Varying Covariates

Covariates are easy to specify in rxode2, you can specify them as a variable. Time-varying covariates, like clock time in a circadian rhythm model, can also be used. Extending the indirect response model already discussed, we have:

```
library(rxode2)
library(units)

mod3 <- rxode2({
    KA=2.94E-01;
    CL=1.86E+01;
    V2=4.02E+01;
    Q=1.05E+01;
    V3=2.97E+02;
    Kin0=1;
    Kout=1;
    EC50=200;
#### The linCmt() picks up the variables from above
    C2   = linCmt();
    Tz= 8
    amp=0.1
    eff(0) = 1  ## This specifies that the effect compartment starts at 1.
#### Kin changes based on time of day (like cortosol)
```

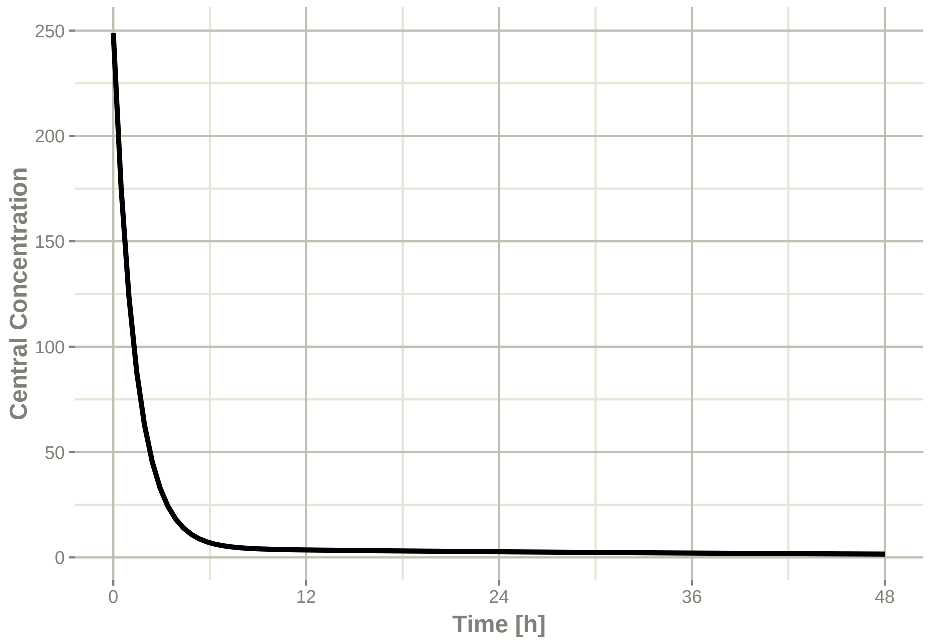
```
    Kin =   Kin0 +amp *cos(2*pi*(ctime-Tz)/24)
    d/dt(eff) =  Kin - Kout*(1-C2/(EC50+C2))*eff;
})


ev <- eventTable(amount.units="mg", time.units="hours") %>%
    add.dosing(dose=10000, nbr.doses=1, dosing.to=1) %>%
    add.sampling(seq(0,48,length.out=100));


#### Create data frame of 8 am dosing for the first dose This is done
#### with base R but it can be done with dplyr or data.table
ev$ctime <- (ev$time+set_units(8,hr)) %% 24
```

Now there is a covariate present in the event dataset, the system can be solved by combining the dataset and the model:

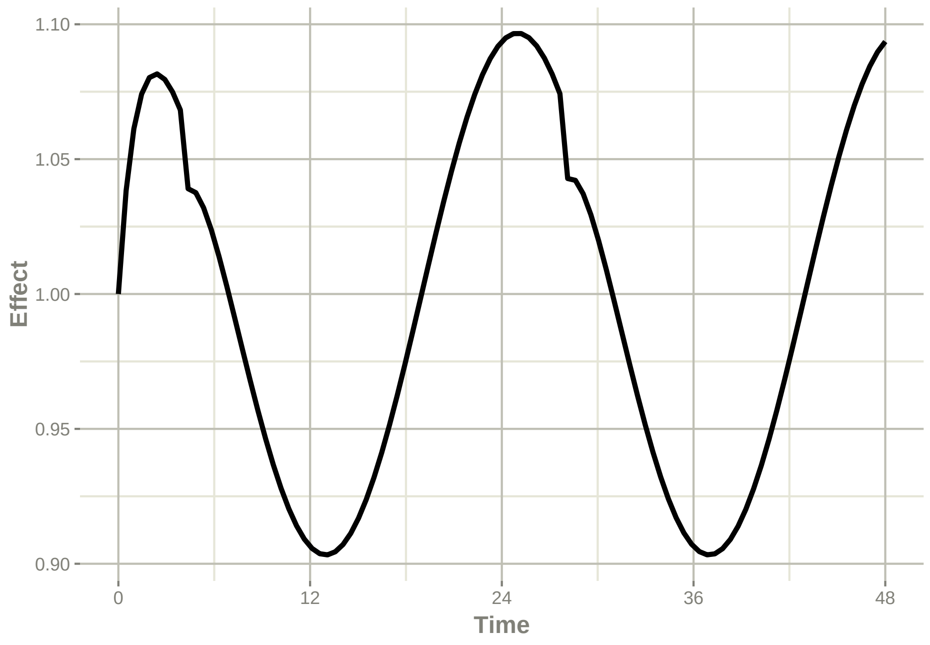
```
r1 <- solve(mod3, ev, covsInterpolation="linear")
print(r1)
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#>         KA         CL         V2          Q         V3       Kin0       Kout
#>   0.294000  18.600000  40.200000  10.500000 297.000000   1.000000   1.000000
#>       EC50         Tz        amp         pi
#> 200.000000   8.000000   0.100000   3.141593
#> -- Initial Conditions ($inits): --
#> eff
#>   1
#> -- First part of data (object): --
#> # A tibble: 100 x 5
#>    time    C2   Kin   eff ctime
#>     [h] <dbl> <dbl> <dbl>   [h]
#> 1 0     249.   1.1   1      8
#> 2 0.485 175.   1.10  1.04  8.48
#> 3 0.970 124.   1.10  1.06  8.97
#> 4 1.45   88.0  1.09  1.07  9.45
#> 5 1.94   62.9  1.09  1.08  9.94
#> 6 2.42   45.2  1.08  1.08 10.4
#> # i 94 more rows
```

When solving ODE equations, the solver may sample times outside of the data. When this happens, this ODE solver can use linear interpolation between the covariate values. It is equivalent to R's `approxfun` with `method="linear"`.

```
plot(r1,C2, ylab="Central Concentration")
```



```
plot(r1,eff) + ylab("Effect") + xlab("Time")
```



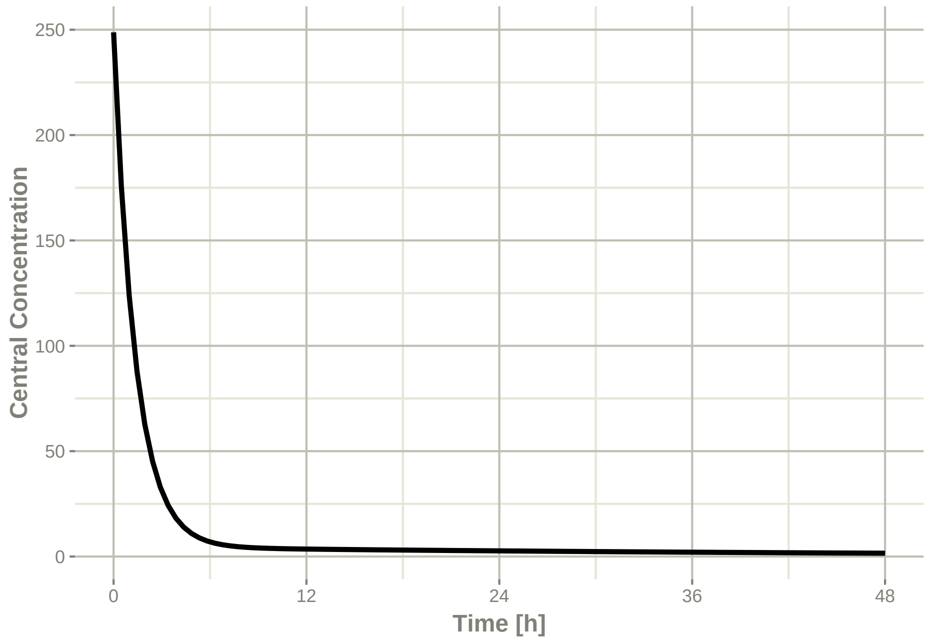
Note that the linear approximation in this case leads to some kinks in the solved system at 24-hours where the covariate has a linear interpolation between near 24 and near 0. While linear seems reasonable, cases like clock time make other interpolation methods more attractive.

In rxode2 the default covariate interpolation is be the last observation carried forward (`locf`), or constant approximation. This is equivalent to R's `approxfun` with `method="constant"`.

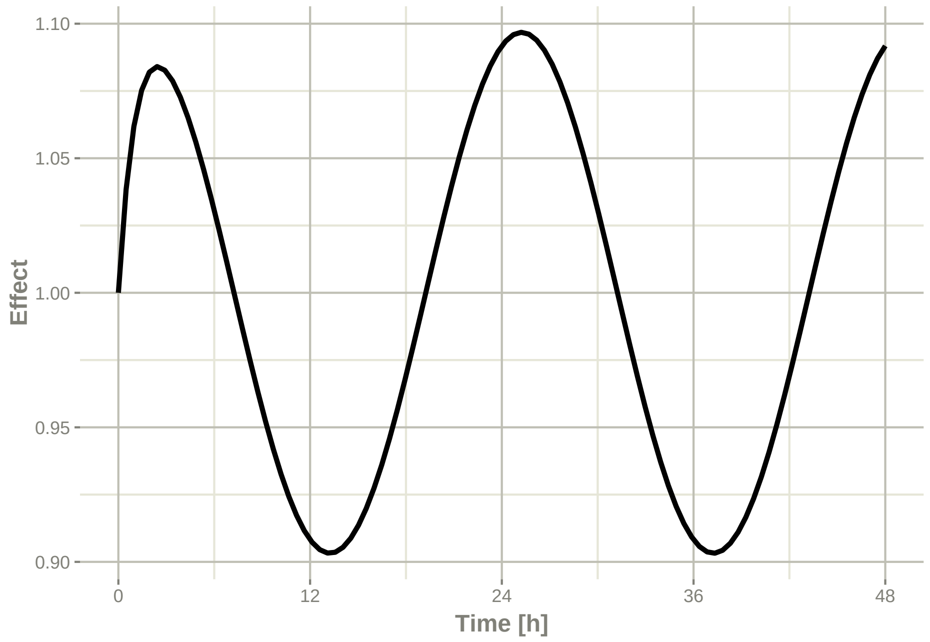
```
r1 <- solve(mod3, ev,covsInterpolation="locf")
print(r1)
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#>         KA         CL         V2          Q         V3       Kin0       Kout
#>   0.294000  18.600000  40.200000  10.500000 297.000000   1.000000   1.000000
#>       EC50         Tz        amp         pi
#> 200.000000   8.000000   0.100000   3.141593
#> -- Initial Conditions ($inits): --
#> eff
#>   1
#> -- First part of data (object): --
#> # A tibble: 100 x 5
#>    time    C2   Kin   eff ctime
#>     [h] <dbl> <dbl> <dbl>   [h]
#> 1 0     249.   1.1   1     8
#> 2 0.485 175.   1.10  1.04  8.48
#> 3 0.970 124.   1.10  1.06  8.97
#> 4 1.45   88.0  1.09  1.08  9.45
#> 5 1.94   62.9  1.09  1.08  9.94
#> 6 2.42   45.2  1.08  1.08 10.4
#> # i 94 more rows
```

which gives the following plots:

```
plot(r1,C2, ylab="Central Concentration", xlab="Time")
```



```
plot(r1,eff, ylab="Effect", xlab="Time")
```



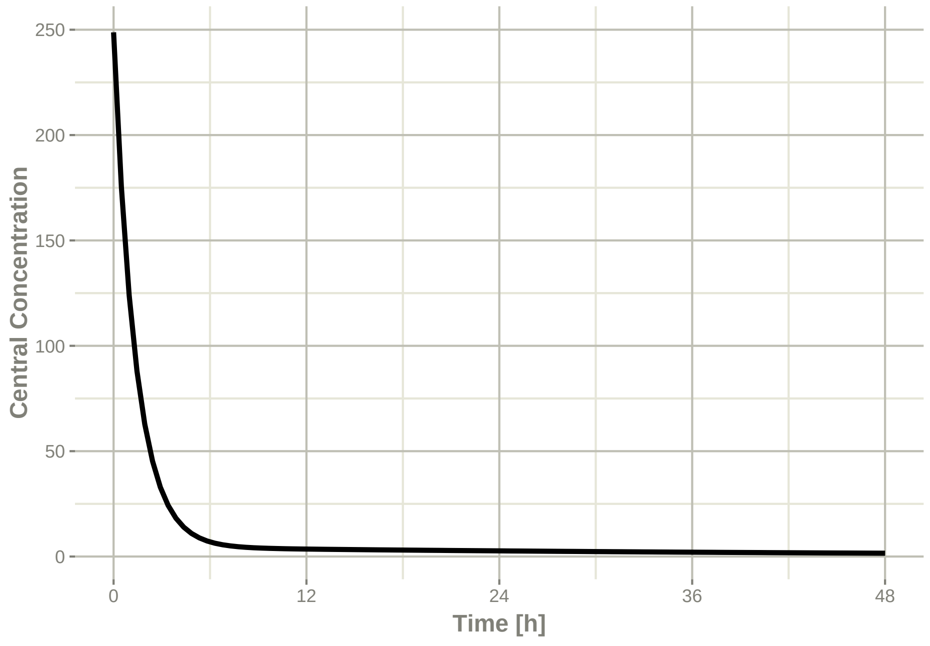
In this case, the plots seem to be smoother.

You can also use NONMEM's preferred interpolation style of next observation carried backward (NOCB):

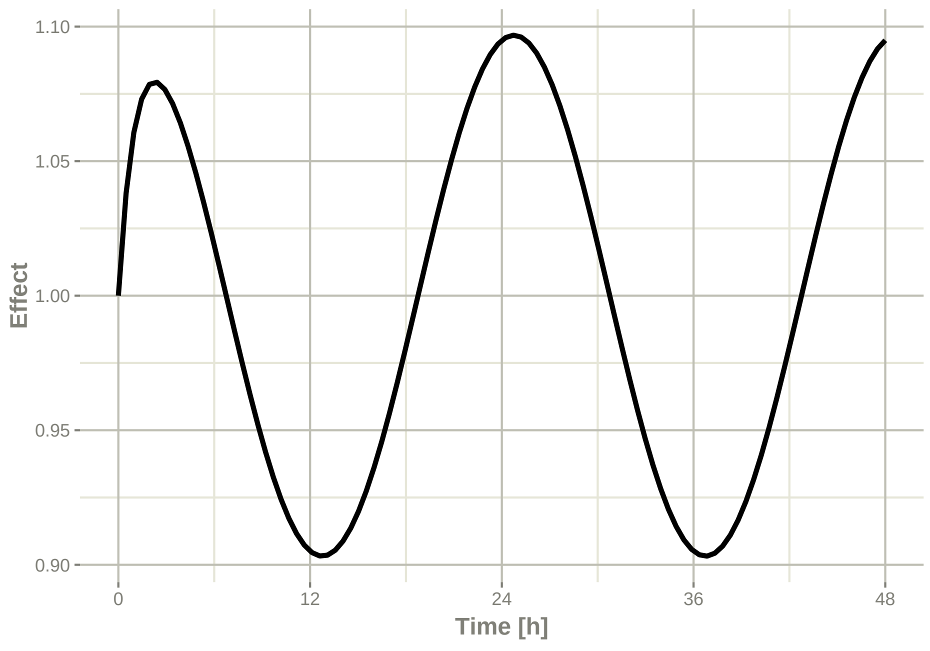
```
r1 <- solve(mod3, ev,covsInterpolation="nocb")
print(r1)
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#>          KA          CL          V2           Q          V3        Kin0        Kout
#>    0.294000   18.600000   40.200000   10.500000 297.000000    1.000000    1.000000
#>        EC50          Tz         amp          pi
#> 200.000000    8.000000    0.100000    3.141593
#> -- Initial Conditions ($inits): --
#> eff
#>   1
#> -- First part of data (object): --
#> # A tibble: 100 x 5
#>    time    C2   Kin   eff ctime
#>     [h] <dbl> <dbl> <dbl>   [h]
#> 1 0     249.   1.1   1     8
#> 2 0.485 175.   1.10  1.04  8.48
#> 3 0.970 124.   1.10  1.06  8.97
#> 4 1.45   88.0  1.09  1.07  9.45
#> 5 1.94   62.9  1.09  1.08  9.94
#> 6 2.42   45.2  1.08  1.08 10.4
#> # i 94 more rows
```

which gives the following plots:

```
plot(r1,C2, ylab="Central Concentration", xlab="Time")
```



```r
plot(r1,eff, ylab="Effect", xlab="Time")
```



13.2 Shiny and rxode2

13.2.1 Facilities for generating R shiny applications

An example of creating an R shiny application to interactively explore responses of various complex dosing regimens is available at http://qsp.engr.uga.edu:3838/rxode2/RegimenSimulator. Shiny applications like this one may be programmatically created with the experimental function `genShinyApp.template()`.

The above application includes widgets for varying the dose, dosing regimen, dose cycle, and number of cycles.

```
genShinyApp.template(appDir = "shinyExample", verbose=TRUE)

library(shiny)
runApp("shinyExample")
```

Click here to go to the Shiny App

13.2.2 Exploring parameter fits graphically using shiny

An rxode2 object can be explored with `rxShiny(obj)`. `rxShiny()` will also allow you to try new models to see how they behave.

13.3 Using rxode2 with a pipeline

13.3.1 Setting up the rxode2 model for the pipeline

In this example we will show how to use rxode2 in a simple pipeline.

We can start with a model that can be used for the different simulation workflows that rxode2 can handle:

```
library(rxode2)

Ribba2012 <- rxode2({
    k = 100

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

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

    tkqpp = 0.0031
    eta.kqpp = 0
    kqpp ~ tkqpp * exp(eta.kqpp)

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

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

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

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

This is a tumor growth model described in Ribba 2012. In this case, we compiled the model into an R object `Ribba2012`, though in an rxode2 simulation pipeline, you do not *have* to assign the compiled model to any object, though I think it makes sense.

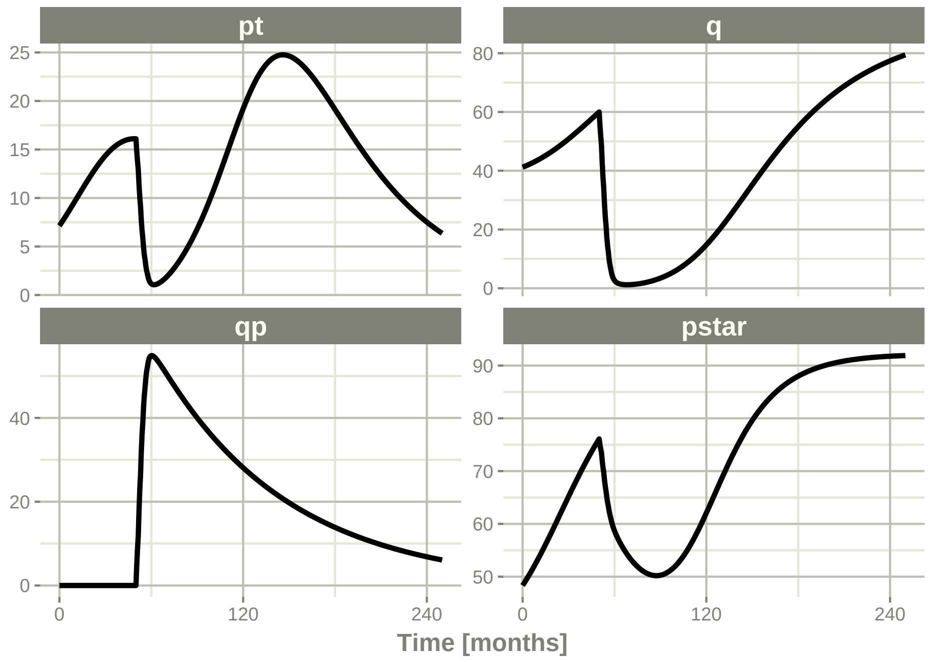
13.3.2 Simulating one event table

Simulating a single event table is quite simple:

- You pipe the rxode2 simulation object into an event table object by `et()`.
- When the events are completely specified, you simply solve the ODE system with `rxSolve()`.
- In this case you can pipe the output to `plot()` to conveniently view the results.

- Note for the plot we are only selecting the selecting following:
  - pt (Proliferative Tissue),
  - q (quiescent tissue)
  - qp (DNA-Damaged quiescent tissue) and
  - pstar (total tumor tissue)

```
Ribba2012 %>% # Use rxode2
    et(time.units="months") %>% # Pipe to a new event table
    et(amt=1, time=50, until=58, ii=1.5) %>% # Add dosing every 1.5 months
    et(0, 250, by=0.5) %>% # Add some sampling times (not required)
    rxSolve() %>% # Solve the simulation
    plot(pt, q, qp, pstar) # Plot it, plotting the variables of interest
```



13.3.3 Simulating multiple subjects from a single event table

13.3.3.1 Simulating with between subject variability

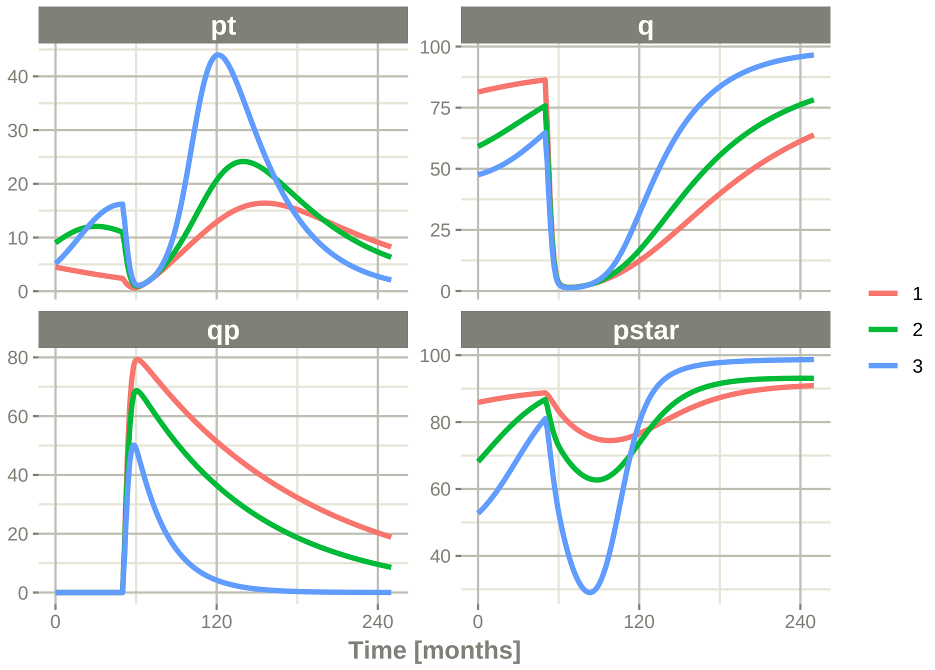
The next sort of simulation that may be useful is simulating multiple patients with the same treatments. In this case, we will use the omega matrix specified by the paper:

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

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

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

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

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

```r
set.seed(1089)
rxSetSeed(1089)
Ribba2012 %>% # Use rxode2
    et(time.units="months") %>% # Pipe to a new event table
    et(amt=1, time=50, until=58, ii=1.5) %>% # Add dosing every 1.5 months
    et(0, 250, by=0.5) %>% # Add some sampling times (not required)
    rxSolve(nSub=3, omega=omega) %>% # Solve the simulation
    plot(pt, q, qp, pstar) # Plot it, plotting the variables of interest
```

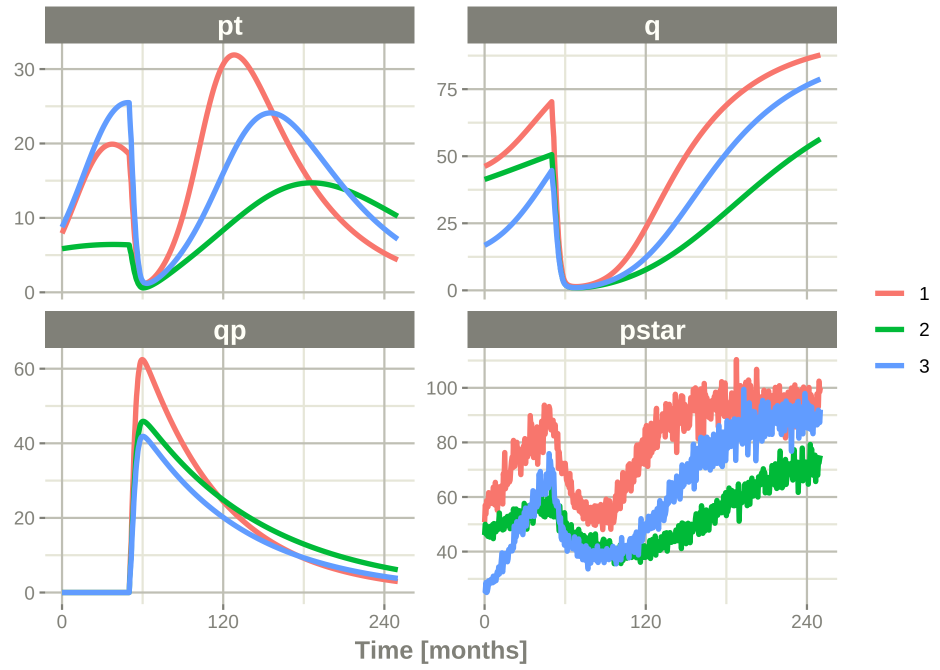


Note there are two different things that were added to this simulation: - `nSub` to specify how many subjects are in the model - `omega` to specify the between subject variability.

13.3.3.2 Simulation with unexplained variability

You can even add unexplained variability quite easily:

```
Ribba2012 %>% # Use rxode2
    et(time.units="months") %>% # Pipe to a new event table
    et(amt=1, time=50, until=58, ii=1.5) %>% # Add dosing every 1.5 months
     et(0, 250, by=0.5) %>% # Add some sampling times (not required)
    rxSolve(nSub=3, omega=omega, sigma=lotri(prop.err ~ 0.05^2)) %>% # Solve the simul
    plot(pt, q, qp, pstar) # Plot it, plotting the variables of interest
```



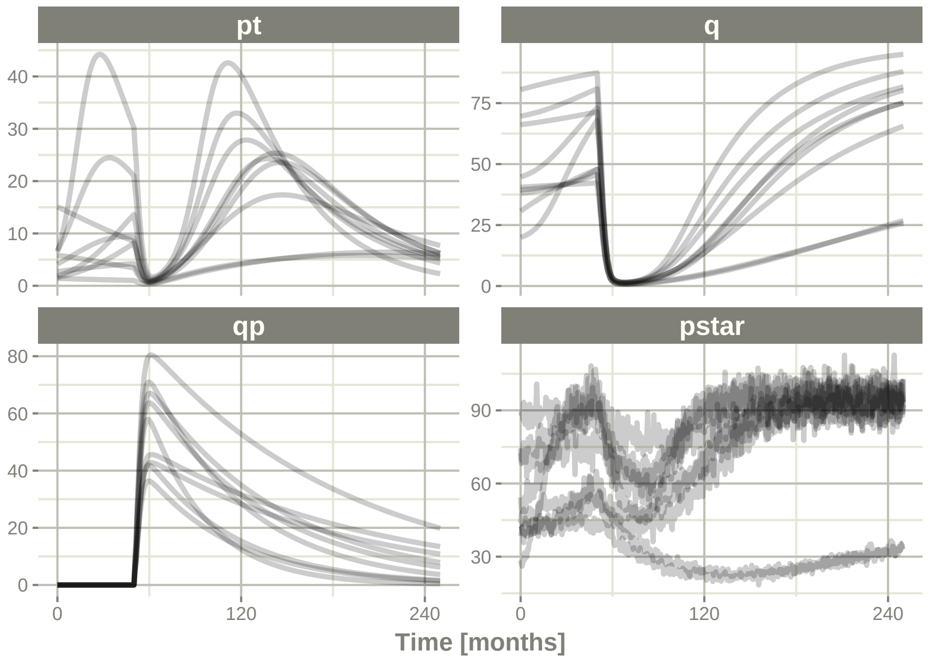
In this case we only added the `sigma` matrix to have unexplained variability on the `pstar` or total tumor tissue.

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

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

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

```
Ribba2012 %>% # Use rxode2
    et(time.units="months") %>% # Pipe to a new event table
    et(amt=1, time=50, until=58, ii=1.5) %>% # Add dosing every 1.5 months
    et(0, 250, by=0.5) %>% # Add some sampling times (not required)
    rxSolve(nSub=3, nStud=3, omega=omega, sigma=lotri(prop.err ~ 0.05^2),
            dfSub=760, dfObs=95) %>% # Solve the simulation
    plot(pt, q, qp, pstar) # Plot it, plotting the variables of interest
```



Often in simulations we have a full covariance matrix for the fixed effect parameters. In this case, we do not have the matrix, but it could be specified by `thetaMat`.

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

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

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

thetaMat
#>              tpt0   tq0 tlambdap    tkqpp  tdeltaqp  tgamma   tkde
#> tpt0       1.7825 0.000  0.00000 0.000000 0.0000000 0.00000 0.0000
#> tq0        0.0000 2.884  0.00000 0.000000 0.0000000 0.00000 0.0000
#> tlambdap   0.0000 0.000  0.01936 0.000000 0.0000000 0.00000 0.0000
#> tkqpp    0.0000 0.000  0.00000 0.001085 0.0000000 0.00000 0.0000
#> tdeltaqp 0.0000 0.000  0.00000 0.000000 0.0018207 0.00000 0.0000
#> tgamma   0.0000 0.000  0.00000 0.000000 0.0000000 0.26973 0.0000
#> tkde     0.0000 0.000  0.00000 0.000000 0.0000000 0.00000 0.0792
```

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

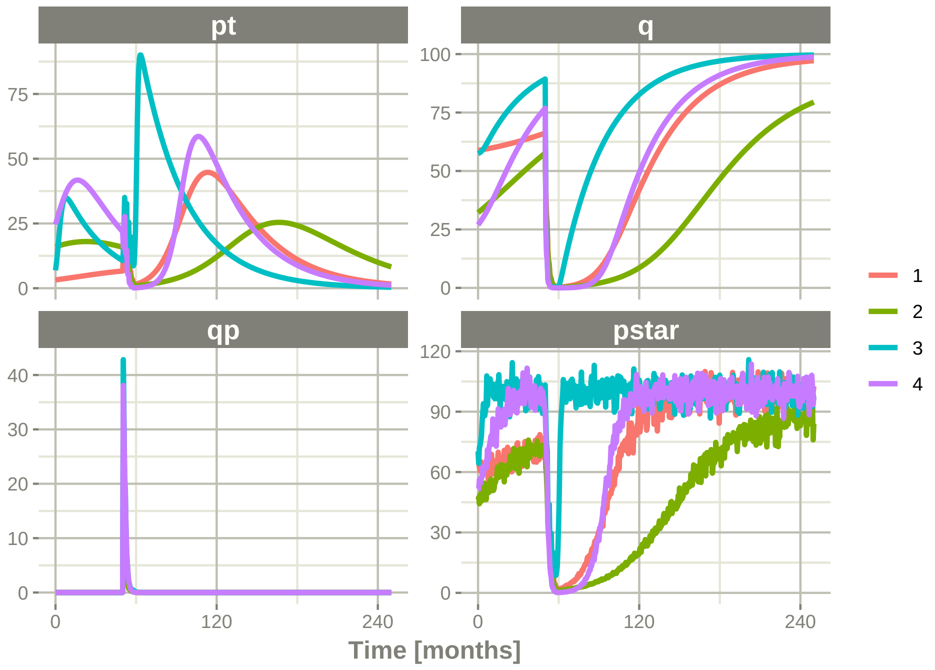
With such large variability in `theta` it is easy to sample a negative rate constant, which does not make sense. For example:

```
Ribba2012 %>% # Use rxode2
et(time.units="months") %>% # Pipe to a new event table
et(amt=1, time=50, until=58, ii=1.5) %>% # Add dosing every 1.5 months
et(0, 250, by=0.5) %>% # Add some sampling times (not required)
rxSolve(nSub=2, nStud=2, omega=omega, sigma=lotri(prop.err ~ 0.05^2),
thetaMat=thetaMat,
dfSub=760, dfObs=95) %>% # Solve the simulation
plot(pt, q, qp, pstar) # Plot it, plotting the variables of interest

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

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

In this case the simulation simply has to be modified to have `thetaLower=0` to make sure all rates are positive:

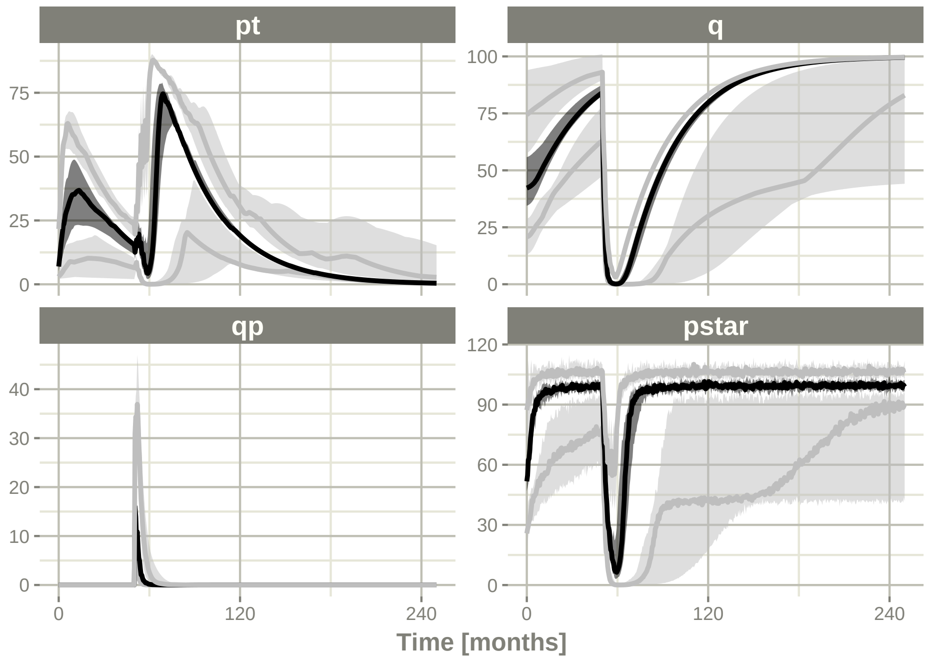
```
Ribba2012 %>% # Use rxode2
    et(time.units="months") %>% # Pipe to a new event table
    et(amt=1, time=50, until=58, ii=1.5) %>% # Add dosing every 1.5 months
    et(0, 250, by=0.5) %>% # Add some sampling times (not required)
    rxSolve(nSub=2, nStud=2, omega=omega, sigma=lotri(prop.err ~ 0.05^2),
            thetaMat=thetaMat,
            thetaLower=0, # Make sure the rates are reasonable
            dfSub=760, dfObs=95) %>% # Solve the simulation
    plot(pt, q, qp, pstar) # Plot it, plotting the variables of interest
```



13.3.4 Summarizing the simulation output

While it is easy to use `dplyr` and `data.table` to perform your own summary of simulations, `rxode2` also provides this ability by the `confint` function.

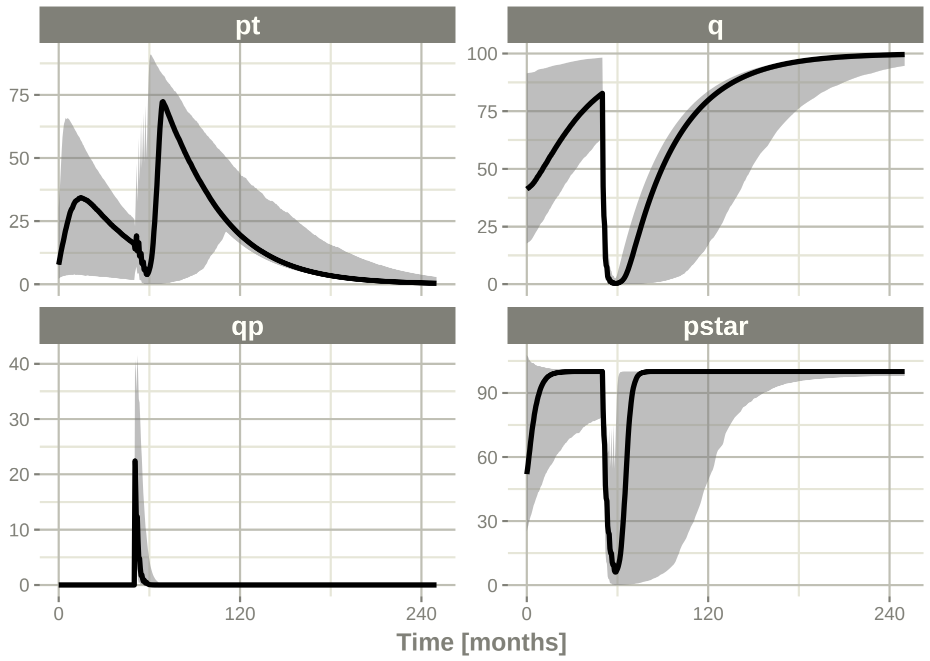
```
#### This takes a little more time; Most of the time is the summary
#### time.

sim0 <- Ribba2012 %>% # Use rxode2
    et(time.units="months") %>% # Pipe to a new event table
    et(amt=1, time=50, until=58, ii=1.5) %>% # Add dosing every 1.5 months
    et(0, 250, by=0.5) %>% # Add some sampling times (not required)
    rxSolve(nSub=10, nStud=10, omega=omega, sigma=lotri(prop.err ~ 0.05^2),
            thetaMat=thetaMat,
            thetaLower=0, # Make sure the rates are reasonable
            dfSub=760, dfObs=95) %>% # Solve the simulation
    confint(c("pt","q","qp","pstar"),level=0.90); # Create Simulation intervals

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



13.3.4.1 Simulating from a data-frame of parameters

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

```
library(dplyr)
pars <- rxInits(Ribba2012);
pars <- pars[regexpr("(prop|eta)",names(pars)) == -1]
print(pars)
#>        k     tkde     tkpq    tkqpp tlambdap   tgamma tdeltaqp     tpt0
#> 1.00e+02 2.40e-01 2.95e-02 3.10e-03 1.21e-01 7.29e-01 8.67e-03 7.13e+00
#>      tq0
#> 4.12e+01
#### This is the exported method for simulation of Theta/Omega internally in rxode2
df <- rxSimThetaOmega(params=pars, omega=omega,dfSub=760,
                      thetaMat=thetaMat, thetaLower=0, nSub=60,nStud=60) %>%
    filter(tkde > tlambdap) %>% as.tbl()
#### You could also simulate more and bind them together to a data frame.
print(df)
#> # A tibble: 1,860 x 16
#>        k  tkde   tkpq tkqpp tlambdap tgamma tdeltaqp  tpt0   tq0 eta.pt0  eta.q0
#>    <dbl> <dbl>  <dbl> <dbl>    <dbl>  <dbl>    <dbl> <dbl> <dbl>   <dbl>   <dbl>
#>  1   100  1.50 0.0295 0.404     1.25    1.16    0.318  7.21  41.5 -0.911  -0.0212
#>  2   100  1.50 0.0295 0.404     1.25    1.16    0.318  7.21  41.5 -0.275   0.537
#>  3   100  1.50 0.0295 0.404     1.25    1.16    0.318  7.21  41.5  0.213  -0.408
#>  4   100  1.50 0.0295 0.404     1.25    1.16    0.318  7.21  41.5 -0.0557  0.0625
#>  5   100  1.50 0.0295 0.404     1.25    1.16    0.318  7.21  41.5 -0.239   0.609
#>  6   100  1.50 0.0295 0.404     1.25    1.16    0.318  7.21  41.5  0.555  -0.173
#>  7   100  1.50 0.0295 0.404     1.25    1.16    0.318  7.21  41.5  0.418   0.929
#>  8   100  1.50 0.0295 0.404     1.25    1.16    0.318  7.21  41.5  0.329  -0.348
#>  9   100  1.50 0.0295 0.404     1.25    1.16    0.318  7.21  41.5 -0.0251  0.920
#> 10   100  1.50 0.0295 0.404     1.25    1.16    0.318  7.21  41.5 -0.106  -0.405
#> # i 1,850 more rows
#> # i 5 more variables: eta.lambdap <dbl>, eta.kqp <dbl>, eta.qpp <dbl>,
#> #   eta.deltaqp <dbl>, eta.kde <dbl>
#### Quick check to make sure that all the parameters are OK.
all(df$tkde>df$tlambdap)
#> [1] TRUE
sim1 <- Ribba2012 %>% # Use rxode2
    et(time.units="months") %>% # Pipe to a new event table
    et(amt=1, time=50, until=58, ii=1.5) %>% # Add dosing every 1.5 months
    et(0, 250, by=0.5) %>% # Add some sampling times (not required)
    rxSolve(df)
#### Note this information looses information about which ID is in a
#### "study", so it summarizes the confidence intervals by dividing the
#### subjects into sqrt(#subjects) subjects and then summarizes the
#### confidence intervals
sim2 <- sim1 %>% confint(c("pt","q","qp","pstar"),level=0.90); # Create Simulation inte
save(sim2, file = file.path(system.file(package = "rxode2"), "pipeline-sim2.rds"), vers
sim2 %>% plot()
```



13.4 Speeding up rxode2

13.4.1 Increasing rxode2 speed by multi-subject parallel solving

`rxode2` originally developed as an ODE solver that allowed an ODE solve for a single subject. This flexibility is still supported.

The original code from the `rxode2` tutorial is below:

```
library(rxode2)

library(microbenchmark)
library(ggplot2)

mod1 <- rxode2({
    C2 = centr/V2;
    C3 = peri/V3;
    d/dt(depot) = -KA*depot;
    d/dt(centr) = KA*depot - CL*C2 - Q*C2 + Q*C3;
    d/dt(peri) = Q*C2 - Q*C3;
    d/dt(eff) = Kin - Kout*(1-C2/(EC50+C2))*eff;
    eff(0) = 1
})
```

```r
#### Create an event table

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

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

13.4.1.1 For Loop

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

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

13.4.1.2 Running with apply

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

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

13.4.1.3 Run using a single-threaded solve

You can also have rxode2 solve all the subject simultaneously without collecting the results in R, using a single threaded solve.

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

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

13.4.1.4 Run a 2 threaded solve

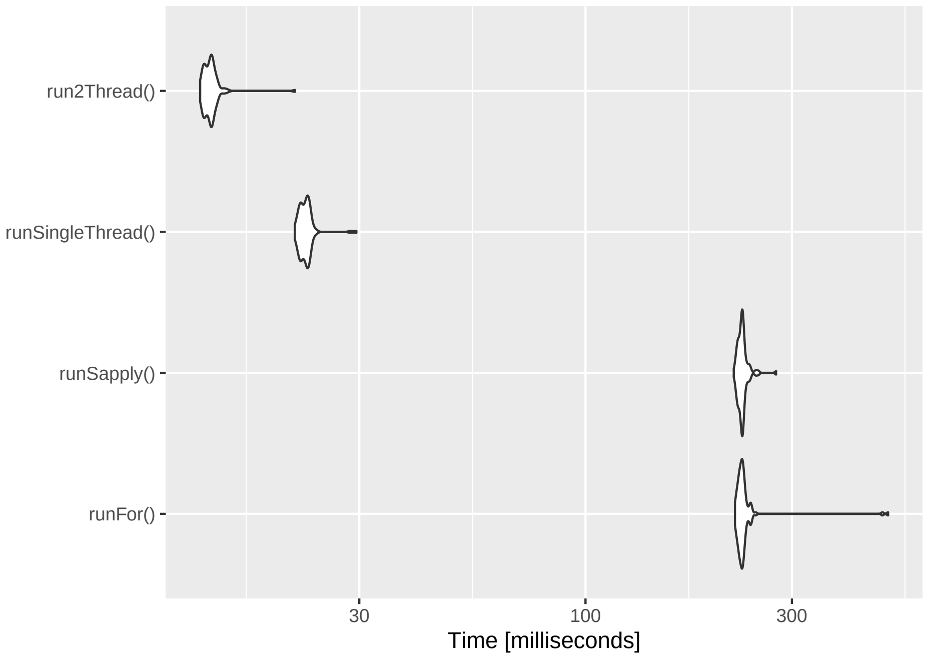
rxode2 supports multi-threaded solves, so another option is to have 2 threads (called `cores` in the solve options, you can see the options in `rxControl()` or `rxSolve()`).

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

13.4.1.5 Compare the times between all the methods

Now the moment of truth, the timings:

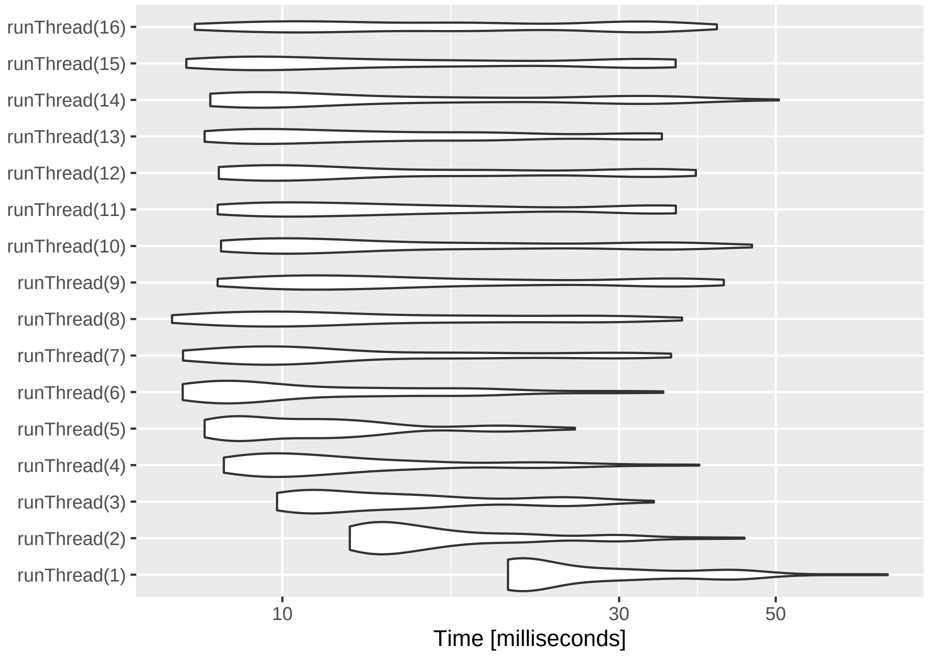
```r
bench <- microbenchmark(runFor(), runSapply(), runSingleThread(),run2Thread())
print(bench)
#> Unit: milliseconds
#>                expr       min        lq      mean    median        uq       max
#>            runFor() 221.55920 226.80759 235.36425 229.97514 232.67346 500.61138
#>         runSapply() 220.16413 226.41573 230.54824 229.85654 231.85143 275.84387
#>   runSingleThread()  21.28809  21.93480  22.56443  22.53847  22.90051  29.54173
#>        run2Thread()  12.85998  13.17329  13.61592  13.58354  13.75136  21.30739
#>  neval
#>    100
#>    100
#>    100
#>    100
autoplot(bench)
```



It is clear that the **largest** jump in performance when using the `solve` method and providing *all* the parameters to rxode2 to solve without looping over each subject with either a `for` or a `sapply`. The number of cores/threads applied to the solve also plays a role in the solving.

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

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

bench <- eval(parse(text=sprintf("microbenchmark(%s)",
                                        paste(paste0("runThread(", seq(1, 2 * rxCores()),
                                              collapse=","))))
print(bench)
#> Unit: milliseconds
#>           expr       min        lq     mean    median       uq      max neval
#>   runThread(1) 20.876151 21.368307 28.42933 23.511336 33.17421 72.05305   100
#>   runThread(2) 12.462495 13.463424 17.76470 15.152448 19.66082 45.15075   100
#>   runThread(3)  9.830081 10.856443 15.34307 13.055271 17.63875 33.60111   100
#>   runThread(4)  8.262952  9.284399 13.32124 11.043230 15.26146 38.96675   100
#>   runThread(5)  7.761468  8.661742 11.68702 10.700278 12.85569 25.97556   100
#>   runThread(6)  7.225629  8.096959 12.12702  9.579379 13.92138 34.64411   100
#>   runThread(7)  7.232503  9.118056 14.35233 10.491250 17.50675 35.54005   100
#>   runThread(8)  6.977873  9.130790 15.07035 11.675910 19.32375 36.83287   100
#>    runThread(9)  8.097309 10.591659 18.68855 14.054417 24.71292 42.20888   100
#>   runThread(10)  8.187620  9.806922 17.61390 12.407843 22.47038 46.27735   100
#>   runThread(11)  8.098492  9.852568 17.51573 13.567982 20.94633 36.09937   100
#>   runThread(12)  8.128809  9.289289 17.01426 11.837734 20.69145 38.53178   100
#>   runThread(13)  7.761838  9.166097 15.98182 13.250288 19.53973 34.48666   100
#>   runThread(14)  7.905669  8.962629 16.82134 11.580982 24.97496 50.52276   100
#>   runThread(15)  7.311771  9.156278 17.40512 13.189013 30.09495 36.07126   100
#>   runThread(16)  7.516997 10.637961 19.26154 17.671474 31.49865 41.26845   100
autoplot(bench)
```



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

13.4.2 A real life example

Before some of the parallel solving was implemented, the fastest way to run `rxode2` was with `lapply`. This is how Rik Schoemaker created the data-set for `nlmixr` comparisons, but reduced to run faster automatic building of the pkgdown website.

```
library(rxode2)
library(data.table)
#Define the rxode2 model
```

```
  ode1 <- "
  d/dt(abs)    = -KA*abs;
  d/dt(centr)  =  KA*abs-(CL/V)*centr;
  C2=centr/V;
  "

#Create the rxode2 simulation object
mod1 <- rxode2(model = ode1)

#Population parameter values on log-scale
  paramsl <- c(CL = log(4),
               V = log(70),
               KA = log(1))
#make 10,000 subjects to sample from:
  nsubg <- 300 # subjects per dose
  doses <- c(10, 30, 60, 120)
  nsub <- nsubg * length(doses)
#IIV of 30% for each parameter
  omega <- diag(c(0.09, 0.09, 0.09))# IIV covariance matrix
  sigma <- 0.2
#Sample from the multivariate normal
  set.seed(98176247)
  rxSetSeed(98176247)
  library(MASS)
  mv <-
    mvrnorm(nsub, rep(0, dim(omega)[1]), omega) # Sample from covariance matrix
#Combine population parameters with IIV
  params.all <-
    data.table(
      "ID" = seq(1:nsub),
      "CL" = exp(paramsl['CL'] + mv[, 1]),
      "V" = exp(paramsl['V'] + mv[, 2]),
      "KA" = exp(paramsl['KA'] + mv[, 3])
    )
#set the doses (looping through the 4 doses)
params.all[, AMT := rep(100 * doses,nsubg)]

Startlapply <- Sys.time()

#Run the simulations using lapply for speed
  s = lapply(1:nsub, function(i) {
#selects the parameters associated with the subject to be simulated
    params <- params.all[i]
#creates an eventTable with 7 doses every 24 hours
    ev <- eventTable()
    ev$add.dosing(
      dose = params$AMT,
      nbr.doses = 1,
      dosing.to = 1,
      rate = NULL,
      start.time = 0
    )
#generates 4 random samples in a 24 hour period
    ev$add.sampling(c(0, sort(round(sample(runif(600, 0, 1440), 4) / 60, 2))))
#runs the rxode2 simulation
    x <- as.data.table(mod1$run(params, ev))
#merges the parameters and ID number to the simulation output
    x[, names(params) := params]
  })

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

Stoplapply <- Sys.time()

print(Stoplapply - Startlapply)
#> Time difference of 14.39288 secs
```

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

```
rx <- rxode2({
    CL =  log(4)
    V = log(70)
    KA = log(1)
    CL = exp(CL + eta.CL)
    V = exp(V + eta.V)
    KA = exp(KA + eta.KA)
    d/dt(abs)    = -KA*abs;
    d/dt(centr)  =  KA*abs-(CL/V)*centr;
    C2=centr/V;
})

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

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

```
startParallel <- Sys.time()
ev <- do.call("rbind",
        lapply(seq_along(doses), function(i){
            et() %>%
                et(amt=doses[i]) %>% # Add single dose
                et(0) %>% # Add 0 observation
#### Generate 4 samples in 24 hour period
                et(lapply(1:4, function(...){c(0, 24)})) %>%
                et(id=seq(1, nsubg) + (i - 1) * nsubg) %>%
#### Convert to data frame to skip sorting the data
#### When binding the data together
                as.data.frame
        }))
#### To better compare, use the same output, that is data.table
res <- rxSolve(rx, ev, omega=omega, returnType="data.table")
endParallel <- Sys.time()
print(endParallel - startParallel)
#> Time difference of 0.1093781 secs
```

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

- `rxode2` use the thread-safe sitmo `threefry` routines for simulation of `eta` values. Therefore the results are expected to be different (also the random samples are taken in a different order which would be different)
- This prior simulation was run in R 3.5, which has a different random number generator so the results in this simulation will be different from the actual nlmixr comparison when using the slower simulation.
- This speed comparison used `data.table`. `rxode2` uses `data.table` internally (when available) try to speed up sorting, so this would be different than installations where `data.table` is not installed. You can force rxode2 to use `order()` when sorting by using `forderForceBase(TRUE)`. In this case there is little difference between the two, though in other examples `data.table`'s presence leads to a speed increase (and less likely it could lead to a slowdown).

13.4.2.1 Want more ways to run multi-subject simulations

The version since the tutorial has even more ways to run multi-subject simulations, including adding variability in sampling and dosing times with `et()` (see rxode2 events for more information), ability to supply both an `omega` and `sigma` matrix as well as adding as a `thetaMat` to R to simulate with uncertainty in the `omega`, `sigma` and `theta` matrices; see rxode2 simulation vignette.

13.5 Integrating rxode2 models in your package

13.5.1 Using Pre-compiled models in your packages

If you have a package and would like to include pre-compiled rxode2 models in your package it is easy to create the package. You simple make the package with the `rxPkg()` command.

```
library(rxode2);
#### Now Create a model
idr <- rxode2({
    C2 = centr/V2;
    C3 = peri/V3;
    d/dt(depot) =-KA*depot;
    d/dt(centr) = KA*depot - CL*C2 - Q*C2 + Q*C3;
    d/dt(peri)  =                    Q*C2 - Q*C3;
    d/dt(eff)  = Kin - Kout*(1-C2/(EC50+C2))*eff;
})

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

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

This will:

- Add the model to your package; You can use the package data as `idr` once the package loads
- Add the right package requirements to the DESCRIPTION file. You will want to update this to describe the package and modify authors, license etc.
- Create skeleton model documentation files you can add to for your package documentation. In this case it would be the file `idr-doc.R` in your `R` directory
- Create a `configure` and `configure.win` script that removes and regenerates the `src` directory based on whatever version of `rxode2` this is compiled against. This should be modified if you plan to have your own compiled code, though this is not suggested.
- You can write your own R code in your package that interacts with the rxode2 object so you can distribute shiny apps and similar things in the package context.

Once this is present you can add more models to your package by `rxUse()`. Simply compile the rxode2 model in your package then add the model with `rxUse()`

```
rxUse(model)
```

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

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

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

13.5.2 Using Models in a already present package

To illustrate, lets start with a blank package

```
library(rxode2)
library(usethis)
pkgPath  <- file.path(rxTempDir(),"MyRxModel")
create_package(pkgPath);
use_gpl3_license("Matt")
use_package("rxode2", "LinkingTo")
use_package("rxode2", "Depends") ##  library(rxode2) on load; Can use imports instead.
use_roxygen_md()
##use_readme_md()
library(rxode2);
#### Now Create a model
idr <- rxode2({
    C2 = centr/V2;
    C3 = peri/V3;
    d/dt(depot) =-KA*depot;
    d/dt(centr) = KA*depot - CL*C2 - Q*C2 + Q*C3;
    d/dt(peri)  =                    Q*C2 - Q*C3;
    d/dt(eff)  = Kin - Kout*(1-C2/(EC50+C2))*eff;
});

rxUse(idr); ## Add the idr model to your package
rxUse(); # Update the compiled rxode2 sources for all of your packages
```

The `rxUse()` will: - Create `rxode2` sources and move them into the package's `src/` directory. If there is only R source in the package, it will also finish off the directory with an `library-init.c` which registers all the rxode2 models in the package for use in R. - Create stub R documentation for each of the models your are including in your package. You will be able to see the R documentation when loading your package by the standard `?` interface.

You will still need to: - Export at least one function. If you do not have a function that you wish to export, you can add a re-export of `rxode2` using roxygen as follows:

```
##' @importFrom rxode2 rxode2
##' @export
rxode2::rxode2
```

If you want to use `Suggests` instead of `Depends` in your package, you way want to export all of rxode2's normal routines

```
##' @importFrom rxode2 rxode2
##' @export
rxode2::rxode2

##' @importFrom rxode2 et
##' @export
rxode2::et

##' @importFrom rxode2 etRep
##' @export
rxode2::etRep

##' @importFrom rxode2 etSeq
##' @export
rxode2::etSeq

##' @importFrom rxode2 as.et
##' @export
rxode2::as.et

##' @importFrom rxode2 eventTable
##' @export
rxode2::eventTable

##' @importFrom rxode2 add.dosing
##' @export
rxode2::add.dosing

##' @importFrom rxode2 add.sampling
##' @export
rxode2::add.sampling

##' @importFrom rxode2 rxSolve
##' @export
rxode2::rxSolve

##' @importFrom rxode2 rxControl
##' @export
rxode2::rxControl

##' @importFrom rxode2 rxClean
##' @export
rxode2::rxClean
```

```
##' @importFrom rxode2 rxUse
##' @export
rxode2::rxUse

##' @importFrom rxode2 rxShiny
##' @export
rxode2::rxShiny

##' @importFrom rxode2 genShinyApp.template
##' @export
rxode2::genShinyApp.template

##' @importFrom rxode2 cvPost
##' @export
rxode2::cvPost

### This is actually from `magrittr` but allows less imports
##' @importFrom rxode2 %>%
##' @export
rxode2::`%>%`
```

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

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

If this is a R package with rxode2 models and you do not intend to add any other compiled sources (recommended), you can add the following configure scripts

```
#!/bin/sh
### This should be used for both configure and configure.win
echo "unlink('src', recursive=TRUE);rxode2::rxUse()" > build.R
${R_HOME}/bin/Rscript build.R
rm build.R
```

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

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

If you want to integrate with other sources in your `Rcpp` or `C/Fortan` based packages, you need to include `rxModels-compiled.h` and: - Add the define macro `compiledModelCall` to the list of registered `.Call` functions. - Register C interface to allow model solving by `R_init0_rxModels_rxode2_models()` (again `rxModels` would be replaced by your package name).

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

```
devtools::load_all()
devtools::document()
devtools::install()
```

If you load the package with a new version of rxode2, the models will be recompiled when they are used.

However, if you want the models recompiled for the most recent version of rxode2, you simply need to call `rxUse()` again in the project directory followed by the standard methods for install/create a package.

```
devtools::load_all()
devtools::document()
devtools::install()
```

Note you do not have to include the `rxode2` code required to generate the model to regenerate the rxode2 c-code in the `src` directory. As with all rxode2 objects, a `summary` will show one way to recreate the same model.

An example of compiled models package can be found in the rxModels repository.

13.6 Stiff ODEs with Jacobian Specification

13.6.0.1 Stiff ODEs with Jacobian Specification

Occasionally, you may come across a **stiff** differential equation, that is a differential equation that is numerically unstable and small variations in parameters cause different solutions to the ODEs. One way to tackle this is to choose a stiff-solver, or hybrid stiff solver (like the default LSODA). Typically this is enough. However exact Jacobian solutions may increase the stability of the ODE. (Note the Jacobian is the derivative of the ODE specification with respect to each variable). In rxode2 you can specify the Jacobian with the `df(state)/dy(variable)=` statement. A classic ODE that has stiff properties under various conditions is the Van der Pol differential equations.

In rxode2 these can be specified by the following:

```
library(rxode2)

Vtpol2 <- rxode2({
    d/dt(y)  = dy
    d/dt(dy) = mu*(1-y^2)*dy - y
##### Jacobian
    df(y)/dy(dy)  = 1
    df(dy)/dy(y)  = -2*dy*mu*y - 1
    df(dy)/dy(dy) = mu*(1-y^2)
##### Initial conditions
    y(0) = 2
    dy(0) = 0
##### mu
    mu = 1 ## nonstiff; 10 moderately stiff; 1000 stiff
})

et <- eventTable();
et$add.sampling(seq(0, 10, length.out=200));
et$add.dosing(20, start.time=0);

s1 <- Vtpol2 %>%  solve(et, method="lsoda")
print(s1)
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#> mu
#>  1
#> -- Initial Conditions ($inits): --
#>  y dy
#>  2  0
#> -- First part of data (object): --
#> # A tibble: 200 x 3
#>     time     y      dy
#>    <dbl> <dbl>   <dbl>
#> 1 0       22    0
#> 2 0.0503  22.0 -0.0456
#> 3 0.101   22.0 -0.0456
#> 4 0.151   22.0 -0.0456
#> 5 0.201   22.0 -0.0456
#> 6 0.251   22.0 -0.0456
#> # i 194 more rows
```

While this is not stiff at mu=1, mu=1000 is a stiff system

```
s2 <- Vtpol2 %>%  solve(c(mu=1000), et)
print(s2)
#> -- Solved rxode2 object --
#> -- Parameters ($params): --
#>   mu
#> 1000
#> -- Initial Conditions ($inits): --
#>  y dy
#>  2  0
#> -- First part of data (object): --
#> # A tibble: 200 x 3
#>     time     y         dy
#>    <dbl> <dbl>      <dbl>
#> 1 0       22    0
#> 2 0.0503  22.0 -0.0000455
#> 3 0.101   22.0 -0.0000455
#> 4 0.151   22.0 -0.0000455
#> 5 0.201   22.0 -0.0000455
#> 6 0.251   22.0 -0.0000455
#> # i 194 more rows
```

While this is easy enough to do, it is a bit tedious. If you have rxode2 setup appropriately, you can use the computer algebra system sympy to calculate the Jacobian automatically.

This is done by the rxode2 option `calcJac` option:

```
Vtpol <- rxode2({
    d/dt(y)  = dy
    d/dt(dy) = mu*(1-y^2)*dy - y
##### Initial conditions
    y(0) = 2
    dy(0) = 0
##### mu
    mu = 1 ## nonstiff; 10 moderately stiff; 1000 stiff
}, calcJac=TRUE)
```

To see the generated model, you can use `rxCat()`:

```
> rxCat(Vtpol)
d/dt(y)=dy;
d/dt(dy)=mu*(1-y^2)*dy-y;
y(0)=2;
dy(0)=0;
mu=1;
df(y)/dy(y)=0;
df(dy)/dy(y)=-2*dy*mu*y-1;
df(y)/dy(dy)=1;
df(dy)/dy(dy)=mu*(-Rx_pow_di(y,2)+1);
```