

nlmixr: an R package for population PKPD modeling

*Matthew Fidler, Teun M. Post, Richard Hooijmaijers, Rik Schoemaker, Mirjam N. Trame,
Justin Wilkins, Yuan Xiong and Wenping Wang*

2018-07-03

Contents

Welcome to population modeling in R	5
Preface: The Road So Far	7
About this book	8
Acknowledgements	8
1 Introduction	9
1.1 Prerequisites	9
1.2 Workflow	9
2 Installation	11
2.1 <code>nlmixr</code>	11
2.2 project management tool (<code>shinyMixR</code>)	16
3 <code>nlmixr</code> vignettes	17
3.1 <code>nlmixr</code> domain specific language	17
3.2 <code>xpose.nlmixr</code>	34
3.3 <code>shinyMixR</code>	35
4 Applications of <code>nlmixr</code>	43
4.1 Posters and Presentations	43
4.2 Sparse data analysis with <code>nlmixr</code>	43
4.3 Course: PAGE 2018	56

Welcome to population modeling in R

The fact that you are here suggest that you have an interest to do population modeling in R! Maybe you found this through our [nlmixr website](#). Maybe via [Twitter](#), [LinkedIn](#), or [GitHub](#). Either way: **welcome, and join us in our journey towards population PKPD modeling with R!**

This book is intended to get you started with [nlmixr](#), an R package for nonlinear mixed effects models in population pharmacokinetics (PK) and pharmacodynamics (PD) . You'll get information on the package itself, its installation, interesting use cases and more.

`nlmixr` is licensed under [GPL-2](#) | [GPL-3](#) [expanded from: GPL (≥ 2)].



Preface: The Road So Far

In pharmacometrics there are several population pharmacokinetic and pharmacodynamic modeling software packages. The **nlmixr** R package was developed for fitting general dynamic models, pharmacokinetic (PK) models and pharmacokinetic-pharmacodynamic (PKPD) models in particular, with either individual data or population data. The interesting part about **nlmixr** is that it is open-source and the entire workflow of modeling can be done solely in R!

nlmixr has its roots in **RxODE**. Once **RxODE** was built it was soon recognized that by combining **RxODE** with a nonlinear mixed-effect model (NLME) estimation tool, one greatly expands an existing NLME tool to include fitting a population PK/PD model described by a set of ordinary differential equations (ODEs) and by allowing complex and arbitrary dosing history.

The first attempt was to combine **RxODE** with the **nlme()** function in R, resulting in two efficient and useful functions for population PK/PD: **nlme_lin_cmpt()** and **nlme_ode()**. After the initial success of these functions, stochastic approximation expectation maximization (SAEM) was subsequently added to the **nlmixr** toolbox. While maintaining fidelity to the original SAEM algorithm, the **nlmixr** implementation of SAEM was written from scratch using C++ with many optimizations and re-factorizations in comparison to the original implementation. The resulting SAEM code is compact and efficient. Since execution of **nlmixr**'s SAEM is in compiled C++ mode, the speed of the implementation is relatively fast.

A function for the generalized nonlinear mixed-effect model (**gnlmm**) was developed to address project needs. This function **gnlmm()** takes an arbitrary user-defined log-likelihood function and hence is able to fit odd types of data like binary data, count data, heavy tail data and bounded clinical endpoints via beta-regression. Structure models can be optionally defined by user-defined ODEs.

These population PK/PD functions, plus a number of ancillary functions to aid model building (including support for VPCs, bootstraps and stepwise forward selection of covariates), comprised the functionality of the first public release of **nlmixr** in October 2016.

In early 2017, a publication on the use of sensitivity equations in first-order conditional estimation with interaction (FOCEi) by [Almqvist and colleagues](#) was brought to the attention of the **nlmixr** team, and a proof-of-concept implementation was soon put in place. A full production version will soon be completed.

nlmixr's population PK/PD functions are highly optimized and efficient. However, due to historic reasons (e.g., **nlme**'s original user interface) and specific input-output needs, different algorithms in **nlmixr** had similar but subtly different user interfaces. To reduce the learning curve of population PK/PD modeling, the **nlmixr** team proposed and implemented a unified user-interface (UI) across all fitting algorithms in mid-2017. The result is a minimalist, intuitive, expressive, domain-specific population PK/PD modeling language. A comprehensive collection of **nlme** algorithms with the UI marked the first CRAN release of **nlmixr** in October 2017.

nlmixr remains under active, intensive development, and exciting features and functionality emerge almost on a daily basis. For instance, parallel computing has recently become available in **RxODE** via the **openmp** package. This parallel computing capability is the industry's first among the current population PK/PD simulators to the best of our knowledge. Clinical trial simulation (CTS) was added to **nlmixr** in May 2018 and is capable of extensive and sophisticated CTS right after a population PK/PD model with **nlmixr**.

We hope you enjoy `nlmixr`!

Wenping Wang

About this book

This book provides guidance on the use of `nlmixr` and serves as a basic reference manual, although we do not cover every aspect of modeling or `nlmixr`.

We assume readers have a background in pharmacometric modeling and know how to use R. Basic knowledge is required and assumed on installing other software packages, such as Python. The installation of `nlmixr` and related software and packages is described in Chapter [2](#).

Acknowledgements

We are thankful for the time people have been willing to spend on this project. There have been various interactions and contributions on GitHub, which we really appreciate. We would like to especially thank James Cavanaugh, Ron Keizer and Jack Miller.

Chapter 1

Introduction

1.1 Prerequisites

Several packages and pieces of software together create the `nlmixr` environment.

These packages and software are:

- [R](#) (including variants such as [Microsoft R Open](#) and related GUIs, such as [RGui](#) and [Rstudio](#))
- [Python](#) and [SymPy](#)
- [RxODE](#) (development release on [GitHub](#))
- [nlmixr](#) (development release on [GitHub](#))

Secondary R packages required by `nlmixr` are:

- [n1qn1](#)
- [PreciseSums](#)
- [SnakeCharmR](#)

Packages to support the use of `nlmixr` are:

- [shinyMixR](#)
- [xpose.nlmixr](#)

More information on the installation of packages and software can be found in [Chapter 2](#).

1.2 Workflow

The ability to perform population modeling in R provides an opportunity to work via a single unified workflow for data management, data exploration, data analysis and report writing.

`nlmixr` can be used directly from the R command line or via the user-friendly R Shiny tool `shinyMixR`. The `shinyMixR` package provides a means to build a project-centric workflow around `nlmixr` from the R command line and from a streamlined [Shiny](#) application. This project tool was developed to enhance the usability and attractiveness of `nlmixr`, facilitating dynamic and interactive use in real-time for rapid model development. More on the use of `shinyMixR` in [Chapter 3.3](#).

Chapter 2

Installation

`nlmixr` can be installed and used on several platforms. Installation can range from easy to challenging, depending on the platform. We are in the process of streamlining this process, and any help or suggestions are greatly appreciated!

2.1 `nlmixr`

2.1.1 Windows installer

For those not interested in customized installation on Windows, we **recommend** you download a Windows installer for your platform from the following [link](#)

2.1.2 Installation on Windows

To replicate the environment for `nlmixr` development on Windows, you may need administrator rights. The following steps should be performed:

1. Install R 3.4.1 (or later - we recommend 3.5.0, and will assume this in the rest of these instructions) from the R website.
 - For best results, we suggest you use `C:\R\R-3.5.0`, but you can also use the default location (`C:\Program Files\R\R-3.5.0`) as well, if really needed. If writing to `C:\` is not allowed, another option could be to install to `%AppData%`, which is typically in `C:\Users\userName\AppData\`. Most users have write access to this location.
 - When installing on 64-bit Windows, it is best practice to include *only* the 64-bit version. If you include 32-bit files, some packages may not run correctly. Additionally, both the 32- and 64-bit binaries have to be compiled for every package. Similarly, if running on 32-bit Windows, install only the 32-bit version of R (and Python, and Rtools).
 - Using “Program Files” may lead to difficulties because paths with spaces are not always supported by all R packages and typically the “Program Files” directory is read-only, which has also been known to cause problems. We do not recommend this.
2. Install the appropriate version of Rtools for Windows, currently version 3.4, from [here](#).
 - This is an absolute requirement, since it includes C++ and related compilers not usually available under Windows. Further, alternative toolsets installed from other sources may not work correctly.

- For best results, use the default location of `C:\Rtools`
 - `RxODE`, a required component of `nlmixr`, checks and sets up the path based on the following:
 - a. `Rtools` is in the path (fastest and recommended option)
 - b. `Rtools` was installed with information saved to the Windows registry, and `RxODE` can find the installation.
 - c. `Rtools` is on a hard drive installed in either `Rtools` or `RBuildTools`
 - If you are on 64-bit windows, please *do not install* the R 3.3.x 32-bit toolchain. These files can interfere with some packages that compile binaries, with unpredictable consequences. Similarly, only install 32-bit `Rtools` on 32-bit versions of Windows.
- Make sure the compilers have been added to the Windows `PATH` environment variable, or `RxODE` and `nlmixr` may not work (this should be done automatically during installation). Under Windows, this is not strictly necessary, rather `RxODE` finds the paths and sets them up if they are not in the path environment. However, it is good practice.

3. Install a version of Python for Windows.

- This is used for its symbolic algebra package [SymPy](#).
- A very robust Python distribution that includes [SymPy](#) and many packages that may be useful to the data scientist and/or pharmacometrician is [Anaconda](#). Although very straightforward and easy to install, it is quite a large download and contains much more than you will need to run `nlmixr`. When installing, use the Python 3.6 version. During the installation, Anaconda provides the option of adding itself to the `PATH` environment variable, but advises against it; please do this anyway (despite the red warning).
- Another option is to use [official Python](#), although you will need to install [SymPy](#) separately if you go this route, which is sometimes not straightforward under Windows 10 owing to folder permissions (see [here](#) for a few workarounds). Nonetheless, see [here](#) for instructions for installation from source or using `pip`. Note that if you approach us for support, we are going to recommend that you use [Anaconda](#).
- Regardless of the option you choose, please use 64-bit Python for 64-bit Windows (or vice versa for 32-bit).
- Once again, make sure Python has been added to the Windows `PATH` environment variable, or `RxODE` and `nlmixr` will not work, no matter what Anaconda might say.

3. Install devtools.

- This package is required to install packages from GitHub, amongst other things.
- This can be done from a clean R session by `install.packages("devtools")`.

4. Load devtools using `library(devtools)`.

5. Install RxODE.

- Currently the new version of `RxODE` is in the process of being uploaded to CRAN. `nlmixr` needs this newer version of `RxODE` to function correctly. To install this version, use the command: `install_github("nlmixrdevelopment/RxODE")`.
- Once installed, type `RxODE::rxWinPythonSetup()` to install the required package `SnakeCharmR` and to make sure Python and `SymPy` are working properly.
- Restart your R session.
- As a quick test, you can make sure that R and Python can communicate by typing the command `library(SnakeCharmR)`. This will display the version of python you are using.
- Another test to make sure that `SymPy` is setup appropriately is to do `library(RxODE)` and `rxSymPyVersion()`. This will display the `SymPy` version.

- To validate or test the installation of RxODE completely, you can type the following `library(RxODE); rxTest()` and it will run all of the unit tests in RxODE to make sure it is running correctly on your system. (Note that the `testthat` package is required for this, and it will take a long time.) For a short test use `rxTest(FALSE)`.

6. Install nlmixr.

- Load `devtools` again using `library(devtools)`
- Install `nlmixr` by running `install_github("nlmixrdevelopment/nlmixr")`

2.1.3 Installation on Linux

Instructions for Ubuntu-alike distributions are given here (specifically, [Ubuntu 16.04 Xenial Xerus](#)), but all current Linux distributions are supported, in principle.

1. Install R 3.4.1 (or later - we recommend 3.5.0) from an appropriate repository (Ubuntu Xenial shown below, based on instructions provided [here](#)).
 - You will need administrator privileges (i.e. access to `sudo`). Provide your admin password when asked.
 - Add the official CRAN repository for Ubuntu: `sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys E298A3A825C0D65DFD57CBB651716619E084DAB9`
 - Add the official CRAN repository for Ubuntu: `sudo add-apt-repository 'deb [arch=amd64,i386] https://cran.rstudio.com/bin/linux/ubuntu xenial/'`. If you aren't using Ubuntu Xenial, change `xenial` to match your distribution's codename.
 - Now refresh the package list using `sudo apt-get update`.
 - We can now install base R and required development libraries, and their dependencies: `sudo apt-get install r-base r-base-dev libssl-dev`
2. Install Python dependencies.
 - Enter this: `sudo apt-get install python-sympy python-pip python-setuptools python3-pip python-dev python3-dev`.
3. Install `devtools` and dependencies.
 - This package is required to install packages from Github, amongst other things.
 - Some Linux distributions don't include build tools out of the box. To be safe, check this: `sudo apt-get install build-essential`
 - Install `devtools` from a clean R session by entering `install.packages("devtools")`.
4. In R, load `devtools` using `library(devtools)`.
5. Install RxODE.
 - Currently the new version of RxODE is in the process of being uploaded to CRAN. `nlmixr` needs this newer version of RxODE to function correctly. To install this version, use the command: `install_github("nlmixrdevelopment/RxODE")`.
 - Install `SnakeCharmR` using `install_github("nlmixrdevelopment/SnakeCharmR")`.
 - Restart your R session.
 - As a quick test, you can make sure that R and Python can communicate by typing the command `library(SnakeCharmR)`. This will display the version of python you are using.

- Another test to make sure that `SymPy` is setup appropriately is to do `library(RxODE)` and `rxSymPyVersion()`. This will display the `SymPy` version.
- To validate or test the installation of `RxODE` completely, you can type the following `library(RxODE); rxTest()` and it will run all of the unit tests in `RxODE` to make sure it is running correctly on your system. (Note that the `testthat` package is required for this, and it will take a long time.) For a short test use `rxTest(FALSE)`.

6. Install `nlmixr`.

- This can be done by `install_github("nlmixrdevelopment/nlmixr")`

2.1.4 Installation on macOS

Instructions for macOS 10.12 Sierra are provided here. They should be broadly extensible to all recent releases of macOS.

The general logic is to first install any external software and then install the R packages.

1. Check the Python version in the terminal: `python --version`. On macOS the standard version is 2.7, which is used by `nlmixr`. If there are other versions of Python available, set the `$PATH` to the 2.7 version and/or remove other versions of Python, if possible (this step can be a challenge).
2. Install R 3.4.1 (or later - we recommend 3.5.0, and will assume this in the rest of these instructions) from the R website.
 - Download and install the appropriate R package (e.g. `R-3.5.0.pkg`) from [CRAN](#).
3. Install Python dependencies.
 - Install `pip` from the macOS or RStudio terminal prompt: `sudo easy_install pip`.
 - Install `sympy` using `pip`: `sudo -H pip install sympy`.
4. Install build tools.
 - Install Xcode from the App Store (see which version matches your MacOS if you do not want to upgrade to the latest OS).
 - Read the license by entering the following at the macOS terminal: `sudo xcodebuild -license`
 - Scroll through it all, reading it carefully, and type `agree` at the end. (If you don't, you can't use `nlmixr` or anything else that requires compilation on macOS. Don't yell at us, yell at Apple.)
 - Install `gfortran`: download the appropriate macOS installer from [here](#) and run it. Another option is to run the installation for the appropriate macOS version from the terminal by typing: `brew cask install gfortran` to [automatically install the version of gfortran](#) on macOS needed by R from the Terminal (Mac or RStudio). Another option is to use the Mac build tools, which are the [Mac equivalent of Rtools](#). With R 3.5, this includes `clang6`, which has open mp support required for threaded/parallel `RxODE` solving.

Now install the required R packages:

5. Install `devtools` and dependencies.
 - This package is required to install packages from GitHub, amongst other things.
 - Install `devtools` from a clean R session by entering `install.packages("devtools")`.

6. In R, load devtools using `library(devtools)`.

7. Install RxODE.

- Currently the new version of RxODE is in the process of being uploaded to CRAN. `nlmixr` needs this newer version of RxODE to function correctly. To install this version, use the command: `install_github("nlmixrdevelopment/RxODE")` or use `install.packages('RxODE')`.
- Install SnakeCharmR using `install_github("nlmixrdevelopment/SnakeCharmR")`.
- Restart your R session.
 - As a quick test, you can make sure that R and Python can communicate by typing the command `library(SnakeCharmR)`. This will display the version of python you are using.
- SnakeCharmR wants to install the Python package `SymPy` and the latest macOS version does not allow this (installation of packages cannot be done system wide and has to be done on user level, or install as Administrator which may allow to install system wide). To do this on user level, open the terminal (in macOS or RStudio) and type: `pip install sympy --user` (the user flag is important!).
- Another test to make sure that `SymPy` is setup appropriately is to do `library(RxODE)` and `rxSymPyVersion()`. This will display the `SymPy` version.
- To validate or test the installation of RxODE completely, you can type the following `library(RxODE); rxTest()` and it will run all of the unit tests in RxODE to make sure it is running correctly on your system. (Note that the `testthat` package is required for this, and it will take a long time.) For a short test use `rxTest(FALSE)`.

8. Install `nlmixr`.

- This can be done by `install_github("nlmixrdevelopment/nlmixr")` or `install.packages('nlmixr')`.

2.1.5 Installation via Docker

Docker is a tool designed to make it easier to create, deploy, and run applications by using so-called “containers”. Containers allow complex applications to be wrapped as packages, ensuring they will run on any computer or platform regardless of any customized settings that machine might have that could differ from the machine used for writing and testing the code (a little like virtual machines, but more lightweight and portable). This is very attractive in our case, since it allows all the complexity above to be sidestepped neatly, and in principle makes the results obtained more reproducible.

The first thing to do is install Docker. An installer is available for all major operating systems:

- Windows: <https://docs.docker.com/docker-for-windows/install/>
- macOS: <https://docs.docker.com/docker-for-mac/install/>
- Ubuntu Linux: <https://docs.docker.com/install/linux/docker-ce/ubuntu/>

Make sure you read the prerequisites, but for most modern systems it is sufficient to download and run the installer.

1. Running the container

To run a docker container you should use a terminal window (or command prompt in Windows).

The first time a container is run, it will download all applicable files from the repository. This can take some time depending on your internet speed. After the first time, the container will be saved locally and the machine will start almost immediately.

It is usually necessary to have access to your local drive, and this can be done by adding an option (`-v`) in the docker command. In the following example the folder `/Users/richard/Document` on the host computer will be mounted to the `docs` folder in the `/home/rstudio` directory of the container:

```
docker run -v /Users/richard/Documents/:/home/rstudio/docs -d -p 8787:8787 nlmixr/nlmixr:1.0
```

The version above applies to Linux, but it's easy to do in Windows as well, by making the appropriate adjustments in the “Shared Drives” tab of the Docker settings app (accessible by right-clicking the Docker whale icon in the Windows system tray). Note that you may need adjust your firewall settings to enable folder sharing, and that you'll probably need administrator privileges.

Once the Docker container is running, you can go to a browser and go to the following URL:

```
localhost:8787
```

This will launch an RStudio Server instance hosted inside the container. If everything has gone correctly you will see an RStudio login screen. You can log in with username and password **rstudio**. If a local folder was mounted, you should be able to find this directly in the RStudio files pane.

Note: In certain cases the localhost is not recognized and the url should be `127.0.0.1:8787`

Note: If you are using Docker Toolbox on Windows 7, use the Docker Machine IP instead of localhost. For example, <http://192.168.99.100:8787/>. To find the right IP address, the `ipconfig` command can be used from the Command Prompt.

Note: In general, most modern browsers will work with RStudio Server. However, there might be some problems when running shiny apps. Both Chrome and Safari have been tested and should work correctly.

2. Stop the container

Once you are done, or you would want to start another instance of the container, you can close the browser window and from the terminal the following can be done to stop all containers:

```
docker kill $(docker ps -q)
```

Note: In case you want to stop a specific container, you can list the running containers with `docker container ls` and close the specific container ID (e.g. `docker container stop 1fa4ab2cf395`).

Note: In Windows it might be necessary to stop the container using Powershell; otherwise, the container can be stopped using Task Manager or through the Docker GUI.

2.2 project management tool (shinyMixR)

A user-friendly tool was developed for `nlmixr` based on [Shiny](#) and [shinydashboards](#), which facilitates a workflow around an `nlmixr` project.

This `shinyMixR` package provides a means to build a project-centric workflow around `nlmixr` from the R command line and from a streamlined Shiny application. This project tool was developed to enhance the usability and attractiveness of `nlmixr`, facilitating dynamic and interactive use in real-time for rapid model development. More on the use of `ShinyMixR` in Chapter 3.3.

To install the package, use:

```
devtools::install_github("richardhooijmaijers/shinyMixR")
```


Chapter 3

nlmixr vignettes

Some *significant* applications are demonstrated in this chapter; the vignettes will be discussing the application in more depth.

3.1 nlmixr domain specific language

```
library(nlmixr)
?nlmixr
```

Under the hood, `nlmixr` has five main modules:

1. `dynmodel()` and its mcmc cousin `dynmodel.mcmc()` for nonlinear dynamic models of individual data;
2. `nlme_lin_cmpt()` for one to three linear compartment models of population data with first order absorption, or i.v. bolus, or i.v. infusion using the nlme algorithm;
3. `nlme_ode()` for general dynamic models defined by ordinary differential equations (ODEs) of population data using the nlme algorithm;
4. `saem_fit` for general dynamic models defined by ordinary differential equations (ODEs) of population data by the Stochastic Approximation Expectation-Maximization (SAEM) algorithm;
5. `gnlmm` and `gnlmm2` for generalized non-linear mixed-models (possibly defined by ordinary differential equations) of population data by the adaptive Gaussian quadrature algorithm.

3.1.1 Rationale

`nlmixr` estimation routines have their own way of specifying models. Often the models are specified in ways that are most intuitive for one estimation routine, but do not make sense for another estimation routine. Sometimes, legacy estimation routines like `[nlme]` have their own syntax that is outside of the control of the `nlmixr` package.

The unique syntax of each routine makes the routines themselves easier to maintain and expand, and allows interfacing with existing packages that are outside of `nlmixr` (like `[nlme]`). However, a model definition language that is common between estimation methods, and an output object that is uniform, will make it easier to switch between estimation routines and will facilitate interfacing output with external packages like `xpose` and other user-written packages.

The `nlmixr` mini-modeling language (Domain Specific Language) attempts to address this issue by incorporating a common language. This language is inspired by both R and NONMEM, since these languages are familiar to many pharmacometricians.

Initial Estimates and boundaries for population parameters

`nlmixr` models are contained in a R function with two blocks: `ini` and `model`. This R function can be named anything, but is not meant to be called directly from R. In fact, if you try you will likely get an error such as `Error: could not find function "ini"`.

The `ini` model block

The `ini` model block is meant to hold the initial estimates for the model, and the boundaries of the parameters for estimation routines that support boundaries (note `nlmixr`'s `saem` and `[nlme]` do not currently support parameter boundaries).

To explain how these initial estimates are specified we will start with an annotated example:

```
f <- function(){ ## Note the arguments to the function are currently
                  ## ignored by `nlmixr`

  ini({
    ## Initial conditions for population parameters (sometimes
    ## called theta parameters) are defined by either `<-` or `=`
    lCl <- 1.6      #log Cl (L/hr)
    ## Note that simple expressions that evaluate to a number are
    ## OK for defining initial conditions (like in R)
    lVc = log(90)  #log V (L)
    ## Also a comment on a parameter is captured as a parameter label
    lKa <- 1 #log Ka (1/hr)
    ## Bounds may be specified by c(lower, est, upper), like NONMEM:
    ## Residuals errors are assumed to be population parameters
    prop.err <- c(0, 0.2, 1)
  })
  ## The model block will be discussed later
  model({})
}
```

As shown in the above examples:

- Simple parameter values are specified as a R-compatible assignment
- Boundaries may be specified by `c(lower, est, upper)`.
- Like NONMEM, `c(lower,est)` is equivalent to `c(lower,est,Inf)`
- Also like NONMEM, `c(est)` does not specify a lower bound, and is equivalent to specifying the parameter without R's `c` function.
- The initial estimates of the between subject variabilities are specified on the variance scale, and in analogy with NONMEM, the square roots of the diagonal elements correspond to coefficients of variation when used in the exponential IIV implementation. This is true, since this is an approximation. When displaying %CV in the `nlmixr` table, it uses: `sqrt(exp(v) - 1) * 100` where `v`=variance of the Omega formula.

These parameters can be called by almost any R-compatible name. Please note that:

- Residual error estimates should be coded as population estimates (i.e. using an `'='` or `'<-'` statement, not a `'~'`). These are specified on a standard deviation scale.
- Variable names starting with `"_"` are not supported. Note that R does not allow variable starting with `"_"` to be assigned without quoting them.

- Variable names starting with “rx_” or “nlmixr_” are not supported, since RxODE and nlmixr use these prefixes internally for certain estimation routines and for calculating residuals.
- Variable names are case sensitive, just like they are in R. “CL” is not the same as “Cl”.

Initial estimates for between-subject error distributions

In mixture models, multivariate normal individual deviations from the population parameters are estimated (in NONMEM these are called ETA parameters). Additionally, the variance/covariance matrix of these deviations are also estimated (in NONMEM this is the OMEGA matrix). The initial estimates for variance are specified by the ~ operator in nlmixr, which is typically used in R to denote “modeled by”, and was chosen to distinguish these estimates from the population and residual error parameters.

Continuing the prior example, we can annotate the estimates for the between subject error distribution:

```
f <- function(){
  ini({
    lCl <- 1.6      #log Cl (L/hr)
    lVc = log(90)   #log V (L)
    lKa <- 1        #log Ka (1/hr)
    prop.err <- c(0, 0.2, 1)
    ## Initial estimate for ka IIV variance
    ## Labels work for single parameters
    eta.ka ~ 0.1    #BSV Ka

    ## For correlated parameters, you specify the names of each
    ## correlated parameter separated by a addition operator `+`
    ## and the left handed side specifies the lower triangular
    ## matrix initial of the covariance matrix.
    eta.cl + eta.vc ~ c(0.1,
                        0.005, 0.1)

    ## Note that labels do not currently work for correlated
    ## parameters. Also do not put comments inside the lower
    ## triangular matrix as this will currently break the model.
  })
  ## The model block will be discussed later
  model({})
}
```

As shown in the above examples:

- Simple variances are specified by the variable name and the estimate separated by ~.
- Correlated parameters are specified by the sum of the variable labels and then the lower triangular matrix of the covariance is specified on the left hand side of the equation. This is also separated by ~.

Currently the model syntax does not allow comments inside the lower triangular matrix.

Model syntax for ODE-based models

The ini model block Once the initialization block has been defined, you can define a model block in terms of the defined variables in the ini block. You can also mix in RxODE blocks into the model. This section is analogous to NONMEM’s \$PK, \$PRED, \$DES and \$ERROR blocks.

The current method of defining an nlmixr model is to specify the parameters, and then possibly the RxODE lines:

Continuing describing the syntax with an annotated example:

```
f <- function(){
  ini({
    lCl <- 1.6      #log Cl (L/hr)
    lVc <- log(90)  #log Vc (L)
    lKA <- 0.1      #log Ka (1/hr)
    prop.err <- c(0, 0.2, 1)
    eta.Cl ~ 0.1 ## BSV Cl
    eta.Vc ~ 0.1 ## BSV Vc
    eta.KA ~ 0.1 ## BSV Ka
  })
  model({
    ## First parameters are defined in terms of the initial estimates
    ## parameter names.
    Cl <- exp(lCl + eta.Cl)
    Vc <- exp(lVc + eta.Vc)
    KA <- exp(lKA + eta.KA)
    ## After the differential equations are defined
    kel <- Cl / Vc;
    d/dt(depot) = -KA*depot;
    d/dt(centr) = KA*depot - kel*centr;
    ## And the concentration is then calculated
    cp = centr / Vc;
    ## Last, nlmixr is told that the plasma concentration follows
    ## a proportional error (estimated by the parameter prop.err)
    cp ~ prop(prop.err)
  })
}
```

A few points to note:

- Parameters are defined before the differential equations. Currently, defining the differential equations directly in terms of the population parameters is not supported.
- The differential equations, parameters and error terms are in a single block, instead of multiple sections.
- State names, calculated variables cannot start with either “rx_” or “nlmixr_” since these are reserved terms for internal use in some estimation routines.
- Errors are specified using ~. Currently you can use either `add(parameter)` for additive error, `prop(parameter)` for proportional error or `add(parameter1) + prop(parameter2)` for combined additive and proportional error. You can also specify `norm(parameter)` for the additive error, since it follows a normal distribution.
- Some algorithms, like `saem`, require parameters in terms of `Pop.Parameter + Individual.Deviation.Parameter + Covariate*Covariate.Parameter`. The order of these parameters do not matter. The approach is similar to NONMEM’s mu-referencing.
- The type of parameter (population parameter estimate, or individual parameter deviation) in the model is determined by the initial block; Covariates used in the model are missing in the `ini` block. These variables need to be present in the modeling dataset for the model to run.

Model Syntax for solved PK systems

Solved PK systems are currently supported by `nlmixr` with the `linCmt()` pseudo-function. An annotated example of a solved system is below:

```
f <- function(){
  ini({
```

```

lCl <- 1.6      #log Cl (L/hr)
lVc <- log(90)  #log Vc (L)
lKA <- 0.1      #log Ka (1/hr)
prop.err <- c(0, 0.2, 1)
eta.Cl ~ 0.1 ## BSV Cl
eta.Vc ~ 0.1 ## BSV Vc
eta.KA ~ 0.1 ## BSV Ka
})
model({
  Cl <- exp(lCl + eta.Cl)
  Vc <- exp(lVc + eta.Vc)
  KA <- exp(lKA + eta.KA)
  ## Instead of specifying the ODEs, you can use
  ## the linCmt() function to use the solved system.
  ##
  ## This function determines the type of PK solved system
  ## to use by the parameters that are defined. In this case
  ## it knows that this is a one-compartment model with first-order
  ## absorption.
  linCmt() ~ prop(prop.err)
})
}

```

A few things to keep in mind:

- Currently the solved systems support either oral dosing, IV dosing or IV infusion dosing, but does not allow mixing the dosing types.
- While RxODE allows mixing of solved systems and ODEs, this has not been implemented in `nlmixr` yet.
- The solved systems implemented are the one-, two- and three-compartment models with or without first-order absorption. Each of the models support a lag time with a `tlag` parameter (although currently `tlag` will cause the CWRES/OBJF calculation to fail).
- The `linCmt()` function figures out the model from the parameter names. `nlmixr` currently knows about numbered volumes, V_c/V_p , clearances in terms of both CL and Q/CLD . Additionally, `nlmixr` knows about elimination micro-constants (ie k_{12}). Mixing of parameters and microconstants for is currently not supported.

Checking model syntax

After specifying the model syntax you can check that `nlmixr` is interpreting it correctly by using the `nlmixr` function on it.

Using the above function we can get:

```

> nlmixr(f)
## 1-compartment model with first-order absorption in terms of Cl
## Initialization:
#####
Fixed Effects ($theta):
      lCl      lVc      lKA
1.60000 4.49981 0.10000

Omega ($omega):
      [,1] [,2] [,3]

```

```

[1,] 0.1 0.0 0.0
[2,] 0.0 0.1 0.0
[3,] 0.0 0.0 0.1

## Model:
#####
Cl <- exp(lCl + eta.Cl)
Vc <- exp(lVc + eta.Vc)
KA <- exp(lKA + eta.KA)
## Instead of specifying the ODEs, you can use
## the linCmt() function to use the solved system.
##
## This function determines the type of PK solved system
## to use by the parameters that are defined. In this case
## it knows that this is a one-compartment model with first-order
## absorption.
linCmt() ~ prop(prop.err)

```

In general this gives you information about the model (what type of solved system/RxODE), initial estimates as well as the code for the model block.

Using the model syntax for estimating a model

Once the model function has been created, you can use it to estimate the parameters for a model given a dataset.

This dataset has to have RxODE compatible event IDs. Both Monolix and NONMEM use a different dataset description. You may convert these datasets to RxODE-compatible datasets with the `nmDataConvert` function. Note that steady state doses are not supported by RxODE, and therefore not supported by the conversion function.

As an example, we can use a simulated rich 1-compartment oral PK dataset.

```

d <- Oral_1CPT
d <- d[,names(d) != "SS"];
d <- nmDataConvert(d);

```

Once the data has been converted to the appropriate format, you can use the `nlmixr` function to run the appropriate code.

The method to execute the model is:

```
fit <- nlmixr(model.function, rxode.dataset, est="est", control=estControl(options))
```

Currently `nlme` and `saem` are implemented. For example, to run the above model with `saem`, we could have the following:

```

> f <- function(){
  ini({
    lCl <- 1.6      #log Cl (L/hr)
    lVc <- log(90)  #log Vc (L)
    lKA <- 0.1      #log Ka (1/hr)
    prop.err <- c(0, 0.2, 1)
    eta.Cl ~ 0.1 ## BSV Cl
    eta.Vc ~ 0.1 ## BSV Vc
  })
}

```

```

    eta.KA ~ 0.1 ## BSV Ka
  })
  model({
    ## First parameters are defined in terms of the initial estimates
    ## parameter names.
    Cl <- exp(lCl + eta.Cl)
    Vc <- exp(lVc + eta.Vc)
    KA <- exp(lKA + eta.KA)
    ## After the differential equations are defined
    kel <- Cl / Vc;
    d/dt(depot) = -KA*depot;
    d/dt(centr) = KA*depot-kel*centr;
    ## And the concentration is then calculated
    cp = centr / Vc;
    ## Last, nlmixr is told that the plasma concentration follows
    ## a proportional error (estimated by the parameter prop.err)
    cp ~ prop(prop.err)
  })
}
> fit.s <- nlmixr(f,d,est="saem",control=saemControl(n.burn=50,n.em=100,print=50));
Compiling RxODE differential equations...done.
c:/Rtools/mingw_64/bin/g++ -I"c:/R/R-34~1.1/include" -DNDEBUG -I"d:/Compiler/gcc-4.9.3/local330/i
In file included from c:/R/R-34~1.1/library/RCPAR~1/include/armadillo:52:0,
      from c:/R/R-34~1.1/library/RCPAR~1/include/RcppArmadilloForward.h:46,
      from c:/R/R-34~1.1/library/RCPAR~1/include/RcppArmadillo.h:31,
      from saem3090757b4bd1x64.cpp:1:
c:/R/R-34~1.1/library/RCPAR~1/include/armadillo_bits/compiler_setup.hpp:474:96: note: #pragma message
      #pragma message ("WARNING: use of OpenMP disabled; this compiler doesn't support OpenMP 3.0+")
c:/Rtools/mingw_64/bin/g++ -shared -s -static-libgcc -o saem3090757b4bd1x64.dll tmp.def saem3090757b4b
done.
1:    1.8174    4.6328    0.0553    0.0950    0.0950    0.0950    0.6357
50:    1.3900    4.2039    0.0001    0.0679    0.0784    0.1082    0.1992
100:   1.3894    4.2054    0.0107    0.0686    0.0777    0.1111    0.1981
150:   1.3885    4.2041    0.0089    0.0683    0.0778    0.1117    0.1980
Using sympy via SnakeCharmR
## Calculate ETA-based prediction and error derivatives:
Calculate Jacobian.....done.
Calculate sensitivities.....
done.
## Calculate d(f)/d(eta)
## ...
## done
## ...
## done
The model-based sensitivities have been calculated.
It will be cached for future runs.
Calculating Table Variables...
done

```

The options for `saem` are controlled by `saemControl`. You may wish to make sure the minimization is complete in the case of `saem`. You can do that with `traceplot`, which shows the iteration history with burn-in and EM phases. In this case, the burn-in seems reasonable; you may wish to increase the number of iterations in the EM phase of the estimation. Overall it is probably a semi-reasonable solution. (`xpose.nlmixr`'s

prn_vs_iteration() can also do this.)

nlmixr output objects

In addition to unifying the modeling language sent to each of the estimation routines, the outputs currently have a unified structure.

You can see the fit object by typing the object name:

```
> fit.s
nlmixr SAEM fit (ODE)

      OBJF      AIC      BIC Log-likelihood
62335.96 62349.96 62397.88      -31167.98

Time (sec; $time):
      saem setup FOCEi Evaluate covariance table
elapsed 379.32   2.9      1.71      0 19.11

Parameters ($par.fixed):
      Parameter Estimate      SE      CV Untransformed      (95%CI)
lCl      log Cl (L/hr)      1.39 0.0240      1.7%      4.01 (3.82, 4.20)
lVc      log Vc (L)      4.20 0.0256      0.6%      67.0 (63.7, 70.4)
lKA      log Ka (1/hr)      0.00890 0.0307 344.9%      1.01 (0.950, 1.07)
prop.err      0.198      19.8%

Omega ($omega):
      eta.Cl      eta.Vc      eta.KA
eta.Cl 0.06833621 0.00000000 0.000000
eta.Vc 0.00000000 0.07783316 0.000000
eta.KA 0.00000000 0.00000000 0.111673

Fit Data (object is a modified data.frame):
      ID      TIME      DV      IPRED      PRED      IRES      RES      IWRES
1: 1 0.25 204.8 194.859810 198.21076 9.94018953 6.589244 0.25766777
2: 1 0.50 310.6 338.006073 349.28827 -27.40607290 -38.688274 -0.40955290
3: 1 0.75 389.2 442.467750 463.78410 -53.26775045 -74.584098 -0.60809361
---
6945: 120 264.00 11.3 13.840800 70.58248 -2.54080024 -59.282475 -0.92725039
6946: 120 276.00 3.9 4.444197 34.41018 -0.54419655 -30.510177 -0.61851500
6947: 120 288.00 1.4 1.427006 16.77557 -0.02700637 -15.375569 -0.09559342
      WRES      CWRES      CPRED      CRES      eta.Cl      eta.Vc
1: 0.07395107 0.07349997 198.41341 6.38659 0.09153143 0.1366395
2: -0.26081216 -0.27717947 349.82730 -39.22730 0.09153143 0.1366395
3: -0.39860485 -0.42988445 464.55651 -75.35651 0.09153143 0.1366395
---
6945: -0.77916115 -1.34050999 41.10189 -29.80189 0.32007359 -0.1381479
6946: -0.65906613 -1.28359979 15.51100 -11.61100 0.32007359 -0.1381479
6947: -0.56746681 -1.22839732 5.72332 -4.32332 0.32007359 -0.1381479
      eta.KA
1: 0.1369685
2: 0.1369685
3: 0.1369685
---
6945: -0.2381078
```



```
6946: -0.2381078
6947: -0.2381078
```

This example shows the standard printout of an `nlmixr` fit object. The elements of the fit are:

- The type of fit (`nlme`, `saem`, etc)
- Metrics of goodness of fit (AIC, BIC, and `logLik`).
 - To align the comparison between methods, the FOCEi likelihood objective is calculated regardless of the method used and used for goodness of fit metrics.
 - This FOCEi likelihood has been compared to NONMEM's objective function and gives the same values (based on the data in **Wang 2007** (2007))
 - Note that FOCEi is used to calculate the objective function for a `saem` fit.
 - Even though the objective functions are calculated in the same manner, caution should be used when comparing fits from different estimation routines.

The next item is the timing of each of the steps of the fit.

- These can be accessed by (`fit.s$time`).
- As a mnemonic, the access for this item is shown in the printout. This is true for almost all of the other items in the printout.

After the timing of the fit, the parameter estimates are displayed (can be accessed by `fit.s$par.fixed`

- While the items are rounded for R printing, each estimate without rounding is still accessible by the `$` syntax. For example, the `$Untransformed` gives the untransformed parameter values.
- The Untransformed parameter takes log-space parameters and back-transforms them to normal parameters. Not the CIs are listed on the back-transformed parameter space.
- Proportional Errors are converted to %CV on the untransformed space
- Omega block (accessed by `fit.s$omega`)

A table of fit data is also supplied. Please note:

- An `nlmixr` fit object is actually a data frame. Saving it as a Rdata object and then loading it without `nlmixr` will just show the data by itself. The additional fit information is still present, but `nlmixr` must be loaded in order to see it.
- Special access to fit information (like the `$omega`) needs `nlmixr` to extract the information.

If you use the `$` to access information, the order of precedence is:

- Fit data from the overall data.frame
- Information about the parsed `nlmixr` model (via `$uif`)
- Parameter history if available (via `$par.hist` and `$par.hist.stacked`)
- Fixed effects table (via `$par.fixed`)
- Individual differences from the typical population parameters (via `$eta`)
- Fit information from the list of information generated during the post-hoc residual calculation.
- Fit information from the environment where the post-hoc residual were calculated
- Fit information about how the data and options interacted with the specified model (such as estimation options or if the solved system is for an infusion or an IV bolus).

While the printout may display the data as a `data.table` object or `tbl` object, the data is NOT any of these objects, but rather a derived data frame. - Since the object `**is*` a `data.frame`, you can treat it like one.

In addition to the above properties of the fit object, there are a few additional that may be helpful for the modeler:

- `$theta` gives the fixed effects parameter estimates (in NONMEM the `thetas`). This can also be accessed in `nlme` function. Note that the residual variability is treated as a fixed effect parameter and is included in this list.
- `$eta` gives the random effects parameter estimates, or in NONMEM the `etas`. This can also be accessed in using the `random.effects` function.

3.1.2 Some examples

3.1.2.1 A two-compartment PK model

Now let's have a look at some examples to demonstrate the syntax. Here's a relatively straightforward two-compartmental PK model with covariate effects.

```
my2CptModel <- function() {
  ini({
    tka <- log(1.14)
    tc1 <- log(0.0190)
    tv2 <- log(2.12)
    tv3 <- log(20.4)
    tq <- log(0.383)
    wteff <- 0.35
    sexeff <- -0.2
    eta.ka ~ 1
    eta.c1 ~ 1
    eta.v2 ~ 1
    eta.v3 ~ 1
    eta.q ~ 1
    prop.err <- 0.075
  })
  model({
    ka <- exp(tka + eta.ka)
    c1 <- exp(tc1 + wteff*LWT + eta.c1)
    v2 <- exp(tv2 + sexeff*SEX + eta.v2)
    v3 <- exp(tv3 + eta.v3)
    q <- exp(tq + eta.q)
    d/dt(depot) = -ka * depot
    d/dt(center) = ka * depot - c1 / v2 * center + q/v3 * periph - q/v2 * center
    d/dt(periph) = q/v2 * center - q/v3 * periph
    cp = center / v2
    cp ~ prop(prop.err)
  })
}

my2CptFit <- nlmixr(my2CptModel, dat, est="saem")
```

We will fit this model using SAEM, so we have included random effects on every model parameter. We have also chosen to use ODEs rather than `linCmt()`.

Notice how covariates have been included. `lwt` is pre-processed body weight ($\log(\text{WT}/70)$) and `SEX` is a binary variable equal to 0 (male) or 1 (female).

Let's inspect the results...

```
print(my2CptFit)
```

```
## -- nlmixr SAEM fit (ODE); OBFJ calculated from FOCEi approximation -----
##      OBFJ      AIC      BIC Log-likelihood Condition Number
## 3178.892 3204.892 3255.411      -1589.446      650007.1
##
```

Objective function (OBFJ), Akaike information criterion (AIC), Bayesian information criterion (BIC), the log-likelihood of the model given the data, and the condition number are provided at the top of the output block. This more, judging from the condition number (the ratio between the largest and smallest eigenvalues), may be over-parameterized. For this SAEM model, the OBFJ has been calculated using the FOCEi method.

```
## -- Time (sec; $time): -----
##      saem setup Likelihood Calculation covariance table
## elapsed 547.67 23.31      0.72      0 1.81
##
```

Next, we have a block showing benchmarks for the run. This fit took just under 10 minutes in total.

```
## -- Parameters ($par.fixed): -----
##      Estimate      SE      %RSE Back-transformed(95%CI) BSV(CV%)
## tka      0.136    0.0502    36.8      1.15 (1.04, 1.26)    29.9%
## tcl      -2.08    0.0360     1.73     0.125 (0.116, 0.134)    22.5%
## tv2       0.700    0.0330     4.72      2.01 (1.89, 2.15)    23.0%
## tv3       1.69    0.00155   0.0919      5.42 (5.41, 5.44)    18.0%
## tq       -1.22   0.000407   0.0332     0.294 (0.294, 0.294)    30.2%
## wteff     0.765    0.0379     4.96      2.15 (1.99, 2.31)
## sexeff    -0.215    0.0493    23.0     0.807 (0.732, 0.889)
## prop.err   0.0726      7.26%
##      Shrink(SD)%
## tka      13.6%
## tcl       1.78%
## tv2      10.1%
## tv3      20.8%
## tq       6.82%
## wteff
## sexeff
## prop.err   24.6%
##
## No correlations in between subject variability (BSV) matrix
## Full BSV covariance ($omega) or correlation ($omega.R; diagonals=SDs)
## Distribution stats (mean/skewness/kurtosis/p-value) available in $shrink
##
```

The next block provides parameter estimates and associated precisions, between subject variability (BSV) where appropriate, and exponentiated versions of the parameter estimates together with 95% confidence intervals ("back-transformed"). (`nlmixr` assumes that all parameters are log-transformed. Sometimes this is not the case, as illustrated here for `wteff` and `sexeff`.) Shrinkages are also supplied. The fit object has properties that can be interrogated:

- `my2CptFit$par.fixed` provides the fixed-effect parameter estimates
- `my2CptFit$omega` provides the variance-covariance matrix for BSV (`$OMEGA`)
- `my2CptFit$omega.R` provides the correlation matrix
- `my2CptFit$shrink` provides shrinkages and other statistical descriptors of the parameter distributions

```
## -- Fit Data (object is a modified data.frame): -----
## # A tibble: 360 x 28
##   ID     TIME    DV  SEX    WT  PRED    RES    WRES IPRED    IRES    IWRES
## * <fct> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>    <dbl>    <dbl>
## 1 1      0.302  190.   0.   52.  169.   21.4  0.375  191.  -0.578 -0.0417
## 2 1      2.91   296.   0.   52.  389.  -93.1 -1.35   329.  -32.5  -1.36
## 3 1      3.14   312.   0.   52.  379.  -66.7 -1.01   314.  -2.27  -0.0996
## # ... with 357 more rows, and 17 more variables: CPRED <dbl>, CRES <dbl>,
## # CWRES <dbl>, eta.ka <dbl>, eta.cl <dbl>, eta.v2 <dbl>, eta.v3 <dbl>,
## # eta.q <dbl>, depot <dbl>, center <dbl>, periph <dbl>, ka <dbl>,
## # cl <dbl>, v2 <dbl>, v3 <dbl>, q <dbl>, cp <dbl>
```

Finally, a summary of the underlying `tibble` (a modified `data.frame`) is provided - this contains all parameter estimates and diagnostics.

3.1.2.2 Physiologically-based PK

Building on the first simple example, we can be more ambitious, and try a full PBPK model. This one was published for mavoglurant (Wendling et al. 2016).

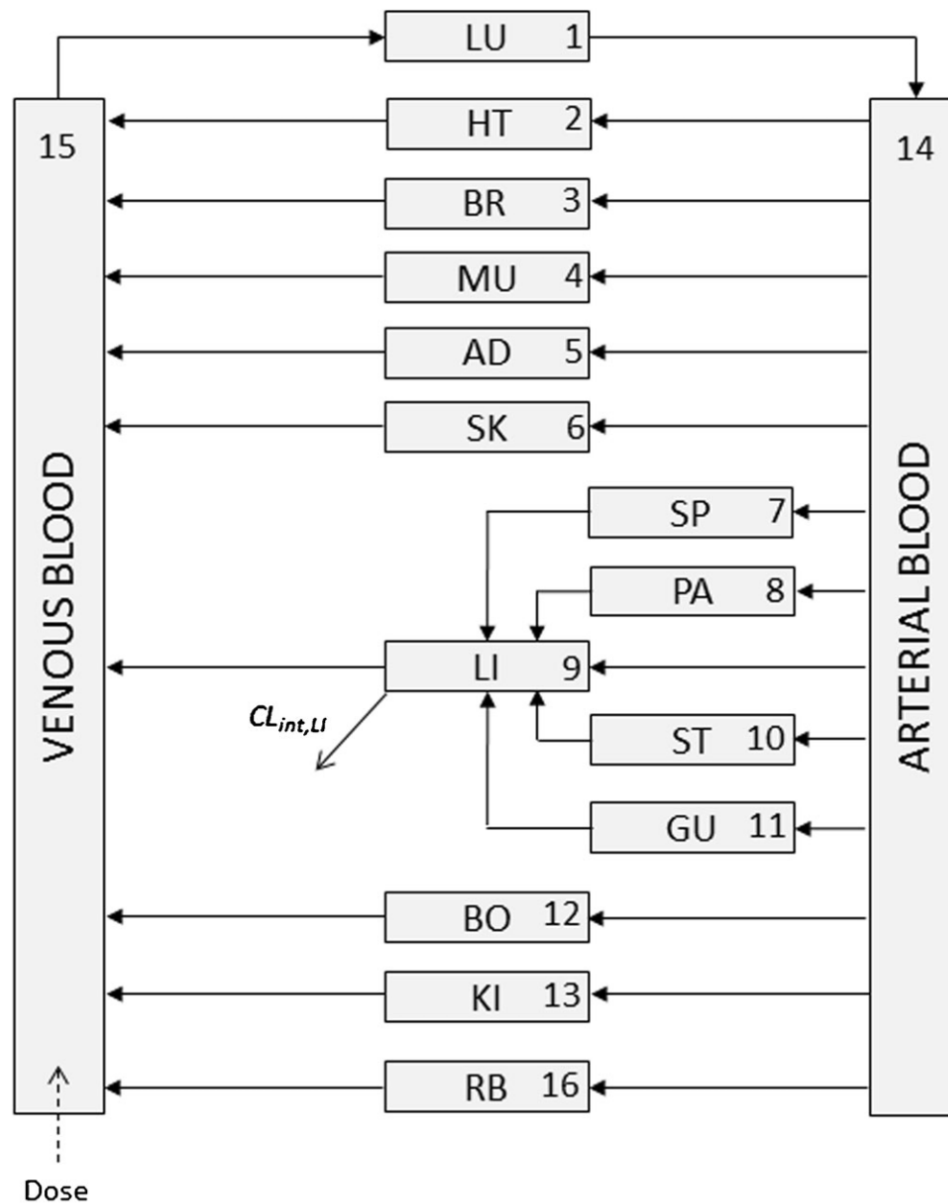


Fig. 1. Schematic representation of the WBPBPK model for MVG. See text for definition of symbols. The *numbers* refer to the state numbers of the system

```
pbpk <- function(){
  ini({
    lKbBR = 1.1
    lKbMU = 0.3
    lKbAD = 2
    lCLint = 7.6
    lKbBO = 0.03
    lKbRB = 0.3
    eta.LCLint ~ 4
    add.err <- 1
    prop.err <- 10
  })
}
```

```

})
model({
  KbBR = exp(lKbBR)
  KbMU = exp(lKbMU)
  KbAD = exp(lKbAD)
  CLint= exp(lCLint + eta.LCLint)
  KbBO = exp(lKbBO)
  KbRB = exp(lKbRB)

  ## Regional blood flows
  CO = (187.00*WT^0.81)*60/1000;          # Cardiac output (L/h) from White et al (1968)
  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 + Q
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*Adi
d/dt(Rest_of_Body)   = QRB*(Arterial_Blood/VAB - Rest_of_Body/KbRB/VRB);

C15 ~ add(add.err) + prop(prop.err)
})
}

```

This model estimates 6 structural parameters, 1 random effect, and includes combined additive and proportional residual error. The dependent variable, C15, is venous blood concentration.

3.1.2.3 Nonlinear PK

Complex models can be estimated as well. In the example below, a target-mediated drug disposition PK model for nimotuzumab is illustrated (Rodríguez-Vera et al. 2015).

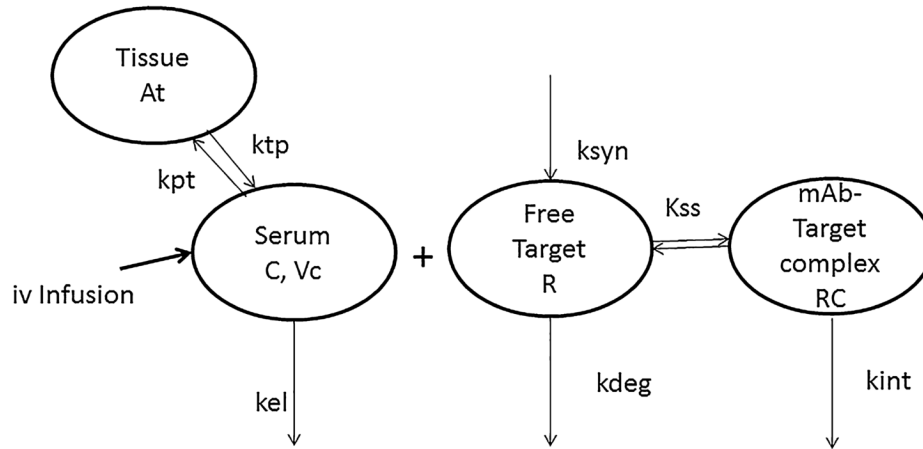


Figure 1. Schematic representation of the QSS approximation of the target-mediated disposition model, which explicitly includes mAb-target binding in central compartment, receptor turnover and internalization of the mAb-target complex. K_{el} , first-order mAb elimination rate constant; k_{pt} , first-order serum-tissue distribution rate constant; k_{tp} , first-order tissue-serum distribution rate constant; A_t , amount of the mAb in the tissue compartment; C , concentration of free mAb in the serum compartment; R , concentration of free target; RC , concentration of the mAb-target complex; k_{syn} , synthesis rate constant of the target; k_{deg} , degradation rate constant of the target; k_{int} , internalization rate constant of the mAb-target complex; K_{ss} , steady-state rate constant.

```
myPKPDMModel <- function() {
  ini({
    tc1 <- log(0.001)
    tv1 <- log(1.45)
    tq <- log(0.004)
    tv2 <- log(44)
    tkss <- log(12)
    tkint <- log(0.3)
    tkdyn <- log(1)
    tkdeg <- log(7)
    eta.cl ~ 2
    eta.v1 ~ 2
    eta.kss ~ 2
    add.err <- 10
  })
  model({
    cl <- exp(tc1 + eta.cl)
    v1 <- exp(tv1 + eta.v1)
    Q <- exp(tq)
    v2 <- exp(tv2)
    kss <- exp(tkss + eta.kss)
    kint <- exp(tkint)
    ksyn <- exp(tkdyn)
    kdeg <- exp(tkdeg)

    k <- cl/v1
    k12 <- Q/v1
    k21 <- Q/v2

    eff(0) <- ksyn/kdeg # initialize compartment
```



```

# Calculate concentration
conc = 0.5*(central/v1 - eff - kss) + 0.5*sqrt((central/v1 - eff - kss)**2 + 4*kss*central/v1)

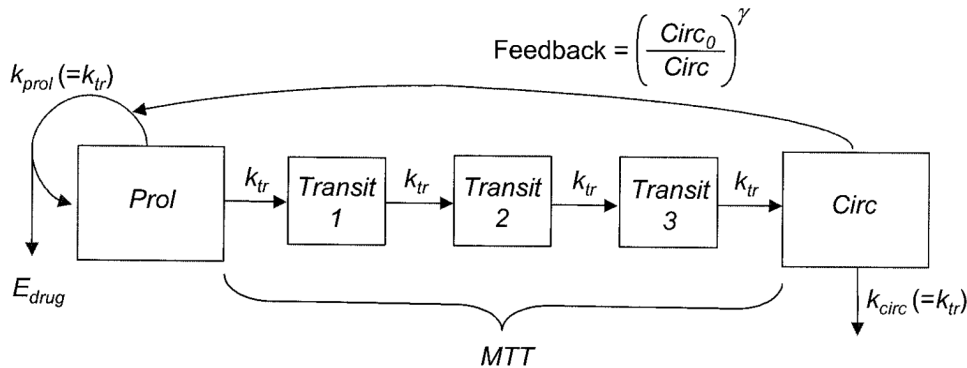
d/dt(central)    = -(k+k12)*conc*v1+k21*peripheral-kint*eff*conc*v1/(kss+conc)
d/dt(peripheral) = k12*conc*v1-k21*peripheral  ##Free Drug second compartment amount
d/dt(eff)        = ksyn - kdeg*eff - (kint-kdeg)*conc*eff/(kss+conc)

IPRED = log(conc)
IPRED ~ add(add.err)
})
}

```

3.1.2.4 PKPD models

Finally, we present an implementation of the Friberg myelosuppression model (Friberg et al. 2002). `nlmixr` is capable of handling almost any kind of model that can be implemented using ODEs.



```

wbc <- function() {
  ini({
    log_CIRC0 <- log(7.21)
    log_MTT   <- log(124)
    log_SLOPU <- log(28.9)
    log_GAMMA <- log(0.239)
    eta.CIRC0 ~ .1
    eta.MTT   ~ .03
    eta.SLOPU ~ .2
    prop.err  <- 10
  })
  model({
    CIRC0 = exp(log_CIRC0 + eta.CIRC0)
    MTT   = exp(log_MTT   + eta.MTT)
    SLOPU = exp(log_SLOPU + eta.SLOPU)
    GAMMA = exp(log_GAMMA)

    # PK parameters from input dataset
    CL = CLI;
    V1 = V1I;
    V2 = V2I;
    Q = 204;

```

```

CONC = A_centr/V1;

# PD parameters
NN    = 3;
KTR    = (NN + 1)/MTT;
EDRUG = 1 - SLOPU * CONC;
FDBK   = (CIRC0 / A_circ)^GAMMA;

CIRC = A_circ;

A_prol(0) = CIRC0;
A_tr1(0)  = CIRC0;
A_tr2(0)  = CIRC0;
A_tr3(0)  = CIRC0;
A_circ(0) = CIRC0;

d/dt(A_centr) = A_periph * Q/V2 - A_centr * (CL/V1 + Q/V1);
d/dt(A_periph) = A_centr * Q/V1 - A_periph * Q/V2;
d/dt(A_prol)   = KTR * A_prol * EDRUG * FDBK - KTR * A_prol;
d/dt(A_tr1)    = KTR * A_prol - KTR * A_tr1;
d/dt(A_tr2)    = KTR * A_tr1 - KTR * A_tr2;
d/dt(A_tr3)    = KTR * A_tr2 - KTR * A_tr3;
d/dt(A_circ)   = KTR * A_tr3 - KTR * A_circ;

CIRC ~ prop(prop.err)
})
}

```

3.2 xpose.nlmixr

3.2.1 Introduction

`xpose.nlmixr` was written to provide an interface between `nlmixr` and the NONMEM-centric `xpose` model diagnostics package developed by Ben Guiastrénec and colleagues at Uppsala University.

3.2.2 Getting started

To get started, first install the package using:

```
devtools::install_github("nlmixrdevelopment/xpose.nlmixr")
```

3.2.3 Overview

To leverage the full power of `xpose` to provide diagnostics for an `nlmixr` model, an `xpose` object must be created from the `nlmixr` fit.

```
xpdb <- xpose_data_nlmixr(nlmixrfit)
```

This `xpose` object can now be used in the same way as any other - for full details, see the `xpose` website at uupharmacometrics.github.io/xpose.

3.3 shinyMixR

3.3.1 Introduction

The `nlmixr` package can be accessed through command line in R and RStudio, and through an interface `shinyMixR` supported by the `nlmixr` team. Both command line and interface support the integration with `xpose.nlmixr`.

`shinyMixR` aims to provide a user-friendly tool for `nlmixr` based on `Shiny`, which facilitates a workflow around an `nlmixr` project. The `shinydashboard` package was used to set up a structure for controlling and tracking runs with an `nlmixr` project, and was the basis for setting up the modular interface.

This project tool enhances the usability and attractiveness of `nlmixr`, facilitating dynamic and interactive use in real-time for rapid model development. In this project-oriented structure, the command line and dashboard can be used independently and/or interdependently. This means that many of the functions within the package can also be used outside the interface within an interactive R session.

Using `shinyMixR`, the user specifies and controls an `nlmixr` project workflow entirely in R. Within a project folder, a structure can be created to include separate folders for models, data and runs. Functionality is available to edit and execute a model code, summarize and compare model outputs in a tabular fashion, and view and compare model development using a tree paradigm. Inputs, outputs and metadata are stored in relation to the model code within the project structure (a discrete R object) to ensure traceability.

Results are visualized using modifications of existing packages (such as `xpose.nlmixr`, user-written functions and packages, or pre-existing plotting functionality included in the `shinyMixR` package. Results are reported using the `R3port` package in pdf and html format.

How the package can be used and what the most important functions are, is described in this section. When working with this package, there are two important assumptions:

1. A specific folder structure is in place. The package uses this structure to read and write certain files (the folder structure can be generated automatically using the `create_proj` function).
2. Within a folder structure, multiple models can be present and is considered a “project”. The package creates and manages a project object which is available in the global environment.

An introduction on how to use the `shinyMixR` interface can be found [here](#). To get started, first install the package and use the library.

3.3.2 Getting started

To get started, first install the package using:

```
devtools::install_github("richardhooijmaijers/shinyMixR")
```

Be aware that the `nlmixr`, `nlmixr.xpose` and `R3port` packages should be installed before installing `shinyMixR`, e.g.:

```
devtools::install_github("richardhooijmaijers/R3port")
devtools::install_github("nlmixrdevelopment/nlmixr")
devtools::install_github("nlmixrdevelopment/xpose.nlmixr")
```

The easiest way to get to know the package is to create the folder structure (which includes some example models):

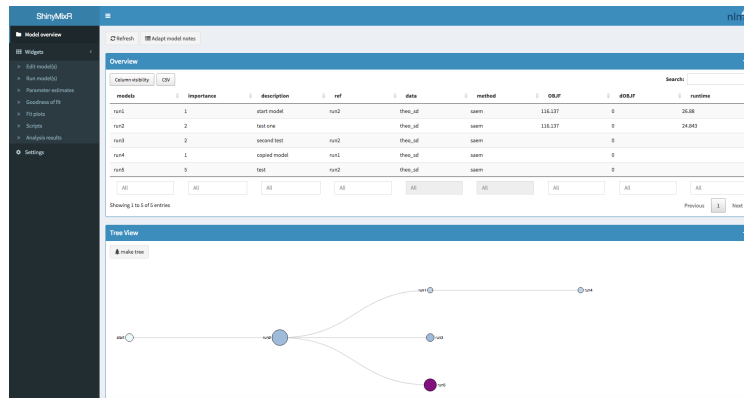


Figure 3.1: shinyMixR interface

```
library(shinyMixR, quietly = TRUE)
create_proj()
```

By default, a folder structure is created within the current directory. The following folders are created:

- **analysis:** in this folder all plots and tables are saved in a structured way to make them accessible to the interface
- **data:** data files used by the models in R data format (.rds) or Excel format (.csv).
- **models:** models, available as separate R scripts according to the `nlmixr` unified-user-interface R model functions.
- **scripts:** generic analysis scripts made available in the interface
- **? shinyMixR:** folder used by the interface to store temporarily files and results files

The interface monitors what happens in these folders. This is important to know because new models and/or data can be copied into these folders or files can be removed. This will then be recognized within the app (after refreshing the overview). This way it is possible to work on data and models separately and plug them in to `shinyMixR` at a later stage.

Once there is a folder structure present the interface can be started:

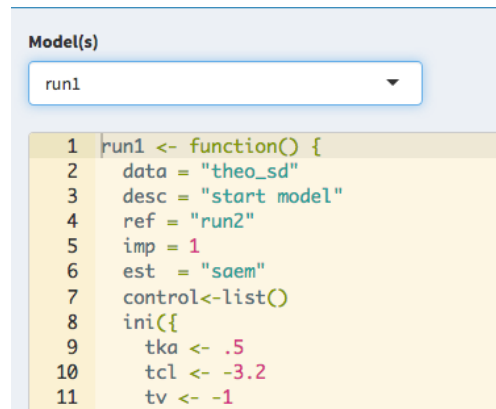
```
run_shinymixr(launch.browser = TRUE)
```

The interface will be started and a project object will be created in the global environment in which all information is kept/managed. If correct, the interface will open in the default browser and the following will be seen:

On the left side, there is a sidebar with various menu items. The content of the main body will open with the model overview but will change based on the selected menu item in the sidebar. The sidebar can be collapsed by clicking the three lines in the top bar, providing more room for the main body.

3.3.3 Overview

The overview page can be used to see which models are present in a project, the relationship between models and to adapt model meta data. The overview can be exported to a CSV file and a selection of columns can be made that should be displayed (all using the DT package).



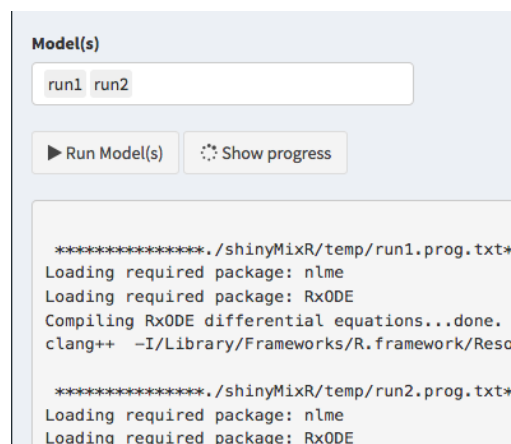
The screenshot shows a web interface for editing model code. At the top, a dropdown menu labeled 'Model(s)' has 'run1' selected. Below it, a code editor displays the following R code:

```

1 run1 <- function() {
2   data = "theo_sd"
3   desc = "start model"
4   ref = "run2"
5   imp = 1
6   est = "saem"
7   control<-list()
8   ini({
9     tka <- .5
10    tcl <- -3.2
11    tv <- -1

```

Figure 3.2: Editing model code



The screenshot shows the 'Model(s)' dropdown menu with 'run1' and 'run2' selected. Below the dropdown are two buttons: 'Run Model(s)' and 'Show progress'. A terminal window below these buttons displays the following output:

```

*****./shinyMixR/temp/run1.prog.txt*
Loading required package: nlme
Loading required package: RxODE
Compiling RxODE differential equations...done.
clang++ -I/Library/Frameworks/R.framework/Reso

*****./shinyMixR/temp/run2.prog.txt*
Loading required package: nlme
Loading required package: RxODE

```

Figure 3.3: Executing model code

3.3.4 Edit models

The edit tab can be used to edit existing models within an editor including syntax coloring (using **shinyAce**). It is also possible to create new models using various templates or to duplicate existing models.

3.3.5 Run models

The run tab can be used to execute models within a project. It is possible to run one or multiple models at once. Also it is possible to assess the intermediate output or progress of an **nlmixr** run.

3.3.6 Parameter estimates

The parameter estimates tab can be used to generate a table with parameter estimates. In case multiple models are selected the table will show the results of each run in a separate column. This page is reactive which means that in case a different model is selected, the table is directly updated. There is the possibility to save the table to a latex/pdf or html file.

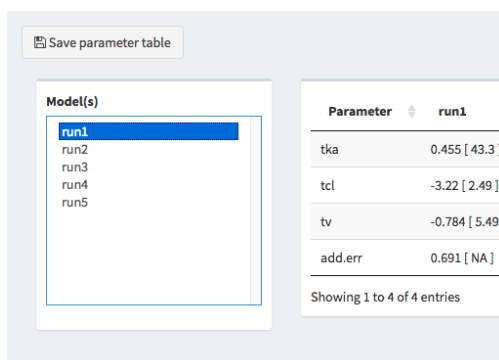


Figure 3.4: View parameter estimates



Figure 3.5: Creating goodness-of-fit plots



Figure 3.6: Creating individual fit plots

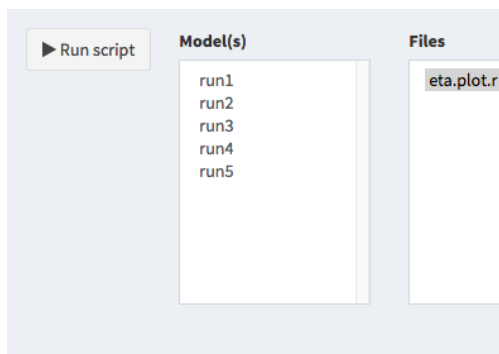


Figure 3.7: Run user defined scripts

3.3.7 Goodness-of-fit

The goodness-of-fit tab can be used to generate a combination of 4 goodness-of-fit plots combined. By default `nlmixr.xpose` is used but direct `ggplot2` can also be used by specifying this in the settings. Plots can be saved to a LaTeX/pdf or html file.

3.3.8 Fit plots

The fit plots tab can be used to generate individual fit plots. The same options are present here as for the goodness-of-fit plots.

3.3.9 Scripts

It is possible to write your own scripts that can be used to analyse model results. This script can be used to process the result for one or multiple models at once (the interface will include the name of the selected models in the script). An example of how such a script will look like is included in the package.

3.3.10 Results

It is possible to view and combine the results from the models within a project within the last tab.

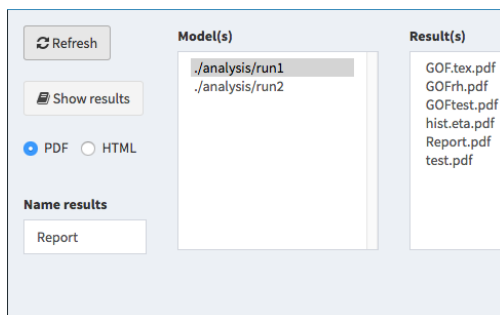


Figure 3.8: View and combine modeling results

3.3.11 Interactive session

When working in an interactive R session, many functions used by the interface are available from an interactive R session. When working outside the interface it is important to know how to interact with the project object. This object should be created as one of the first steps because other function rely on the availability of this object:

```
proj_obj <- get_proj()
```

This function will look in the folder structure to create or update the available information. The result is a list that is build-up as follows:

```
object
|--- run 1
|   |--- model location
|   |--- model meta data
|   |--- model high level results
|
|--- ...
|
|--- meta data (time of last refresh)
```

In case this object is not present it will be created by looking at the files present in the current folder structure. In case the object is already present it will check if newer files are present in the current folder and will update the object with this information. Therefore one has to be aware that this function should be submitted manually in case new information is present in one of the folders.

Once a folder structure is in place and the project object is created, an overview can be created for the available models and when models are executed and running high level results can be viewed. It is also possible to display a collapsible d3 tree view of the models. This is mainly useful in case reference to models is provided to show the hierarchy of the models within a project:

```
overview()
tree_overview()
```

Although the `nlmixr` package has the possibility to run `nlmixr` models, the `shinyMixR` package has a function available named `run_nmx`. This function allows to run the model in an external R session when executing a run through `shinyMixR`. This is necessary within the interface to overcome the application from freezing when a model is submitted. Also within an interactive R session it is convenient because you do not have to wait for a run to finish. An example on how this can be used is given below:


```
#proj_obj <- get_proj()
#run_nmx("run1",proj=proj_obj)
```

```
run_nmx("run1",proj=proj_obj)
# progress of a run is written to external text file
# this can be read-in for intermediate assessment
readLines("shinyMixR/temp/run1.prog.txt")
```

The current version of the package has three functions for assessing the model results: `par_table`, `gof_plot` and `fit_plot`. The first function is to create a simple parameter table `par_table`. By default this function returns a data.frame. In case multiple models are selected, each column will have the results of the selected model. The results can also be written to a PDF (using LaTeX) or html file using the `R3port` package:

```
par_table(proj_obj,c("run1","run2"))
# par_table(proj_obj, models="run1", outnm="par.tex")
```

For assessing the goodness of fit, the `gof_plot` function can be used. This function will by default use the `nlmixr.xpose` package to create 4 different types of plots. It is also possible to directly create `ggplot2` types of plots. By default the plots will be created within the R session but can also be written to pdf/html using the `R3port` package:

```
res <- readRDS("./shinyMixR/run1.res.rds")
gof_plot(res)
# gof_plot(res, mdlnm="run1", outnm="gof.tex")
```

The last plot is an individual fit plot `fit_plot`. This function will by default use the `nlmixr.xpose` package to create a plot per individual including the observed data, individual and population predictions. Also here it is possible to create `ggplot2` types of plots and plots can be outputted to pdf/html:

```
fit_plot(res, type="user")
# fit_plot(res, mdlnm="run1", outnm="fit.html")
```


Chapter 4

Applications of `nlmixr`

4.1 Posters and Presentations

A series of posters were presented at different conferences where `nlmixr` was compared to `NONMEM`. These findings provide evidence that `nlmixr` provides a viable open-source parameter estimation alternative for fitting nonlinear mixed effects pharmacometric models within the R environment.

- ACoP 2016, Seattle, USA: PosterACoP2016
- WCoP 2016, Brisbane, Australia: PosterWCoP2016
- PAGE 2017, Budapest, Hungary: PosterPAGE2017
- ACoP 2017, Fort Lauderdale, USA: PosterACoP2017

On 8 December 2016, Rik Schoemaker presented a seminar on `nlmixr` at Uppsala University with the title: **`nlmixr`: an open-source package for pharmacometric modelling in R** (PresentationUppsala161208).

`nlmixr` was presented at the 12th Pharmacometrics Network Benelux Meeting (29 March 2018) with the title: **Simulation (R_xODE) and parameter estimation in `nlmixr`** (PresentationPNB180326).

4.2 Sparse data analysis with `nlmixr`

4.2.1 Examination of `nlmixr` estimation algorithm properties for sparse sample data

The NLME and SAEM parameter estimation algorithms were compared with `NONMEM` FOCEi in a sparse-sampling data setting. To this end, 10,000 patients were simulated after 7 doses with 24 hour intervals with doses split between 10, 30, 60 and 120 mg. Four time points were randomly sampled in the 24 hours after the last dose. A first order absorption, one compartment distribution, and linear elimination model was used with population values of Clearance=4.0 L/hr, Vc=70 L, and KA=1 /hr, 30% IIV for all three parameters (diagonal omega matrix), and 20% residual variability. Of these 10,000 patients, 600 patients were randomly sampled (stratified by dose, 150 subjects per dose), 500 times using PsN and analysed using `NONMEM`.

The code for the full analysis is provided here along with some output graphs to demonstrate the results. While these are interesting in their own right, the code also demonstrates how to perform parallel analysis of the estimations; with 500 datasets per analysis, it makes perfect sense to be able to run these analyses side-by-side providing you have access to a computer with multiple cores. As these approaches are completely OS-dependent, the code below for running in parallel is only applicable to Windows.

First, we load the packages and define the model using ODEs. Note that `data.table` is needed to run these examples.

```
library(nlmixr)
library(data.table)

#Define the RxODE model
ode4 <- "
d/dt(abs)    = -KA*abs;
d/dt(centr)  = KA*abs-(CL/V)*centr;
C2=centr/V;
"

#Create the RxODE simulation object
mod4 <- RxODE(model = ode4, modName = 'mod4')
```

Generate the 10,000 sampled parameters:

```
#Population parameter values on log-scale
params1 <- c(CL = log(4),
             V = log(70),
             KA = log(1))
#make 10,000 subjects to sample from:
nsubg <- 2500 # subjects per dose
doses <- c(10, 30, 60, 120)
nsub <- nsubg * length(doses)
#IIV of 30% for each parameter
omega <- diag(c(0.09, 0.09, 0.09)) # IIV covariance matrix
sigma <- 0.2
#Sample from the multivariate normal
set.seed(98176247)
library(MASS)
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(params1['CL'] + mv[, 1]),
    "V" = exp(params1['V'] + mv[, 2]),
    "KA" = exp(params1['KA'] + mv[, 3])
  )
#set the doses (looping through the 4 doses)
params.all[, AMT := 1000 * doses]
```

Then we run the simulation of all these profiles. Here we use `lapply`:

```
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 = 7,
  dosing.to = 1,
  dosing.interval = 24,
  rate = NULL,
  start.time = 0
)
#generates 4 random samples in a 24 hour period for the last dose
ev$add.sampling(6 * 24 + c(0, sort(round(sample(runif(600, 0, 1440), 4) / 60, 2))))
#runs the RxODE simulation
x <- as.data.table(mod4$run(params, ev))
#merges the parameters and ID number to the simulation output
x[, names(params) := params]
})

#runs the entire sequence of 10000 subjects and binds the results to the object res
res = as.data.table(do.call("rbind", s))

Stoplaply <- Sys.time()

Stoplaply - Startlaply
#10,000 subjects simulated in:
#Time difference of 29.36007 secs

```

Clean up the results and prepare for analysis using NONMEM:

```

setnames(res, "time", "TIME")
#administered dose:
Dose <- expand.grid(TIME = seq(0, 6 * 24, 24), ID = params.all$ID)
Dose <- data.table(merge(Dose, params.all, by = "ID"))
Dose[, C2 := 0]
Dose[, EVID := 101]
Dose[, DOSE := AMT / 1000]
res[, EVID := 0]
res[, centr := NULL]
res[, abs := NULL]
res[, DOSE := AMT / 1000]
res[, AMT := 0]
#take out the 'trough' (sampling)timepoint used for dosing in this case
res <- res[TIME != 144]
res <- rbind(res, Dose)
setkey(res, ID, TIME)
#Add residual error
res[, DV := C2 * exp(rnorm(length(C2), 0, sigma))]
res[, C2 := NULL]
res[, DV := round(DV)]
#NONMEM EVID is just 1 instead of the RxODE EVID of 101
res[, EVIDNM := as.numeric(EVID == 101)]
res <- res[, .(ID, DOSE, V, CL, KA, TIME, EVID, AMT, DV, EVIDNM)]
write.table(res, file="FullSIM160817.csv", sep=";", col.names=TRUE, quote=FALSE, row.names=FALSE)

```

Then `PsN` is used to sample 600 subjects stratified by dose from the 10,000 subjects and analyse these with `NONMEM`. This is repeated 500 times using `PsN` bootstrap functionality, creating both 500 sets of output and 500 data files to be analysed using `nlmixr` with the following `PsN` syntax:

```
bootstrap runN024.mod -samples=500 -sample_size=600 -stratify_on=DOSE -no-run_base_model -seed=12345 -t
```

and the following `NONMEM` syntax file:

```
$PROB    ORAL1_1CPT_KAVCL MULTIPLE DOSE FOCEI runN024
$INPUT    ID DOSE VI CLI KAI TIME EVIDNLMX AMT DV EVID
$DATA     FullSIM160817.csv IGNORE=@
$SUBR     ADVAN2,TRANS2
$PK
CL=EXP(THETA(1)+ETA(1))
V=EXP(THETA(2)+ETA(2))
KA=EXP(THETA(3)+ETA(3))
S2=V
$ERROR
IPRED = F
RESCV = THETA(4)
W      = IPRED*RESCV
IRES   = DV-IPRED
IWRES  = IRES/W
Y      = IPRED+W*EPS(1)
$THETA   1.6          ;CL
$THETA   4.5          ;V
$THETA   0.2          ;Ka
$THETA   (0,0.3,1) ;RSV
$OMEGA   0.15 0.15 0.15
$SIGMA   1 FIX
$EST      NSIG=3 PRINT=5 MAX=9999 NOABORT POSTHOC METHOD=COND INTER NOOBT
$COV
```

The `NONMEM` results will be provided separately, along with the `PsN`-generated data files that will be analysed in `nlmixr`. A function `do_nlmixr` is created to read in the data file, define the model, run the parameter estimation, and then save the output file, to be analysed at a later date.

The code to analyse these data sets using `SAEM` with the solved equations implementation is:

```
#SAEM with solved equations:

do_nlmixr <- function(i) {
  datr <-
    read.csv(
      paste("D:\\nmrun\\nmrun1\\m1\\bs_pri_", i, ".dta", sep = ""),
      header = TRUE,
      stringsAsFactors = F
    )

  #If using Microsoft R Open, you need to specify:
  #setMKLthreads(1)
  #Otherwise it accesses too much resources without any gain in speed
```

```

one.compartment.oral.model.solved <- function() {
  ini({
    # Where initial conditions/variables are specified
    # '<-' or '=' defines population parameters
    # Simple numeric expressions are supported
    lCl <- 1          #log Cl (L/hr)
    lVc <- 4          #log V (L)
    lKA <- 0.1        #log V (L)
    # Bounds may be specified by c(lower, est, upper), like NONMEM:
    # Residuals errors are assumed to be population parameters
    prop.err <- c(0, 0.2, 1)
    # Between subject variability estimates are specified by '~'
    # Semicolons are optional
    eta.Cl ~ 0.1
    eta.Vc ~ 0.1
    eta.KA ~ 0.1
  })
  model({
    # Where the model is specified
    # The model uses the ini-defined variable names
    Cl <- exp(lCl + eta.Cl)
    Vc <- exp(lVc + eta.Vc)
    KA <- exp(lKA + eta.KA)
    # Solved equations:
    linCmt() ~ prop(prop.err)
  })
}

fit <-
  nlmixr(
    one.compartment.oral.model.solved,
    datr,
    est = "saem",
    control = saemControl(print = 50)
  )

save(fit, file = paste("fit_SAEM_Solved_UUI_", i, ".Rdata", sep = ""))
}

```

The code for SAEM with an ODE implementation is:

```

#SAEM with ODE:

do_nlmixrODE <- function(i) {
  datr <-
    read.csv(
      paste("D:\\nmrun\\nmrun1\\m1\\bs_pr1_", i, ".dta", sep = ""),
      header = TRUE,
      stringsAsFactors = F
    )

  #If using Microsoft R Open, you need to specify:
  #setMKLthreads(1)

```

```

#Otherwise it accesses too much resources without any gain in speed

one.compartment.oral.model <- function() {
  ini({
    # Where initial conditions/variables are specified
    # '<-' or '=' defines population parameters
    # Simple numeric expressions are supported
    lCl <- 1          #log Cl (L/hr)
    lVc <- 4          #log V (L)
    lKA <- 0.1        #log V (L)
    # Bounds may be specified by c(lower, est, upper), like NONMEM:
    # Residuals errors are assumed to be population parameters
    prop.err <- c(0, 0.2, 1)
    # Between subject variability estimates are specified by '~'
    # Semicolons are optional
    eta.Cl ~ 0.1
    eta.Vc ~ 0.1
    eta.KA ~ 0.1
  })
  model({
    # Where the model is specified
    # The model uses the ini-defined variable names
    Cl <- exp(lCl + eta.Cl)
    Vc <- exp(lVc + eta.Vc)
    KA <- exp(lKA + eta.KA)
    # RxODE-style differential equations are supported
    d / dt(depot)    = -KA * depot

    d / dt(centr)    = KA * depot - (Cl / Vc) * centr

    ## Concentration is calculated
    cp = centr / Vc

    # And is assumed to follow proportional error estimated by prop.err
    cp ~ prop(prop.err)

  })
}

fit <-
  nlmixr(
    one.compartment.oral.model,
    datr,
    est = "saem",
    control = saemControl(print = 50)
  )

save(fit, file = paste("fit_SAEM_ODE_UUI_", i, ".Rdata", sep = ""))
}

```

NLME with solved equations:


```

#nlme with solved equations:

do_nlmixr_nlme <- function(i) {
  datr <-
    read.csv(
      paste("D:\\nmrun\\nmrun1\\m1\\bs_pr1_", i, ".dta", sep = ""),
      header = TRUE,
      stringsAsFactors = F
    )

  one.compartment.oral.model.solved <- function() {
    ini({
      # Where initial conditions/variables are specified
      # '<-' or '=' defines population parameters
      # Simple numeric expressions are supported
      lCl <- 1          #log Cl (L/hr)
      lVc <- 4          #log V (L)
      lKA <- 0.1        #log V (L)
      # Bounds may be specified by c(lower, est, upper), like NONMEM:
      # Residuals errors are assumed to be population parameters
      prop.err <- c(0, 0.2, 1)
      # Between subject variability estimates are specified by '~'
      # Semicolons are optional
      eta.Cl ~ 0.1
      eta.Vc ~ 0.1
      eta.KA ~ 0.1
    })
    model({
      # Where the model is specified
      # The model uses the ini-defined variable names
      Cl <- exp(lCl + eta.Cl)
      Vc <- exp(lVc + eta.Vc)
      KA <- exp(lKA + eta.KA)
      # Solved equations:
      linCmt() ~ prop(prop.err)
    })
  }

  fit <-
    nlmixr(
      one.compartment.oral.model.solved,
      datr,
      est = "nlme",
      control = nlmeControl(pnlsTol = .1)
    )

  save(fit, file = paste("fit_NLME_Solved_UUI_", i, ".Rdata", sep = ""))
}

```

and finally nlme with ODE:

#nlme with ODE:

```
do_nlmixrODE_nlme <- function(i) {
  datr <-
    read.csv(
      paste("D:\\nmrun\\nmrun1\\m1\\bs_pr1_", i, ".dta", sep = ""),
      header = TRUE,
      stringsAsFactors = F
    )

  one.compartment.oral.model <- function() {
    ini({
      # Where initial conditions/variables are specified
      # '<-' or '=' defines population parameters
      # Simple numeric expressions are supported
      lCl <- 1          #log Cl (L/hr)
      lVc <- 4          #log V (L)
      lKA <- 0.1        #log V (L)
      # Bounds may be specified by c(lower, est, upper), like NONMEM:
      # Residuals errors are assumed to be population parameters
      prop.err <- c(0, 0.2, 1)
      # Between subject variability estimates are specified by '~'
      # Semicolons are optional
      eta.Cl ~ 0.1
      eta.Vc ~ 0.1
      eta.KA ~ 0.1
    })
    model({
      # Where the model is specified
      # The model uses the ini-defined variable names
      Cl <- exp(lCl + eta.Cl)
      Vc <- exp(lVc + eta.Vc)
      KA <- exp(lKA + eta.KA)
      # RxODE-style differential equations are supported
      d / dt(depot) = -KA * depot

      d / dt(centr) = KA * depot - (Cl / Vc) * centr

      ## Concentration is calculated
      cp = centr / Vc

      # And is assumed to follow proportional error estimated by prop.err
      cp ~ prop(prop.err)
    })
  }

  fit <-
    nlmixr(
      one.compartment.oral.model,
      datr,

```

```

    est = "nlme",
    control = nlmeControl(pnlsTol = .1)
  )

  save(fit, file = paste("fit_NLME_ODE_UUI_", i, ".Rdata", sep = ""))
}

```

To run these analyses in parallel, you need to set up a local virtual cluster using the `doParallel` package, in this case with 15 cores, but adjust to your own hardware:

```

#install.packages("doParallel")
library(doParallel)
cl <- makeCluster(15)
registerDoParallel(cl)

```

And then run the 500 analyses using `foreach` syntax. SAEM with ODEs takes a long time for 600 subjects and 4 samples per subject, and so this example only runs 15 analyses:

```

timeS_Sparse <- Sys.time()
nlmixr_out <-
  foreach(i = 1:15, .packages = c('nlmixr')) %dopar% do_nlmixrODE(i)
timeE_Sparse <- Sys.time()
timeE_Sparse - timeS_Sparse
#Time difference of 1.216711 hours for only 15 runs in parallel on a 15 core cluster

```

But SAEM with solved equations is much faster!

```

timeS_Sparse <- Sys.time()
nlmixr_out <-
  foreach(i = 1:500, .packages = c('nlmixr')) %dopar% do_nlmixr(i)
timeE_Sparse <- Sys.time()
timeE_Sparse - timeS_Sparse
#Time difference of 3.656132 hours

```

NLME with solved systems is even faster:

```

timeS_Sparse <- Sys.time()
nlmixr_out <-
  foreach(i = 1:500, .packages = c('nlmixr')) %dopar% do_nlmixr_nlme(i)
timeE_Sparse <- Sys.time()
timeE_Sparse - timeS_Sparse
#Time difference of 43.58214 mins

```

and NLME with ODEs is still very doable:

```

timeS_Sparse <- Sys.time()
nlmixr_out <-
  foreach(i = 1:500, .packages = c('nlmixr')) %dopar% do_nlmixrODE_nlme(i)
timeE_Sparse <- Sys.time()
timeE_Sparse - timeS_Sparse
#Time difference of 2.432612 hours

```

You can then read in the output from the 500 `nlmixr` analyses. With the new unified user interface, a single read routine suffices!

```
Read_nlmixr <- function(Identifier) {
  for (i in 1:500) {
    filename <- paste(Identifier, "_", i, ".Rdata", sep = "")

    if (file.exists(filename)) {
      load(filename)
      TMSE <- fit$par.fixed$SE
      TM <- fit$theta
      names(TMSE) <- paste(names(TM), "_SE", sep = "")
      Time <- c("Time" = fit$table.time["elapsed"])
      IIV <- sqrt(diag(fit$omega))
      run <- c("Run" = i)
      MISSING_CWRES <- c("MISSING_CWRES" = sum(is.na(fit$CWRES)))
      TM <- c(run, TM, TMSE, Time, IIV, MISSING_CWRES)
      TM <- as.data.frame(t(TM))
      fit <- NULL
      print(i)
      if (i == 1) {
        nlmixrparams <- TM
      } else {
        nlmixrparams <- rbind(nlmixrparams, TM)
      }
    }
  }

  save(nlmixrparams, file = paste(Identifier, ".Rdata", sep = ""))
}

Read_nlmixr(Identifier = "fit_NLME_Solved_UUI")
#Read_nlmixr(Identifier = "fit_NLME_ODE_UUI")
#Read_nlmixr(Identifier = "fit_SAEM_Solved_UUI")
```

These results can then be compared with the NONMEM output and plotted to see the results. Let's start with a comparison of `nlme` results and NONMEM estimates. The results generated for `nlme` using solved equations are provided in Figure 4.1.

Estimates for population parameters (CL, Vc, Ka) match well between NONMEM and NLME, and IIV for CL looks fine as well. However, when IIVs become a bit more difficult to estimate, like for Vc, and for Ka especially, NLME tends to return IIV values of zero quite frequently.

The next step is to compare the results for NLME with models implemented using solved equations vs. the ODE implementation (Figure 4.2).

As is clear from the graph, there is some discrepancy between the outcomes of the two model-definition methods, but this is to be expected with any numerical approach.

Next, SAEM results are compared with NONMEM FOCEi. At this stage, estimating ODE models with a large number of subjects is prohibitively time-consuming using SAEM, and so only SAEM with solved solution results are presented (Figure 4.3).

These results show a near perfect match between NONMEM and `nlmixr`/SAEM population estimates, and a very good match for IIV estimates as well, where it is noted that for Ka, NONMEM estimates a number

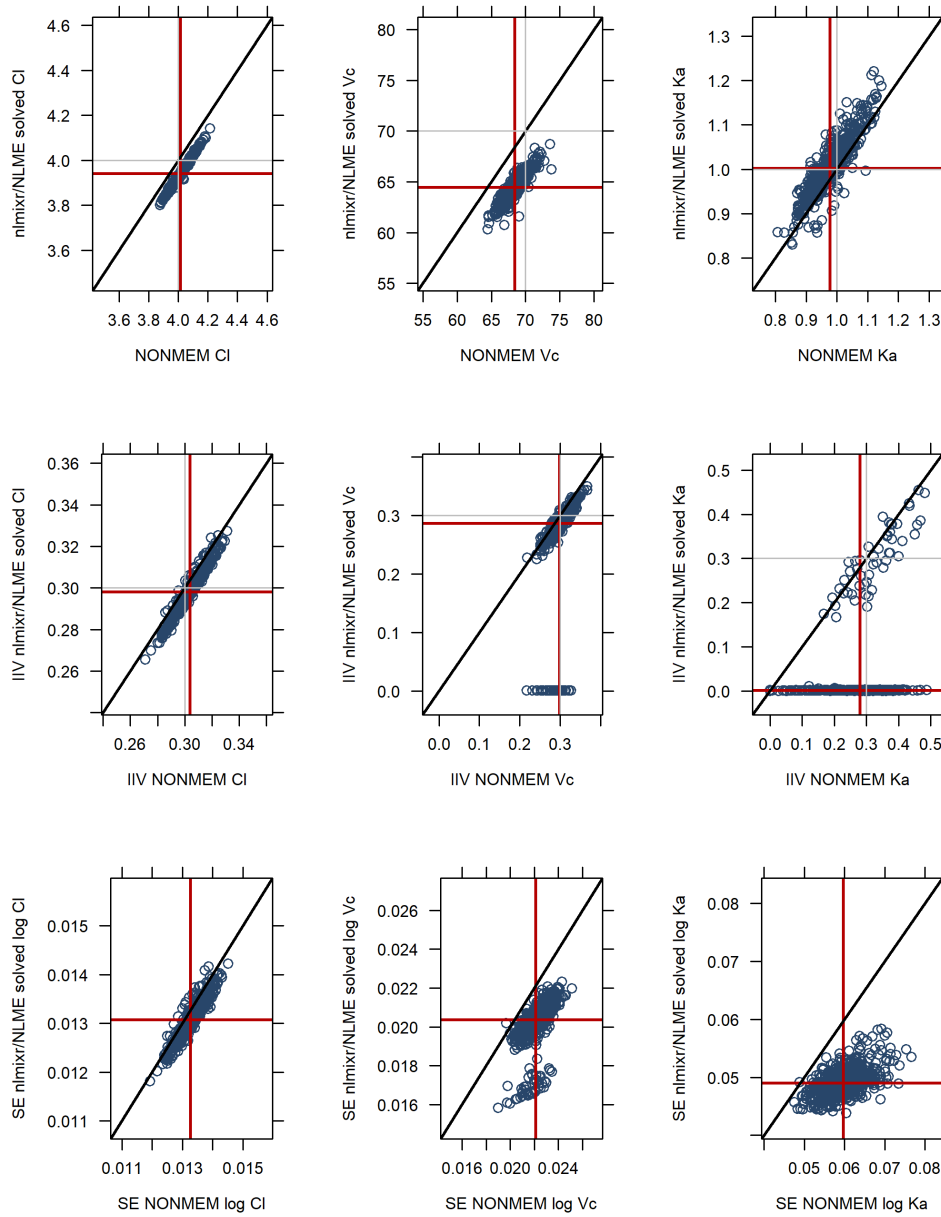


Figure 4.1: Sparse data analysis results: NONMEM FOCE-I solved vs. nlmixr/NLME solved. CI (left column), Vc (middle column), and Ka (right column), for the population parameter (top row), IIV (middle row), and SE of the log population estimate (bottom row)

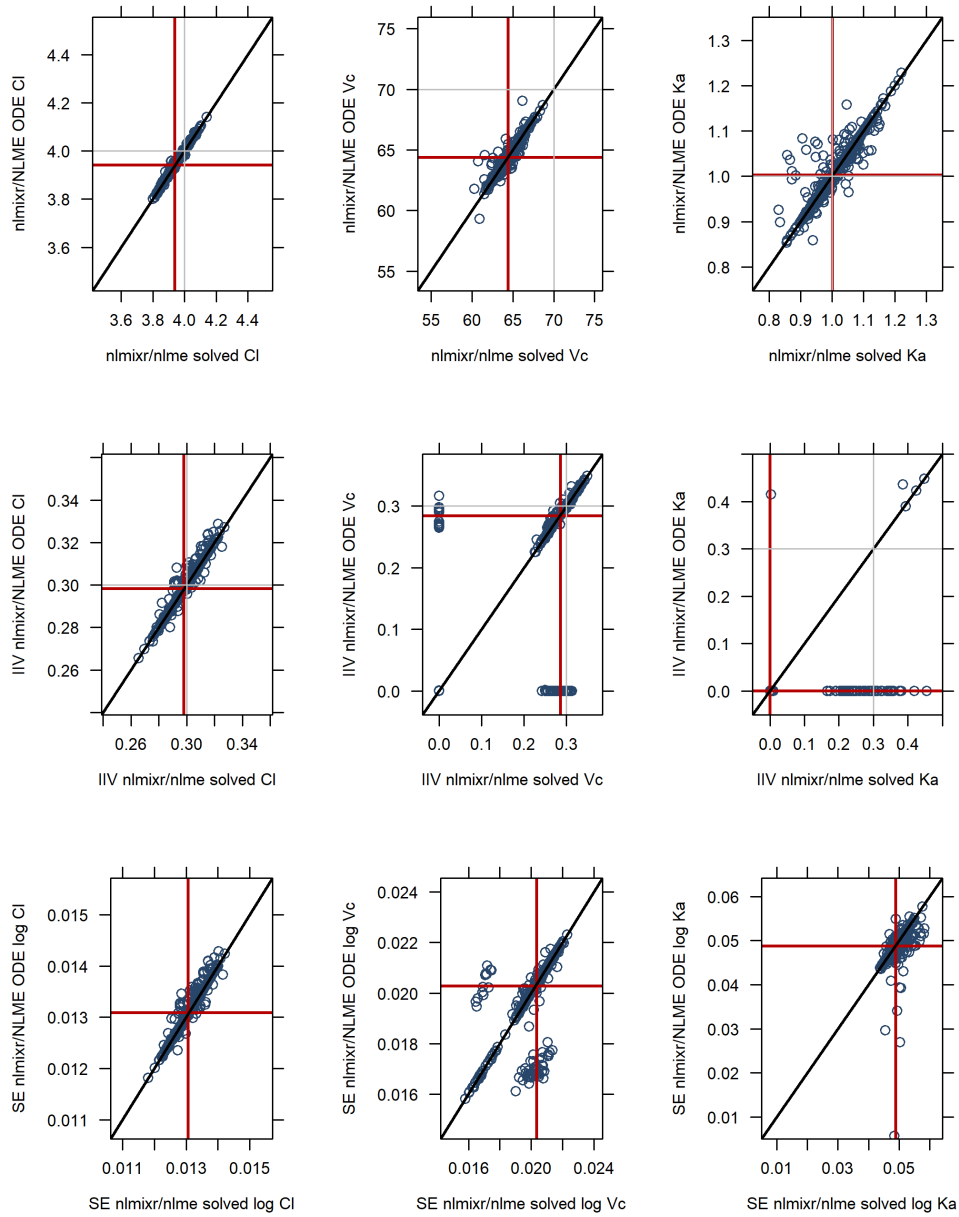


Figure 4.2: Sparse data analysis results: `nlmixr/NLME solved` vs. `nlmixr/NLME ODE`. CI (left column), Vc (middle column), and Ka (right column), for the population parameter (top row), IIV (middle row), and SE of the log population estimate (bottom row)

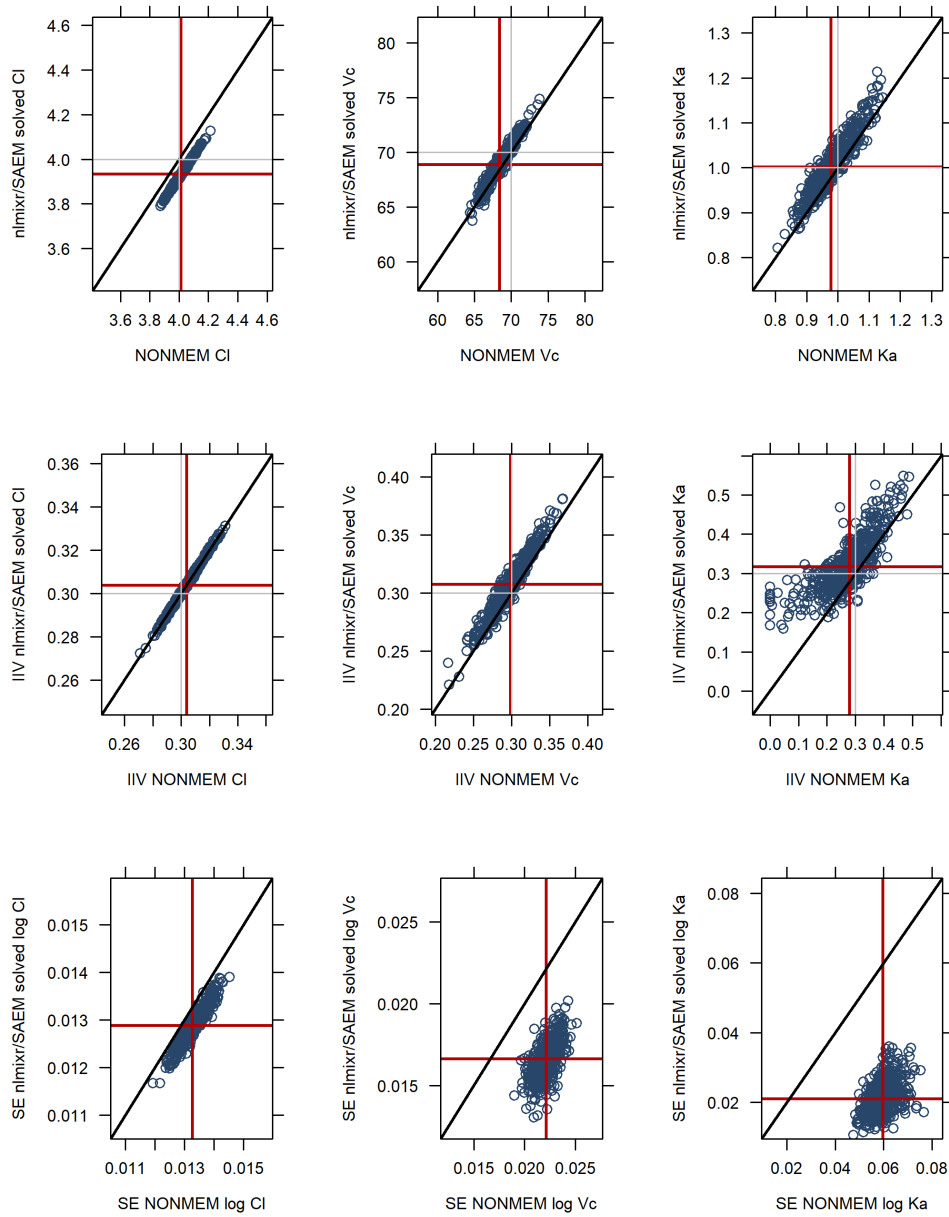


Figure 4.3: Sparse data analysis results: NONMEM FOCE I solved vs. nlmixr/SAEM solved. CI (left column), Vc (middle column), and Ka (right column), for the population parameter (top row), IIV (middle row), and SE of the log population estimate (bottom row)

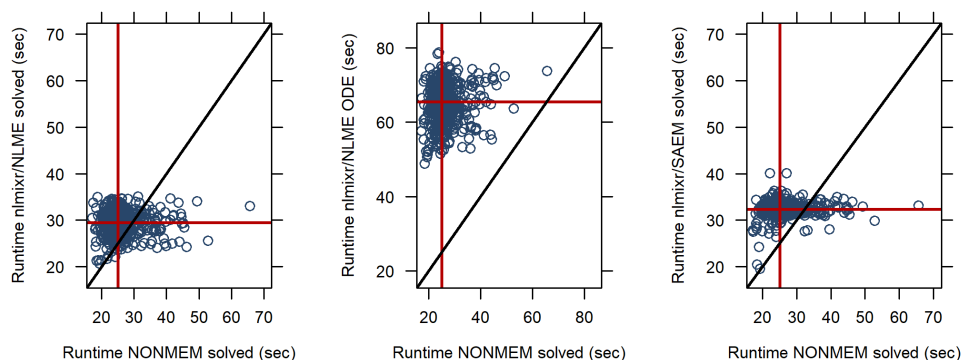


Figure 4.4: Comparison of runtimes vs NONMEM FOCE I: nlmixr/NLME solved (left), nlmixr/NLME ODE (middle), SAEM solved (right)

of IIVs at zero, while SAEM never does this. The standard errors for NONMEM are larger than for SAEM in all cases, but it is difficult to say which one is correct: for population parameters and IIVs, the values simulated from are known, but for standard errors this is not the case.

Finally, for a comparison between methods in terms of speed of calculations see Figure 4.4. It is clear that NONMEM is faster in virtually all cases (and SAEM with ODEs is not even reported) and these comparisons are with single-thread NONMEM. However, speeds are not so prohibitively high that use in daily practice is crippled.

4.3 Course: PAGE 2018

The nlmixr team presented a one-day course at PAGE 2018, in Montreux, Switzerland. The examples used in the course are presented [here](#).

Friberg, Lena E., Anja Henningsson, Hugo Maas, Laurent Nguyen, and Mats O Karlsson. 2002. “Model of chemotherapy-induced myelosuppression with parameter consistency across drugs.” *Journal of Clinical Oncology* 20 (24): 4713–21. doi:[10.1200/JCO.2002.02.140](#).

Rodríguez-Vera, Leyanis, Mayra Ramos-Suzarte, Eduardo Fernández-Sánchez, Jorge Luis Soriano, Concepción Peraire Guitart, Gilberto Castañeda Hernández, Carlos O. Jacobo-Cabral, Niurys De Castro Suárez, and Helena Colom Codina. 2015. “Semimechanistic model to characterize nonlinear pharmacokinetics of nimotuzumab in patients with advanced breast cancer.” *Journal of Clinical Pharmacology* 55 (8): 888–98. doi:[10.1002/jcph.496](#).

Wang, Yaning. 2007. “Derivation of various NONMEM estimation methods.” *Journal of Pharmacokinetics and Pharmacodynamics* 34 (5): 575–93. doi:[10.1007/s10928-007-9060-6](#).

Wendling, Thierry, Nikolaos Tsamandouras, Swati Dumitras, Etienne Pigeolet, Kayode Ogungbenro, and Leon Aarons. 2016. “Reduction of a Whole-Body Physiologically Based Pharmacokinetic Model to Stabilise the Bayesian Analysis of Clinical Data.” *The AAPS Journal* 18 (1): 196–209. doi:[10.1208/s12248-015-9840-7](#).