Population PK/PD with PoPy

Release 1.3.1

David Cristinacce
Andrew Cristinacce
Phil Tresadern
James Wright

Jun 12, 2026

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PoPy (pronounced pop-eye, as in the sailor man) is new software that supports population modelling of PK/PD (pharmacokinetic/pharmacodynamic) data.

It is written using the *Python* programming language but requires no programming experience for the typical user.

PoPy's intuitive interface and automated visualization make it perfect for people who are new to PK/PD. For experienced analysts, however, *PoPy* contains many of the features that are commonly used for modern PK/PD modelling.

Part I

Preliminaries

CHAPTER

ONE

MOTIVATION

Patients being treated for a given medical condition come in all shapes and sizes, and each patient's illness will affect them differently. Each patient should therefore receive personalized treatment that ensures the best response for their particular circumstances. This requires an understanding of how variation between patients influences the effect of a given drug on a given condition. Variation between patients can be both directly observed - properties such as age and weight, known as *covariates* - or observed indirectly through the patient's response to a specific dosing regimen of the drug.

In order to predict the effect of this variation better, compartmental models were developed that approximate, in a simple way, what the body does to the drug (pharmacokinetics) and what the drug does to the body (pharmacodynamics). These models treat the body as a series of compartments through which the drug circulates until eventually eliminated either through metabolism or excretion. The rate at which the drug moves between compartments is specified by a set of equations that are controlled by a set of model parameters.

Because every individual in a population will process the drug differently, it is important that these model parameters be personalized to each individual. This results, however, in an explosion of parameters for which there is rarely enough data to constrain the solution.

An approach called Nonlinear Mixed Effects Modelling (NLME) addresses this problem by modelling individual parameters as random deviations from a population average, both of which have specific statistical distributions that indicate the likelihood of any particular solution given observed data.

The difficulty with these nonlinear models is that the equations defining them are mathematically complex and must be solved numerically.

In the late 1970s, Stuart L. Beal and Lewis B. Sheiner began developing a computer software package called *Nonmem* - short for *Non*linear *M*ixed *E*ffects *M*odelling - that was written in Fortran and was being used for data analysis at least as early as 1980 [Evaluation of methods for estimating population pharmacokinetics parameters. I. Michaelis-Menten model: routine clinical pharmacokinetic data. Sheiner LB, Beal SL. J Pharmacokinet Biopharm. 1980 Dec;8(6):553-71. doi: 10.1007/BF01060053.].

Nonmem was groundbreaking and quickly set the "Gold Standard" for modelling nonlinear drug effects. Perhaps surprisingly, over four decades later, *Nonmem* is still the market leader in pharmacokinetic and pharmacodynamic (PK/PD) analysis. A quick analysis of PAGE meeting abstracts suggests that *Nonmem* still has around a 50% market share within the PK/PD community, and a similar analysis of PubMed abstracts indicates the same pattern.

In those four decades, however, there have been many advances in programming languages, numerical algorithms and computer hardware. Although *Nonmem* has benefited from some of these (*e.g.* developments in CPU power), its implementation in Fortran limits the advances on which it can capitalize and from which its many users would benefit (*e.g.* an enhanced user interface or improved numerical algorithms).

As a result, a number of other packages have emerged that offer similar or alternative approaches to dealing with nonlinear mixed effect models, often using newer languages that can more easily be extended, suggesting that there is appetite for alternatives.

Many of these alternative PK/PD packages, however, only *simulate* new data and do not *estimate* population parameters from a given dataset and model, which is arguably the most useful function of a PK/PD modelling package. Unsatisfied by most of these alternatives, therefore, we embarked upon a project to build a modelling package that was as powerful as *Nonmem* only easier for developers to maintain and extend, and for modellers to use without the need for add-ons such as "Perl speaks *Nonmem*" to augment its functionality (*e.g.* visualization and reporting).

Because we wanted a system that would be easy to extend, we chose to implement our system in Python that has a vast ecosystem of libraries and an enormous user community, making it arguably the go-to language choice for data science at the time of writing.

The result is a software package for *Po*pulation PK/PD in *Py*thon - "*PoPy*" - with many desirable features:

- Simulation: To support simulating one or many populations from a model, either from scratch or from seed values, *PoPy* offers many closed-form models, complex dosing functions and built-in sampling distributions (rather than just the normal distribution).
- Parameter Estimation: To support fitting complex models to large datasets, *PoPy* offers several fitting algorithms, using symbolic differentation for more efficient and accurate parameter updates.
- Visualization: To assist in reporting, *PoPy* generates a range of plots, tables and reports.
- Ease-of-use: To be as easy to pick up as possible, especially for existing *Nonmem* users, *PoPy* uses a structured input script format that makes it easy to write complex models yet *requires little to no Python experience from the user*.
- Performance: To be as powerful as possible, we compile much of the speed-critical code in Cython for high performance on a single CPU, and we provide parallel computing capabilities (via MPI) to maximize resource use on challenging problems with a large population of individuals.

Here we present *PoPy* and demonstrate its capability, using comparisons to other major packages for illustration, suggesting that *PoPy* shows promise as a research platform for PK/PD analysis.

CHAPTER

TWO

ABOUT *POPY*

PoPy is a powerful PK/PD (pharmacokinetic/pharmacodynamic) modelling system written in *Python*.

Its features include:

- Simulation and fitting of Nonlinear Mixed Effect models
- *PK* and *PD* modelling
- *Inter-occasion variability*
- *Continuous* and *categorical* likelihoods
- Dosing functions including *Bolus*, *infusions* and more

PoPy consists of a powerful set of *Command Line Tools* that typically operate on *script files* to enable you to analyse a *PK/PD data file* quickly and efficiently, while minimizing errors due to an elegant and intuitive syntax.

PoPy makes it easy to fit models to data, but also has the ability to simulate new population data to test hypothetical scenarios.

2.1 System Requirements

PoPy is currently available as a 64-bit *Microsoft Windows* binary that requires ~1.5Gb of disk storage to install.

A Linux *Python* wheel is available on request.

2.2 Bug Reporting

Although *PoPy* undergoes thorough tests before release, it is always possible that there will be cases we have not considered that cause *PoPy* to fail. (*PoPy* scripts contain a lot of "free text" where the user defines their own model with few constraints.) If you do come across something you believe is a bug in the software, please report it to us so that we can fix it make *PoPy* the best it can be.

2.2.1 Bug Reporting Procedure

1. Check you have a genuine bug and that you are using the latest version of *PoPy*.
2. Write down the steps that reproduce the bug so that we can recreate it at *PoPy* HQ. Include as many details as possible, including operating system and *PoPy* version. Feel free to share your log files and input scripts with us but *please remove anything sensitive*. A minimal `tut` script that reproduces the bug without using real data is perfect for reporting bugs.
3. Write down what you expected to happen and what actually did happen.
4. Email your bug report to info@popypkpd.com. (*PoPy* does not yet have a formal bug tracking system but may do in the near future.)

If we can't reproduce the bug, we will contact you to try to get to the bottom of the problem. Once we can reproduce the bug, we will endeavour to give you a time estimate of how long it will take to fix and, if the bug is serious, will release a new *PoPy* binary as soon as it is fixed. If the bug is relatively minor, the fix will be available in the next scheduled *PoPy* release.

2.3 Acknowledgements

PoPy could not have been developed without the efforts of many people for which we are very grateful.

David Cristinacce was *PoPy*'s mastermind, seeing an opportunity to reimagine nonlinear mixed effects modelling in a modern language (Python). He and Phil Tresadern were responsible for software development while Richard Dimelow and Andrew Cristinacce provided valuable user feedback that helped shape *PoPy*'s evolution. Overall direction and business management was provided by James Wright of Wright Dose Limited, *PoPy*'s financial sponsor.

Python was chosen as the modern language in which to reimplement NLME methods because of its rich ecosystem of libraries, and *PoPy* owes a debt to the creators and maintainers of the many libraries we used, including:

- NumPy: fast C-style arrays and numerical algorithms
- SciPy: ODE solving, numerical optimization, and statistics functions
- matplotlib: figure creation
- SymPy: symbolic manipulation of code that improves both speed and accuracy
- sphinx: documentation generation tools
- pylint: encourages good coding standards
- jenkins: continuous integration and delivery to keep *PoPy* "clean"

CHAPTER

THREE

ABOUT THIS BOOK

3.1 Book Structure

After these preliminaries, we present *PoPy* in roughly three parts.

The first part, *Operating Instructions*, is a tutorial-style introduction to using *PoPy* that focusses on how *PoPy* is run at the command line for a given script whose structure is explained but whose specific content is not especially important. This is what you *need* to know to start using *PoPy*.

The final part, *PoPy Reference Guide*, we provide reference details that you don't *need* to know at first but might *want* to know as you become more familiar with *PoPy* and how it works.

In between, we provide a series of "how to"-style guides that introduce PK/PD concepts and their implementations in *PoPy*. These "recipes" focus on the *contents* of the scripts that define models, simulations, and parameter estimation settings in order to help you create the models you need to better understand the data you have.

These recipes start simple, simulating data for only a single individual using different PK/PD models. Later recipes introduce parameter estimation (fitting) and analysis of data from a population of individuals. The last "how-to" part helps users who are familiar with *Nonmem*, the current market leader in NLME for PK/PD, to get started with *PoPy* by demonstrating equivalent scripts for common models and fitters.

3.2 Abbreviations

Throughout this book we use commonly found abbreviations in PK/PD, some of which are given below for convenience.

FOCE
: *first order conditional estimation*

IMP
: *importance sampling*

ITS
: *iterative two stage*

IV
: Intra-venous, *i.e.* injected directly into a vein

JOE
: *joint optimisation and estimation*

LAPLACE
: *Laplace approximation*

ND
: *none derivative estimation*

OBJV
: *objective function* value

ODEs
: *ordinary differential equations*

PK/PD
: Pharmacokinetic/Pharmacodynamic

SAEM
: *stochastic approximation expectation maximisation*

TSLD
: Time since last dose

VPC
: *visual predictive check*

wrt
: With respect to - usually defines rate of change variable in an *ordinary differential equation*

CHAPTER

FOUR

SETTING UP *POPY*

Before using *PoPy* for the first time, it needs to be installed and configured (much of which is done for you using sensible defaults).

4.1 Installation

PoPy is currently available as a 64-bit binary for *Microsoft Windows* version 10 or 11, requiring around 1.5Gb of disk space and a CPU at least as powerful as a Intel Core i3.

First, self-register an account with the *PoPy* product site at https://product.popypkpd.com/register

When logged in with your email address and a password, download the latest binary from https://product.popypkpd.com/releases/

We currently provide a **64-bit version** of *PoPy* for Windows 10/11 and a Python wheel compiled for Linux (available on request).

Note: If you have a previous version of *PoPy* installed, we recommend that you *uninstall_popy* before installing again.

Double-click on the *PoPy* installer, popy-1.3.1-win64-installer.exe and the installer will begin uncompressing files (which will occupy around 1.5Gb of disk space).

The installer makes no changes to the Windows registry and does **not** require admin rights.

By default, the installer places the software in a subdirectory called *PoPy*. You can, however, change this default and install *PoPy* to a folder of your choice. For ease of access, we recommend *PoPy* is installed in the root directory of the main drive (*e.g.* `c:\PoPy`).

(A local user typically has write access to the `c:\\` drive but not to `c:\\Program Files` unless the user is an administrator.)

The installer also creates a desktop shortcut and Windows start menu shortcut under the name '*PoPy*', for the current user only.

Note: It is possible to install *PoPy* on a usb stick. However due to the way the *Python* language operates, *i.e.* many .py files, this will have a significant performance impact. It is much better to install *PoPy* on the main hard drive of a computer.

4.1.1 Uninstalling *PoPy*

If you wish to remove *PoPy* for any reason (*e.g.* installing a new version), click on the shortcut at:

```
Start Menu > PoPy > popy_uninstall
```

or run the .exe at `<PoPy installation folder>\popy_uninstall.exe`.

The uninstaller will remove the *PoPy* desktop shortcut and Windows start menu items, but does not change the Windows Registry.

4.2 Configure Editor

PoPy is heavily script driven, so it helps to have a suitable text editor installed on your system.

We recommended that you use *Notepad++* to edit *PoPy* scripts and provide configuration files to assist in editing *PoPy* files. (See *Notepad++*.)

Note: Whichever editor you choose to use, you _must_ use spaces rather than tabs to indent the content of your *PoPy* scripts.

4.3 Configure *PoPy*

The machine-level settings of your *PoPy* installation are stored in `<PoPy installation folder>\popy_config.pyml` which can be modified in your chosen text editor.

We recommend, however, that you do not alter this file unless you know exactly what you are doing; the default settings should be sufficient for most users.

Should you edit your 'popy_config.pyml' and *PoPy* stops functioning correctly, please restore to the following default settings:

```
# path to graphviz dot.exe for creating diagrams
graphviz_dot_path: auto

# path to sphinx confuration options - used when generating html or latex
sphinx_conf_folder: ${POPY_PYTHON_PATH}/conf
```

Upon running *PoPy* for the first time, a user-level configuration file will also (on Microsoft Windows only) be created in the user's roaming profile which is typically found at

```
C:\Users\<username>\AppData\Roaming\PoPy\<SHA>-config.pyml
```

where *<username>* is the user's login username and *<SHA>* is the unique identifier for the version of *PoPy* being configured, as displayed when running *popy info*. Settings in the user's local configurations override those in the global configuration. In this way, we provide user- and installation-specific configuration of *PoPy* for finer-grained control over *PoPy*'s behaviour.

4.3.1 Sharing *PoPy* Among Users

By default, *PoPy* is installed for a single user on a given machine, and the path to *PoPy* scripts will be added to the Path environment variable of the user who installed *PoPy*.

If another user wants to use the same installation of *PoPy*, they should run popy-cmd from the installation folder where they will be asked whether they wish to add *PoPy* to their user path, and will be given three options.

- Answering "Y" (for Yes) to this question will add *PoPy* to their path so that

they can then call popy commands from any Command Prompt, not just the popy-cmd prompt. We recommend this choice.

- Answering "N" (for No) will continue running *PoPy* but will ask again for

subsequent calls to *PoPy* (including subscripts of scripts). An alternative for skipping this check is to add the *–skip-path-check* argument when calling *PoPy*, e.g.,

```
popy info --skip-path-check
```

- Answering "V" (for neVer) will skip the path check on this call *and all subsequent

calls* to *PoPy*, recording this choice in the user's local configuration file.

Part II

Operating Instructions

CHAPTER

FIVE

POPY TERMINAL

For simplicity, *PoPy* is a text-based *command line* tool that runs in an operating system terminal.

This terminal requires a set of environment variables to be defined that *PoPy* depends upon to work correctly. There are three easy ways to set up the *PoPy* environment:

- Run `<PoPy installation folder>/popy_cmd.exe` via Windows Explorer. This will open a *PoPy* command prompt in the installation folder.
- Left mouse click on the '*PoPy* Command Prompt' shortcut on your Desktop in the in the *PoPy* folder of the Windows Start menu. (This calls `popy_cmd.exe` for the *most recent PoPy* installation.)
- Open a terminal (*i.e.* a command prompt or Windows Powershell) in any folder and type `popy` to set up the environment. This requires *PoPy* to be in the user's `PATH` environment variable already. If it is not, first run `popy_cmd.exe` from the installation path instead.

Note: If there are multiple *PoPy* installations, we recommend that you run `popy_cmd.exe` from the desired installation folder to ensure that the correct version of *PoPy* is run. In any case, you should check that your *PoPy* environment is for the correct version by running `popy info` before proceeding.

Note: There *should* be no significant difference between running *PoPy* in a Command Prompt versus running it in a Powershell. We have, however, seen examples where running in a Command Prompt took significantly longer than in a Powershell for reasons we have not yet established.

In the terminal, you should see something like:

```
Setting up PoPy session for first use.
This may take a minute or so.
PoPy v1.3.1
(c)2026 Wright Dose Limited, UK.

Usage: popy <command> <arguments>

where command is:

    activate     Activate licence for PoPy
    deactivate   Deactivate licence for PoPy
    validate     Run validation test to ensure PoPy installed successfully
```

(continues on next page)

(continued from previous page)

```
    version      Display the PoPy version number
    info         Display detailed information on the PoPy installation

    create       Create a new script for a given action (e.g., fit, gen)
    run          Run a script
    check        Check a script (without running)
    format       Reformat an existing script to match latest specifications

    doc          Open HTML documentation in web browser
    view         Open html output in default browser

Call

    popy <cmd> --help

for help on using a specific PoPy tool.
```

in pale blue text on a black background.

This summary of the various *PoPy* commands indicates that *PoPy* is ready for use.

CHAPTER

SIX

CHECKING THE *POPY* INSTALLATION

Having opened a *opened a PoPy terminal*, the first screen you see shows the help page for *PoPy* which lists the available *PoPy* commands. This help is available at any time by typing

```
popy
```

in any terminal (*i.e.* command prompt or Powershell).

In general, you use *PoPy* by calling

```
popy <command> <arguments>
```

in a *PoPy* terminal where <command> is an action that applies on its own or to one or more <arguments>.

To verify that *PoPy* is working, for example, run

```
popy info
```

which, if *PoPy* has installed successfully, will show detailed information on the installation similar to:

```
INFO - In a PoPy Binary environment
INFO - popy_flavour=binary
INFO - popy_python_path=C:\PoPy\
INFO - popy_release=<X.Y.Z>
INFO - popy_version=academic
INFO - python_version=3.8.6
INFO - windows_version=('10', '10.0.17134', 'SP0', 'Multiprocessor Free')
INFO - machine_name=mickey
INFO - product_key=None
The product is running in trial mode
trial days remaining=3
INFO - should_run=True
```

(The output above shows information on a brand new installation of *PoPy* that is in its 60-day trial period and will look different once *PoPy* has been activated.)

6.1 *PoPy* Validation

Because computers vary in their architecture, it is possible that running the same code and the same script could give different results on different installations.

We therefore bundle a tool, *popy validate*, that runs *PoPy* on a suite of examples and compares the results generated locally on your hardware to some reference results generated on the *PoPy* development team's hardware. If the results agree sufficiently well then *PoPy* is ready to use.

These *Validation Examples* are interesting examples drawn from the *DDMoRe* repository that cover models with different features, *e.g.* inter-occasion variability.

To run this suite of tests, simply run

```
popy validate
```

from a *PoPy terminal*.

Some of the examples take a few minutes to run and the full validation can take about 20 minutes on older hardware but you should only need to run the validation once.

If the validation is successful, you should get output on the command line as the validation examples are processed. The end of the output should look like:

```
INFO -
INFO - Validation results: 4 passed; 0 failed.
INFO - Passes:
INFO -   ddmore0061
INFO -   ddmore0093
INFO -   ddmore0215
INFO -   ddmore0238
INFO -

    VALIDATION RESULT: PASS

INFO -
INFO -  time to complete val  = 915.338s
INFO - Main script val completed.
INFO -  time to complete main + sub scripts = 916.993s
Finished: SUCCESS
```

This output will also be written to the log file `validation_script.pyml.run.main.log` in the `validation` subdirectory of your *PoPy* installation.

This log file can be used as a form of verification for a validation report, if *PoPy* is to be used in a commercial setting.

Note: If validation fails for any reason, please get in touch with us to diagnose the problem.

CHAPTER

SEVEN

ACTIVATING A *POPY* LICENCE

Brand new *PoPy* installations can be used for 60 days on a trial period basis.

To continue using *PoPy* after this free trial period has expired, you must acquire and activate a licence for *PoPy*.

There are currently two licences for *PoPy*:-

***PoPy Academic* Licence**
: A free licence for *PoPy* which may **not** be used for commercial purposes, provided to academic institutions and nonprofits.

***PoPy Commercial* Licence**
: A paid-for licence granting the end user one year of *PoPy* usage on a single machine for commercial purposes. A commercial license is required to use *PoPy* in work including regulator submissions or published studies that are sponsored or funded by for-profit organizations.

(All versions of *PoPy* have the same functionality; the Academically-licensed version of *PoPy* is not limited in any way compared with the Commercially-licensed *PoPy*.)

Once you know the type of licence you require, email us at info@popypkpd.com to get a product key and run

```
popy activate <product_key>
```

(where "<product_key>" is the product key we sent to you) in a *PoPy Terminal* to activate your *PoPy* installation for 12 months, after which it will need to be renewed.

Once you have activated your *PoPy* installation, run:

```
popy info
```

to check the activation status, where you should see something like:

```
INFO - In a PoPy Binary environment
INFO - popy_flavour=binary
INFO - popy_python_path=C:\PoPy\
INFO - popy_release=<X.Y.Z>
INFO - popy_version=academic
INFO - python_version=3.8.6
INFO - windows_version=('10', '10.0.17134', 'SP0', 'Multiprocessor Free')
INFO - machine_name=mickey
```

(continues on next page)

(continued from previous page)

```
INFO - product_key=<XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX>
status=Product key is activated and within licence period
Licence has been running for 3613 days
Licence needs renewing in 1868 days
INFO - name=Phil Tresadern
INFO - email=phil@popypkpd.com
INFO - company=Wright Dose Ltd
INFO - licence_start_date=2009-02-12 00:00:00
INFO - licence_end_date=2024-02-16 00:00:00
INFO - should_run=True
```

which confirms that *PoPy* is now licensed.

If you see

```
INFO - product_key=None
```

or

```
INFO - should_run=False
```

in the output then there has been a problem with the activation. Email us at info@popypkpd.com and we will do our best to resolve the problem.

7.1 Deactivating a *PoPy* Licence

A *PoPy* licence limits its use to a fixed number of machines. # Once this limit has been reached, the licence must first be deactivated on one of the machines before *PoPy* can be activated on another. (Remember to make a note of the licence product key before deactivating the licence.)

To deactivate a *PoPy* licence on a machine, run

```
popy deactivate
```

in a *PoPy Terminal* and then

```
popy info
```

to confirm that the licence has been deactivated, as indicated by

```
INFO - product_key=None
```

in the output.

CHAPTER

EIGHT

CREATING YOUR FIRST *POPY* SCRIPTS

Some *PoPy* commands (*e.g.* `info`, `validate`) take no arguments; others (*e.g.* `activate`) take a single specific argument; the most important commands, however, operate on a structured text file that we refer to as a *script* and that implements a specific *action* (*e.g.* simulate data, fit model parameters, plot graphs).

This script contains metadata about the experiment along with a definition of the mathematical model and parameters of the code that will operate on it (*e.g.* optimizer tolerances).

To ease in the creation of these scripts *PoPy* comes with a command, `create`, that creates a template script for each action and that you can customize to suit your needs.

```
popy create <action> <filename>
```

where `<action>` is the action whose template script you want to create, and `<filename>` is the name of the file to create.

These built in scripts are designed to get you started with using a particular type of *script file*.

See *popy create* documentation.

8.1 Creating a `fit` Script

One of the actions that *PoPy* provides is `fit` that estimates ('fits') model parameters given a model and observed data. To create a `fit` script, *PoPy Terminal* and type:

```
popy create fit my_first_fit_script.pyml
```

where

- *popy create* is one of the *Command Line Tools*
- the `fit` option requests a *Fit Script* to be created and
- the *Fit Script* written to a file called `my_first_fit_script.pyml`.

This will create a brief script that is short but functional. Pass additional arguments to `popy create` to create a more verbose example script that includes all possible options (`-a`) with comments (`-c`) that explain each option and added spaces (`-s`) and line breaks (`-l`) for an easier read:

```
popy create fit my_first_fit_script.pyml -acsl
```

(See *popy create* or type `popy_create -h` for an explanation of the available command line arguments.)

8.2 Message Logging

You may notice that, in addition to the requested file, *popy create* also creates a file called `my_first_fit_script.pyml.create.main.log`. This text file contains a record of the messages output by the *popy create* tool which, in this case, is very short.

All of the *Command Line Tools* create between one and three log files and these files act as part of the audit trail for your experiments.

- The `main` log file is always created and contains messages generated as part of the normal running process
- A `warning` log file contains details of warnings (if any occurred) that were raised and that suggest changes to the script might be beneficial
- An `error` log file contains details of errors (if any occurred) that prevented *PoPy* from completing its operation

For example, if you try running the created fit_script:

```
popy run my_fit_script.pyml
```

the output on the screen should display an error:

```
CAST_ERROR= ERROR in value_record: ROOT->FILE_PATHS->input_data_file
ERROR when casting input file element
input file path = <current working␣
↪directory>/my_pkpd_model_gen.pyml_output/generated_data/cx_obs_params.csv
NOT present on file system.
```

telling you that the data to which the model should be fitted cannot be found in the expected location.

Furthermore, the `my_first_fit_script.pyml.run.main.log` file will declare that an error has occurred, and a new `my_first_fit_script.pyml.run.error.log` file will include details of the error that may be useful for debugging.

The amount of detail given in log files can be controlled with another command line argument (`-v`) to each *PoPy* command that supports it.

8.3 Creating a `tut` Script

The created `fit` script did not run because there was no data to which we could fit the model.

We could solve this problem using another *PoPy* action, `gen`, that generates synthetic data from a mathematical model and user-given parameters that define a population of individuals. For learning purposes, however, *PoPy* also comes with a "tutorial" action (`tut`) that completes four steps:

1. Create and run a `gen` script to generate a population of individuals and simulate observations from a mathematical model

2. Create and run a `fit` script that estimates the (known) model parameters from (inaccurate) initial estimates and the simulated data
3. Create and run a `comp` script that compares the estimated model parameters with the known parameters used to simulate the data
4. Create and run a `tutsum` script that summarizes the results of the experiment as an HTML report for easy reading

In this way, `tut` demonstrates almost all of *PoPy*'s capabilities in a single script for ease of learning, and you should create one now by running

```
popy create tut my_first_tut_script.pyml
```

in the *PoPy* terminal.

Because the `tut` action contains all of the steps we need, you should be able to run the script to completion straight away with no reported errors:

```
popy run my_first_tut_script.pyml
```

(As an aside, note that the created script contains the action it performs in its metadata so you do not need to specify the action to `popy run`, only to `popy create` where the action is not yet known.)

Let us now look closer at what is happening "under the hood" and the results that a `tut` script produces.

CHAPTER

NINE

RUNNING YOUR FIRST `TUT` SCRIPT

Having created a `tut` script:

```
popy create tut my_first_tut_script.pyml
```

you may now run it to examine the results and how they correspond to the contents of the script:

```
popy run my_first_tut_script.pyml
```

which will create an output folder called `my_first_tut_script.pyml_output` into which it saves all outputs (apart than log files and a link to the `tutsum` HTML report, both of which are created in the same folder as the `my_first_tut_script.pyml` input script).

If you ran the script already, a `main` log file will already exist which will prompt *PoPy* to ask whether you want to overwrite any previous results.

If you answer "n" (for No) then *PoPy* will terminate, leave the previous results unchanged, and suggest options that will re-run the script safely.

If you answer "y" (for Yes) then the previous results will be overwritten and lost for good, so be sure this is what you want. Use the `-o` command line argument to forcibly overwrite existing results without prompting (*e.g.* when running *PoPy* in an automated environment), or use the `-t` command line argument to create fresh outputs with timestamped filenames for posterity.

Note: Because the `main` log file is used to indicate existing results, we do not recommend manually deleting log files in case valuable results get overwritten.

To view the summarized results of the `tut` script, open the `my_first_tut_script.pyml.html` file in a web browser and follow the links to various output files.

You can compare your local html output with the pre-computed documentation output (*First order absorption model with peripheral compartment*) though you should expect some minor numerical differences when comparing against results computed on different hardware.

9.1 Script Hierarchies

PoPy scripts form a hierarchical structure whereby a script can be instructed to create and run other child scripts (which we call *subscripts*) for additional "downstream" processing after it has completed its main task.

A `tut` script, for example, can create and run `gen`, `fit`, `comp` and `tutsum` scripts. In turn, a `gen` script can create and run `grph`, `sim` and `gensum` scripts to plot observations, simulate new data at dense time points, and summarize the `gen` results.

The complete script hierarchy for a single population, which starts with a *Tut Script*, is shown in Fig. 1 and shows the many actions you can perform in *PoPy*.

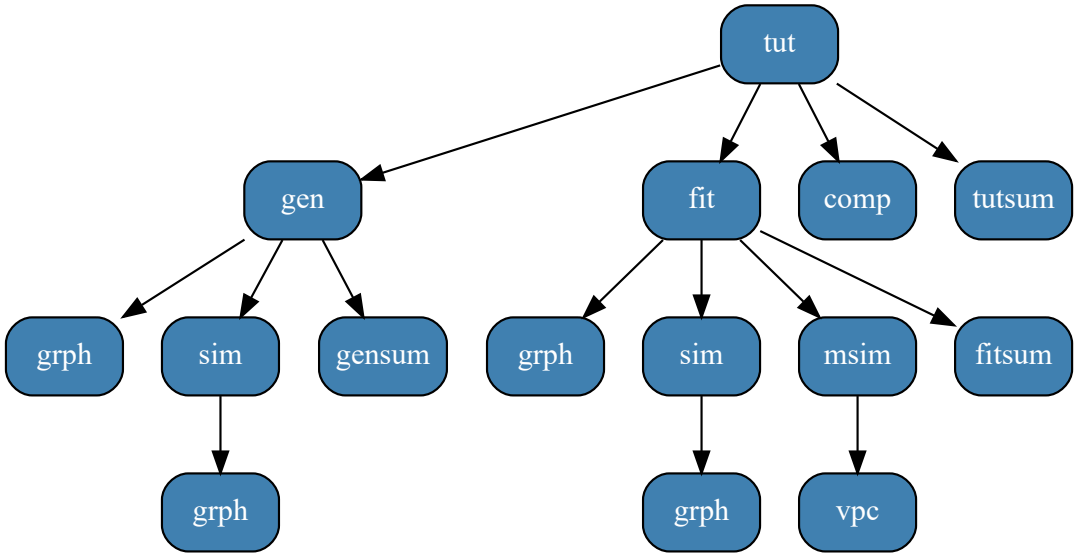


Fig. 1: Hierarchy of all child scripts for a parent *Tut Script*.

When fitting to real-life data (rather than synthetically-generated data), you typically use only the `fit` branch of the hierarchy as shown in Fig. 2.

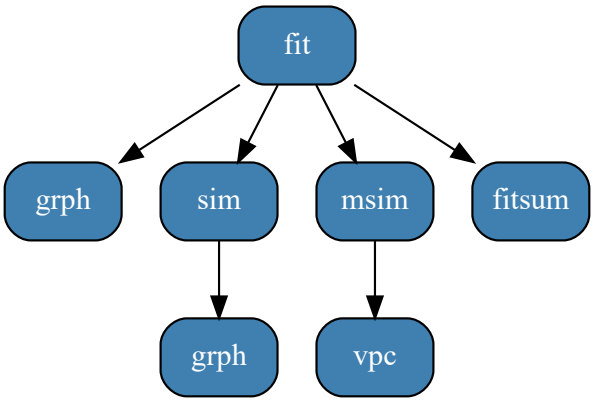


Fig. 2: Hierarchy of child scripts for a parent *Fit Script*.

Subscripts are specified in the *OUTPUT_SCRIPTS* section of the parent script which, in the case of our `tut` script, contains

```
OUTPUT_SCRIPTS:
    GEN: {output_mode: run, sim_time_step: 1.0}
    FIT: {output_mode: run, sim_time_step: 1.0}
    COMP: {output_mode: run}
    TUTSUM: {output_mode: run}
```

which instructs `tut` to create and run a `gen` script, then a `fit` script, then a `comp` script, and finally a `tutsum` script.

Subscripts can either be ignored (`output_mode: none`), created but not run (`output_mode: create`) or created and run (`output_mode: run`). Most subscripts are created and run by default, though some of the more time-consuming subscripts are only created (to be run manually by the user at a later date).

In the case of a `tut` script, its only function is to create (and optionally run) a suite of subscripts for `gen`, `fit`, `comp` and `tutsum`; it does no other processing of its own and is primarily a tool for learning and rapid prototyping. (This book makes extensive use of *tut_scripts* to illustrate different *Simulating A Single Individual*.) When creating these subscripts, their fields are typically populated using content from the parent script to ensure consistency within the hierarchy.

Running the *Tut Script* will create an output folder containing the four derived subscripts:

```
my_first_tut_script.pyml_output/
    my_pkpd_model_gen.pyml
    my_pkpd_model_fit.pyml
    my_pkpd_model_comp.pyml
    my_pkpd_model_tutsum.pyml
```

(These files are named after the default script `name` property - "my_pkpd_model" - which is user-definable.)

We will now examine each of these scripts in turn.

9.2 The `gen` Subscript

The purpose of the *Gen Script* is to create a synthetic set of data for a population of individuals with properties defined by the user.

To do so, it primarily needs to know about:

1. the individuals in the population
2. the specifications of the study or trial (*e.g.* times of dose and observation)
3. the mathematical model of the PK/PD

9.2.1 gen Inputs

The properties of the population and study are defined in the EFFECTS section of the gen script:

```
EFFECTS:
    POP: |
        c[AMT] = 100.0
        f[KA] = 0.2
        f[CL] = 2.0
        f[V1] = 50
        f[Q] = 1.0
        f[V2] = 80
        f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv] = [
            [0.1],
            [0.01, 0.03],
            [0.01, -0.01, 0.09],
            [0.01, 0.02, 0.01, 0.07],
            [0.01, 0.02, 0.01, 0.01, 0.05],
        ]
        f[PNOISE] = 0.15
    ID: |
        c[ID] = sequential(50)
        t[DOSE] = 2.0
        t[OBS] ~ unif(1.0, 50.0; 5)
        # t[OBS] = range(1.0, 50.0; 5)
        r[KA,␣
↪CL, V1, Q, V2] ~ mnorm([0,0,0,0,0], f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv])
```

which is a carbon copy of the GEN_EFFECTS section of the parent tut script.

Very briefly, this EFFECTS section defines a population of 50 individuals, numbered sequentially from 1 to 50,

```
c[ID] = sequential(50)
```

numerous fixed effects with constant values (as indicated by the use of the "=" assignment operator),

```
f[KA] = 0.2
f[CL] = 2.0
...
```

and random effects that are sampled (as indicated by the use of the "~" distribution operator) for each individual from a zero-mean, multi-variate normal distribution whose *symmetric positive definite* covariance matrix (specified completely by its lower triangular elements) is defined by one of the population-level fixed effects,

```
f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv] = [
    [0.1],
    [0.01, 0.03],
    [0.01, -0.01, 0.09],
    [0.01, 0.02, 0.01, 0.07],
```

(continues on next page)

(continued from previous page)

```
    [0.01, 0.02, 0.01, 0.01, 0.05],
]
...
r[KA,␣
↪CL, V1, Q, V2] ~ mnorm([0,0,0,0,0], f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv])
```

Regarding the study properties, it defines a dose of 100 units of drug, administered at time `t = 2.0`,

```
c[AMT] = 100.0
...
t[DOSE] = 2.0
```

and five observations at time points uniformly distributed in the period [1.0, 50.0]:

```
t[OBS] ~ unif(1.0, 50.0; 5)
```

with a hierarchical structure that includes between-subject variability but no between-occasion variability,

```
POP: |
    # population-level parameters
ID: |
    # individual-level parameters
```

In *PoPy*, `POP` is the root of the hierarchy which, in this example, creates an `ID` branch for every unique value of `c[ID]` (50 branches in this case). Covariates and random effects defined at the `ID` level will be resampled for every `ID` branch. In contrast, population-level fixed effects are treated as if defined at the `POP` level, wherever they happen to be defined. (This allows you to define population covariances immediately before random effect distributions that depend upon them.) Deeper levels of the hierarchy (*e.g.* between-occasion variability) can be created by adding more entries to the tree structure, *e.g.* below `ID`. Time definitions *must* be at the lowest level of the hierarchy because they implicitly define the leaves of the tree structure (one leaf node per row).

The mathematical model to be simulated is defined by the `MODEL_PARAMS`, `STATES`, `DERIVATIVES` and `PREDICTIONS` sections, all of which are carbon copies of the corresponding sections in the `tut` script. In this built-in example, the model is a two compartment model with absorption and bolus dosing (Fig. 3:)

but the exact details of the model are not important here.

One distinction is, however, worth noting: the *PREDICTIONS* section defines the probability distribution of the observation (`c[DV_CENTRAL]`)

```
var = m[ANOISE]**2 + m[PNOISE]**2 * p[DV_CENTRAL]**2
c[DV_CENTRAL] ~ norm(p[DV_CENTRAL], var)
```

which, for a `gen` script, is *sampled* at every observation time point in order to simulate noisy observations.

The remaining sections of note at this point include:

- `METHOD_OPTIONS`: derived from the `tut` script but with `py_module` set to `gen` so that *PoPy* knows which action to apply

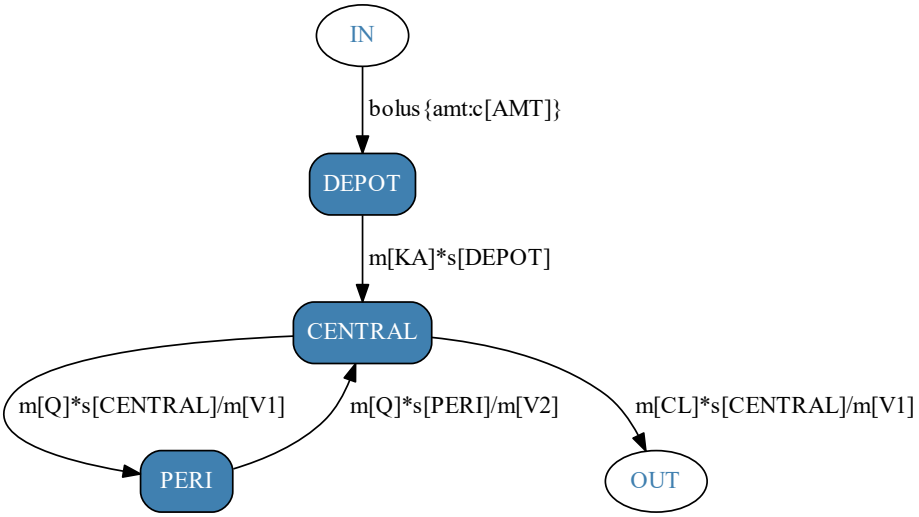


Fig. 3: Two compartment model with depot dosing for *Tut Script*.

- `ODE_SOLVER`: a carbon copy of the same section in the `tut` script, setting the parameters of the ODE solver that will simulate the compartment model
- `OUTPUT_SCRIPTS`: the subscripts of `gen` to be run, which are populated with default values

The ability to generate synthetic data from a model is especially useful if you wish to demonstrate a model, but do not have access to a real data set. Real data is expensive to obtain and even if it exists may have issues regarding completeness, accuracy or confidentiality. The other disadvantage of real data is that we never know the true underlying model.

9.2.2 `gen` Outputs

The main `gen` output is a machine-readable tabular file, stored at

```
<tutorial script output folder>\
    my_pkpd_model_gen.pyml_output\
        synthetic_data.csv
```

that contains both assigned (or sampled) covariates *and* sampled observations.

whose first few ten rows are shown in (Table 1).

Table 1: First 10 rows of 'synthetic_data.csv' file

TIME	ID	TYPE	AMT	DV_CENTRAL	DV_CENTRAL_FLAG	orig_data_row
2	1	dose	100	0.00152359650281	0	0
10.0120217722	1	obs	100	1.67075294237	1	1
11.0234536491	1	obs	100	2.11707619384	1	2
16.5024021745	1	obs	100	1.31206960996	1	3
28.818526425	1	obs	100	0.435327698431	1	4
46.551188548	1	obs	100	0.153098540708	1	5
2	2	dose	100	0.000377833830139	0	0
30.181690446	2	obs	100	0.412763665129	1	1
33.0056777467	2	obs	100	0.274939718447	1	2
…	…	…	…	…	…	…

This data file, combining covariates and observations, is effectively what you would input to a parameter estimation program to fit a model to data.

The data file starts with a header row that labels the covariates so that they can be referenced by name in downstream processes. In this example, the column/covariate labels are:

- TYPE: whether the row is a dose, an observation or a simulation
- ID: the identifier for a given individual
- TIME: the time stamp of the row event
- AMT: the amount of drug administered if the row is of dose TYPE
- DV_CENTRAL: the simulated observation
- DV_CENTRAL_FLAG: whether the observation should be classed as a measurement in parameter estimation
- orig_data_row: the data row number within an individual subject

Intermediate outputs of `gen` are saved to files in

```
<tutorial script output folder>\
    my_pkpd_model_gen.pyml_output\
        generated_data\
```

and include:

- assigned (or sampled) fixed effects, stored in a machine-readable `fx_params.csv` tabular file
- sampled random effects, stored in a machine-readable `rx_params.csv` tabular file

These outputs are generated in two stages: sampling and simulation.

The sampling stage proceeds in four steps:

First, population-level *fixed effects* `f[X]` variables defined in the *EFFECTS* section are assigned or sampled. (In this case the `f[X]` are all assigned constants.)

Second, a hierarchical tree structure of covariates is created by assigning or sampling values according to their position in the *EFFECTS* section. In this example, we sample 50 unique individuals who share a

common `AMT` covariate at the `POP` level. (The `c[TIME]` and `c[TYPE]` fields are created by *PoPy* automatically.)

Third, we sample (never assign) random effects following the same hierarchy, applying dependencies on the (known) fixed effects.

Fourth, we assign or sample dose and observation times at every leaf of the tree structure. In this example, we assign one dose time and sample five observation times for each of the 50 individuals, giving a dataset of 300 rows in total.

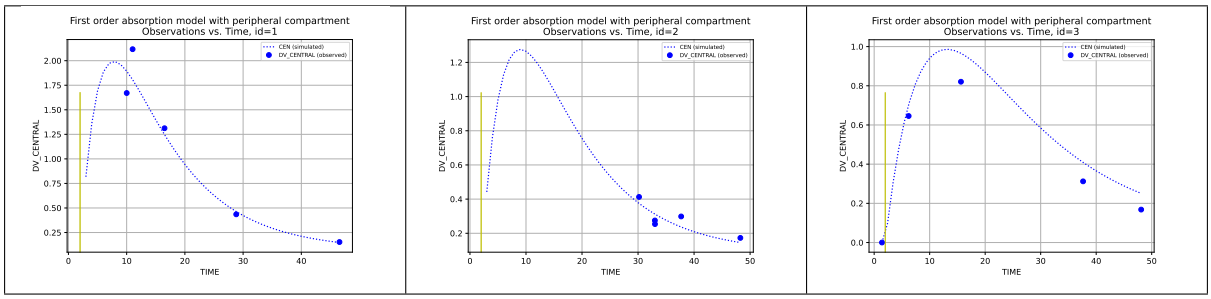
The simulation step then uses these sampled values to compute PK/PD model parameters, initial compartment amounts, the compartment amount at every sampled time point, and a simulated noisy observation at every time point according to the `MODEL_PARAMS`, `STATES`, `DERIVATIVES` and `PREDICTIONS` sections, respectively.

These simulated observations are then combined with the sampled values and saved to `synthetic_data.csv`.

Subscripts of `gen` can be used to plot the simulated observations (`grph`); simulate smooth, noiseless curves of the time series (`sim`); and summarize the generated data in an HTML report (`gensum`).

We can see (in Table 2) the first three plots created by `gen` followed by `sim` followed by `grph`. Here, a dotted blue line represents the noiseless model predictions given the `f[X]` parameters and sampled `r[X]` values for each individual, and solid blue dots represent the simulated, noisy observations (`c[DV_CENTRAL]` in `synthetic_data.csv`).

Table 2: Synthetic data plots for first three individuals



9.3 The `fit` Subscript

Having simulated some data to which we can fit our model, the *Fit Script* is run in order to estimate the "best fit" fixed effects (and random effects) for the simulated dataset.

9.3.1 `fit` Inputs

As in the `gen` script, a *METHOD_OPTIONS* block contains a `py_module` entry that defines the action to take (`fit` in this case).

A new *FILE_PATHS* section includes an `input_data_file` entry that points to the data file (`my_pkpd_model_gen.pyml_output/generated_data/cx_obs_params.csv`) that defines the study and the individuals taking part in it.

```
FILE_PATHS:
    input_data_file:␣
↪builtin_tut_example_gen.pyml_output/generated_data/cx_obs_params.csv
    output_file_ext: ['svg', 'pdf']
```

Because the properties of the population and study are entirely contained in the input data, a `fit` script need only consider what fixed effects and random effects to estimate, so the parent `tut` script's *FIT_EFFECTS* that is copied to the `fit` script's *EFFECTS* block does not include covariates (`c[X]`) or times (`t[X]`):

```
EFFECTS:
    POP: |
        f[KA] ~ P1.0
        f[CL] ~ P1.0
        f[V1] ~ P20
        f[Q] ~ P0.5
        f[V2] ~ P100
        f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv] ~ spd_matrix() [
            [0.05],
            [0.01, 0.05],
            [0.01, 0.01, 0.05],
            [0.01, 0.01, 0.01, 0.05],
            [0.01, 0.01, 0.01, 0.01, 0.05],
        ]
        f[PNOISE] ~ P0.1
    ID: |
        r[KA,␣
↪CL, V1, Q, V2] ~ mnorm([0,0,0,0,0], f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv])
```

Furthermore, because fixed effects are unknown but allowed to vary they are given a *distribution* of values (usually to limit their range) and an initial value to begin the estimation process:

```
f[X] ~ <distribution> <initial value>
```

For example,

```
f[X] ~ unif(0.0, +inf) 1.0
```

defines a uniformly distributed variable that must take a value in the range (0, +infinity) - *i.e.* a positive value - and that has an initial value of 1.0. Because this range is a common one in PK/PD modelling, *PoPy* provides a convenient shortcut, `~ P`:

```
f[X] ~ P 1.0
```

where the 'P' is short for 'positive'.

Random effects definitions are identical to those in the `gen` script.

In total there are 271 parameters to estimate:

- six univariate parameters
- one symmetric covariance matrix with 15 degrees of freedom
- five random effects for each of the 50 individuals in the population

MODEL_PARAMS, *DERIVATIVES* and *PREDICTIONS* define the mathematical model as before, and the *ODE_SOLVER* section sets the parameters of the solver that integrates derivatives into amounts.

OUTPUT_SCRIPTS defines the subscripts we want to run after completing the model fitting, which in this case are `sim`, `msim` and `fitsum`. As with `gen`, `sim` simulates smooth curves at dense time points for the model with estimated (rather than true) values of the parameters, and `fitsum` creates a HTML report of the fitting process.

`msim` is a different subscript that, given population fixed effects, will generate many new populations whose statistics can be used to evaluate the goodness of fit via a Visual Predictive Check (VPC).

The other section - also new to the `fit` script - is the `FIT_METHODS` section that defines the fitting method (*e.g.* FOCE, JOE or ND) and its parameters (*e.g.* tolerances or number of iterations). These parameters control how well and how quickly a fitting algorithm will converge on a final estimate and will be discussed in greater depth later in this book.

```
FIT_METHODS: [ND: {max_n_main_iterations: 30}]
```

9.3.2 `fit` Outputs

Running this `fit` script (either manually or as part of this `tut` tutorial), will attempt to minimize iteratively an error function (known as the Objective Function, OBJV) over a number of steps such that the model parameters generate a better prediction of the data at every step. The Objective Function is defined in such a way that it is minimized when the *likelihood* (the probability of the data given the model) is maximized, and when OBJV stops decreasing the fit is said to have *converged* (Table 3).

Table 3: Objective values vs iteration number and time

Iteration	Time	OBJV
0.	0.00	4224.091990462789
0.	11.27	888.5810225001037
1.	11.57	888.5810225001037
1.1	11.64	-624.5908516989929

continues on next page

Table 3 – continued from previous page

1.2	14.19	-688.3203750485884
1.3	15.71	-783.4113895200995
1.4	17.30	-817.544655433727
1.5	18.82	-819.0278364926105
1.6	20.24	-820.6396717219868
1.7	21.97	-821.8064086681877
1.8	23.39	-822.4392081874006
1.9	25.01	-824.1441967766439
1.10	27.08	-825.5293291066278
1.11	28.70	-828.130755672234
1.12	31.17	-829.0629711533805
1.13	32.51	-831.9135180999128
1.14	34.16	-833.5065464878089
1.15	35.87	-835.658611770827
1.16	37.67	-838.15981287146
1.17	39.96	-841.6970964235949
1.18	41.69	-844.3049259627668
1.19	43.31	-848.650027481997
1.20	45.48	-850.9383284732306
1.21	47.61	-853.0830534238515
1.22	48.91	-854.1773655364336
1.23	50.52	-854.4969371123143
1.24	51.88	-855.8025865592084
1.25	53.21	-858.9790044800343
1.26	55.11	-860.7660976740185
1.27	56.43	-861.8112092897645
1.28	57.77	-862.283489303459
1.29	59.34	-862.8287881866131
1.30	60.89	-863.1496093288893

Because the data to which we are fitting with our model was created by simulating observations *with the same model*, we should expect a good model fit to the data. (Some differences in estimated parameter values should, however, be expected because the observations are noisy and imperfect [Sheiner1980].)

When OBJV has converged to a locally minimal value (a "local minimum") then the corresponding values of the fixed effects and random effects are returned as the "best" estimates of those variables:

```
f[KA] = 0.2148
f[CL] = 1.7896
f[V1] = 55.4580
f[Q] = 1.0564
f[V2] = 486.5705
f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv] = [
    [ 0.1393, -0.0553, 0.0414, -0.0256, 0.1217 ],
    [ -0.0553, 0.0808, -0.0070, -0.0061, -0.1784 ],
    [ 0.0414, -0.0070, 0.1098, -0.0827, 0.2139 ],
    [ -0.0256, -0.0061, -0.0827, 0.3173, -0.3239 ],
    [ 0.1217, -0.1784, 0.2139, -0.3239, 0.9443 ],
]
```

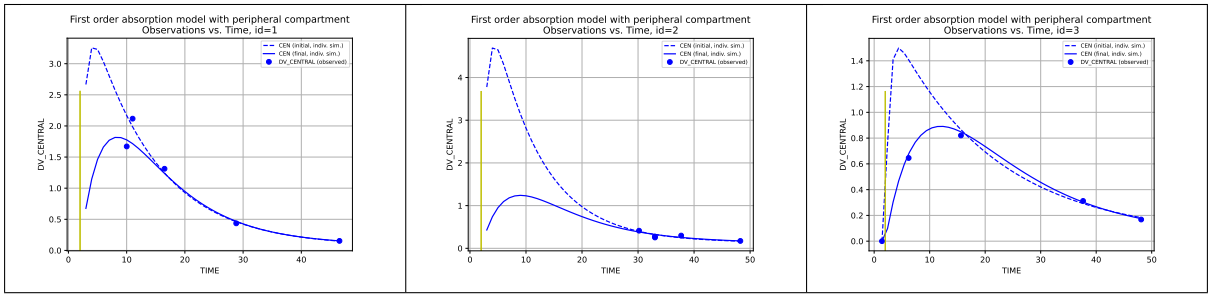
(continues on next page)

(continued from previous page)

```
f[PNOISE] = 0.1415
```

After running `fit` followed by `sim` followed by `grph` we can see a smooth predicted time series of drug concentration for every individual, overlaid with the input observations (Table 4)

Table 4: Model predictions vs original data points for first three individuals



where a dashed blue line represents the noiseless model predictions given the *initial* `f[X]` parameters and sampled `r[X]` values for each individual, a solid blue line represents the predicted curve for the *final* values, and solid blue dots represent the simulated, noisy observations (`c[DV_CENTRAL]` in `synthetic_data.csv`).

These graphs show that *PoPy* has adjusted the `f[X]` and `r[X]` parameters such that the PK curves more closely match the input data and therefore maximise the likelihood of the data being generated from this model.

9.4 The `comp` and `tutsum` Subscripts

In order to quantify *how* well `fit` has estimated the parameters, the `comp` script recomputes OBJV for the true parameters and the final estimated parameters. The `tutsum` script then creates an HTML report containing absolute and relative errors as part of a summary of the whole experiment.

9.4.1 Fitted vs True f[X] values

If the *Comp Script* has been run, the *TutSum Script* output also contains a convenient table that compares the initial, fitted and true `f[X]` values (Table 5).

Table 5: Comparison of main initial, fitted and true `f[X]` values

Name	Initial	Fitted	True	Abs. Error	Prop. Error
f[KA]	1	0.215	0.2	1.48e-02	7.41%
f[CL]	1	1.79	2	2.10e-01	10.52%
f[V1]	20	55.5	50	5.46e+00	10.92%
f[Q]	0.5	1.06	1	5.64e-02	5.64%
f[V2]	100	487	80	4.07e+02	508.21%

Table 5 shows that the `f[KA]`, `f[CL]`, `f[V1]`, `f[Q]` parameters are recovered reasonably well, in the sense that the fitted values are much closer to the true values, compared to the initial starting values. However the `f[V2]` fitted value is quite different from the true value.

You can also compare the proportional noise estimate:

Table 6: Comparison of fitted and true proportional noise `f[X]` values

Name	Initial	Fitted	True	Abs. Error	Prop. Error
f[PNOISE]	0.1	0.142	0.15	8.50e-03	5.67%

where `f[PNOISE]` started at 0.1 and is estimated at 0.141, which can be compared with the true value 0.15. Generally the proportional noise is identifiable in a PK/PD model because every row of the data set and the current `f[PNOISE]` parameter has an influence on the overall likelihood.

The comparison of the covariance matrix estimates is quite long, as it displays a row for each of the 5*5 `f[X]` comparisons:

Table 7: Comparison of fitted and true isv variance `f[X]` values

Name	Initial	Fitted	True	Abs. Error	Prop. Error
f[KA_isv]	0.05	0.139	0.1	3.93e-02	39.25%
f[KA_isv;CL_isv]	0.01	-0.0553	0.01	6.53e-02	653.38%
f[KA_isv;V1_isv]	0.01	0.0414	0.01	3.14e-02	314.14%
f[KA_isv;Q_isv]	0.01	-0.0256	0.01	3.56e-02	356.13%
f[KA_isv;V2_isv]	0.01	0.122	0.01	1.12e-01	1116.86%
f[CL_isv;KA_isv]	0.01	-0.0553	0.01	6.53e-02	653.38%
f[CL_isv]	0.05	0.0808	0.03	5.08e-02	169.21%
f[CL_isv;V1_isv]	0.01	-0.00702	-0.01	2.98e-03	29.83%
f[CL_isv;Q_isv]	0.01	-0.00606	0.02	2.61e-02	130.30%
f[CL_isv;V2_isv]	0.01	-0.178	0.02	1.98e-01	992.15%
f[V1_isv;KA_isv]	0.01	0.0414	0.01	3.14e-02	314.14%
f[V1_isv;CL_isv]	0.01	-0.00702	-0.01	2.98e-03	29.83%
f[V1_isv]	0.05	0.11	0.09	1.98e-02	21.97%
f[V1_isv;Q_isv]	0.01	-0.0827	0.01	9.27e-02	927.02%
f[V1_isv;V2_isv]	0.01	0.214	0.01	2.04e-01	2039.38%
f[Q_isv;KA_isv]	0.01	-0.0256	0.01	3.56e-02	356.13%
f[Q_isv;CL_isv]	0.01	-0.00606	0.02	2.61e-02	130.30%
f[Q_isv;V1_isv]	0.01	-0.0827	0.01	9.27e-02	927.02%
f[Q_isv]	0.05	0.317	0.07	2.47e-01	353.28%
f[Q_isv;V2_isv]	0.01	-0.324	0.01	3.34e-01	3339.14%
f[V2_isv;KA_isv]	0.01	0.122	0.01	1.12e-01	1116.86%
f[V2_isv;CL_isv]	0.01	-0.178	0.02	1.98e-01	992.15%
f[V2_isv;V1_isv]	0.01	0.214	0.01	2.04e-01	2039.38%
f[V2_isv;Q_isv]	0.01	-0.324	0.01	3.34e-01	3339.14%
f[V2_isv]	0.05	0.944	0.05	8.94e-01	1788.59%

which suggests that:

- the inter-subject variances of the clearances, `f[CL_isv]` and `f[Q_isv]`, are estimated reasonably well
- the `f[KA_isv]` estimate is far too high (possibly due to the relative lack of data before the **cmax** peak of the PK curve, there being only 5 data points sampled per individual)
- the `f[V1_isv]` and `f[V2_isv]` estimates are badly estimated (which is not that surprising, since *volumes of distribution* are harder to estimate in PK models compared to *clearances* which are rates.

There can be multiple reasons for the fitted values not agreeing well with true parameters, including:

- Too few observations in the data set
- Too few individuals in the data set
- Too much noise added to the measurement data
- False minima on the likelihood surface
- A fundamental difficulty in identifying some PK parameters, such as if the model is over-parameterised relative to the data set or even unidentifiable

Given the relatively small amount of data generated (250 observation points), our results (Table 5) are adequate. As shown in the next section the Objective Function for the fitted `f[X]` solution is actually lower than for the true `f[X]` solution.

9.4.2 Fitted vs True Objective value

It can be difficult to know by just comparing the fitted `f[X]` and true `f[X]` values (*Fitted vs True f[X] values*) whether the fitting method has done well or badly. We can run a form of sanity check by computing the objective function of the true `f[X]` and comparing this with the objective function of the fitted `f[X]`. This is done by optimising the `r[X]` for both solutions and using the synthetic data file to compute the likelihood.

The rationale is that the fitted `f[X]` objective value should always be **lower** than the true `f[X]` objective value, because the fitted model estimates can take advantage of correlations in the random measurement noise to get a better fit to the synthetic data. If the fitted objective function is *higher* then the *PoPy* fitting method has ended up in a suboptimal local minimum because the known true values are a better minimum.

In this example, the true model objective function is

```
-881.0027
```

which can be compared with the fitted model objective function,

```
-896.8752
```

This indicates that the fitted `f[X]` pass the sanity check and perhaps justifies the lack of agreement with parameters such as `f[V2]`. It does **not**, however, say whether the global optimum solution was found, *i.e.* whether or not `f[X]` is a true global minimum of the likelihood surface.

The difference between the two objective values above is partially dependent on the amount of noise applied to the measurements (*i.e.* `f[PNOISE]`), the number of individuals simulated, the number of observations per individual and the number of parameters in the model. More random noise in the *synthetic data* creates more likelihood minima that are away from the true `f[X]` solution.

If the model is practically identifiable then increasing the number of data points and individuals should lead to a convergence between the true and fitted `f[X]` and the objective functions above.

CHAPTER

TEN

POPY DATA FORMAT

Many *PoPy* actions output several tables with one row per event (typically a dose or observation) over a population of individuals.

The columns or fields in the data file are split into four main types:

- Required fields must be present in every data file; they identify rows in the dataset
- Dose event parameters define the administration of drug to the body
- Observation field parameters define the measurements that should be sampled (during simulation) or that contribute to the error function (during model fitting)
- All other columns represent user-defined inputs to or outputs of the mathematical model (*e.g.* measured covariates)

The data file values for each field can be accessed using the `c[X]` (or similar) notation in the *PoPy script file*.

10.1 Required Fields

A *PoPy* data set requires the following fields:-

- *TYPE* - type of row
- *ID* - identity
- *TIME* - time field

Note the names `TYPE`, `ID` and `TIME` are the default names of these three required fields. (These three fields are the minimum required to uniquely define a row in the dataset, though other fields maybe be required to define rows when there are other levels in the `EFFECTS` hierarchy, *e.g.* when using between-occasion variability.)

If you are using pre-existing data that does not use these field headings, you can map the names you have to `TYPE`, `ID` and `TIME` in the *script file DATA_FIELDS* section.

10.1.1 TYPE

The `TYPE` field specifies the event that is happening in each row of the data file, namely one of:

- obs: Measurements that contribute to the log likelihood as defined in the *PREDICTIONS* section
- dose: Creates a dose according to the dosing functions in the *DERIVATIVES* section
- pred: Extra prediction data points. *PoPy* will output extra `p[X]` data at these time points, but they do **not** contribute to the likelihood.
- reset: Set the `s[X]` compartment states back to the initial values (usually zero)
- reset+dose: A 'reset' immediately followed by a 'dose' event at the same `TIME`

Drug trial data typically consists of 'obs' and 'dose' rows with a few 'reset' rows per subject.

10.1.2 ID

The `ID` field value defines the individual for a given row. As *PoPy* is a PopPK/PD system, the `ID` field is required because the data is split over multiple individuals to form a population.

Note that non-population analysis can be performed in *PoPy* by assigning all rows the same `ID` value.

10.1.3 TIME

The `TIME` field defines the time stamp for each row and is required to be monotonically increasing unless a *TYPE* = 'reset' or 'reset+dose' row is reached.

Note: Note that when the *ID* identifier changes between rows, then an implicit 'reset' occurs.

For an example of a valid combination of TYPE/ID/TIME data see Table 1.

Table 1: *PoPy* time reset example

TYPE	*ID*	*TIME*	comment
obs	Bob	0.0	observation at time zero
dose	Bob	4.0	dose for bob at time 4.0
obs	Bob	4.0	observation for bob at time 4.0
obs	Bob	8.0	later observation
obs	Ruth	0.0	time goes back, ok cos new ID
dose	Ruth	10.0	dose for Ruth at time 10.0
obs	Ruth	20.0	later observation
reset	Ruth	30.0	`s[X]` reset at time 30.0
obs	Ruth	1.0	observation following reset

In Table 1 the time always increases or stays the same in consecutive rows, but time is allowed to go backwards after a new ID or a reset.

10.2 Dosing Fields

Dosing events are indicated in the data file by a *TYPE* field that starts with `dose`.

The amount of drug administered is usually specified in an AMT field.

Note in *PoPy* `AMT` is **not** a keyword; it is just the conventional name (and our default label) for the dose amount used by any dosing function.

Table 2: *PoPy* single dose type example

TYPE	*TIME*	AMT	comment
dose	1.0	100	dose of 100 at time 1.0
dose	2.0	200	dose of 200 at time 2.0
dose	3.0	100	dose of 100 at time 3.0

As an example, we can create a data file (Table 2) that specifies three dose events at times [1.0, 2.0, 3.0] respectively, whose parameters will be 'injected' into the dosing function specified in the *DERIVATIVES* section of the script.

10.3 Observation Fields

Observation rows are indicated by a *TYPE* field that starts with `obs` and are used by the *PREDICTIONS* section of the *PoPy* *script file* either to sample an observation (during simulation) or evaluate the likelihood of an observation given an estimated probability distribution (during model fitting).

Table 3: *PoPy* single observed field example

TYPE	DRUG_CONC
obs	10.5
obs	20.0
obs	15.5

As an example, we can create a data file (Table 3) that specifies three drug concentration observations of 10.5, 20.0 and 15.5 units respectively. The amount or concentration of drug or biomarker will be simulated at these time points by solving the differential equations given in *DERIVATIVES*, and passed to the *PREDICTIONS* block to define the observation's estimated probability distribution for sampling or evaluation.

Although the `TYPE` field disambiguates observation events from other events, we may also wish to disambiguate valid observations that should contribute to likelihood values from invalid ones that should not (*e.g.* where data are missing). For this purpose, *PoPy* data files come with a `FLAG` field that uses a value of zero to filter out invalid observations.

Table 4: *PoPy* single observed field missing data example

TYPE	DRUG_CONC	DRUG_CONC_FLAG	comment
obs	10.5	1	Valid observation
obs	20.0	1	Valid observation
obs	15.5	1	Valid observation
obs	-2.0	0	Observation out of valid range

As an example, we can extend our earlier data file (Table 3) with an invalid observation of -2.0 units that falls outside of the valid range (Table 4). This invalid observation is identified by setting the `DRUG_CONC_FLAG` value to zero. Any likelihood computations will ignore this observation.

Missing `FLAG` fields automatically default to a value of 1 such that all observation rows contribute to the likelihood of the data given the model.

10.4 User-Defined Fields

The other columns of the *PoPy* data file are available to use in the *verbatim* sections *MODEL_PARAMS*, *STATES*, *DERIVATIVES* and *PREDICTIONS* that define the mathematical model of the process.

For example see below for a simple example of *covariate modelling* using the *MODEL_PARAMS*:

```
MODEL_PARAMS: |
    m[X] = f[X] + f[X_Y_EFFECT]*c[Y]
```

Here the `m[X]` parameter is modelled as having a linear relationship with the `c[Y]` covariate from the data file.

It is also possible to use `c[X]` variables in the other sections. One use case is when you already have PK parameters estimated (from a previous study) and wish to use these `c[X]` variables in the *DERIVATIVES* section instead of estimating `m[X]` parameters for each individual.

Having covered the basics of how *PoPy* is used, the functions it provides, and the outputs it generates, we can now look at how to build (or grow) a mathematical model of the process under consideration.

Part III

Simulating A Single Individual

In the first of several "how-to" style guides, we present some of the more fundamental concepts in pharmacokinetics and pharmacodynamics, using *PoPy* scripts to simulate data and figures to support our explanations. For ease of introduction, we consider only a single individual and look at populations of individuals in following parts (*e.g.* *Simulating A Population Of Individuals*).

Pharmacokinetics ([MouldUpton2012], [MouldUpton2013], [UptonMould2013], [RowlandTozer2012]) is the study of the time profile of drugs and metabolites in a physiological organism, often described as "what the body does to the drug".

Typically, our goal is to build (or grow) a mathematical model that predicts a time course of *drug* concentration resembling the observations we have collected from an individual.

Pharmacodynamics, in contrast, describes "what the drug does to the body" and typically requires us to build a mathematical model that predicts a time course of *biomarker* concentration resembling the observations we have collected from an individual.

Often, the two are combined in a single model because the biomarker predictions are dependent (either directly or indirectly) on the drug predictions.

In this part, we consider factors of our mathematical model such as how drug is administered, distributed and eliminated, and also any errors in the way we observe PK/PD concentrations.

We assume little prior knowledge of PK/PD, and experienced modellers may be tempted to skip this section. It is, however, a gentle introduction to the syntax of *PoPy* models and how they are grown, bit by bit.

CHAPTER

ELEVEN

DOSE ADMINISTRATION

When modelling the passage of drug through the body (and possibly the effect it has on the body), we first need to model how the drug is administered into the body.

The simplest case would be a *bolus* dose in which a single amount of drug is administered instantaneously to a given compartment, such as intravenously into the blood stream (*e.g.* an injection) or via absorption into a 'depot' compartment (*e.g.* an orally administered pill).

There are, of course, other means of administering a drug and here we will introduce the dosing functions supported by *PoPy*. All of the dosing functions have two arguments in common:

- the *amt* argument is the quantity of drug administered in total.
- the *lag* argument defines a delay between the drug being administered (as specified in the *data file*) and the drug entering the compartment where it is administered. This can be useful when very little drug is absorbed immediately. Unless otherwise stated, it is assumed that the lag time is zero.

The remaining arguments to the dosing function are dependent on the dose type.

11.1 Bolus Dose

A bolus dose represents an instantaneous increase in the amount of a drug in a specific compartment. Physiologically it represents an injection where the drug is assumed to be well distributed within the compartment within a negligible time period.

Note: See the *Bolus Dose with no elimination.* for *Tut Script* used to generate results in this section.

The mathematical expression for a bolus at time t_B in compartment **Central** is:-

$$s[CENTRAL] = s[CENTRAL] + B(t)$$

where:-

$$B(t) = \begin{cases} \text{c[AMT]}, \text{ if } t = t_B \\ 0.0, \text{ otherwise} \end{cases}$$

In *PoPy*, we add a bolus dose to a given compartment (*e.g.* **Central**) using the *DERIVATIVES* section of the script:

```
d[CENTRAL] = @bolus{amt: c[AMT], lag: m[LAG]} + ...
```

Because the dose amount is usually fixed by the experiment, it is typically included in the input data and is therefore a column (or covariate) and encoded as such, *e.g.* `c[AMT]`.

The lag time, however, is usually estimated as a model parameter and would typically be coded as such, *e.g.* `m[LAG]`. If lag time is not included in the bolus function, it defaults to zero:

```
d[CENTRAL] = @bolus{amt: c[AMT], lag: 0.0} + ...
```

which is exactly the same as

```
d[CENTRAL] = @bolus{amt: c[AMT]} + ...
```

See *@bolus* for some more syntax examples.

The cumulative amount in the compartment (with no elimination) is therefore a step function (Fig. 1):

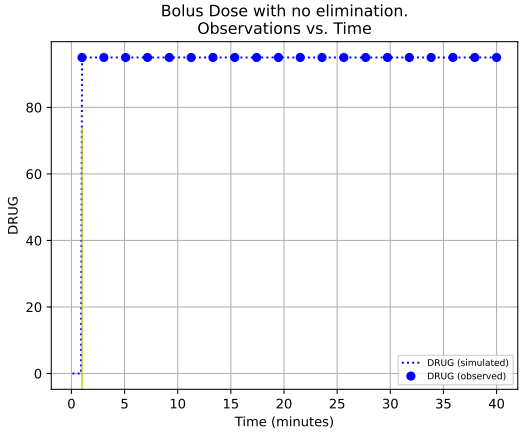


Fig. 1: Cumulative amount following a bolus dose with no elimination

An example of a bolus dose added to a one compartment model with no lag time is:

```
DERIVATIVES: |
    d[CENTRAL] = @bolus{amt:c[AMT]} - c[CL]*s[CENTRAL]/c[V]
```

11.2 Infusion

An infusion administers the dose gradually, usually at a fixed rate over a fixed period of time (*i.e.* the rate of infusion is constant during the time period and zero before and after, typically intravenously).

The mathematical expression for an infusion is:

$$\text{d[CENTRAL]} = \text{d[CENTRAL]} + \text{R(t)}$$

where the infusion has constant rate `c[RATE]`, starts at time *t=0* and has duration `c[DUR]` and in compartment CENTRAL is:-

where:-

$$R(t) = \begin{cases} c[RATE], \text{ if } t_S \leq t \leq t_S + c[DUR] \\ 0.0, \text{ otherwise} \end{cases}$$

An infusion dose can be characterised either by the duration of the infusion or the rate of the infusion. As the rate of change is constant, an infusion is also known as zero order absorption.

Note: For a constant total infusion amount (AMT) the RATE and DURATION can be calculated from each other:

$$\text{AMT} = \text{RATE} \times \text{DURATION}$$

So specifying either a RATE or DURATION, together with a total amount, is sufficient to fully define an infusion.

11.2.1 Infusion Duration

An infusion duration is coded by:

```
@inf_dur{amt: c[AMT], lag: m[LAG], dur: c[DUR]}
```

in the equation for the appropriate compartment in the **DERIVATIVES:** section of a fit or tutorial script.

Dose and lag are coded in the same way as for a bolus dose.

Duration can either be included in the input data or can be estimated as a parameter.

Note: See the *Infusion Duration Dose with no elimination.* for *Tut Script* used to generate results in this section.

An example of an infusion dose spread over 20 time units added to a one compartment model with no lag time is:

```
DERIVATIVES: |
    d[CENTRAL] = (
        @inf_dur{amt: c[AMT], lag: 0.0, dur: 20}
        - m[KE]*s[CENTRAL]
    )
```

If the infusion duration varies between individuals use:

```
DERIVATIVES: |
    d[CENTRAL] = (
        @inf_dur{amt: c[AMT], lag: 0.0, dur: c[DUR]}
        - m[KE]*s[CENTRAL]
    )
```

See *@inf_dur* for some more syntax examples.

The Amount-vs-Time curve for an infusion (without elimination) is a ramp function where the cumulative dose rises linearly until it reaches the total amount administered (Fig. 2).

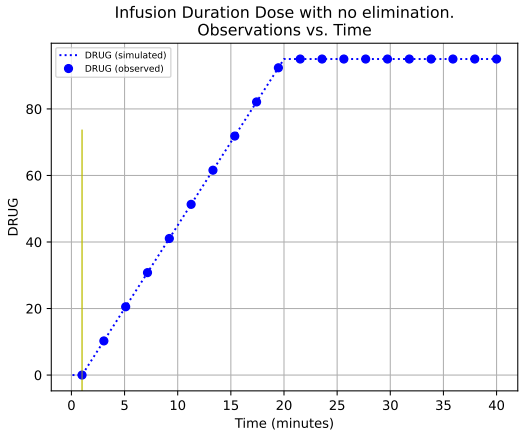


Fig. 2: Cumulative amount following a infusion dose of 95 mg over a duration of 19 min with no elimination

11.2.2 Infusion Rate

An infusion duration is coded by:

```
@inf_rate{amt: c[AMT], lag: m[LAG], rate: c[RATE]}
```

in the equation for the appropriate compartment in the **DERIVATIVES:** section of a fit or tutorial script.

The amount and lag are coded in the same way as for a bolus dose.

Infusion rate can either be included in the input data or can be estimated as a parameter.

Note: See the *Infusion Rate Dose with no elimination.* for *Tut Script* used to generate results in this section.

An example of an infusion dose at a rate of 5 dose units per time unit added to a one compartment model with no lag time is:

```
DERIVATIVES: |
    d[CENTRAL] = (
        @inf_rate{amt: c[AMT], lag: 0.0, rate: 5}
        - m[KE]*s[CENTRAL]
    )
```

If the infusion rate is different between individuals use:

```
DERIVATIVES: |
    d[CENTRAL] = (
        @inf_rate{amt: c[AMT], lag: 0.0, rate: c[RATE]}
        - m[KE]*s[CENTRAL]
    )
```

See *@inf_rate* for some more syntax examples.

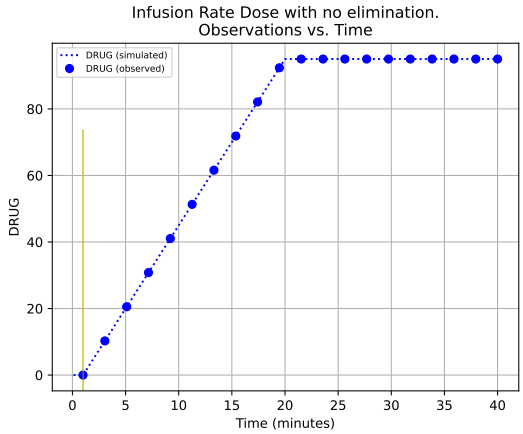


Fig. 3: Cumulative amount following a infusion dose of 95 mg at a rate of 5 mg/min with no elimination

11.3 Gamma Dose

This occurs when the release of the drug into a specific compartment follows a Gamma curve. See:-

https://en.wikipedia.org/wiki/Gamma_distribution

Note: See the *Gamma Dose with no elimination.* for *Tut Script* used to generate results in this section.

The mathematical expression for an Gamma dose starting at time t_S with Weibull parameters λ and κ and total dose amount AMT in compartment **Central** is:-

$$\text{d[CENTRAL]} = \text{d[CENTRAL]} + \text{R(t)}$$

where:-

$$R(t) = \begin{cases} \frac{\beta^{\alpha}(t-t_S)^{\alpha-1}e^{-\beta(t-t_S)}}{\Gamma(\alpha)}, \text{ if } t \geq t_S \\ 0.0, \text{ otherwise} \end{cases}$$

Where R(t) is the rate of drug absorption at time (t) and λ and κ are parameters of the Gamma. R(t) is the Gamma density function scaled by the AMT parameter. Since a density function has unit area under the curve, the AMT scaling ensures that the total dose administered is equal to AMT.

This can be implemented in PoPy using:

```
@gamma{amt: c[AMT], lag: m[LAG], alpha: m[ALPHA], beta: m[BETA]}
```

in the equation for the appropriate compartment in the *DERIVATIVES* section of a *Fit Script* or ref:*tut_script*. See *@gamma*.

Here the parameters map to the Gamma equation as follows:-

Parameter	Sym-bol
alpha	α
beta	β

The lag is coded in the same way as for a bolus dose. A lag time merely delays the start time of the gamma dose.

An example of a gamma dose added to a one compartment model with no lag time is:

```
DERIVATIVES: |
    d[CENTRAL]␣
↪= @gamma{amt: c[AMT], alpha: m[ALPHA], beta: m[BETA]} - m[KE]*s[CENTRAL]
```

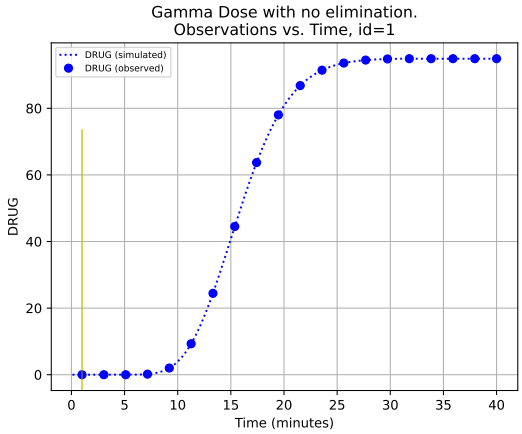


Fig. 4: Cumulative amount following a Gamma dose with no elimination

11.4 Weibull Dose

The examples given so far have shown the amount of drug absorbed with respect to time, rising from an initial value (typically zero) to the amount administered.

They can, alternatively, be viewed as the cumulative probability of 1 mg of drug having been absorbed at any given time, multiplied by the total amount (in mg) of drug administered. (Because a density function has unit area under the curve, the scaling ensures that the total dose administered is equal to *AMT*.) This formulation allows us to consider alternative dosing functions by using different probability distributions.

Note: See the *Weibull Dose with no elimination.* for *Tut Script* used to generate results in this section.

One choice that has been proposed [Piotrovskii1987] uses the Weibull distribution [Christensen1980] where the amount, $S(t)$, absorbed at time t following a dose of AMT at time $t = t_0$ is given by

$$S(t) = AMT \cdot \exp\left\{-\left(\frac{t-t_0}{\lambda}\right)^{\kappa}\right\}$$

such that

$$\frac{dS(t)}{dt} = \left\{ \begin{array}{ll} \frac{\kappa}{\lambda}\left(\frac{t-t_0}{\lambda}\right)^{\kappa-1} \cdot AMT \cdot \exp\left\{-\left(\frac{t-t_0}{\lambda}\right)^{\kappa}\right\}, & \text{for } t \geq t_0 \\ 0, & \text{otherwise} \end{array} \right\}$$

where λ (lambda) and κ (kappa) are, respectively, scale and shape parameters of the Weibull distribution.

In other words, for $t \geq t_0$

$$\frac{dS(t)}{dt} = \frac{\kappa}{\lambda}\left(\frac{t-t_0}{\lambda}\right)^{\kappa-1} \cdot S(t)$$

which can be viewed as a rate absorption constant that varies over time, reflecting the possibility that a molecule of drug may be more likely to be absorbed the longer it is resident in the body (for example, due to passage of the drug into the intestine).

A shortcut for the Weibull dosing function is applied in *PoPy* using

```
@weibull{amt: c[AMT], lag: m[LAG], lambda: m[LAMBDA], kappa: m[KAPPA]}
```

in the equation for the appropriate compartment in the *DERIVATIVES* section of an input script.

The lag works in the same way as for a bolus dose, delaying the onset of the weibull dose. Again, this can be left out if we assume no lag:

```
DERIVATIVES: |
    d[CENTRAL] = @weibull{amt: c[AMT], lambda: m[LAMBDA], kappa: m[KAPPA]}
```

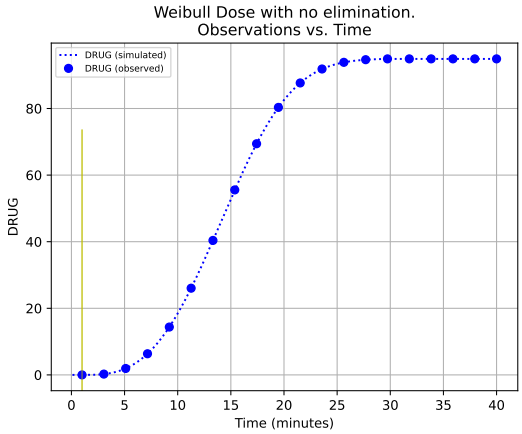


Fig. 5: Cumulative amount following a Weibull dose with no elimination

CHAPTER

TWELVE

ELIMINATION, CLEARANCE AND VOLUME OF DISTRIBUTION

Pharmacokinetics is the study of "what the body does to the drug" (whose administration is modelled in *Dose Administration*).

By and large, the body removes it either by *metabolism* (primarily in the liver) or by direct *elimination* of the unchanged drug (primarily in the kidneys).

Note: See the *Elimination Example with KE parameter* for the *Tut Script* used to generate results in this section.

Consider, as a first example, a bolus dose of 100 mg administered intravenously (IV) at time *t0=1*. Under first order kinetics, the rate of elimination is directly proportional to the amount of drug, $S(t)$, in the body at time *t* and to the constant of proportionality is known as the elimination rate constant, *KE*:

$$\frac{dS}{dt} = -KE \cdot S(t)$$

This *ordinary differential equation* (ODE) has a closed form solution:

$$S(t) = S(t0) \cdot \exp\{-KE \cdot (t - t0)\}$$

i.e. an exponential decay curve, where the initial conditions are

$$S(t0) = 100 \text{ mg}.$$

We can specify this directly in the *PREDICTIONS* section of the control script,

```
PREDICTIONS: |
    plabel[DRUG_AMT] = "Drug Amount (mg)"
    clabel[TIME] = "Time (minutes)"
    p[DRUG_AMT] = c[AMT]*exp(-c[KE]*(c[TIME]-1.0))
    c[DRUG_AMT] ~ norm(p[DRUG_AMT], 0.0)
```

where, for now, we treat *KE* as if it were a constant measurable quantity, `c[KE]`, defined in the *data file*. From this closed form solution, we can predict the concentration at different time points and synthesize observations (Fig. 1).

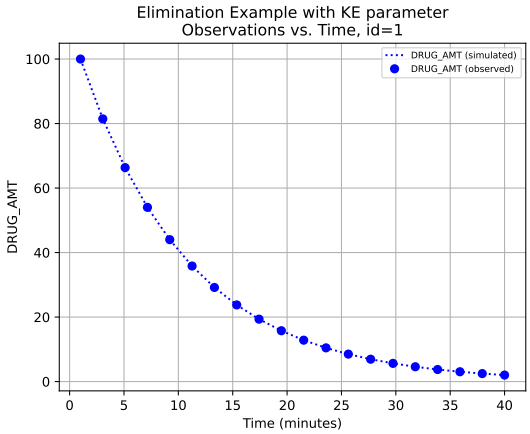


Fig. 1: Amount Administered vs Time curve for an intravenously administered 100 mg bolus dose at time *t0=1* with first order elimination. (Observations are noiseless in this example.)

12.1 Volume of Distribution

In practice, however, we cannot measure the *amount* of drug in the body from a blood plasma sample, only its *concentration* (*i.e.* amount of drug per unit volume).

Note: See the *Elimination Example with Volume of Distribution* for the *Tut Script* used to generate results in this section.

This measured concentration will clearly be influenced by the physiological volume of blood plasma in an individual's body. Less obvious, however, is that the concentration will also be influenced by the physiochemical properties of the drug that determine how the drug is distributed throughout the body; some drugs are distributed mostly in the blood plasma (which is directly observed) whereas others get distributed to all tissues (which are not). The scaling factor – a combination of physiological and physiochemical properties – that relates amounts to concentrations is known as the *volume of distribution*, *V*, which varies greatly between drugs and, to a lesser extent, between individuals.

We model the observed concentration by including the volume of distribution parameter (with a value of 20 L in this example) in the *PREDICTIONS* section of the control script:

```
PREDICTIONS: |
    plabel[DRUG_CONC] = "Drug Concentration (mg/L)"
    clabel[TIME] = "Time (minutes)"
    AMOUNT = c[AMT]*exp(-c[KE]*(c[TIME]-1.0))
    p[DRUG_CONC] = AMOUNT/c[V]
    c[DRUG_CONC] ~ norm(p[DRUG_CONC], 0.0)
```

which has the effect of scaling the observations (Fig. 2; note the scale of the *y*-axis).

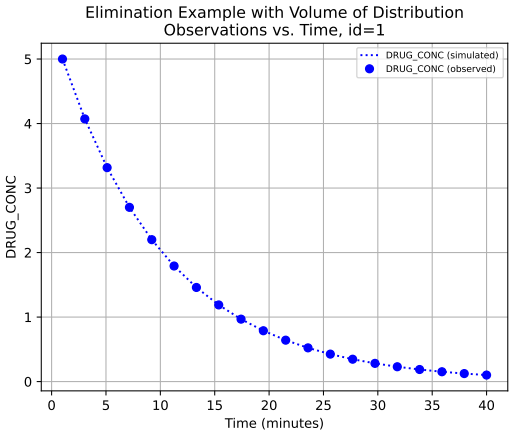


Fig. 2: Concentration vs Time curve for an intravenously administered 100 mg bolus dose with first order elimination

12.2 Clearance

Note: See the *Elimination Example with Clearance* for the *Tut Script* used to generate results in this section.

Because we can only measure concentrations, it makes sense to define the rate of elimination also in terms of concentrations:

$$\begin{aligned}\frac{dS}{dt} &= -KE \cdot V \cdot \frac{S(t)}{V} \\ &= -KE \cdot V \cdot C(t) \\ &= -CL \cdot C(t)\end{aligned}$$

where

$$C(t) = \frac{S(t)}{V}$$

is the concentration at time t and

$$CL = KE \cdot V$$

is a constant of proportionality known as the *clearance*, *CL*, that relates elimination to concentration. Again, this permits a closed form solution,

$$S(t) = S(0) \cdot \exp\{-(CL/V) \cdot (t - t0)\}$$

where we have substituted *CL/V* for the rate constant of elimination, *KE*:

```
PREDICTIONS: |
    plabel[DRUG_CONC] = "Drug Concentration (mg/L)"
    clabel[TIME] = "Time (minutes)"
    AMOUNT = c[AMT]*exp(-(c[CL]/c[V])*(c[TIME]-1.0))
    p[DRUG_CONC] = AMOUNT/c[V]
    c[DRUG_CONC] ~ norm(p[DRUG_CONC], 0.0)
```

Because this is a mathematical equivalence, the Concentration vs Time curve (Fig. 2) is unchanged.

CHAPTER

THIRTEEN

COMPARTMENT MODELS

Although analytic solutions exist for simple models, more complex models can be difficult (or impossible) to define in closed form. Compartment models are a basic tool used to describe more complex PK in animals and man, the idea being that the body can be treated as though it were composed of a number of compartments through which the drug disperses. The concentration of drug in some of these compartments can be measured directly, *e.g.* by taking blood samples.

A typical model contains a **Central** compartment that represents the site of sampling (usually the blood plasma) with zero or more additional compartments. Drug diffuses between compartments (sometimes under the action of transporters) as the system heads toward an equilibrium state in which concentrations in each compartment may or may not be equal.

Drug is lost over time from the **Central** compartment via *elimination* (usually through the action of the liver and kidneys).

Note: Compartmental models are not intended to replicate biology *per se*; they simply produce mathematical curves that resemble observations, and it is difficult to associate any compartment with a specific tissue or organ unless sampling occurs at that site.

13.1 One Compartment Model

13.1.1 One Compartment Model with Intravenous Dosing

We return to the example from the previous chapter whereby a 100 mg bolus dose of drug is administered intravenously into the body, which we now refer to as the **Central** compartment. The drug is then removed from the **Central** compartment by the same process of *elimination* described previously, such that

$$\frac{dS}{dt} = -(CL/V) \cdot S(t)$$

which has the closed form solution

$$S(t) = S(0) \cdot \exp(-(CL/V) \cdot t).$$

Note: See the *One Compartment Model with Intravenous Dosing* for *Tut Script* used to generate results in this section.

Graphically, we can draw a *compartment diagram* that shows the flows into and out of the **Central** compartment (Fig. 1).

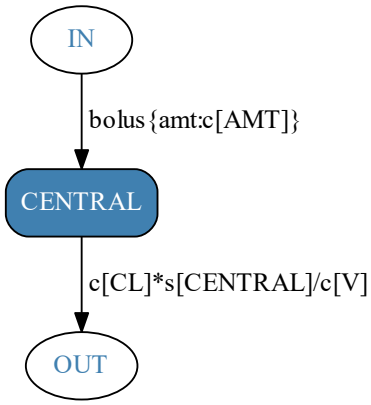


Fig. 1: Compartment diagram for a one compartment model with intravenous dosing.

In the previous chapter, we specified this model programatically by writing the closed form solution directly into the *PREDICTIONS* block of the control script:

```
PREDICTIONS: |
    plabel[DRUG_CONC] = "Drug Concentration (mg/L)"
    clabel[TIME] = "Time (minutes)"
    AMOUNT = c[AMT]*exp(-(c[CL]/c[V])*(c[TIME]-1.0))
    p[DRUG_CONC] = AMOUNT/c[V]
    c[DRUG_CONC] ~ norm(p[DRUG_CONC], 0.0)
```

Using the compartment model approach, however, we can express the same model more naturally using the differential equations themselves, and obtain amounts (and therefore concentrations) using a numerical *ODE* solver. To do so, we need to add a new section, *DERIVATIVES*, to the control script in which we specify the differential equations:

```
DERIVATIVES: |
    d[CENTRAL] = @bolus{amt:c[AMT]} - c[CL]*s[CENTRAL]/c[V]
```

where `d[CENTRAL]` represents the first derivative of the amount of drug, `s[CENTRAL]`, in the **Central** compartment.

There are two components of the flow: a positive flow, representing an intravenous *bolus dose*, *into* the **Central** compartment

```
@bolus{amt: c[AMT]}
```

and a negative flow *out of* the **Central** compartment,

```
-c[CL]*s[CENTRAL]/c[V]
```

that is directly proportional to the concentration, `s[CENTRAL]/c[V]`.

Because we are now using a compartment model, the *PREDICTIONS* section can simply refer to the amount in the **Central** compartment, `s[CENTRAL]`, as determined by the solver:

```
PREDICTIONS: |
    plabel[DV_CENTRAL] = "Drug Concentration (mg/L)"
    clabel[TIME] = "Time (minutes)"
    p[DV_CENTRAL] = s[CENTRAL]/c[V]
    c[DV_CENTRAL] ~ norm(p[DV_CENTRAL], 0.0)
```

and the predicted observations again follow an exponential decay curve (Fig. 2).

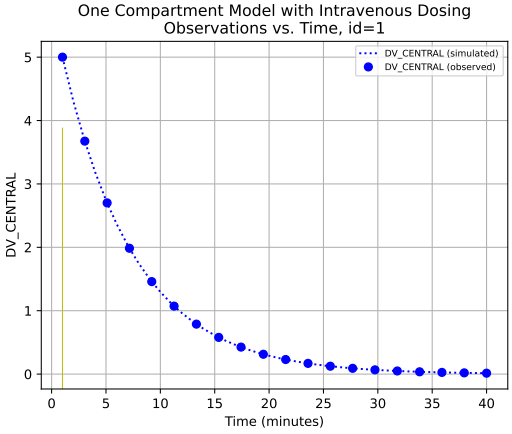


Fig. 2: Concentration vs Time curve for a bolus dose, intravenously applied to the **Central** compartment, with first order elimination.

(For clarity, this example applies the bolus dose at t = 1 s, as indicated by the vertical line in the plot.)

Given that this is a relatively simple model, it has a closed form solution which means we can write down an equation that defines concentration (not its derivative) as a function of time. As a result, we can obtain the concentration at any time point without needing to use a comparatively slow numerical differential solver.

PoPy therefore includes a convenient shortcut to this equation that can be used in the *DERIVATIVES* section of the script to compute directly the amount of drug:

```
DERIVATIVES: |
    s[CENTRAL] = @iv_one_cmp_cl{
        dose: @bolus{amt:c[AMT]},
        CL: c[CL], V: c[V] }
```

- The left hand side of the equation is now s[CENTRAL] rather than d[CENTRAL] because we are defining the amount and not its derivative.
- The shortcut's name has three parts: the first, *iv*, denotes that the dose is intravenous; the second, *one_cmp*, says that there is one compartment; and the third, *cl*, says that we are parameterizing the model using clearance and volumes of distribution.
- The shortcut takes a number of arguments such as *dose* (which defines the properties of the dose such as its amount) and *CL* (which allows you to use any name for the rate constant model parameter, *e.g.* `c[clearance]`).

One quantity of interest in PK is half-life ($t_{1/2}$), the time taken for drug concentration to fall to half that of its peak value. For a one compartment model with IV administration, we can calculate $t_{1/2}$

directly from the closed form solution:

$$
\begin{aligned}
S(t_{1/2}) &= S(0)/2 \\
\Rightarrow \exp\{-(CL/V)\cdot t_{1/2}\} &= 0.5 \\
-(CL/V)\cdot t_{1/2} &= \log(0.5) \\
t_{1/2} &= -\log(0.5)\cdot\frac{V}{CL} \\
t_{1/2} &\approx 0.693\cdot\frac{V}{CL}
\end{aligned}
$$

Next, we turn to the case where the drug is administered via a route that requires *absorption* (*i.e.* anything except intravenous or intra-arterial injection).

13.1.2 One Compartment Model with Absorption

Many drugs are not administered directly into the bloodstream but are given orally, into the gastrointestinal tract, from which absorption must occur before the drug is observed in the plasma. Drugs are also frequently administered by other methods such as subcutaneously or intramuscularly, in which case absorption will also need to be modelled.

We model this using one or more absorption compartments (which we refer to as **Depot** compartments), connected via a one-way flow to the **Central** compartment.

Note: In PKPD terminology, the **Depot** compartment is *not* counted as a compartment, hence this is called a "one compartment model with absorption" even though it has two "compartments".

Note: See the *One Compartment Model with Absorption* for *Tut Script* used to generate results in this section.

In a first order absorption model, the flow from the **Depot** to the **Central** compartment is proportional to the amount of drug in **Depot**, and the constant of proportionality is known as the *absorption rate*, *KA*:

We must therefore model three flows:

- A bolus dose to the **Depot**
- A negative (outward) first order flow from the **Depot** with a rate constant of *KA*. This must be matched by a positive (inward) first order flow to the **Central** compartment.
- A negative (outward) flow from the **Central** compartment to account for *elimination*.

which we specify directly in the *DERIVATIVES* section of the *PoPy* script:

```
DERIVATIVES: |
    d[DEPOT]   = @bolus{amt:c[AMT]} - c[KA]*s[DEPOT]
    d[CENTRAL] = c[KA]*s[DEPOT] - c[CL]*s[CENTRAL]/c[V]
```

PoPy allows you to define the flows between compartments incrementally to simplify the model specification, after first initializing the flows to zero:

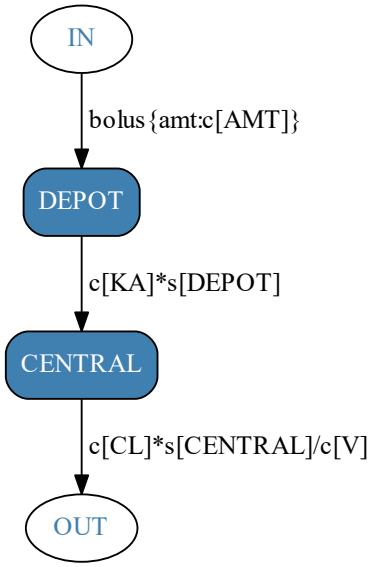


Fig. 3: Compartment diagram for a one compartment model with absorption.

```
DERIVATIVES: |
    # initialize
    d[DEPOT]   = 0.0
    d[CENTRAL] = 0.0
    # update
    d[DEPOT] += @bolus{amt:c[AMT]}  # dose in
    d[DEPOT->CENTRAL] += c[KA]*s[DEPOT]  # absorption
    d[CENTRAL] -= c[CL]*s[CENTRAL]/c[V]  # elimination out
```

In this case, flows in from an external "source" are added using `+=`, flows out to an external "sink" are subtracted using `-=`, and flows between compartments are added using `+=` with the arrow (`->`) notation to define the compartments being linked.

This notation has the advantage that the pairing of compartments is explicit and every flow is defined only once rather than having to maintain two equal and opposite flows, thus reducing the potential for human error when defining the model.

We note that, as in *One Compartment Model with Intravenous Dosing*, *PoPy* provides a closed form solution for this model:

```
DERIVATIVES: |
    s[DEPOT,CENTRAL] = @dep_one_cmp_cl{
        dose: @bolus{amt:c[AMT]},
        KA: c[KA], CL: c[CL], V: c[V]}
```

where `dep` (rather than `iv`) in the first part of the shortcut name denotes that a **Depot** compartment is included.

Regardless of which formulation is used, the resulting Concentration vs Time curve is the same (Fig. 4). In this case, the amount of drug in **Central** rises more slowly than in the intravenous case as the drug is absorbed, peaks at a lower amount, then drops off at an approximately exponential rate as elimination removes the drug from the body.

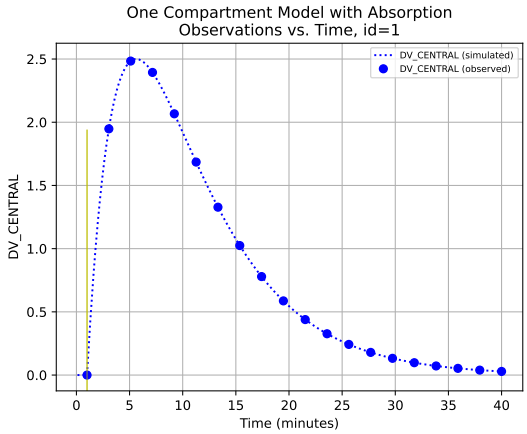


Fig. 4: Concentration vs Time curve for a one compartment model with absorption

---

Note: It is common for a drug to be absorbed more quickly than it is eliminated. In some cases, however, the opposite is true and the elimination rate becomes limited by the absorption rate (because in practice the drug cannot be eliminated faster than it is absorbed). This phenomenon is known as flip flop kinetics.

We now turn to the case where the drug diffuses at different rates between the blood and at least one other organ, modelled by adding a peripheral compartment.

13.2 Two Compartment Model

13.2.1 Two Compartment Model with Intravenous Dosing

For simplicity, we return to intravenous administration of the drug directly into the **Central** compartment. This time, however, we add a peripheral distribution compartment, **Peri**, that models the different distribution of the drug between the blood and/or well perfused organs and other tissues. The peripheral compartment is a useful approximation, which captures the impact of tissues with slower distribution on the shape of the plasma concentration-time profile.

Note: See the *Two Compartment Model with Intravenous Dosing* for *Tut Script* used to generate results in this section.

Because the diffusion can occur in both directions (unlike in absorption), we must now model the flow from **Central** to **Peri** *and* from **Peri** back to **Central**. We therefore introduce two new parameters:

- *Q*, the intercompartmental clearance
- *V2*, the volume of distribution of **Peri** (compartment 2)

and rename the volume of distribution of the **Central** compartment from *V* to *V1* in order to distinguish it from that of the **Peri** compartment:

The corresponding *DERIVATIVES* section is:

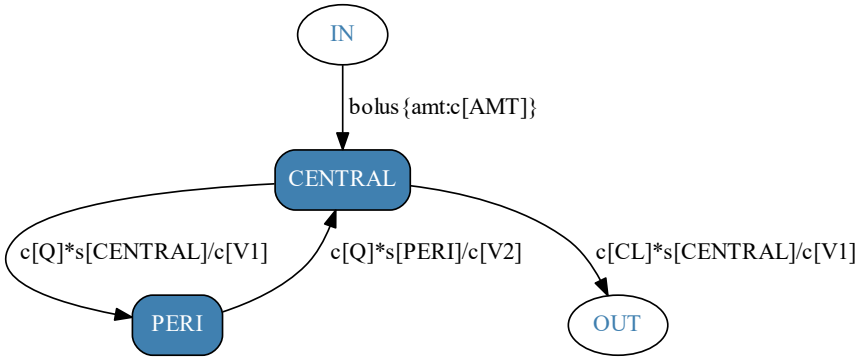


Fig. 5: Compartment diagram for a two compartment model with intravenous dosing.

```
DERIVATIVES: |
    # initialize
    d[CENTRAL] = 0.0
    d[PERI] = 0.0
    # update
    d[CENTRAL] += @bolus{amt:c[AMT]}

↪d[CENTRAL->PERI] += c[Q]*s[CENTRAL]/c[V1]  # intercompartmental diffusion
    d[PERI->CENTRAL] += c[Q]*s[PERI]/c[V2]  # intercompartmental diffusion
    d[CENTRAL] -= c[CL]*s[CENTRAL]/c[V1]
```

and, again, there is a convenient shortcut, `@iv_two_cmp_cl`

```
DERIVATIVES: |
    s[CENTRAL,PERI] = @iv_two_cmp_cl{
        dose: @bolus{amt:c[AMT]},
        CL: c[CL], V1: c[V1],
        Q: c[Q], V2: c[V2] }
```

As in previous examples, the resulting Concentration vs Time curve (Fig. 6) is the same regardless of whether we use flows or the closed form solution because they are equivalent.

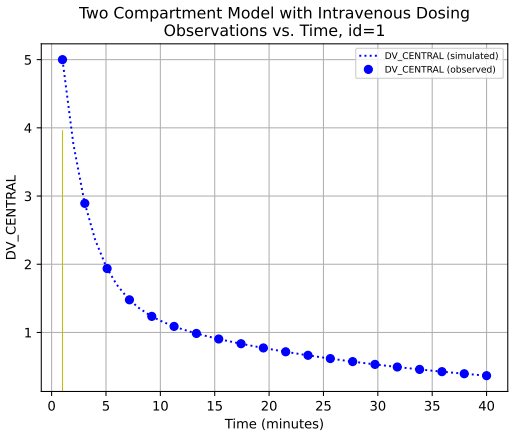


Fig. 6: Concentration vs Time curve for a two compartment model with intravenous dosing

13.2.2 Two Compartment Model with Absorption

We can now model the effect of absorption (*i.e.* a **Depot** compartment), either by adding a **Depot** compartment to iv_two_cmp_cl_tut or by adding a **Peri** compartment to dep_one_cmp_cl_tut.

Note: See the *Two Compartment Model with Absorption* for *Tut Script* used to generate results in this section.

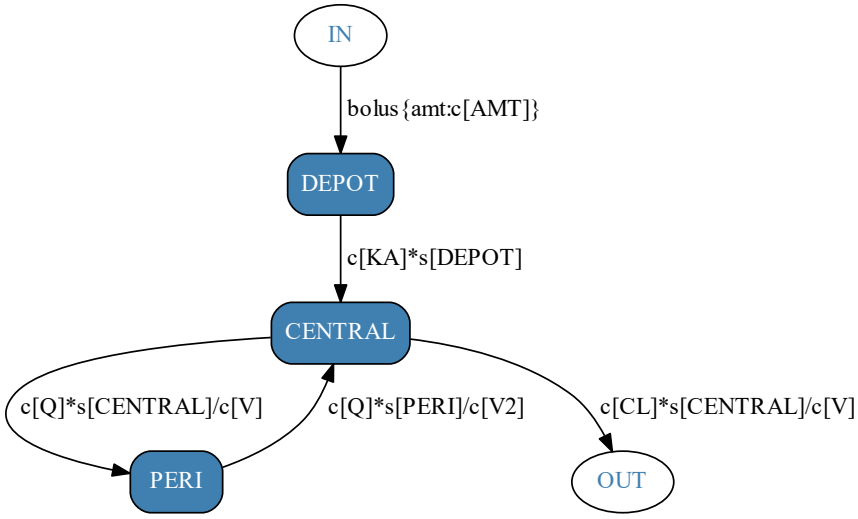


Fig. 7: Compartment diagram for a two compartment model with absorption

The resulting *DERIVATIVES* section is:

```
DERIVATIVES: |
    # initialize
    d[DEPOT]   = 0.0
    d[CENTRAL] = 0.0
    d[PERI] = 0.0
    # add flows
    d[DEPOT] += @bolus{amt:c[AMT]}
    d[DEPOT->CENTRAL] += c[KA]*s[DEPOT]  # absorption
    d[CENTRAL->PERI] += c[Q]*s[CENTRAL]/c[V]
    d[PERI->CENTRAL] += c[Q]*s[PERI]/c[V2]
    d[CENTRAL] -= c[CL]*s[CENTRAL]/c[V]
```

which also has a closed-form shortcut:

```
DERIVATIVES: |
    s[DEPOT,CENTRAL,PERI] = @dep_two_cmp_cl{
        dose: @bolus{amt:c[AMT]},
        KA: c[KA], CL: c[CL], V1: c[V1],
        Q: c[Q], V2: c[V2] }
```

In the resulting Concentration vs Time curve (Fig. 8) we see a combination of the behaviours exhibited in the two earlier examples (iv_two_cmp_cl_tut and dep_one_cmp_cl_tut).

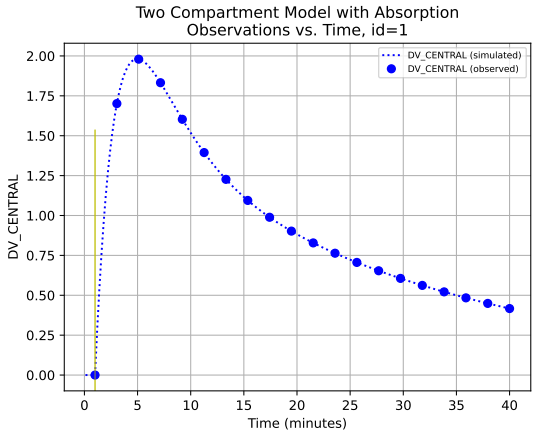


Fig. 8: Concentration vs Time for a two compartment model with absorption

13.3 Three Compartment Model

13.3.1 Three Compartment Model with Intravenous Dosing

For completeness, we look at models with three compartments: the **Central** compartment and two peripheral compartments, **Peri1** and **Peri2**.

Note: See the *Three Compartment Model with Intravenous Dosing* for *Tut Script* used to generate results in this section.

As in *Two Compartment Model with Intravenous Dosing*, we add two new parameters:

- *Q3*, the inter-compartmental clearance between **Central** and **Peri2** (compartment 3)
- *V3*, the volume of distribution of **Peri2** (compartment 3)

and rename the inter-compartmental clearance between **Central** and **Peri1** (compartment 2) from *Q* to *Q2*.

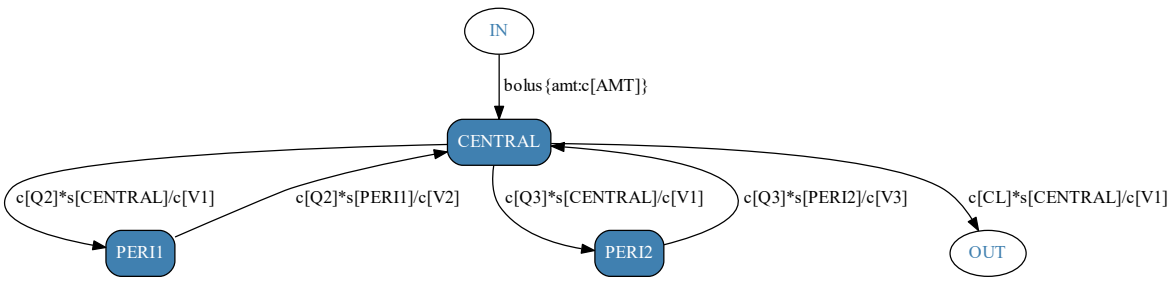


Fig. 9: Compartment diagram for a three compartment model with intravenous dosing

After adding the two new flows to **Peri2**, the *DERIVATIVES* sections is now:

```
DERIVATIVES: |
    # initialize
    d[CENTRAL] = 0.0
```

(continues on next page)

(continued from previous page)

```
    d[PERI1] = 0.0
    d[PERI2] = 0.0
    # update
    d[CENTRAL] += @bolus{amt:c[AMT]}  # dose in
    d[CENTRAL->PERI1] += c[Q2]*s[CENTRAL]/c[V1]
    d[PERI1->CENTRAL] += c[Q2]*s[PERI1]/c[V2]
    d[CENTRAL->PERI2] += c[Q3]*s[CENTRAL]/c[V1]
    d[PERI2->CENTRAL] += c[Q3]*s[PERI2]/c[V3]
    d[CENTRAL] -= c[CL]*s[CENTRAL]/c[V1]  # elimination out
```

and again there is a closed-form shortcut, `iv_three_cmp_cl`.

```
DERIVATIVES: |
    s[CENTRAL,PERI1,PERI2] = @iv_three_cmp_cl{
        dose: @bolus{lag:0, amt:c[AMT]},
        CL: c[CL], V1: c[V1],
        Q2: c[Q2], V2: c[V2],
        Q3:c[Q3], V3: c[V3] }
```

both of which give the same Concentration vs Time curve (Fig. 10), though it is not easy to see the impact of the second peripheral compartment that provides the transitional phase between the very steep initial fall and the shallow first-order terminal phase.

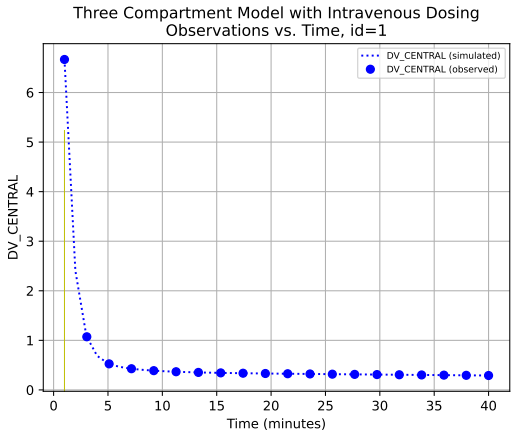


Fig. 10: Amount vs. Time for a three compartment model with intravenous dosing

13.3.2 Three Compartment Model with Absorption

The final model we consider in this chapter is a three compartment model with first order absorption (Fig. 11), specified either by adding a **Depot** compartment to iv_three_cmp_cl_tut or by adding a second peripheral compartment to dep_two_cmp_cl_tut.

Note: See the *Three Compartment Model with Absorption* for *Tut Script* used to generate results in this section.

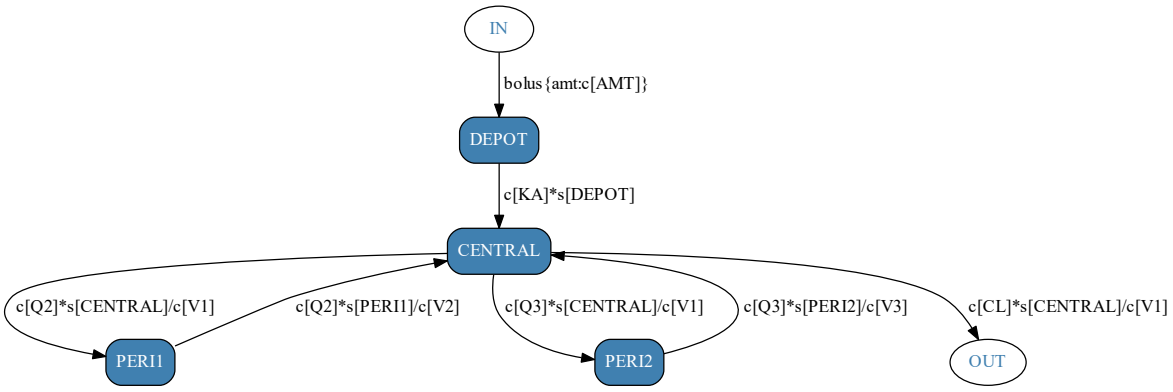


Fig. 11: Compartment diagram for a three compartment model with absorption

The *DERIVATIVES* section,

```
DERIVATIVES: |
    # initialize
    d[DEPOT] = 0.0
    d[CENTRAL] = 0.0
    d[PERI1] = 0.0
    d[PERI2] = 0.0
    # update
    d[DEPOT] += @bolus{amt:c[AMT]} # dose in
    d[DEPOT->CENTRAL] += c[KA]*s[DEPOT]  # absorption
    d[CENTRAL->PERI1] += c[Q2]*s[CENTRAL]/c[V1]
    d[PERI1->CENTRAL] += c[Q2]*s[PERI1]/c[V2]
    d[CENTRAL->PERI2] += c[Q3]*s[CENTRAL]/c[V1]
    d[PERI2->CENTRAL] += c[Q3]*s[PERI2]/c[V3]
    d[CENTRAL] -= c[CL]*s[CENTRAL]/c[V1]  # elimination out
```

also has a closed form shortcut,

```
DERIVATIVES: |
    s[DEPOT,CENTRAL,PERI1,PERI2] = @dep_three_cmp_cl{
        dose: @bolus{lag:0, amt:c[AMT]},
        KA: c[KA], CL: c[CL], V1: c[V1],
        Q2: c[Q2], V2: c[V2],
        Q3: c[Q3], V3: c[V3]}
```

both of which produce the same Concentration vs Time curve, which resembles that from *Two Compartment Model with Absorption* only with a lower peak (Fig. 12).

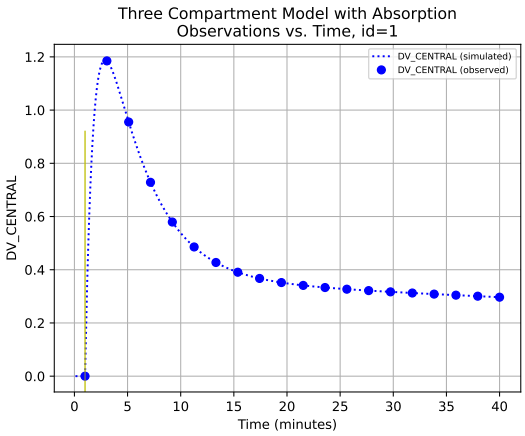


Fig. 12: Concentration vs Time for a three compartment model with absorption

CHAPTER

FOURTEEN

RESIDUAL ERROR MODEL

So far, we have looked at compartment models and different dosing regimes all under theoretical conditions. For example, our Concentration vs Time curve for a one compartment model with absorption produces smooth curves with evenly spaced observation points (Fig. 1).

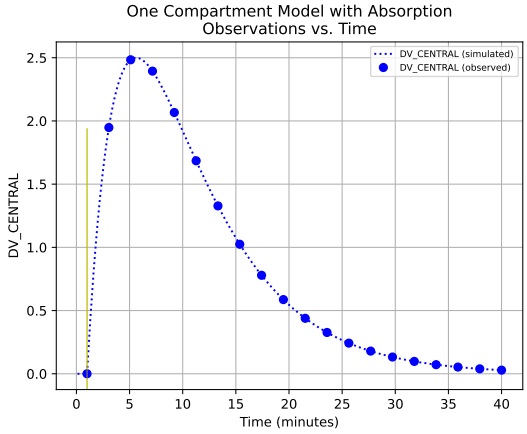


Fig. 1: Concentration vs Time for a one compartment model with absorption, without measurement error (noise).

In practice, however, it is impossible to measure any quantity perfectly; all measurements are imperfect and contain some *noise*. To weight each observation correctly, we may need to account for patterns of noise in the data. For example, noise often increases proportionally with signal so that higher concentrations contain more absolute error. Sometimes we know that observations have been collected in suboptimal conditions and may wish to allow the model to downweight them if they are inconsistent with other data.

We therefore turn to the different ways in which noise manifests itself on the observations and how we can incorporate this into a fitted model.

14.1 Controlling Randomness in Simulations

Note that the .csv data file generated by *Gen Script* on your own machine, will likely contain different values due to the random sampling of *random effect* realizations and then random noise added to each observation.

If you wish to obtain new random results each time your re-run the *Gen Script* then change the 'rand_seed' option to 'auto' as follows:

```
METHOD_OPTIONS: {rand_seed: auto}
```

However if you re-run the *Gen Script* with a fixed number, you should obtain exactly the same results on your machine as before, due to this setting:

```
METHOD_OPTIONS: {rand_seed: 12345}
```

Using a fixed number for the 'rand_seed' makes any sampling process in *PoPy* replicable.

14.2 Error Models for Continuous Data

Specifically, we look at how to model errors that appear on a continuous quantity such as the measured concentration of drug, demonstrating the concept using a normal distribution (a popular choice for continuous measurements).

14.2.1 Additive Noise

Note: See the *Model containing additive error only and additive error only input data* for the *Tut Script* used to generate results in this section.

In the simplest case, the difference between the observed measurement and the model prediction will be a random variable of constant *variance* (or, equivalently, constant standard deviation) such that the observations are scattered evenly around the ideal curve.

```
PREDICTIONS: |
    p[DV_CENTRAL] = s[CENTRAL]/m[V]
    plabel[DV_CENTRAL] = "Drug Concentration (mg/L)"
    add_std = m[ANOISE_STD]
    total_var = add_std**2
    c[DV_CENTRAL] ~ norm(p[DV_CENTRAL], total_var)
    clabel[DV_CENTRAL] = "Drug Concentration (mg/L)"
    clabel[TIME] = "Time (minutes)"
```

With a standard deviation of 1.0, for example, and 100 randomly sampled time points between 1 and 40, we get a more realistic Concentration vs Time curve (Fig. 2).

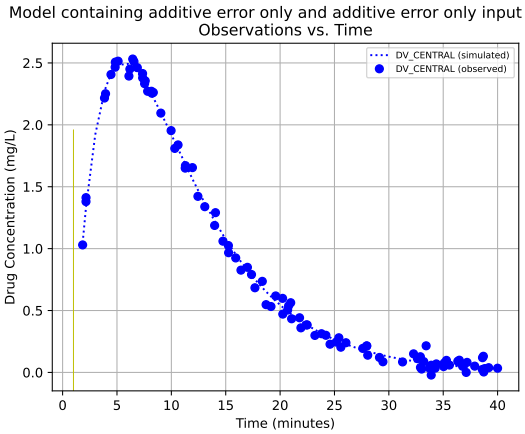


Fig. 2: Concentration vs Time for a one compartment model with absorption, plus additive noise.

14.2.2 Proportional Noise

Note: See the *Model containing proportional error only, with proportional only data* for the *Tut Script* used to generate results in this section.

While additive noise is independent of the strength of the signal (*e.g.* the magnitude of the measurement) at any point in time, proportional noise increases in proportion to the magnitude of the signal. For example, the standard deviation of the noise may be equal to 10% of the signal magnitude such that the error is greatest at the peak of the curve, reducing as the concentration falls (Fig. 3)

```
PREDICTIONS: |
    p[DV_CENTRAL] = s[CENTRAL]/m[V]
    plabel[DV_CENTRAL] = "Drug Concentration (mg/L)"
    prop_std = p[DV_CENTRAL] * m[PNOISE_STD]
    total_var = prop_std**2
    c[DV_CENTRAL] ~ norm(p[DV_CENTRAL], total_var)
    clabel[DV_CENTRAL] = "Drug Concentration (mg/L)"
    clabel[TIME] = "Time (minutes)"
```

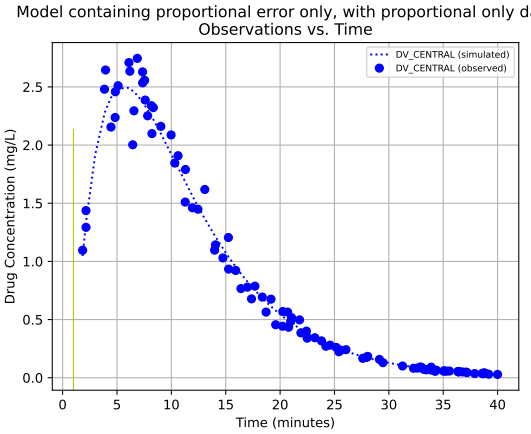


Fig. 3: Concentration vs Time for a one compartment model with absorption, plus proportional noise.

14.2.3 Additive and Proportional Noise

Note: See the *Model containing both proportional and additive error* for the *Tut Script* used to generate results in this section.

In reality, most measurement processes are affected by both kinds of error such that the additive noise dominates at low signal magnitudes whereas proportional noise dominates at high signal magnitudes (Fig. 4).

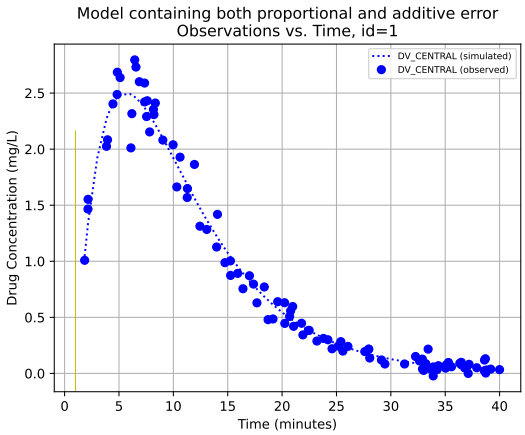


Fig. 4: Concentration vs Time for a one compartment model with absorption, plus both proportional and additive noise

We must, however, take care when defining the variance of a mixed error model. The variance of the sum of two normally distributed variables is the sum of their *variances*, and therefore the *standard deviation* of the sum does *not* equal the sum of their standard deviations:

$$
\begin{aligned}
a &\sim N(\mu_a, \sigma_a^2) \\
b &\sim N(\mu_b, \sigma_b^2) \\
Var(a+b) &= \sigma_a^2 + \sigma_b^2 \\
&\neq (\sigma_a + \sigma_b)^2 \\
Std(a+b) &\neq \sigma_a + \sigma_b
\end{aligned}
$$

as shown in the *PREDICTIONS* block of the *PoPy* input script:

```
PREDICTIONS: |
    p[DV_CENTRAL] = s[CENTRAL]/m[V]
    plabel[DV_CENTRAL] = "Drug Concentration (mg/L)"
    prop_std = p[DV_CENTRAL] * m[PNOISE_STD]
    add_std = m[ANOISE_STD]
    total_var = prop_std**2 + add_std**2
    c[DV_CENTRAL] ~ norm(p[DV_CENTRAL], total_var)
    clabel[DV_CENTRAL] = "Drug Concentration (mg/L)"
    clabel[TIME] = "Time (minutes)"
```

It is crucially important to specify the form of the total variance correctly when fitting if we are to estimate the additive and proportional variances correctly. To see the effect of specifying the variance incorrectly, compare the tutorial script *Mixed error model fitted to mixed error data, but with incorrect variance definition* with its correct counterpart, *Model containing both proportional and additive error*.

CHAPTER

FIFTEEN

PHARMACODYNAMIC MODELS

Pharmacodynamic (PD) models can be categorised as what the drug does to the body. Some physiological process in the body is altered by the presence of a drug and this can be measured somehow.

The pharmacodynamic response could be related to something that the drug is designed to improve such as the QT interval or blood pressure. It could also be a potential toxic effect eg. increased levels of the enzyme Alanine aminotransferase, which could indicate potential liver damage.

The amount, or concentration, of a drug in the plasma will influence the size of the PD response, either directly or indirectly.

15.1 Time Delay

In an indirect model there is a time delay between the drug concentration and the PD response due to the intermediate processes. There may also be a time delay in a direct response because the concentration of drug in the plasma may not be the same as the concentration at the site of the response.

15.2 Direct Effect Models

A direct response means that the drug has a direct effect on a PD process. For example, as levels of Quinidine increase, the length of the Q-T interval increases.

direct_pd_tut is an example of a tutorial script of a simple direct PD model. The pharmacokinetic model is a one compartment model. Values of c[K] for each individual are simulated in the gen_params section. In a real world situation, the values of K would have been estimated by a previous pharmacokinetic model. The amount of drug in the CENTRAL compartment affects the rate at which the MARKER is removed from the system.

This is implemented in the DERIVATIVES section in the script script as shown below:

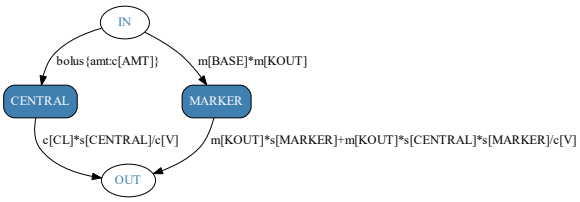
```
DERIVATIVES: |
    d[CENTRAL] = @bolus{amt:c[AMT]} - s[CENTRAL]*c[CL]/c[V]
    d[MARKER] = m[BASE]*m[KOUT] - (1+s[CENTRAL]/c[V])*m[KOUT]*s[MARKER]
```

Note that the following states section is also required to initialise the s[MARKER] value at the steady state concentration at time zero:

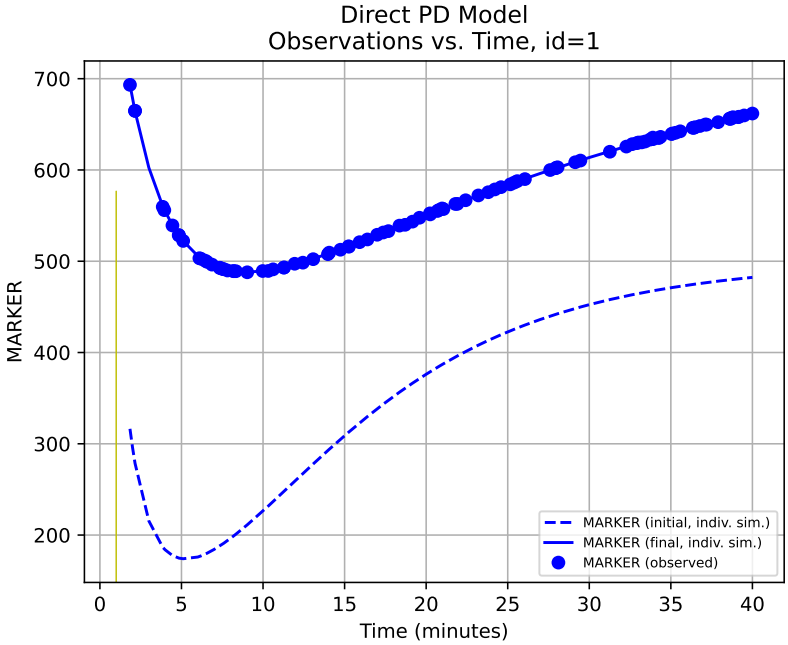
```
STATES: |
    s[CENTRAL] = 0.0
    s[MARKER] = m[BASE]
```

With these initial conditions d[MARKER] =0. s[MARKER] will only change if the drug is administered.

A compartment diagram of this model is shown below:



In contrast to the pharmacokinetic models, there is no actual movement between the compartments. The amount in the CENTRAL compartment just has an effect on the KOUT of the MARKER compartment.



15.3 Indirect Effect Models

In an indirect response PD Model the drug will affect an inter-mediate process, which then influences the measured PD response. A classic example is warfarin. Warfarin inhibits the synthesis of prothrombin, which in turn inhibits the formation of blood clots.

indirect_pd_tut is a similar model, which includes a time delay in the effect on the MARKER compartment. This is achieved by two lag compartments, delaying the drug effect:

```
DERIVATIVES: |
    d[CENTRAL] = @bolus{amt:c[AMT]} - c[CL]*s[CENTRAL]/c[V]
    d[LAG1] = m[KOUT]*s[CENTRAL]/c[V] - m[KOUT]*s[LAG1]
```

(continues on next page)

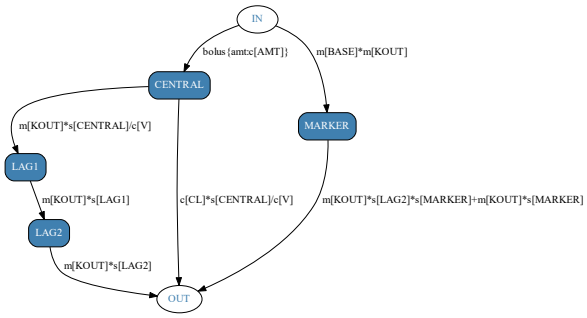
(continued from previous page)

```
    d[LAG2] = m[KOUT]*s[LAG1] - m[KOUT]*s[LAG2]
    d[MARKER] = m[BASE]*m[KOUT] - (1+s[LAG2])*m[KOUT]*s[MARKER]
```

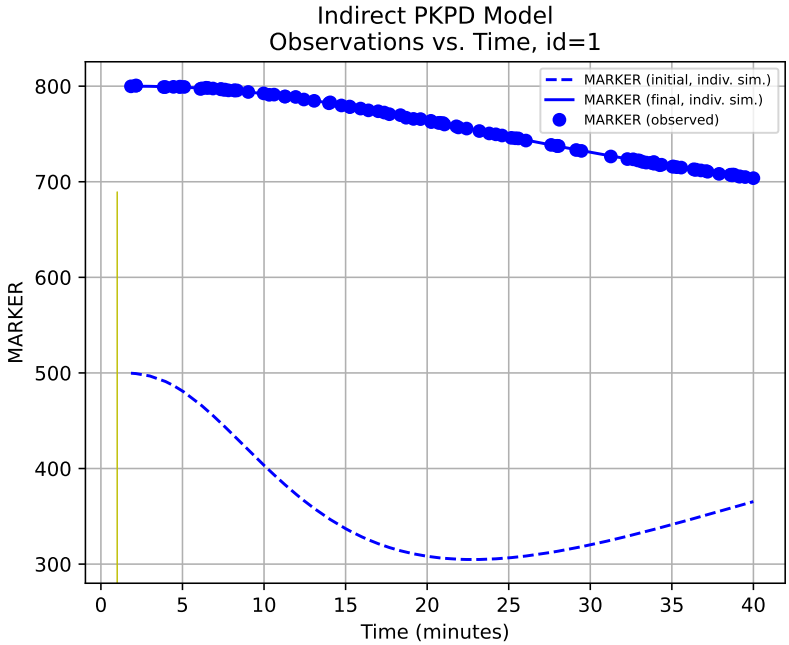
As in the direct model, the initial steady state is achieved as follows:

```
STATES: |
    s[CENTRAL] = 0.0
    s[MARKER] = m[BASE]
```

The actual processes that occur between the CENTRAL compartment and the MARKER compartment are NOT modelled directly because we want to model a time delay and then the eventual effect of the amount in the CENTRAL compartment on the MARKER compartment clearance. A compartment diagram of this model is shown below:



This is similar to *Direct Effect Models*, except that the effect of the amount in the CENTRAL compartment on the MARKER compartment is delayed by passing through the two lag compartments.

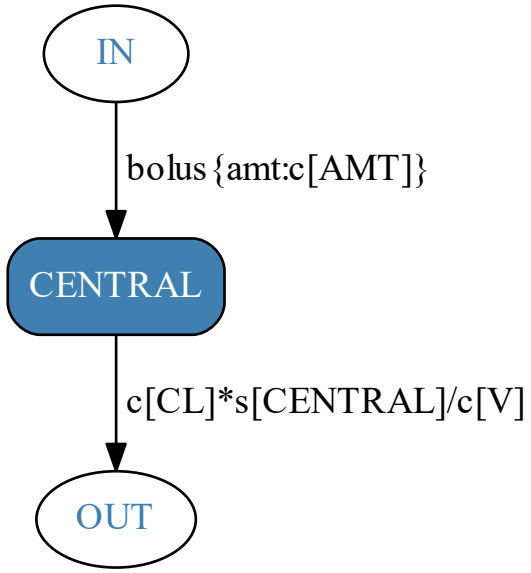


15.4 Linear Effect Model

linear_pd_tut is a direct effect model where the PD Marker response is much simpler. The derivatives block is a standard one compartment PK model a follows:

```
DERIVATIVES: |
    d[CENTRAL] = @bolus{amt:c[AMT]} - c[CL]*s[CENTRAL]/c[V]
```

Resulting in the following compartment diagram:

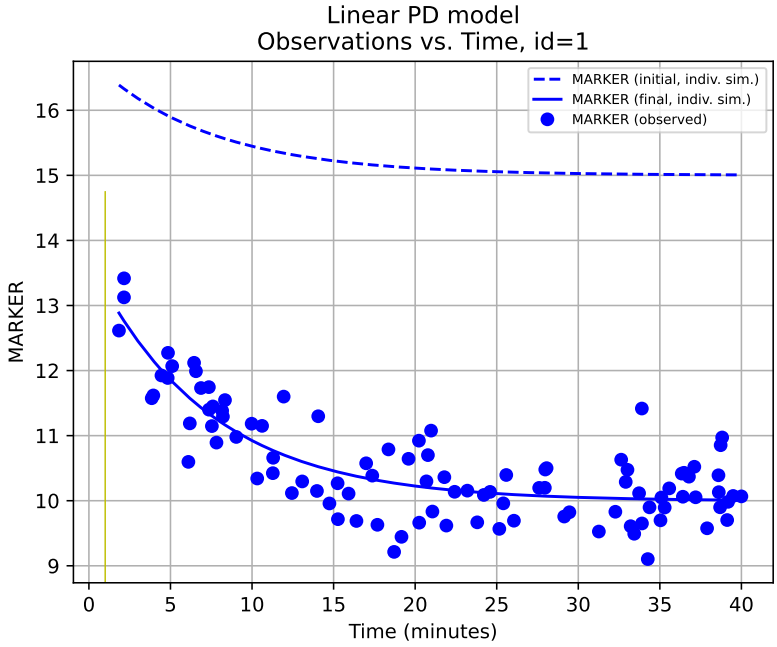


The linear effect is defined in the predictions block as follows:

```
PREDICTIONS: |
    clabel[TIME] = "Time (minutes)"
    plabel[MARKER] = "Biomarker concentration (mg/L)"
    p[MARKER] = m[BL] + m[SLOPE]*s[CENTRAL]/c[V]
    var = m[ANOISE]**2
    c[MARKER] ~ norm(p[MARKER], var)
```

The concentration in the central compartment has a linear effect on the marker response, which is added to a baseline m[BL].

Linear PD models are rarely used because a linear response implies that the bio marker response will be proportional to the drug concentration. This is unlikely to be true in humans because other factors will limit a body's response to a drug.



15.5 Circadian Models

A circadian rhythm is a roughly 24-hour cycle that affects physiological responses in the body. For example, the absorption and metabolisation of a drug can also be faster when taken in the morning compared to the night. Temporal cycles may affect the pharmacokinetics or pharmacodynamics of a drug and so must be taken into account by a model.

15.5.1 Sine Model

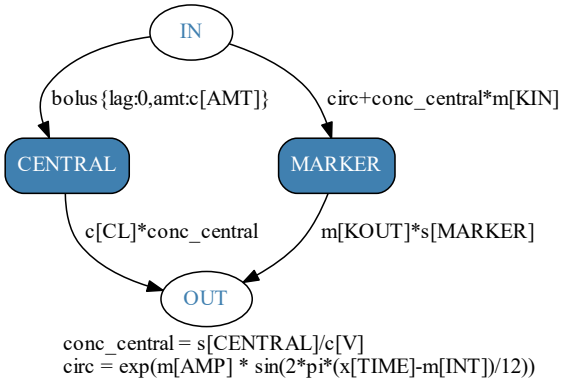
circ_sin_tut is a similar example, but this time the sine function is used to replicate the circadian rhythm. The circadian rhythm has been changed to a half-daily rhythm by substituting 12 for 24 in the MODEL_PARAMS section:

```
MODEL_PARAMS: |
    m[AMP] = f[AMP]
    m[INT] = f[INT]
    m[KOUT] = f[KOUT]
    m[KIN] = c[BASE]*m[KOUT]
    m[ANOISE] = f[ANOISE]
```

The PD model has also been changed so that the concentration in the central compartment now affects the production of the biomarker and the elimination rate of the biomarker is dependent on the amount of biomarker:

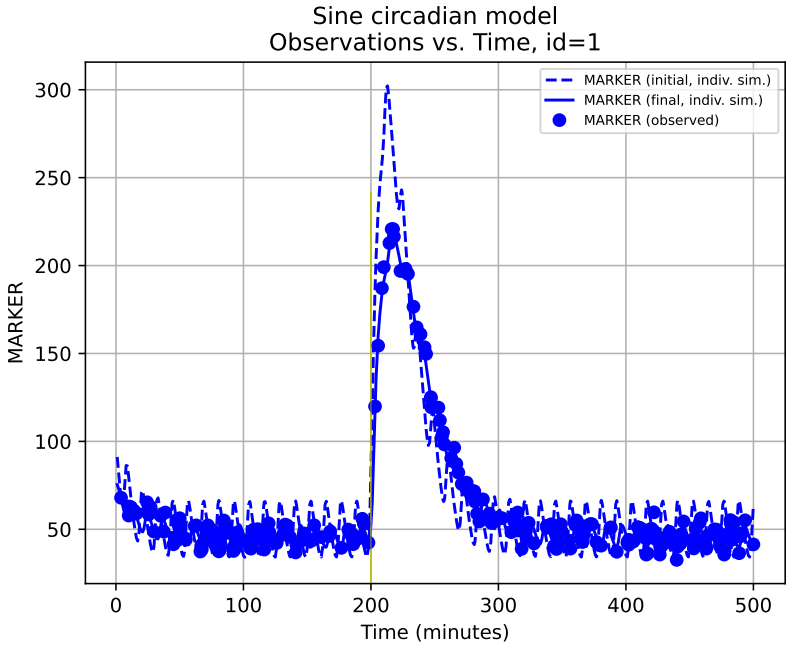
```
DERIVATIVES: |
    conc_central = s[CENTRAL]/c[V]
    circ = exp(m[AMP] * sin(2*pi*(x[TIME]-m[INT])/12))
    d[CENTRAL] = @bolus{lag:0, amt:c[AMT]} - c[CL]*conc_central  # PK
    d[MARKER] =  m[KIN]*conc_central + circ - m[KOUT]*s[MARKER]  # PD
```

The compartment diagram is:



Again the PoPy fit script gives good estimates of the ground truth values used to generate the data.

Name	Initial	Fitted	True	Abs. Error	Prop. Error
f[AMP]	3	2…	2	1.79e-03	0.09%
f[INT]	16	7.87	8	1.29e-01	1.61%
f[KOUT]	0.1	0.05…	0.05	1.56e-05	0.03%
f[ANOISE]	5	3.08	3	7.59e-02	2.53%



15.6 Emax Models

The response of the human body to a drug is unlikely follow a linear pattern. At low drug concentrations, effects may not occur and/or may be hard to measure. At higher concentrations the effect usually saturates. This is because many of the chemical reactions that occur in the body and are targeted by drugs depend on enzymes.

The effects of drugs often follow a sigmoidal (or S-shaped) curve and an emax model can produce this effect. The emax model is based on the equation

$$\text{Response} = \frac{\text{Emax} \times \text{Amount}}{\text{E50} + \text{Amount}}$$

where

$$\text{Emax} = \text{Maximum response}$$

$$\text{Amount} = \text{Amount or concentration of drug}$$

$$\text{E50} = \text{Amount or concentration at which the response is half of the maximum}$$

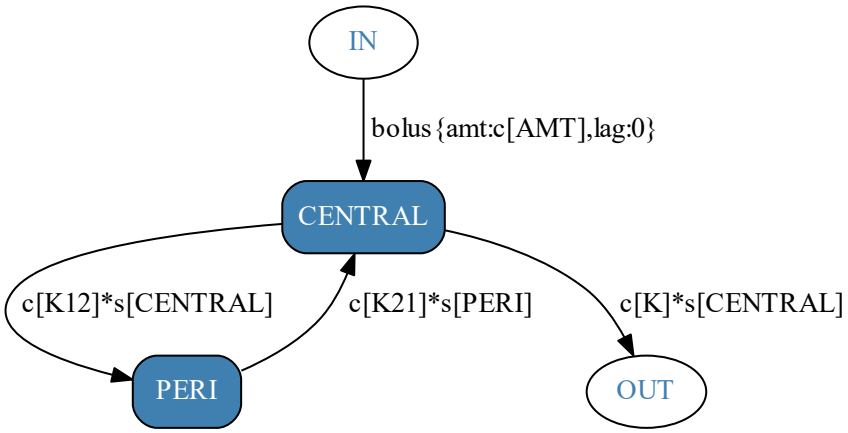
and the steepness of the curve is determined by the hill coefficient (γ), with larger amounts having steeper curves:

$$\text{Response} = \frac{\text{Emax} \times \text{Amount}^{\gamma}}{\text{E50}^{\gamma} + \text{Amount}^{\gamma}}$$

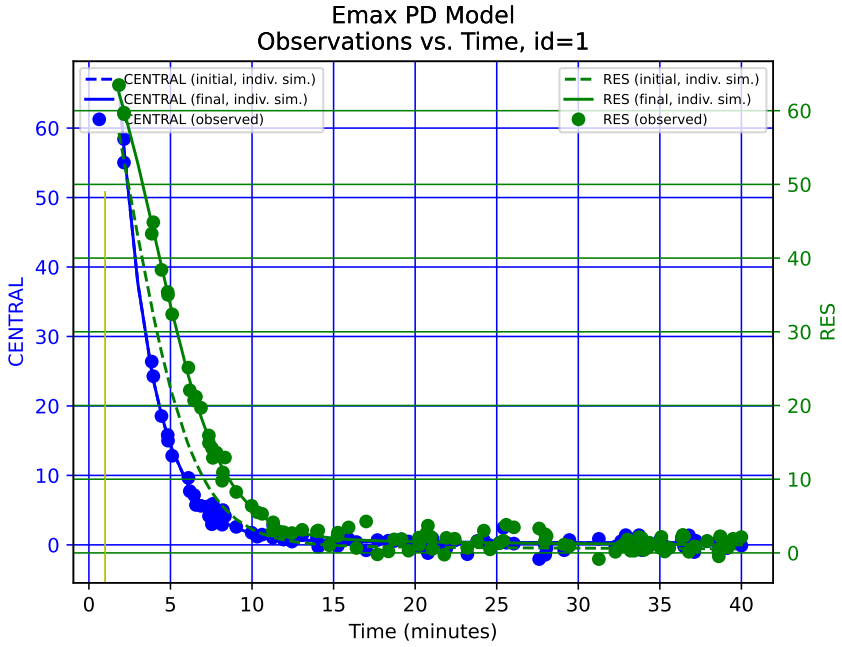
15.6.1 Simple Emax Model

emax_simple_tut is a simple emax model. The two compartment PK model is based on previous estimates of K, K12 and K21. The response compartment is based on the amount of drug in the central compartment:

$$\text{Response} = \frac{\text{Emax} \times \text{Amount}}{\text{E50} + \text{Amount}}$$



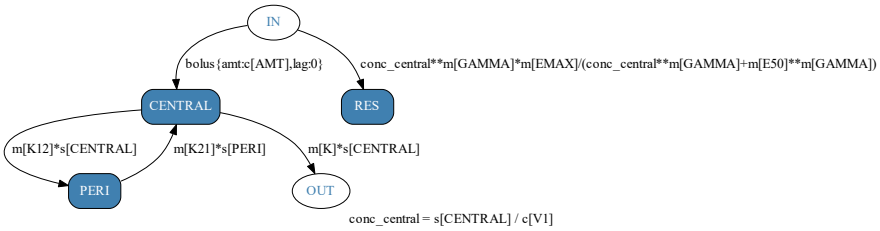
Note that there is no flow out of the response compartment in this model and so the amount in the repsonse compartment grows with each time step.



15.6.2 More Complex Emax Model

emax_complex_tut is a more complex version of emax_simple_tut. This model uses the concentration in the central compartment, rather than the amount to influence the response compartment. Concentration is often used instead of amounts because it is the concentration of a drug in the plasma that is measured in clinical trials. This model also has a gamma (γ) parameter to incorporate the steepness of the slope:

$$\text{Response} = \frac{\text{Emax} \times \text{conc}^{\gamma}}{\text{E50}^{\gamma} + \text{conc}^{\gamma}}$$



Again, the response accumulates in the response compartment.

15.6.3 Non-Compartmental Response Emax Model

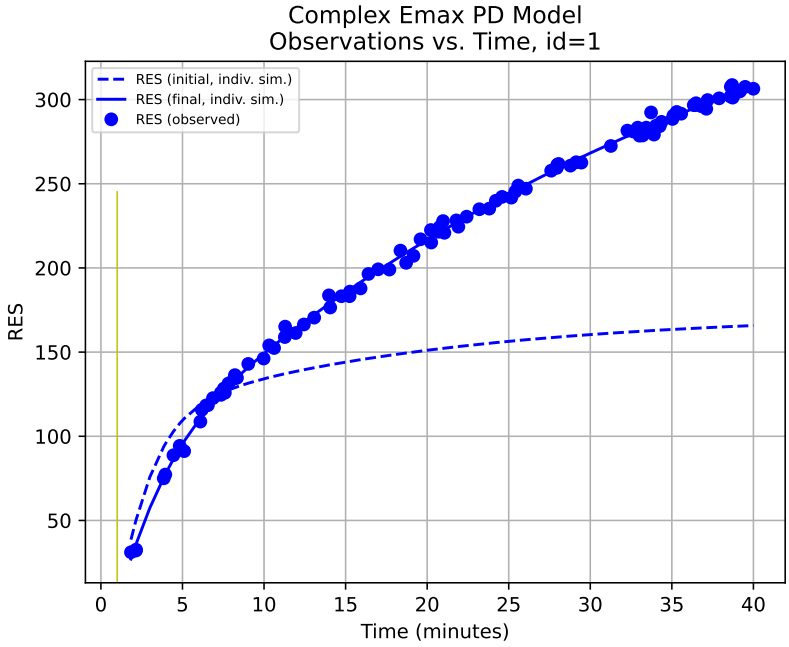
emax_nondiff_tut has a one compartment PK model, but the response variable is outside of the DERIVATIVES block:

```
DERIVATIVES: |
    d[CENTRAL] = @bolus{amt:c[AMT], lag:0} - m[K]*s[CENTRAL]
```

and is instead in the PREDICTIONS block:

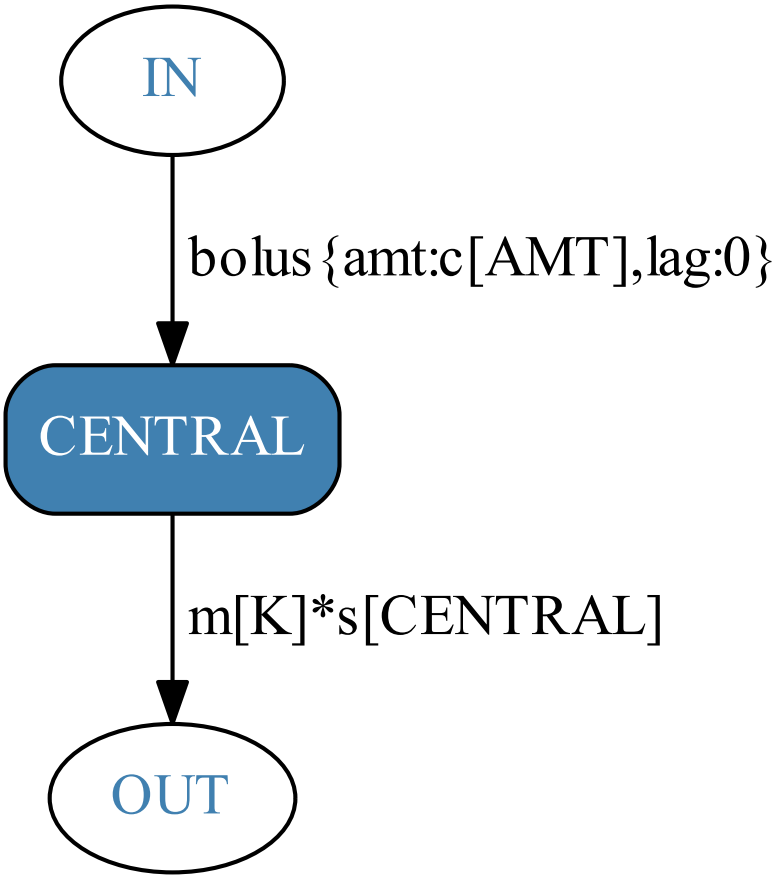
```
PREDICTIONS: |
    clabel[TIME] = "Time (minutes)"
```

(continues on next page)

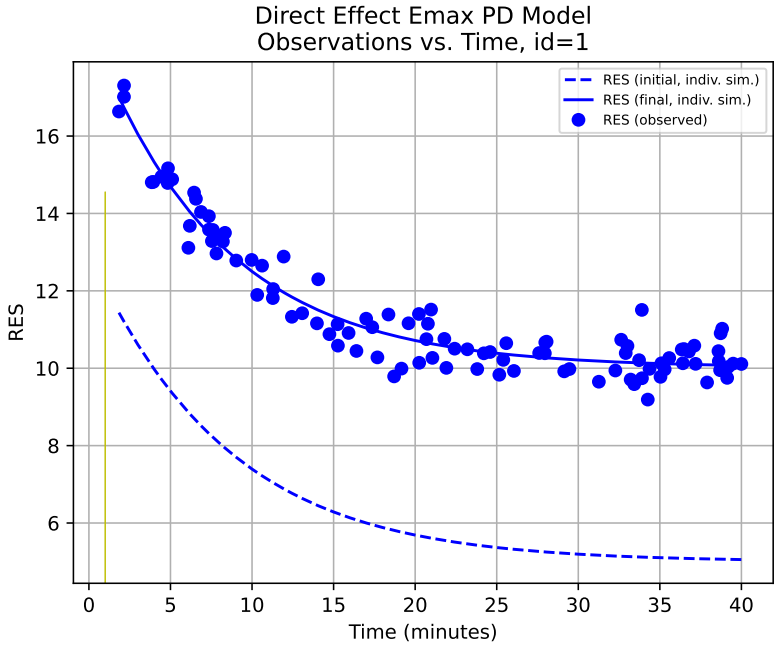


(continued from previous page)

```
    plabel[RES] = 'Response'
    conc = s[CENTRAL]/c[V]
    p[RES] = m[EBASE] + (m[EMAX]*conc)/(m[E50]+conc)
    var =  m[ANOISE]**2
    c[RES] ~ norm(p[RES], var)
```



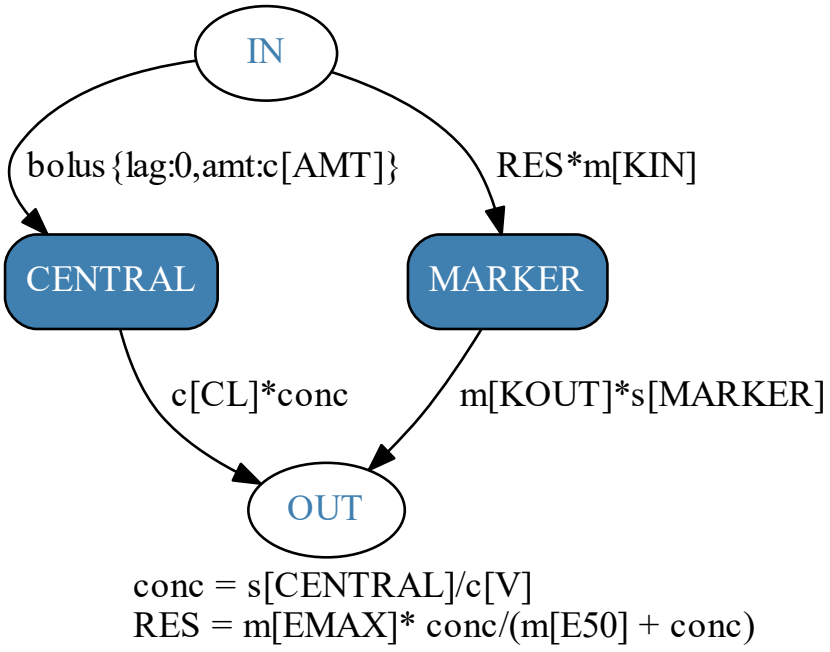
This means the response changes with the concentration in the central compartment and does not accumulate over time. The greater the concentration of drug in the plasma, the greater the response will be.

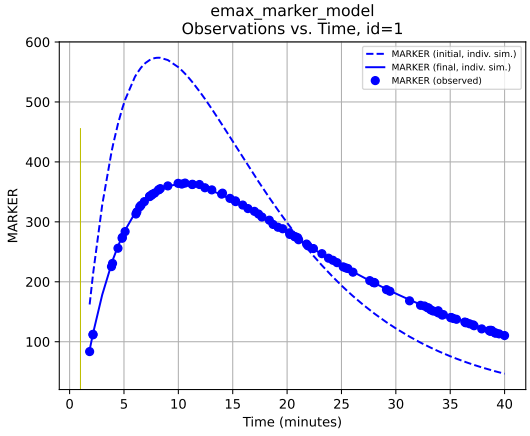


15.6.4 Biomarker Emax Model

emax_biomarker_tut is an example of a biomarker model. The one compartment PK model estimates the changes in drug concentration in the plasma. This causes a response, which affects the production of a biomarker. The amount of the biomarker being produced increases with increasing drug concentrations:

```
DERIVATIVES: |
    conc = s[CENTRAL]/c[V]
    d[CENTRAL] = @bolus{lag:0, amt:c[AMT]} - c[CL]*conc
    RES = m[EMAX]* conc/(m[E50] + conc)
    d[MARKER] = m[KIN]*RES - m[KOUT]*s[MARKER]
```





15.7 Other Stuff

- binary response models
- categorical response models
- ordered categorical modelling
- time to event/survival models
- count data

15.8 PD Parameter Fitting

In practice PKPD models can be fitted simultaneously or separately (see simultaneous_vs_separate_fit).

In the simultaneous case, the model is fitted to PK and PD data at the same time and the m[X] individual PK + PD parameters. deduced at the same time.

In the separate case the PK model is fitted and the PK model parameters for each individuals recorded. The data is then merged with original data set and the PK parameters appear as c[X] variables when fitting the PD model.

In this case study the PD models are demonstrated as the second step in a separate PKPD fit. Therefore PK parameters are assumed to be known (from a prior model fit) and therefore appear as c[X] variables. The PD parameters are m[X] variables, to be deduced by the model.

Part IV

Simulating A Population Of Individuals

Because people vary in many, many ways, we should rarely draw conclusions of any analysis based on a single individual.

This part therefore introduces concepts related to variability within a *population* of individuals (and *Population PK/PD* from which *PoPy* gets its name), using *PoPy* example scripts to generate data and figures for ease of learning.

Variability between individuals comes in two flavours:

1. *Observable* variability that can be *measured* directly before a trial even takes place, such as age, body weight and exposure to other drugs that may have an impact on an individual's PK/PD. These variables are known as *covariates* and are usually recorded in the data before analysis takes place.
2. *Unobservable* Variability that must be *estimated* as part of the data analysis.

Early attempts at estimating the unobservable variability relied on the residual error model "taking up the slack" but it quickly became clear that the estimates of population parameters were biased; when dealing with a population of subjects, a single set of model parameters cannot capture the variation in concentration time courses in a sensible way.

A significant advance in the field came with the development of *mixed effect models* that model unobservable variability between individuals as random deviations (known as *random effects*) from population averages (known as *fixed effects*). Furthermore, the distributional assumptions we choose for the newly introduced *random effects* constrain the problem mathematically, making it practical to find a "best" local fit even with sparse observations.

The population on average is therefore characterized by the fixed effects, while each individual in the population has PK/PD parameters that are also dependent on both covariates and random effects.

CHAPTER

SIXTEEN

INTER-SUBJECT VARIABILITY (ISV)

One way to model random variability between individuals is to use a *mixed effect model*. With this approach, we assign a set of one or more *random effects* to every individual such that their model parameters (*e.g.* the clearance, `m[CL]`) can deviate from the population values.

The random effect, `r[CL_isv]` for example, is a realization of a random variable drawn from a probability distribution. Typically, this is a normal distribution with the mean fixed at zero and a variance that is estimated from the data, though other distributions are possible.

This is then combined with the population value to give an individualized value for the PK parameter. For example, we can use an additive model such that an individual's pharmacokinetic parameters deviate from the population mean:

```
m[CL] = f[CL] + r[CL_isv]
```

Additive models can also be used for parameters such as lag times when the exact time of a dose event is not known, or when a measurement has no meaningful zero reference point (like temperature).

Most physiological quantities, however, are constrained to be positive (*e.g.* clearances, volumes) such that an additive model can cause problems if `m[CL]` becomes negative. It is therefore common instead to impose a log-normal distribution over `m[CL]` by applying the additive model in a log-transformed space:

```
m[LOG_CL] = log(f[CL]) + r[CL_isv]
...
CL = exp(m[LOG_CL])
```

or, equivalently,

```
m[CL] = f[CL] * exp(r[CL_isv])
```

16.1 Data Synthesis with ISV

Note: See *One Compartment Model with Absorption and Inter-subject Variance f[CL_isv]=0.2*, *One Compartment Model with Absorption and Inter-subject Variance f[CL_isv]=0.01* and *One Compartment Model with Absorption and Inter-subject Variance f[CL_isv]=0.5* for the *Tut Script* used to generate results in this section.

We illustrate Inter Subject Variability (sometimes referred to as "Between Subject Variability") using an example based on the one compartment model with absorption that was introduced earlier (*One Compartment Model with Absorption*).

Because we now want several individuals, we make three changes to the *EFFECTS* section of the script:

1. move values that vary between individuals (`c[ID]`, `c[AMT]` and time points `t[RESET]`, `t[DOSE]` and `t[OBS]`) from the **POP** level into a new level, *ID*.
2. add to the *ID* level an individual-specific random effect, `r[CL_isv]`, that is responsible for inter-subject variation in the model parameter *CL*.
3. add to the **POP** level a population-specific fixed effect, `f[CL_isv]`, that specifies the *variance* (*i.e.* the spread) of the random effects, `r[CL_isv]`.

```
EFFECTS:
    POP: |
        f[KA] = 0.3
        f[CL] = 3
        f[V] = 20
        f[PNOISE_STD] = 0.1
        f[ANOISE_STD] = 0.05
        # new: true inter-subject variability
        f[CL_isv] = 0.2
    ID: |
        # new: 30 individuals for this population
        c[ID] = sequential(30)
        c[AMT] = 100.0
        t[RESET] = 0.0
        t[DOSE] = 1.0
        t[OBS] ~ unif(1.0, 50.0; 4)
        # new: inter-subject variability
        r[CL_isv] ~ norm(0, f[CL_isv])
```

Note: The *ID* level sits below **POP** to indicate that there should be a branch of the hierarchy created for every individual in the population.

During data synthesis, *PoPy* adds random variation between individuals by sampling realizations of `r[CL_isv]` from the probability distribution. These realizations of `r[CL_isv]` are then used in the *MODEL_PARAMS* section to add inter-subject variability to the model parameter, `m[CL]`.

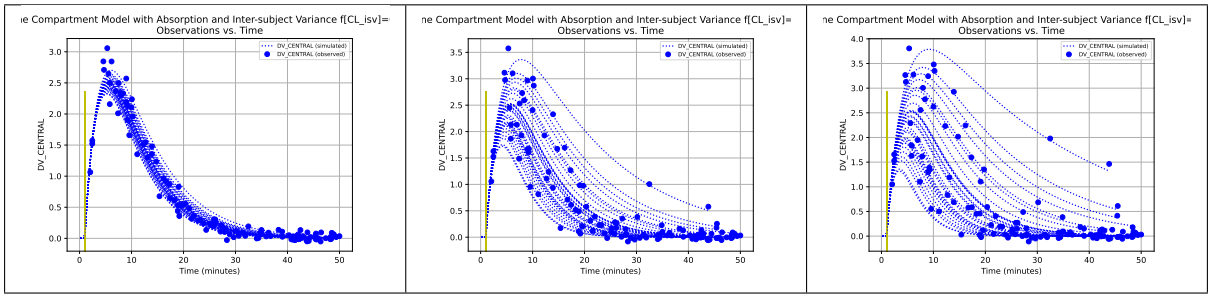
```
MODEL_PARAMS: |
    m[KA] = f[KA]
    # new: log-normally distributed ISV
    m[CL] = f[CL]*exp(r[CL_isv])
    m[V] = f[V]
    m[PNOISE_STD] = f[PNOISE_STD]
    m[ANOISE_STD] = f[ANOISE_STD]
```

The rest of the script (*e.g.* *DERIVATIVES* and *PREDICTIONS*) are the same as for a single individual.

This variation changes the shape of the predicted curves over the set of individuals in the population

(Table 1).

Table 1: Population graphs with increasing ISV: (left) CL_isv=0.01; (centre) CL_isv=0.2; (right) CL_isv=0.5



16.2 Naïve ISV Fit

Note: See the *One Compartment Model with Absorption and no inter-subject Variance f[CL_isv]=0* for the *Tut Script* used to generate results in this section.

Given a dataset where there are 30 subjects with ISV, we first investigate the effect of excluding ISV from the model we fit.

We can do this without changing the data synthesis code (*e.g.* the *MODEL_PARAMS* section) by fixing `f[CL_isv]` at zero such that all deviations from the population prediction are accounted for under the residual error model.

```
EFFECTS:
    POP: |
        f[KA] ~ unif(0.01, 1) 0.5
        f[CL] ~ unif(0.01, 10) 1
        f[V] ~ unif (0.01, 100) 15
        f[PNOISE_STD] ~ unif(0.001, 1) 0.2
        f[ANOISE_STD] ~ unif(0.001, 1) 0.2
        # new: assumed zero inter-subject variability
        f[CL_isv] = 0.0
    ID: |
        # new: inter-subject variability
        r[CL_isv] ~ norm(0, f[CL_isv])
```

After the fit, we see that the population parameters (*i.e.* the fixed effects) are poorly estimated. In particular, the noise levels `f[PNOISE_STD]` and `f[ANOISE_STD]` are far higher than the true values because they are responsible for capturing the variation that, in reality, is ISV rather than noise.

```
f[KA] = 0.2087
f[CL] = 3.0963
f[V] = 14.8134
f[PNOISE_STD] = 0.3766
```

(continues on next page)

(continued from previous page)

```
f[ANOISE_STD] = 0.1619
f[CL_isv] = 0.0000
```

As a reference point, the final objective function value is

```
-200.9397695675923
```

16.3 Mixed Effect ISV Fit

Note: See the *One Compartment Model with Absorption and Inter-subject Variance f[CL_isv]=0.2* for the *Tut Script* used to generate results in this section.

We now look at a mixed effects model that allows model parameters (and predictions) to be tailored to each individual by making the ISV variance, `f[CL_isv]`, a parameter to be optimized:

```
EFFECTS:
    POP: |
        f[KA] ~ unif(0.01, 1) 0.5
        f[CL] ~ unif(0.01, 10) 1
        f[V] ~ unif (0.01, 100) 15
        f[PNOISE_STD] ~ unif(0.001, 1) 0.2
        f[ANOISE_STD] ~ unif(0.001, 1) 0.2
        # new: estimated inter-subject variability
        f[CL_isv] ~ unif(0.001, 1) 0.01
    ID: |
        # new: inter-subject variability
        r[CL_isv] ~ norm(0, f[CL_isv])
```

As a result, the population parameters are now much closer to the values used for data synthesis (shown above). In particular, we see that the noise parameters are much more accurate since the ISV can be accounted for using `f[CL_isv]` rather than the residual noise model.

```
f[KA] = 0.3019
f[CL] = 3.2981
f[V] = 19.6503
f[PNOISE_STD] = 0.0712
f[ANOISE_STD] = 0.0545
f[CL_isv] = 0.1844
```

For comparison, we can also see that the final objective function value is also far lower than that obtained without modelling ISV:

```
-394.58995634065906
```

Variation between subjects is just one of many sources of randomness in population data. The next source we will look at is variation within a subject from one visit to another - *Inter-Occasion Variability (IOV)*.

CHAPTER

SEVENTEEN

INTER-OCCASION VARIABILITY (IOV)

Just as PK parameters can vary from one individual to the next, they can also vary *within* an individual over a period of time. Clearances and volumes, for example, can often vary from day-to-day, particularly where inflammation plays a role but even in healthy volunteers. Absorption parameters can also vary substantially with each dosing occasion, sometimes to the extent that IOV is even greater than ISV. As a result, when drugs are administered on multiple occasions it may be important to model this variation, too [KarlssonSheiner1993].

17.1 Data Synthesis with IOV

Note: See the *One Compartment Model with Absorption and Inter-occasion Variance f[CL_isv]=0.2* and *One Compartment Model with Absorption and Inter-occasion Variance f[CL_isv]=0.5* for the *Tut Script* used to generate results in this section.

First, we generate some new data with both ISV and IOV by making more changes to the *EFFECTS* section of the script:

1. move values that vary between occasions (just the time points `t[RESET]`, `t[DOSE]` and `t[OBS]`, unless the dose amount, `c[AMT]` varies with occasion) from the *ID* level into a new level, **OCCASION**.
2. add to the **OCCASION** level an occasion-specific covariate, `c[OCC]`, that indicates the occasion, and an occasion-specific random effect, `r[CL_iov]`, that is responsible for inter-occasion variation in the model parameter *CL*.
3. add to the **POP** level a population-specific fixed effect, `f[CL_iov]`, that specifies the *variance* (*i.e.* the spread) of the inter-occasional random effects, `r[CL_iov]`.

```
EFFECTS:
    POP: |
        f[KA] = 0.3
        f[CL] = 3
        f[V] = 20
        f[PNOISE_STD] = 0.1
        f[ANOISE_STD] = 0.05
        f[CL_isv] = 0.2
        # new: true inter-occasion variability
```

(continues on next page)

(continued from previous page)

```
        f[CL_iov] = 0.1
    ID: |
        # new: 10 subjects in this population
        c[ID] = sequential(10)
        c[AMT] = 100.0
        r[CL_isv] ~ norm(0, f[CL_isv])
    OCC: |
        # new: 3 occasions per individual
        c[OCC] = sequential(3)
        t[RESET] = 0.0
        t[DOSE] = 1.0
        t[OBS] ~ unif(1.0, 50.0; 4)
        # new: inter-occasion variability
        r[CL_iov] ~ norm(0, f[CL_iov])
```

(Here we have also reduced the population to ten individuals with three occasions each in order to maintain 30 time courses.)

Note: Adding the **OCCASION** level below the *ID* level tells *PoPy* that there are multiple occasions for an individual, much as adding *ID* below **POP** says that there are multiple subjects in the population. Every occasion therefore inherits the parameters defined in the levels above. (See *EFFECTS*.)

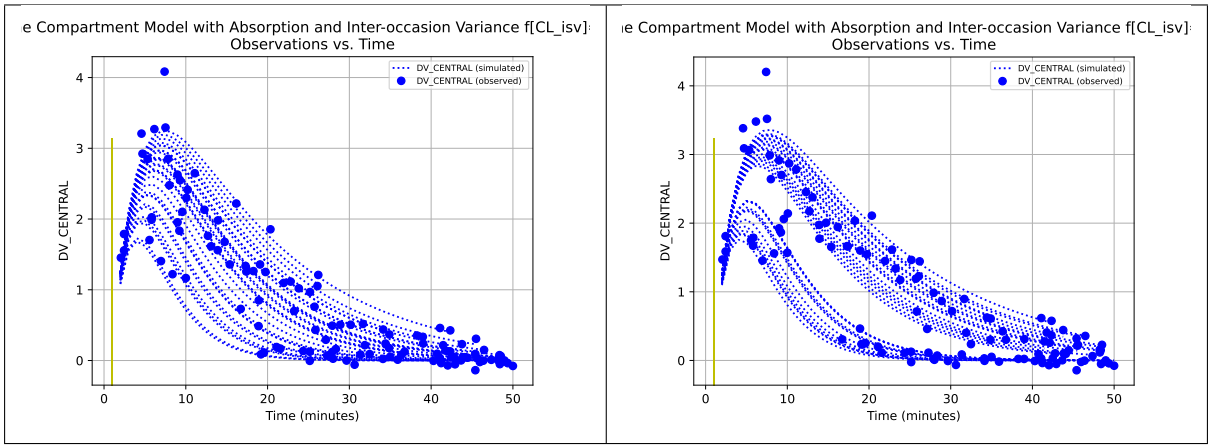
The IOV random effect is then added to its ISV sibling in the *MODEL_PARAMS* section of the script:

```
MODEL_PARAMS: |
    m[KA] = f[KA]
    # log-normally distributed ISV and IOV
    m[CL] = f[CL]*exp(r[CL_isv] + r[CL_iov])
    m[V] = f[V]
    m[PNOISE_STD] = f[PNOISE_STD]
    m[ANOISE_STD] = f[ANOISE_STD]
```

The remaining sections (*e.g.* *DERIVATIVES* and *PREDICTIONS*) remain the same.

As with ISV, this added variation changes the shape of the predicted curves over the set of individuals in the population (Table 1). Depending on the magnitude of the variances, the set of curves may overlap such that it is difficult to separate the individuals or they may form tight clusters. (The overlap will also increase with bigger populations.)

Table 1: Population graphs with increasing ISV: (left) CL_isv=0.2, CL_iov=0.1; (right) CL_isv=0.5, CL_iov=0.01



17.2 Naïve IOV Fit

Note: See *One Compartment Model with Absorption and no inter-occasion Variance f[CL_iov]=0* for the *Tut Script* used to generate results in this section.

As with ISV, we can investigate the effect of excluding IOV from a fit to data where IOV exists by fixing the `f[CL_iov]` variance to zero.

```
EFFECTS:
    POP: |
        f[KA] ~ unif(0.01, 1) 0.5
        f[CL] ~ unif(0.01, 10) 1
        f[V] ~ unif (0.01, 100) 15
        f[PNOISE_STD] ~ unif(0.001, 1) 0.2
        f[ANOISE_STD] ~ unif(0.001, 1) 0.2
        f[CL_isv] ~ unif(0.001, 10) 0.01
        # new: assumed zero inter-occasion variability
        f[CL_iov] = 0
    ID: |
        r[CL_isv] ~ norm(0, f[CL_isv])
    OCC: |
        # new: inter-occasion variability
        r[CL_iov] ~ norm(0, f[CL_iov])
```

Again, although the population parameters are close to their "true" values, the noise parameters and the ISV are all bigger than they should be in order to capture the IOV that has not been modelled

```
f[KA] = 0.3202
f[CL] = 2.5349
f[V] = 20.9761
f[PNOISE_STD] = 0.2297
```

(continues on next page)

(continued from previous page)

```
f[ANOISE_STD] = 0.0976
f[CL_isv] = 0.1395
f[CL_iov] = 0.0000
```

and we use the resulting objective function value,

```
-253.73372112248674
```

as a reference.

17.3 Mixed Effect IOV Fit

Note: See *One Compartment Model with Absorption and Inter-occasion Variance f[CL_isv]=0.2* for the *Tut Script* used to generate results in this section.

Adding IOV to the model by making `f[CL_iov]` a free parameter

```
EFFECTS:
    POP: |
        f[KA] ~ unif(0.01, 1) 0.5
        f[CL] ~ unif(0.01, 10) 1
        f[V] ~ unif (0.01, 100) 15
        f[PNOISE_STD] ~ unif(0.001, 1) 0.2
        f[ANOISE_STD] ~ unif(0.001, 1) 0.2
        f[CL_isv] ~ unif(0.001, 10) 0.01
        # new: estimated inter-occasion variability
        f[CL_iov] ~ unif(0.001, 10) 0.01
    ID: |
        r[CL_isv] ~ norm(0, f[CL_isv])
    OCC: |
        # new: inter-occasion variability
        r[CL_iov] ~ norm(0, f[CL_iov])
```

results in population parameters that are again close to their "true" values, this time including the variance parameters. This reflects the intuition that a correctly defined model requires less variance to capture the observed deviations from the population estimates.

```
f[KA] = 0.3345
f[CL] = 2.5855
f[V] = 20.2051
f[PNOISE_STD] = 0.1000
f[ANOISE_STD] = 0.0479
f[CL_isv] = 0.1200
f[CL_iov] = 0.0783
```

As with the ISV example, the final objective function value,

```
-363.81175569174985
```

is also significantly lower, indicating a better fit of the model to the observed concentrations.

CHAPTER

EIGHTEEN

MODELLING CORRELATION IN RANDOM EFFECTS

As shown in *Inter-Subject Variability (ISV)*, we can use *random effects* to account for variability in model parameters between individuals in a population. In that example, we demonstrated the principle for a single parameter whereas we now consider the case where two or more parameters vary between individuals. In particular, we are interested in what happens when pairs of parameters vary in similar ways, for example when they have a common underlying cause.

In the following examples we model inter-subject variability in both `m[CL]` and `m[V]` using a combined proportional and additive noise model to generate observations for 200 individuals, and fit various models to the synthesized data.

18.1 Uncorrelated Effects

Note: See the *Diagonal matrix generation diagonal matrix fit using separate univariate normals* and *Diagonal matrix generation diagonal matrix fit* for the *Tut Script* used to generate results in this section.

We can synthesize observations where parameters are uncorrelated simply by creating, for each parameter, an independent random effect with its own variance:

```
EFFECTS:
    POP: |
        f[KA] = 0.3
        f[CL] = 3
        f[V] = 20
        f[PNOISE_STD] = 0.1
        f[ANOISE_STD] = 0.05
        f[CL_isv] = 0.2
        # new: ISV on volume of distribution
        f[V_isv] = 0.1
    ID: |
        c[ID] = sequential(200)
        c[AMT] = 100.0
        t[RESET] = 0.0
        t[DOSE] = 1.0
        t[OBS] ~ unif(1.0, 50.0; 4)
        r[CL_isv] ~ norm( 0, f[CL_isv] )
```

(continues on next page)

(continued from previous page)

```
        # new: ISV on volume of distribution
        r[V_isv] ~ norm( 0, f[V_isv] )
```

Mathematically, this is equivalent to modelling the two parameters jointly via a *covariance matrix* with the individual variances along the diagonal and zeros everywhere else. In *PoPy* this structure is enforced using the `~diag_matrix()` notation:

```
EFFECTS:
    POP: |
        f[KA] = 0.3
        f[CL] = 3
        f[V] = 20
        f[PNOISE_STD] = 0.1
        f[ANOISE_STD] = 0.05
        # new: Joint ISV on both CL and V
        f[CL_isv, V_isv] ~ diag_matrix() [[0.2, 0.1]]
    ID: |
        c[ID] = sequential(200)
        c[AMT] = 100.0
        t[RESET] = 0.0
        t[DOSE] = 1.0
        t[OBS] ~ unif(1.0, 50.0; 4)
        # new: Joint ISV on both CL and V
        r[CL_isv, V_isv] ~ mnorm( [0,0], f[CL_isv, V_isv] )
```

for the covariance matrix and using a multivariate normal distribution, `~mnorm()`, to model the two random effects as a vector (in this case with zero mean). Throughout this chapter we will continue to use the covariance matrix form rather than independent variables.

With independent inter-subject variability in `m[CL]` and `m[V]` the generated curves (Fig. 1) vary widely from individuals with high CL and low V (*i.e.* a high KE) to individuals with a low CL and high V (*i.e.* a low KE).

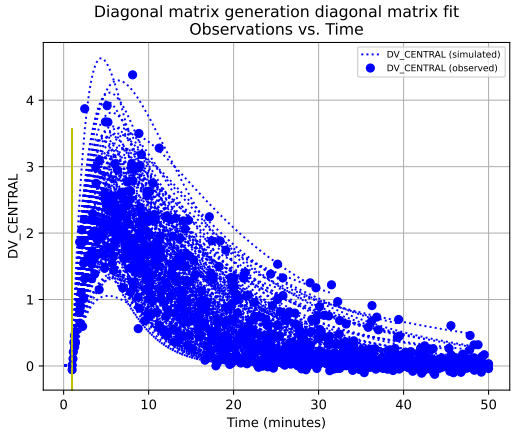


Fig. 1: Simulated prediction curves for a population with independent inter-subject variability on parameters CL and V

In all cases here, we ensure that the generated observations are the same by specifying a fixed

`rand_seed` option in the *METHOD_OPTIONS* section of the tutorial script:

```
METHOD_OPTIONS: {py_module: gen, rand_seed: 314159}
```

18.1.1 Uncorrelated Fit to Uncorrelated Effects

Note: See the *Diagonal matrix generation diagonal matrix fit* for the *Tut Script* used to generate results in this section.

We now consider fitting (*i.e.* estimating the parameters of) a model using the observations, where the model also imposes a diagonal structure on the covariance matrix:

```
EFFECTS:
    POP: |
        f[KA] = 0.3
        f[CL] = 3
        f[V] = 20
        f[PNOISE_STD] = 0.1
        f[ANOISE_STD] = 0.05
        # new: Joint ISV on both CL and V
        f[CL_isv, V_isv] ~ diag_matrix() [[0.01, 0.01]]
    ID: |
        # new: Joint ISV on both CL and V
        r[CL_isv, V_isv] ~ mnorm( [0,0], f[CL_isv, V_isv] )
```

In this case, the only population parameters we estimate are the population variances, `f[CL_isv]` and `f[V_isv]`. The fitted parameter values,

```
f[KA] = 0.3000
f[CL] = 3.0000
f[V] = 20.0000
f[PNOISE_STD] = 0.1000
f[ANOISE_STD] = 0.0500
f[CL_isv,V_isv] = [
    [ 0.1977, 0.0000 ],
    [ 0.0000, 0.1167 ],
]
```

largely agree with the "true" values used to generate the observations, albeit with lower values for the variances than the true values (a phenomenon known as *shrinkage*).

The final objective function value (OBJV) for this fit is

```
-2215.965842502594
```

18.1.2 Correlated Fit to Uncorrelated Effects

Note: See the *Diagonal matrix generation full matrix fit* for the *Tut Script* used to generate results in this section.

We could, however, use a different covariance structure when fitting the model to the observations. For example, we could relax the constraints on the off-diagonal elements by allowing any covariance matrix that is *symmetric positive definite* (SPD) - a necessary property for all covariance matrices.

In *PoPy*, we do this using the `~spd_matrix()` notation:

```
EFFECTS:
    POP: |
        f[KA] = 0.3
        f[CL] = 3
        f[V] = 20
        f[PNOISE_STD] = 0.1
        f[ANOISE_STD] = 0.05
        # new: Model that links variability of
        # CL to that of V
        # zero off diagonal element causes fail for SPD matrix hmmm...
        # f[CL_isv, V_isv] ~ spd_matrix() [
        #    [ 0.01 ],
        #    [ 0.0, 0.01 ]
        # ]
        f[CL_isv, V_isv] ~ spd_matrix() [
            [ 0.01 ],
            [ 0.0001, 0.01 ]
        ]
    ID: |
        r[CL_isv, V_isv] ~ mnorm( [0,0], f[CL_isv, V_isv] )
```

Using the "full" covariance matrix the fitted values,

```
f[KA] = 0.3000
f[CL] = 3.0000
f[V] = 20.0000
f[PNOISE_STD] = 0.1000
f[ANOISE_STD] = 0.0500
f[CL_isv,V_isv] = [
    [ 0.1928, -0.0166 ],
    [ -0.0166, 0.1155 ],
]
```

also largely agree with those fitted using the diagonally-constrained covariance since a diagonal matrix *is* a specific example of an SPD matrix. This can also be seen in the fitted objective function value,

```
-2217.2437915379865
```

which is only fractionally lower than that for the diagonally-constrained model, since the full covariance

matrix has more freedom to fit to the data.

18.2 Correlated Effects

Note: See the *Full matrix generation diagonal matrix fit* for the *Tut Script* used to generate results in this section.

We now run the same experiments, only using new observations that have been synthesized from a model where the variability in `m[CL]` and `m[V]` *is* correlated. This is a more realistic example, since both clearance and volume of distribution can increase with weight.

```
EFFECTS:
    POP: |
        f[KA] = 0.3
        f[CL] = 3
        f[V] = 20
        f[PNOISE_STD] = 0.1
        f[ANOISE_STD] = 0.05
        # new: Data in which variability of CL is
        # linked to that of V
        f[CL_isv, V_isv] ~ spd_matrix() [
            [ 0.15 ],
            [ 0.05, 0.15 ],
        ]
    ID: |
        c[ID] = sequential(200)
        c[AMT] = 100.0
        t[RESET] = 0.0
        t[DOSE] = 1.0
        t[OBS] ~ unif(1.0, 50.0; 4)
        r[CL_isv, V_isv] ~ mnorm( [0,0], f[CL_isv, V_isv] )
```

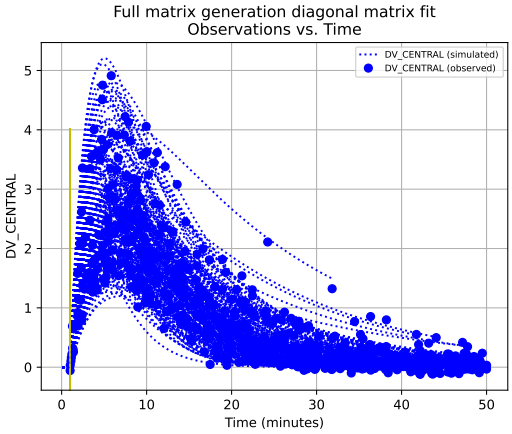


Fig. 2: Simulated prediction curves for a population with correlated inter-subject variability on parameters CL and V.

The generated predictions (Fig. 2) are qualitatively different from those generated without correlation because the resulting *KE* values have a smaller spread. (Correlated changes in *CL* and *V* cancel out to a degree.)

18.2.1 Uncorrelated Fit to Correlated Effects

Note: See the *Full matrix generation diagonal matrix fit* for the *Tut Script* used to generate results in this section.

We now run the fit using a model that prohibits correlation between *CL* and *V* by imposing a diagonal form on the covariance structure,

```
EFFECTS:
    POP: |
        f[KA] = 0.3
        f[CL] = 3
        f[V] = 20
        f[PNOISE_STD] = 0.1
        f[ANOISE_STD] = 0.05
        # new: Model in which variability of CL is
        # independent of that of V
        f[CL_isv, V_isv] ~ diag_matrix() [[0.01, 0.01]]
    ID: |
        r[CL_isv, V_isv] ~ mnorm( [0,0], f[CL_isv, V_isv] )
```

which is at odds with the data.

As a result, the fitted parameters

```
f[KA] = 0.3000
f[CL] = 3.0000
f[V] = 20.0000
f[PNOISE_STD] = 0.1000
f[ANOISE_STD] = 0.0500
f[CL_isv,V_isv] = [
    [ 0.1280, 0.0000 ],
    [ 0.0000, 0.1581 ],
]
```

are much smaller than the "true" values used to generate the observations because they are unable to capture the correlations between values. The final objective function value (OBJV) for the fit,

```
-2090.701479868882
```

cannot be compared with that of the data without correlation because they are different datasets, but serves as a useful baseline when fitting with different models.

18.2.2 Correlated Fit to Correlated Effects

Note: See the *Full matrix generation full matrix fit* for the *Tut Script* used to generate results in this section.

Finally, we run a fit with a model that does have the flexibility to capture correlations, again using the `~spd_matrix()` notation

```
EFFECTS:
    POP: |
        f[KA] = 0.3
        f[CL] = 3
        f[V] = 20
        f[PNOISE_STD] = 0.1
        f[ANOISE_STD] = 0.05
        # new: Model that links variability of
        # CL to that of V
        # f[CL_isv, V_isv] ~ spd_matrix() [
        #    [ 0.01 ],
        #    [ 0.00, 0.01 ]
        # ]
        f[CL_isv, V_isv] ~ spd_matrix() [
            [ 0.01 ],
            [ 0.001, 0.01 ]
        ]
    ID: |
        r[CL_isv, V_isv] ~ mnorm( [0,0], f[CL_isv, V_isv] )
```

This time, the fitted parameters

```
f[KA] = 0.3000
f[CL] = 3.0000
f[V] = 20.0000
f[PNOISE_STD] = 0.1000
f[ANOISE_STD] = 0.0500
f[CL_isv,V_isv] = [
    [ 0.1361, 0.0500 ],
    [ 0.0500, 0.1660 ],
]
```

are closer to the "true" values: the individual variances (along the diagonal) are higher and the covariance (the off-diagonal) is close to the generating value.

The improvement in fit is also apparent from the final objective function value (OBJV),

```
-2108.2792085086057
```

which is lower than for the diagonally-constrained fit, showing that the extra flexibility is beneficial when the data require it.

These examples illustrate the importance of including the flexibility to capture correlations in random effects where the model parameters vary together (for example, due to a common underlying property).

Where the observations do not come from a correlated source, having the flexibility to capture correlations has little real benefit to the model fit (a constrained covariance matrix would do just as well) but may require more data to estimate the greater number of parameters; the model should be as complex as it needs to be, but no more.

In practice, it is rare for PK parameters to be completely independent with a correlation of zero. Accounting for correlations tends to lead to better VPCs and more realistic simulations of future patients.

CHAPTER

NINETEEN

COVARIATES

Within a population, there will be variability in structural model parameters (such as clearance and volume of distribution) from one individual to another that can be predicted as a function of a measurable property (such as renal function impairment). These measurable properties are known as *covariates*.

If the variability can be expressed as a deterministic function of these underlying properties, we can model it to increase the accuracy of our predicted time course and decrease the burden on the residual error model.

19.1 Continuous Covariates

Continuous covariates appear on a sliding scale that can take on any value (often within a sensible range), as opposed to *Discrete Covariates* that take one of a finite (and often small) number of values.

We encode the effect of covariate `c[Y]` on model parameter `m[X]` via mathematical functions in the *MODEL_PARAMS* section of the *PoPy* script. For example, we may assume a linear relationship

```
m[X] = f[X] + f[X_Y_EFFECT]*c[Y]
```

though this can permit `m[X]` to be negative which is undesirable for many model parameters. Alternatives that avoid this problem include the exponential function

```
m[X] = f[X] * exp(f[X_Y_EFFECT]*c[Y])
```

or the power function

```
m[X] = f[X] * c[Y]**f[X_Y_EFFECT]
```

among others.

It may also be sensible to standardize the covariate in some way, for example by shifting

```
m[X] = f[X] * (c[Y]-Yref)**f[X_Y_EFFECT]
```

or scaling

```
m[X] = f[X] * (c[Y]/Yref)**f[X_Y_EFFECT]
```

the covariate with respect to some reference value, `Yref`. Choosing a value that is close to the middle of the range of the data (body weight, for example, is conventionally centred at 70 kg) reduces the

correlation between the intercept and the covariate slope, which leads to smaller standard errors and means that the intercept has a sensible interpretation (*e.g. CL* in an individual with GFR of 120 mL/min).

19.1.1 Body Weight as a Covariate

Note: See *Body Weight Covariate* for the *Tut Script* used to generate results in this section.

Empirical studies have shown that the volume of distribution (and clearance, among others) increases with body weight. In the supplied tutorial example, we have added a weight effect coefficient to the fixed effects in the **POP** level of *EFFECTS* and a weight covariate at the *ID* level:

```
EFFECTS:
    POP: |
        f[KA] = 0.3
        f[CL] = 3
        f[V] = 20
        f[PNOISE] = 0.1
        f[ANOISE] = 0.05
        f[WT_EFFECT] = 0.75  # new fixed effect
    ID: |
        c[ID] = sequential(30)
        c[AMT] = 100.0
        t[RESET] = 0.0
        t[DOSE] = 1.0
        t[OBS] ~ unif(1.0, 50.0; 4)
        c[WT] ~ norm(70, 100)  # new covariate
```

We then update the *MODEL_PARAMS* section to include the covariate effect as per the "allometric function" [Holford1996]

```
MODEL_PARAMS: |
    m[KA] = f[KA]
    m[CL] = f[CL]*(c[WT]/70)**f[WT_EFFECT]  # new covariate effect
    m[V] = f[V]*(c[WT]/70)  # new covariate effect
    m[PNOISE] = f[PNOISE]
    m[ANOISE] = f[ANOISE]
```

which is a form of power function, using a scaled weight with 70 kg as the reference (as is now accepted in the literature).

This generates a population with time courses that are weight dependent (Fig. 1).

For fitting, we estimate only the baseline (median) volume of distribution, `f[V]`, and the weight effect coefficient, `f[WT_EFFECT]`, and we see that the estimated values,

```
f[KA] = 0.3000
f[CL] = 3.0000
f[V] = 20.2610
f[PNOISE] = 0.1000
```

(continues on next page)

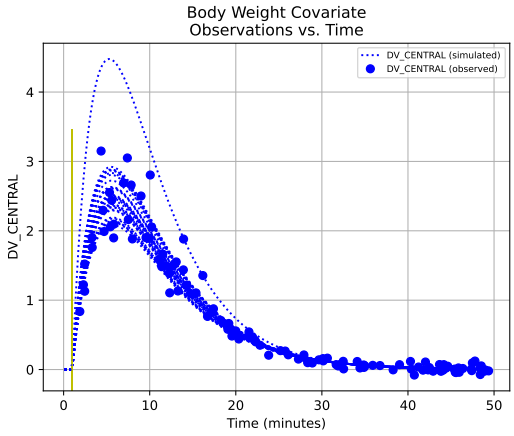


Fig. 1: Simulated prediction curves for a population of 40 individuals with weight-dependent clearance, CL, and volume of distribution, V

(continued from previous page)

```
f[ANOISE] = 0.0500
f[WT_EFFECT] = 0.6657
```

closely match those used to generate the observations.

19.1.2 Other Continous Covariates

Other continuous covariates include

- Age: Renal and hepatic functions decrease with age, which can lead to a reduced drug clearance. The volume of distribution of some lipid soluble drugs also increases with age.
- Ideal Body Weight (IBW):
- Lean Body Weight (LBW):

$$\text{Males} = 50 + 0.9\text{kg for every cm over 150cm}$$
$$\text{Females} = 45 + 0.9\text{kg for every cm over 150cm}$$

- Height: The height of an individual, usually measured in cm or m.
- Body Surface Area (BSA): The surface area of the body in m^2 can be calculated using the weight in Kg and height in cm. There are various different methods for achieving this such as the Du Bois formula:

$$\text{BSA} = 0.007184 * \text{WT}^{0.425} * \text{H}^{0.725}$$

or Mosteller formula:

$$\text{BSA} = \frac{\sqrt{\text{WT} * \text{H}}}{60}$$

- Body Mass Index(BMI): The mass or weight of a subject in kg divided by the square of the height in m, gives the BMI measured in kg/m^2

$$\text{BMI} = \frac{\text{WT}}{\text{H}^2}$$

19.2 Discrete Covariates

Discrete covariates take one of a finite (and often small) number of values, as opposed to continuous covariates that lie on a spectrum. Discrete covariates can then be used to model differences between groups by allowing a different median value in each group.

Where the covariate represents an either-or classification (*i.e.* the covariate is dichotomous), we can encode group membership with a zero (reference group) or one (other group). We can then use this indicator variable in a mathematical formula,

```
m[X] = (1-c[Y])*f[X_WHEN_Y_IS_ZERO] + c[Y]*f[X_WHEN_Y_IS_ONE]
```

For ease of interpretation, *PoPy* allows you to encode data columns as strings so that you can see immediately which group is being referred to in the input file. In these cases, an `if-else` statement can be used to apply the correct fixed effect to the model parameter. This method, unlike indicator variables, can also be used for discrete variables that take more than two values (*e.g.* race).

```
if c[Y] == 'zero':
    m[X] = f[X_WHEN_Y_IS_ZERO]
elif c[Y] == 'one':
    m[X] = f[X_WHEN_Y_IS_ONE]
elif c[Y] == 'two':
    m[X] = f[X_WHEN_Y_IS_TWO]
...
```

We refer to discrete covariates that assign to a group as *categorical covariates*.

In contrast, *ordinal covariates* are a discretization of the continuous spectrum and have a definite ordering. These values can sometimes be treated as if they are continuous (*e.g.* the Child-Pugh scale for hepatic impairment) but it is up to the modeller to decide whether this is justifiable.

19.2.1 Other Discrete Covariates

Other discrete covariates include:

- Sex

  Females may have different volumes of distribution or clearances (or both) than men for some drugs, even after the difference in body weight is accounted for.

- Race

  In some cases, race categorizations may be associated with different pharmacokinetic parameters. For example, the frequency of genetic polymorphisms affecting clearance can vary with race.

- Concomitant Medication

  When two or more drugs are administrated concurrently, they may interfere with each others pharmacokinetic profiles by competing for metabolizing enzymes or transporters. Drugs can also alter the transcription of enzymes, transporters or proteins within drug-binding sites, which can alter both their own pharmacokinetics and the pharmacokinetics of co-administered drugs.

  For example, Drug A, or its metabolites, could induce an enzyme for a conjugation reaction in the metabolisation of Drug B. In this case, Drug A could increase the clearance of Drug B.

  Alternatively, Drug A, or its metabolites could be an inhibitor on one of the enzymes facilitating the metabolism of Drug B. In this case, Drug A could decrease the clearance of Drug B.

  The interactions of different drugs and their metabolites can be very complex and are an important area for population pharmacokinetic modelling.

- Smoking

  Smoking cigarettes increases the clearance of some drugs by increasing the activity of enzymes used in hepatic metabolism. Caffeine is an example of a drug that has increased clearance in smokers. Note that it is not usually the nicotine in cigarettes that affects metabolism, but other chemicals in the cigarette smoke. Nicotine replacement therapy may not have the same effect.

- Disease state

  Many diseases or conditions can alter pharmacokinetics. An example is oedema, which causes an excess of fluid in cavities or tissues in the body. If a drug is soluble in this fluid, it could increase the volume of distribution.

- Food intake

  If a drug is administered orally, the rate of absorption can be affected by whether an individual has eaten recently or not. This is because most drugs are absorbed through the small intestine, but to reach this they have to pass through the stomach. When food is present in the stomach, gastric secretion and residence time are increased. This could lead to a slower absorption rate.

Part V

Estimating Population Parameters

Assume we have data for a population of individuals, where for each individual we are given:

- measured covariates such as age or body weight
- the parameters of the study they were involved in such as the amount of drug administered
- a time series of one or more observed variables such as drug or biomarker concentration

For a given mathematical model of the PK/PD for any individual drawn from the population, we now want to estimate the population averages (fixed effects) and deviations from those averages (random effects) giving rise to model parameters that best predict the observed time series.

Being able to estimate parameters enables us to:

- compare different mathematical models based on their ability to predict observed data
- sample new populations with similar properties and analyse the predicted PK/PD to predict the effect of a drug on a new, random population

In this part, we explore concepts related to estimating parameters of the mathematical model ("fitting the model") using *PoPy* examples to generate data and figures to aid our explanations.

CHAPTER

TWENTY

ESTIMATING MODEL PARAMETERS

Note: See *One Compartment Model with Absorption estimating V and CL* for the *Tut Script* used here.

The previous chapters show how the observed concentrations are influenced by the parameters that define them such as the number of compartments and their function, how flows between compartments are specified, the shape of the dosing function, and the residual error model. We have explored these topics by taking models with given parameters, and using them to generate datasets of synthetic observations. This is often known as a *forward* problem.

We now turn to the more difficult *inverse* problem where we are given a dataset of measurements and a parameterized model, and want to estimate the most likely parameters for the given data. To solve this problem, we first need to make three design decisions.

First, we need a way to quantify the goodness-of-fit for a set of parameters, *i.e.* a single number that enables us to compare one set of parameters with another and determine which agrees better with the data. A popular choice for this is the *likelihood* that quantifies how likely the data are, given the model parameters.

Second, we must specify an algorithm for finding the "best" model parameters from the set of possible options. In practice, finding *the* best set of parameters is difficult or impossible, so we instead look for a *locally* optimal set that is better than all other nearby sets [DennisSchnabel1987] [NocedalWright2006].

Finally, we need a set of criteria that determine whether we have reached the best local solution, *i.e.* whether the algorithm has converged.

20.1 Likelihood and the Objective Function

Before we can find the "best" fit of a model to a set of observations, we first need a measure of what is "good". A common measure of goodness-of-fit is a model's likelihood [Millar2011]: how likely are the observations to arise if the given set of model parameters is correct? Although this is *not* a probability (the integral over model parameters does not equal one), for a given set of observations and a given model structure a higher likelihood indicates a better fit.

In practice, we compute the Maximum Likelihood by minimizing the *negative* of the likelihood (because most optimization algorithms are written to find the minimum of an error or cost function). Moreover, where the likelihood is a member of the exponential family of distributions it is mathematically convenient to minimize *-2 log(likelihood)* which is commonly referred to as the *objective function*, *ObjV*.

Using this measure of cost, we can take a "bird's eye view" of the surface in two dimensions by plotting it as a contour map for pairs of parameter values. In *One Compartment Model with Absorption*, for

example, we looked at a model with three parameters: *KA*, *CL* and *V*. Fixing *KA* at its optimal value, we can see how the cost surface varies with respect to *CL* and *V* (Fig. 1).

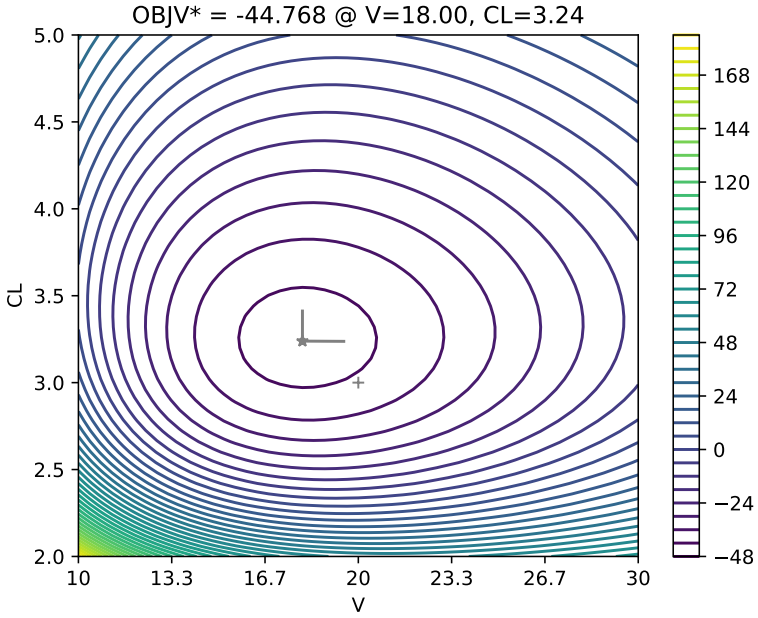


Fig. 1: *CL* vs *V* surface plot of the Objective Function Value (*ObjV*) for a bolus dose, administered to a one compartment model with absorption

In this example, we see that the cost function has a single minimum in this region and is at its lowest close to the true values: $CL = 3$ and $V = 20$. The lowest cost parameters are close to, not exactly at, the true values because of noise added to the observations and a finite data set.

20.2 Minimization Algorithm

Once we have specified the cost function we want to minimize, we need an algorithm that will find the minimum. Because some models have large datasets and complex models with many parameters, it is not practical to find the minimum via an exhaustive search over every possible combination of parameters.

We therefore adopt an incremental approach that takes an initial guess at the parameter values, then looks for a different set of values nearby that has a lower cost. Repeating this process takes us on a path across the likelihood surface until there are no nearby solutions that are better than the current one. This, by definition, is the *local* minimum.

Depending on the shape of the cost surface and the particular algorithm used, finding the local minimum can take only a few steps or it can require many steps over a winding path. In the example given (Fig. 2), a local minimum was found by *PoPy* in a single step. *PoPy*'s search algorithm is very good at finding nearby minima with very few steps, however it is impossible to guarantee that this is the global minimum. This is true for all local search algorithms.

The current recommended search algorithm used by *PoPy* is called *ND*, see *ND Fitting Method* for more details.

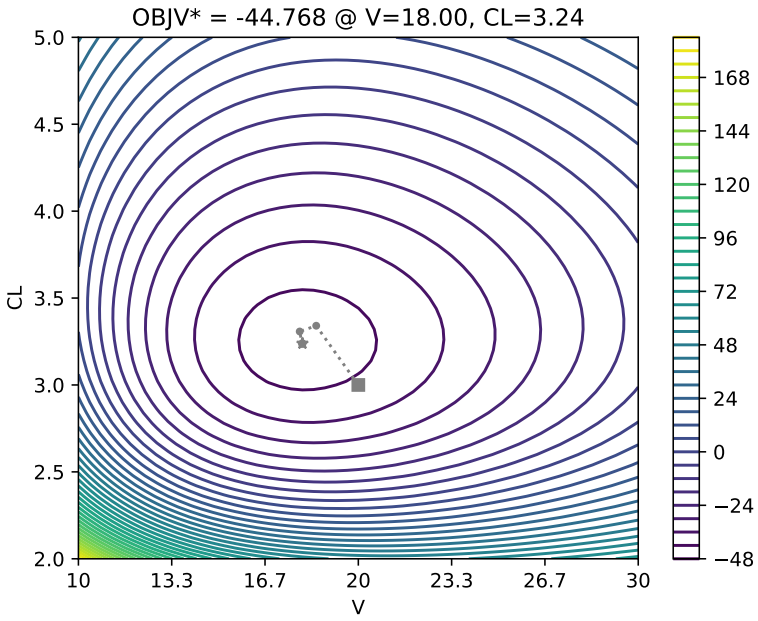


Fig. 2: CL vs V surface plot overlaid with the path taken by the optimization algorithm to the minimum. '+' shows starting values, square shows minimum values.

20.3 Initial Values and Convergence

When using a local optimization algorithm, the current estimate heads "down the hill" to a nearby point where the cost is lower until there is nowhere lower to go. In some cases, there may be more than one point that is *locally* the lowest point. As a result, starting the algorithm from some points will lead to one local minimum whereas starting from other initial points will lead to other local minima.

The set of points that lead to a given local minimum are known as its *basin of convergence*. In simple models, this is rarely a major problem. However, in complex models or if the fitted parameters are infeasible it is worthwhile to test alternative starting values, especially if the initial guess at a starting estimate was a long way from the fitted value.

CHAPTER

TWENTYONE

UNCERTAINTY AND STANDARD ERRORS

Parameter estimation (also referred to as "model fitting") aims to find a solution in parameter space that has a likelihood at least as good as all other nearby solutions. We know that a locally optimal solution has been found when changing any parameter in any direction decreases the likelihood (or, equivalently, increases the cost).

21.1 Likelihood Hessian

We can also quantify the confidence in our estimates from the shape of the likelihood surface at the local minimum: changing some parameters will lower the likelihood by a lot, suggesting that we have high confidence (or low uncertainty) in their estimated value; changing others, however, will lower the likelihood by only a little or possibly not at all, suggesting that we have low confidence (high uncertainty) in our estimate.

Note: See *One Compartment Model with Absorption estimating KA* for the *Tut Script* used here.

First we return to the example used in the last chapter – a one compartment model with absorption – only here we reduce it to a one dimensional problem by fixing *CL* and *V* to the optimal values and allowing only *KA* to vary. We can then plot *ObjV* for a range of values of *KA* as a line (Fig. 1).

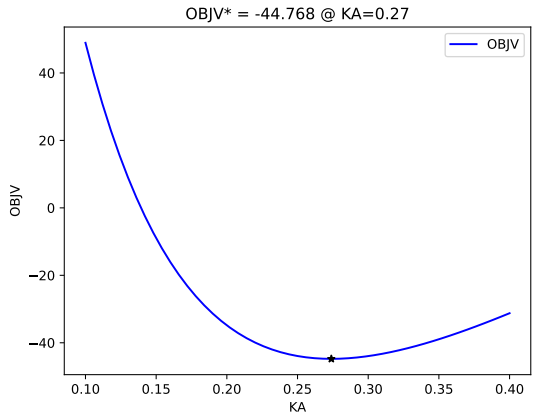


Fig. 1: *ObjV* vs *KA* with other parameters fixed at their true values

Although *ObjV* as a function of *KA* is not symmetric, it is smooth and has a clearly defined minimum. We can therefore compute properties of the curve (*e.g.* its gradient) at any point and approximate it with a polynomial using a Taylor series expansion.

More specifically, using a second order Taylor series at the minimum requires only the minimum value, *ObjV*, and its second derivative because the gradient is zero by definition. This approximates the *ObjV* function with a quadratic (Fig. 2).

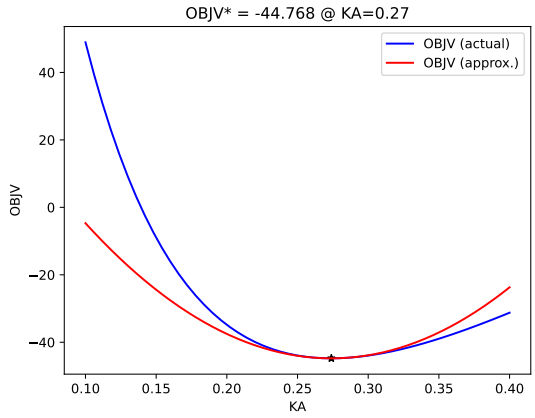


Fig. 2: *ObjV* vs *KA* with other parameters fixed at their true values, plus the quadratic approximation at the minimum

In effect, this approximates the likelihood surface with a Gaussian whose peak coincides with that of the true likelihood (Fig. 3).

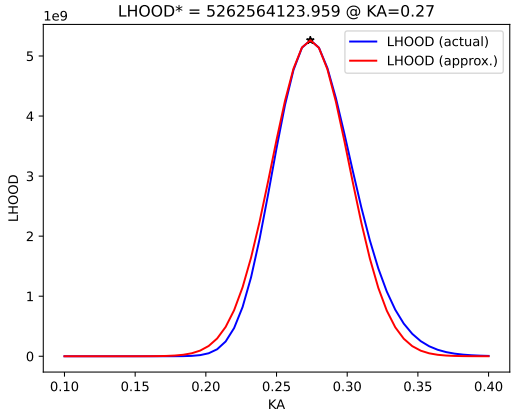


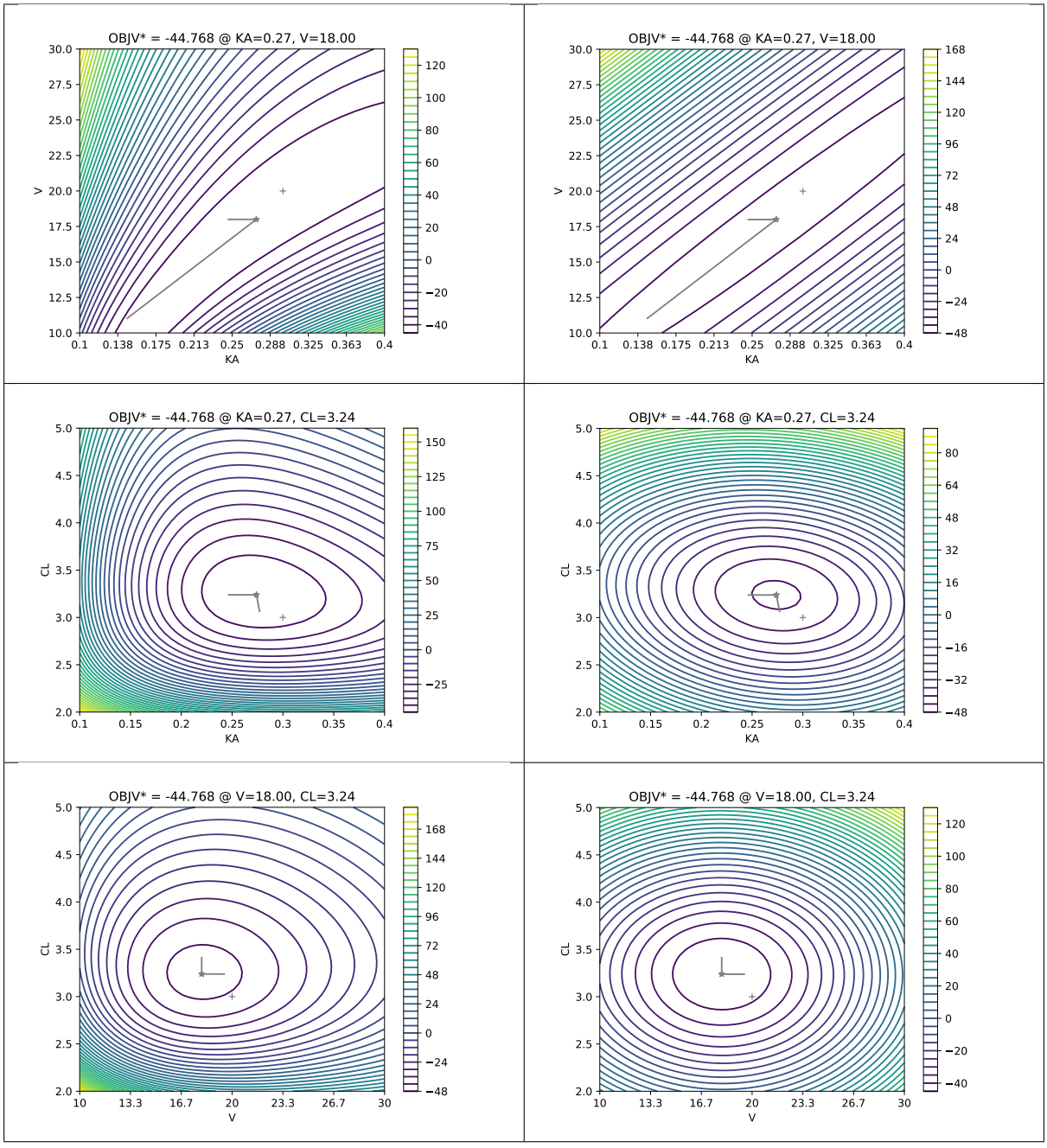
Fig. 3: Likelihood vs *KA* with other parameters fixed at their true values, plus the gaussian approximation at the maximum

---

Note: See *One Compartment Model with Absorption estimating KA and V*, *One Compartment Model with Absorption estimating KA and CL* and *One Compartment Model with Absorption estimating V and CL*, for the *tutorial scripts* used here.

In more than one dimension, the second derivative is a matrix and is known as the *hessian*. In two dimensions, for example, the quadratic approximation to *ObjV* resembles a basin whose shape reflects the true function at its minimum (Table 1).

Table 1: Comparison of ObjV surface (left) with quadratic approximation (right)



21.2 Confidence Intervals

Note: See *One Compartment Model with Absorption estimating KA and CL* for the *Tut Script* used here.

With a gaussian approximation to the likelihood, we can sample new values for the population parameters from the distribution. By sampling many possible solutions from the distribution, we can compute a range for each parameter in which a given percentage of the sampled values lie. These ranges are

called *confidence intervals* and typically encompass 90% or 95% of the sampled values (Fig. 4).

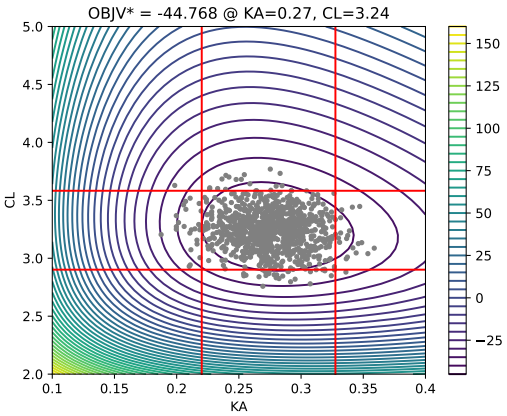


Fig. 4: *ObjV* surface with population parameters sampled from the approximating gaussian and 90% confidence intervals in red for *KA* and *CL*

```
CI(KA) = 0.274 + [-0.0538, 0.0537]
CI(CL) = 3.24 + [-0.337, 0.344]
```

The purpose of the confidence interval is to compare one estimate with another, for example by confirming that our estimated confidence interval contains a value published in research literature by a third party.

When computing confidence intervals, we can get different estimates depending on how we parameterized the model. When population parameters are required to be positive, for example, we might optimise over values for their logarithm (which spans the whole of the real line). This can change the *ObjV* surface to be closer to a quadratic, and effectively samples from a lognormal distribution over the population parameters (Fig. 5).

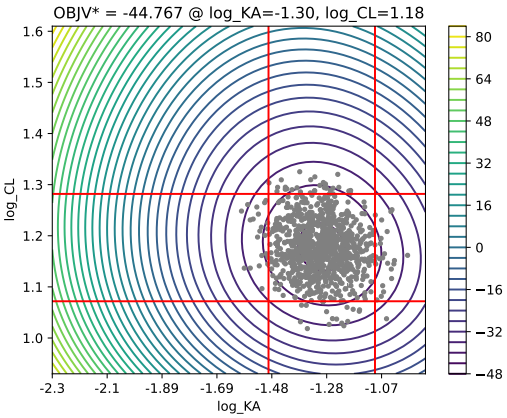


Fig. 5: *ObjV* surface with population parameters sampled from the approximating gaussian and 90% confidence intervals in red for log(*KA*) and log(*CL*)

One outcome of this is that (in the limit) the confidence interval is less likely to be centred on the estimated value:

```
CI(KA) = 0.273 + [-0.049, 0.0602]
CI(CL) = 3.25 + [-0.325, 0.358]
```

21.3 Standard Errors

The *Likelihood Hessian* can be used to compute standard errors for parameter estimates, which are similar to the *Confidence Intervals* described above and computed in a similar manner.

CHAPTER

TWENTYTWO

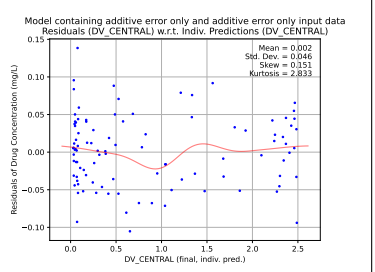
DIAGNOSTICS

After fitting a model to data, it is important to quantify the goodness-of-fit (GoF) to the data in some way. Summary statistics, often known as diagnostics, are useful in comparing models and visualising goodness-of-fit to data in a way that can steer the modeller toward better models.

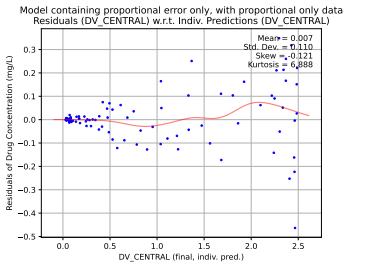
22.1 Residuals

Residuals - the difference between a predicted quantity, such as concentration, and its observed value - indicate the error (or noise) model on the observations, for which there are three popular choices.

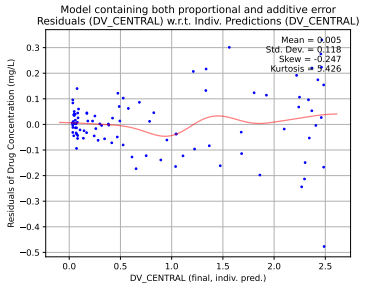
Table 1: Residuals vs IPRED for the additive (left), proportional (centre) and additive+proportional (right) noise models



Additive



Proportional



Additive+Proportional

In the Additive Noise model, noise has a fixed standard deviation (or variance) that is independent of the predicted value.

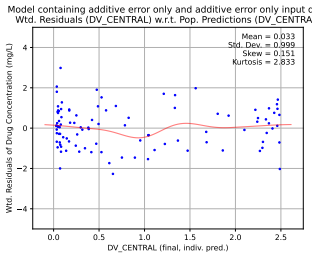
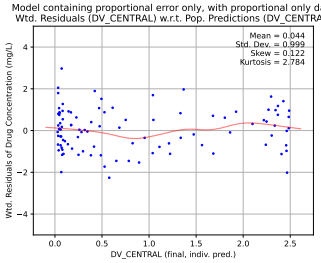
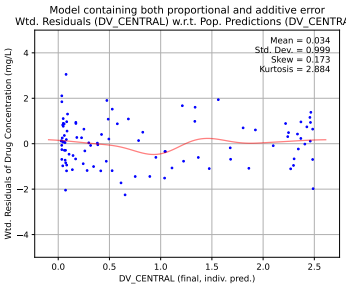
In the Proportional Noise model, the level (variance) of noise increases with the magnitude of the predicted variable.

In the Additive+Proportional Noise model, there is an element of both: additive noise that ensures valid likelihoods when the predicted variable is zero; and a proportional component that better reflects a prediction-dependent noise level.

22.2 Weighted Residuals (WRES)

To achieve a more intuitive and easy-to interpret diagnostic, we can scale residuals by the variance we expect given the predicted value.

Table 2: Weighted Residuals vs PRED for the additive (left), proportional (centre) and additive+proportional noise models

		
Additive	Proportional	Additive+Proportional

Residuals should have a mean of zero and a variance of one when normalized in this way, and if they do not follow the standard normal distribution this could be an indication that the model is misspecified. (Higher order moments of the distribution such as skew and kurtosis should also be close to their standardized values of zero and three, respectively.)

22.3 Conditional Weighted Residuals (CWRES)

It was noted by Hooker et al. [HookerEtAl2007], however, that WRES can sometimes imply that a misspecified model is better than a correctly specified one. This problem arises because (in Nonmem) the WRES are computed from the FO statistics even when FOCE was used to fit the model to the data.

Here we reproduce Hooker's example model in which the response of the disease to the drug is limited through an Emax equation, generating data for a population of 200 individuals with fixed observation time points and inter-subject variability in Emax and C50:

```
GEN_EFFECTS:
    POP: |
        f[EMAX] = 100
        f[C50] = 20
        f[Hill] = 4.5
    ID: |
        c[ID] = sequential(200)
        t[OBS] = [ 0.001, 1.
↪0, 2.0, 4.0, 8.0, 16.0, 32.0, 64, 96, 120, 150, 180, 240, 320, 420, 525 ]
        f[EMAX_isv] = 0.5
        r[EMAX_isv] ~ norm(0, f[EMAX_isv])
        f[C50_isv] = 0.5
        r[C50_isv] ~ norm(0, f[C50_isv])
```

In order to compare the input concentration with the Emax response for fixed drug concentrations, we simply treat the observation times as if they were drug concentrations in the PREDICTIONS block

and assign the corresponding Emax response to the output column, c[Effect]:

```
PREDICTIONS: |
    p[Drug] = c[TIME]
    p[Effect]␣
↪= m[EMAX] * p[Drug]**m[Hill]/(p[Drug]**m[Hill] + m[C50]**m[Hill])
    effect_var = m[ANOISE_var]
    c[Effect] ~ norm(p[Effect], effect_var)
```

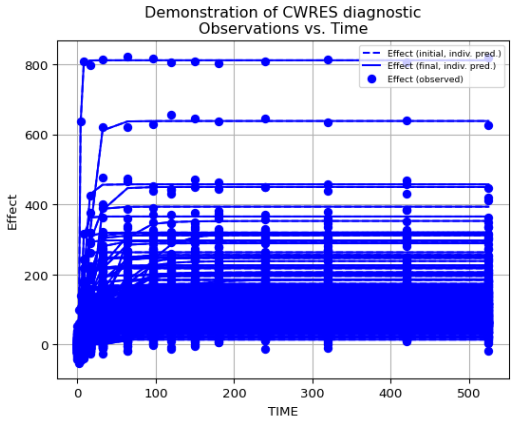
We then apply a *PoPy* fit for two models: one with a correctly specified Hill coefficient of 4.5 (*sum_link_cwres_emax_correct_model_tut*),

```
FIT_EFFECTS:
    POP: |
        f[EMAX] ~ P 100
        f[C50] ~ P 20
        f[Hill] ~ P 4.5  # Correct Hill coefficient
    ID: |
        f[EMAX_isv] ~ P 0.5
        r[EMAX_isv] ~ norm(0, f[EMAX_isv])
        f[C50_isv] ~ P 0.5
        r[C50_isv] ~ norm(0, f[C50_isv])
```

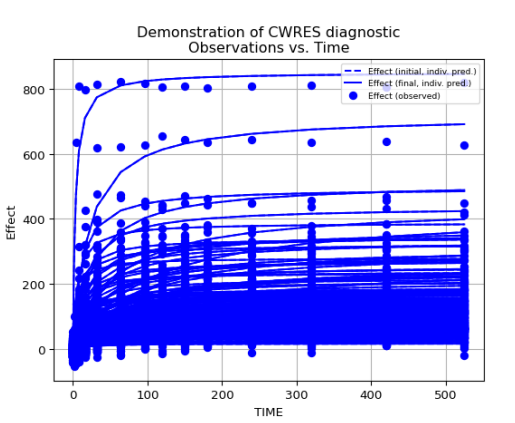
and one with a misspecified Hill coefficient of 1.0 (*sum_link_cwres_emax_misspecified_model_tut*).

```
FIT_EFFECTS:
    POP: |
        f[EMAX] ~ P 100
        f[C50] ~ P 20
        f[Hill] = 1.0  # Incorrect Hill coefficient
    ID: |
        f[EMAX_isv] ~ P 0.5
        r[EMAX_isv] ~ norm(0, f[EMAX_isv])
        f[C50_isv] ~ P 0.5
        r[C50_isv] ~ norm(0, f[C50_isv])
```

Table 3: Fitted predictions for the correctly specified (left) and misspecified (right) models



Correctly specified model

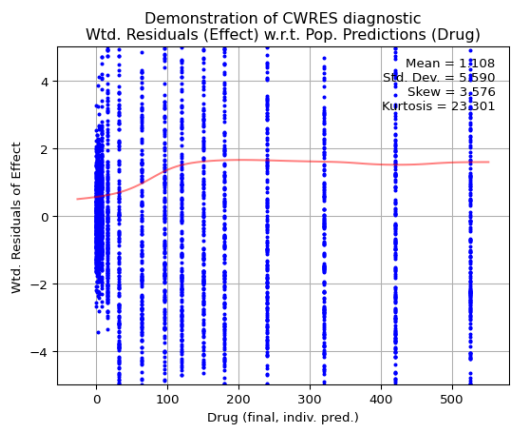


Misspecified model

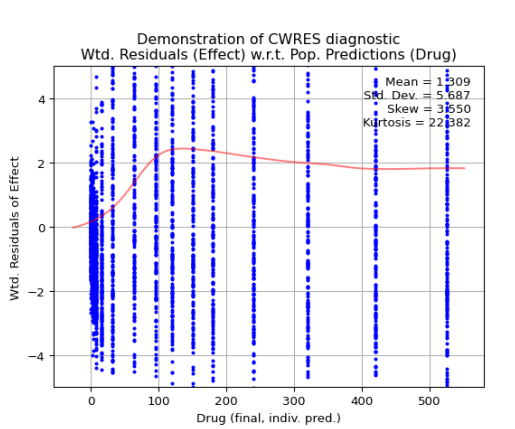
The figure above shows the predictions fitted to the data, where it is clear that the correctly specified model has achieved a better fit to the data.

As noted by Hooker, however, the popular WRES diagnostic for the misspecified model has a distribution whose parameters are closer to the standard normal (mean and skew of zero, standard deviation of one, kurtosis of three) compared with the correctly specified model after fitting with FOCE:

Table 4: WRES vs. PRED for the correctly specified (left) and misspecified (right) models



Correctly specified model

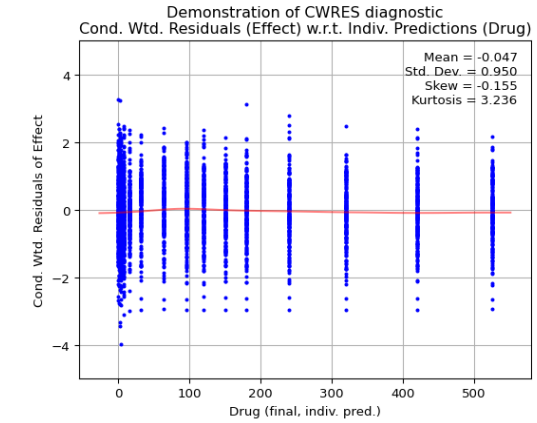


Misspecified model

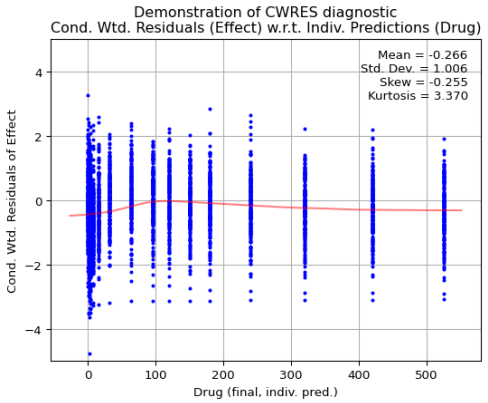
To address this anomaly, Hooker suggested a new diagnostic - the Conditional Weighted Residual (CWRES) - that translated and scaled residuals with values more suited to an FOCE-fitted model.

When using this diagnostic rather than WRES, the moments of the error distribution suggest that the correctly specified model is a better fit to the data than the misspecified mode (with the exception of Standard Deviation which is close to one in both cases):

Table 5: CWRES vs IPRED for the correctly specified (left) and misspecified (right) models



Correctly specified model



Misspecified model

We therefore suggest using CWRES rather than WRES for diagnostics by requesting this plot ('{CWRES}_vs_IPRED') in the OUTPUT_SCRIPTS block of the model file:

```
OUTPUT_SCRIPTS:
    GEN: {output_mode: run, grph_list: []}
    FIT:
        output_mode: run
        grph_list: ['{OBS}_vs_TIME', '{WRES}_vs_PRED', '{CWRES}_vs_IPRED']
    COMP: {output_mode: run}
    TUTSUM: {output_mode: run}
```

CHAPTER

TWENTYTHREE

GENERATE BLQ OBSERVATIONS AND FIT DIFFERENT ERROR MODELS

In this example we will demonstrate generating and fitting to **BLQ** data using the *~rectnorm()* distribution with a simple depot + one compartment model as follows:-

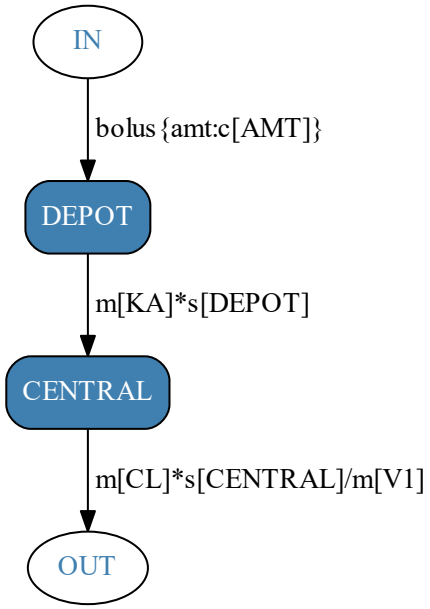


Fig. 1: One compartment model with depot dosing used to generate and fit **BLQ** PK data.

Note: See the *Depot + One compartment PK with BLQ* obtained by the *PoPy* developers for this example, including input script and output data file.

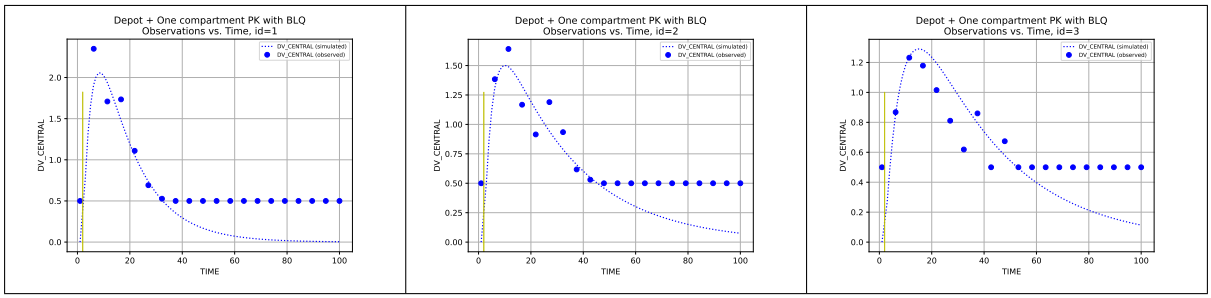
23.1 Generating BLQ observations

The *PREDICTIONS* section to create **BLQ** data utilises the *~rectnorm()* distribution, as follows:-

```
PREDICTIONS: |
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[DV_CENTRAL]**2
    # c[DV_CENTRAL] ~ norm(p[DV_CENTRAL], var)
    c[DV_CENTRAL] ~ rectnorm(p[DV_CENTRAL], var, LLQ=0.5)
```

This creates observations with a **LLQ** of 0.5, see Table 1.

Table 1: Generated observations + true underlying model PK curves for first three individuals



Notice that no observations are output below 0.5, as expected. PoPy outputs observations of 0.5, if the generated observation is in the interval [-inf, 0.5].

23.2 Fitting BLQ observations using rectnorm (correct error model)

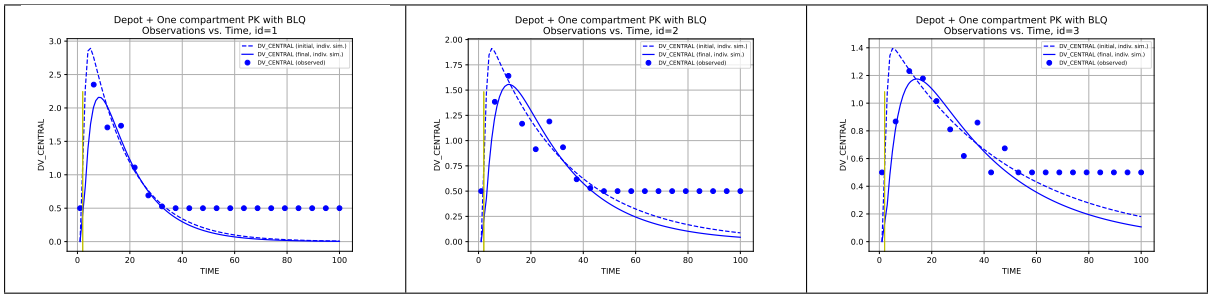
We fit the same (correct) error model, using the same *PREDICTIONS* section as follows:-

```
PREDICTIONS: |
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[DV_CENTRAL]**2
    # c[DV_CENTRAL] ~ norm(p[DV_CENTRAL], var)
    c[DV_CENTRAL] ~ rectnorm(p[DV_CENTRAL], var, LLQ=0.5)
```

i.e. a *~rectnorm()* distribution, with a **LLQ** of 0.5. Note that, this fitting method does **not** require the *LAPLACE* objective function, we can use the *FOCE* objective function instead in *PoPy*.

See Table 2 for PK curves after fitting.

Table 2: Fitted model PK curves vs true model PK curves for first three individuals



Note that the fitted curves (solid blue) line are estimated to be below the **LLQ** level of 0.5 at later time points.

The estimated `f[X]` compared to the true `f[X]` are shown in Table 3 and Table 4.

Table 3: Comparison of initial, fitted and true `f[X]` main values

Name	Initial	Fitted	True	Abs. Error	Prop. Error
f[KA]	1	0.199	0.2	9.47e-04	0.47%
f[CL]	1	1.95	2	5.25e-02	2.62%
f[V1]	20	48.7	50	1.28e+00	2.56%

Table 4: Comparison of initial, fitted and true `f[X]` noise values

Name	Initial	Fitted	True	Abs. Error	Prop. Error
f[PNOISE]	0.1	0.149	0.15	1.12e-03	0.75%

Table 3 and Table 4 show that the `f[KA]`, `f[CL]`, `f[V1]`, `f[PNOISE]` parameters are recovered well when fitting with *~rectnorm()* distribution.

The full set of fitted `f[X]` variable is shown below:-

```
f[KA] = 0.1991
f[CL] = 1.9475
f[V1] = 48.7184
f[KA_isv,CL_isv,V1_isv] = [
    [ 0.0681, 0.0330, -0.0008 ],
    [ 0.0330, 0.0385, 0.0310 ],
    [ -0.0008, 0.0310, 0.1029 ],
]
f[PNOISE] = 0.1489
f[ANOISE] = 0.0100
```

These fitted values can be compared with fitting alternative error models below.

23.3 Fitting BLQ observations using norm (incorrect error model)

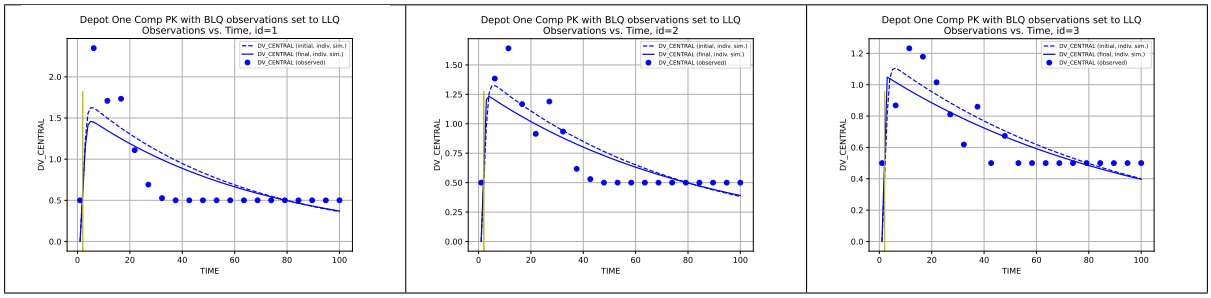
Note: See the *Depot One Comp PK with BLQ observations set to LLQ* script and results used by the *PoPy* developers for this example.

We fit a (incorrect) error model which treats observations of **LLQ** as actual 0.5 observations, using a *~norm()* distribution, as follows:-

```
PREDICTIONS: |
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[DV_CENTRAL]**2
    c[DV_CENTRAL] ~ norm(p[DV_CENTRAL], var)
    # c[DV_CENTRAL] ~ rectnorm(p[DV_CENTRAL], var, LLQ=0.5)
```

For fitted curves see Table 5.

Table 5: Fitted model ~norm() distribution error curves vs **LLQ** model PK curves for first three individuals



Note that the fitted curves (solid blue) line try to stay close to the **BLQ** values at 0.5, which is incorrect, as the PK curve should approach zero concentration instead.

The fitted `f[X]` values are:-

```
f[KA] = 4.6504
f[CL] = 0.9495
f[V1] = 85.1296
f[KA_isv,CL_isv,V1_isv] = [
    [ 1.3306, -0.0040, 0.2688 ],
    [ -0.0040, 0.0000, -0.0008 ],
    [ 0.2688, -0.0008, 0.0586 ],
]
f[PNOISE] = 0.2299
f[ANOISE] = 0.0100
```

These fitted values are inaccurate compared to the parameters recovered using the *~rectnorm()* distribution in section *Fitting BLQ observations using rectnorm (correct error model)* above.

23.4 Fitting BLQ observations using half LLQ (approx error model)

Note: See the *Depot One Comp PK with BLQ observations set to 0.5*LLQ* script and results used by the *PoPy* developers for this example.

We fit a simple approx **BLQ** error model, which models observations of **LLQ** as bare 0.5* **LLQ** observations, by preprocessing the original observation data, as follows:-

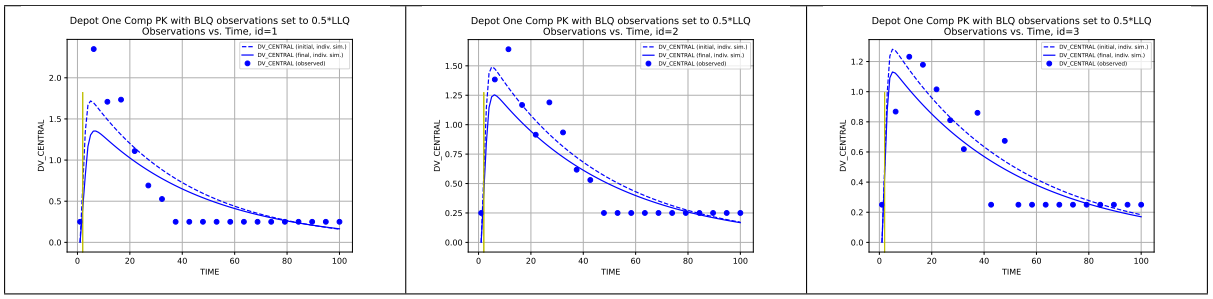
```
PREPROCESS: |
    # use halve value blq data
    if c[DV_CENTRAL] <= 0.5 and c[TYPE] == 'obs':
        c[DV_CENTRAL] = 0.25
```

The *~norm()* distribution is used to model these amended observations (similar to *Fitting BLQ observations using norm (incorrect error model)* above), see:-

```
PREDICTIONS: |
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[DV_CENTRAL]**2
    c[DV_CENTRAL] ~ norm(p[DV_CENTRAL], var)
    # c[DV_CENTRAL] ~ rectnorm(p[DV_CENTRAL], var, LLQ=0.5)
```

For fitted curves using this half **LLQ** approximation, see Table 6.

Table 6: Fitted model ~norm() distribution error curves vs half **LLQ** model PK curves for first three individuals



Note that the fitted curves (solid blue) line get closer to zero (the true asymptotic concentration) as time increases, but are still heavily distorted by assuming the half **LLQ** values are true observations.

The fitted `f[X]` values are:-

```
f[KA] = 1.1546
f[CL] = 1.6274
f[V1] = 77.9502
f[KA_isv,CL_isv,V1_isv] = [
    [ 0.3676, 0.0643, 0.1502 ],
    [ 0.0643, 0.0115, 0.0269 ],
    [ 0.1502, 0.0269, 0.0628 ],
```

(continues on next page)

(continued from previous page)

```
]
f[PNOISE] = 0.3270
f[ANOISE] = 0.0100
```

These fitted values are similar to the results obtained in *Fitting BLQ observations using norm (incorrect error model)* and also inaccurate compared to the parameters recovered using the *~rectnorm()* distribution in section *Fitting BLQ observations using rectnorm (correct error model)* above.

23.5 Fit to non-BLQ observations only (reduced data model)

Note: See the *Depot One Comp PK ignoring BLQ observations.* script and results used by the *PoPy* developers for this example.

We can also just ignore the **BLQ** data, by using *PREPROCESS* to remove the **LLQ** observations, as follows:-

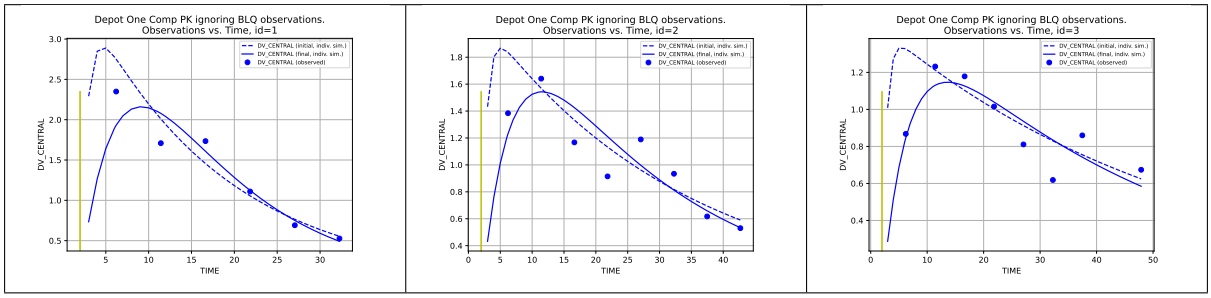
```
PREPROCESS: |
    # ignore blq data
    if c[DV_CENTRAL] <= 0.5 and c[TYPE] == 'obs':
        return
```

The *~norm()* distribution is then used to model the above **LLQ** observations only, see:-

```
PREDICTIONS: |
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[DV_CENTRAL]**2
    c[DV_CENTRAL] ~ norm(p[DV_CENTRAL], var)
    # c[DV_CENTRAL] ~ rectnorm(p[DV_CENTRAL], var, LLQ=0.5)
```

The fitted curves are shown in Table 7.

Table 7: Fitted model ~norm() distribution error curves vs ignore **BLQ** model PK curves for first three individuals



Note that curves are fitted to the above **LLQ** observations only, *i.e.* earlier time points, but there is no data for later **LLQ** time points.

The fitted `f[X]` values are:-

```
f[KA] = 0.2157
f[CL] = 1.8093
f[V1] = 51.2961
f[KA_isv,CL_isv,V1_isv] = [
    [ 0.0000, 0.0001, 0.0003 ],
    [ 0.0001, 0.0516, -0.0033 ],
    [ 0.0003, -0.0033, 0.1236 ],
]
f[PNOISE] = 0.1482
f[ANOISE] = 0.0100
```

These fitted values are much better then the *Fitting BLQ observations using norm (incorrect error model)* and *Fitting BLQ observations using half LLQ (approx error model)* approximatons, but not quite as accurate as the *~rectnorm()* distribution results in section *Fitting BLQ observations using rectnorm (correct error model)*.

Part VI

Working With Many Populations

Given a mathematical model of PK/PD for a drug or biomarker (or both), it can be useful to know how well the model can be fitted (*e.g.* whether parameters are *identifiable*).

Because we should rarely base conclusions on a single datum point, *PoPy* offers the functionality to repeat a fit to *many* synthetically generated populations and aggregate the results in order to assess how well a model or fitting algorithm perform.

This part therefore presents an example in which we generate and fit a model to synthetic data and examine the distribution of fitted results.

CHAPTER

TWENTYFOUR

GENERATE MULTIPLE DATA SETS AND FIT USING SIMPLE POPPK MODEL

The tutorial example above generates a single data set from user specified true `f[X]` values. In *PoPy* it is possible to generalise this approach and sample true `f[X]` values multiple times to create multiple data sets. Then fit the same model to each data set.

In this section we walk through how to generate multiple data sets using a *MTut Script*, using the same simple one compartment model as shown in `fig_simple_popkp_diagram_tut`.

24.1 Running the MTut Script

This multi tutorial example makes use of a single script file, `c:\PoPy\examples\mtut_example1.pyml`

PoPy Terminal to setup the *PoPy* environment in the folder `c:\PoPy\examples\`

With the *PoPy* environment enabled, you can call *popy run* on the *MTut Script* from the command line:-

```
popy run mtut_example1.pyml
```

to run the script.

Running a *MTut Script* can take a considerable amount of time, as it is equivalent to running a *Tut Script* multiple times. However in this toy example we only run the fit/gen cycle 30 times and only a small number of the `f[X]` parameters are estimated.

24.2 Syntax of MTut Script

The *MTut Script* specifies the number of populations to sample as follows:-

```
OUTPUT_OPTIONS: {n_pop_samples: 30}
```

The *MTut Script* encodes **both** the data generation and fitting in the *GEN_EFFECTS* and *FIT_EFFECTS* sections, like a *Tut Script*. The syntax is the same. In this example the *GEN_EFFECTS* section is as follows:-

```
GEN_EFFECTS:
    POP: |
        c[AMT] = 100.0
        # f[KE] = 0.1
        f[KE] ~ unif(0.05,0.15)
        # f[PNOISE] = 0.05
        f[PNOISE] ~ unif(0.02,0.08)
        # f[KE_isv] = 0.03
        f[KE_isv] ~ unif(0.01,0.05)
    ID: |
        c[ID] = sequential(20)
        t[DOSE] = 1.0
        t[OBS] ~ unif(1.0, 50.0; 5)
        r[KE] ~ norm(0, f[KE_isv])
```

And the *FIT_EFFECTS* section is as follows:-

```
FIT_EFFECTS:
    POP: |
        # f[KE] ~ unif(0.001, 100) 0.05
        f[KE] ~ P 0.1
        # f[PNOISE] ~ unif(0.001, 100) 0.1
        f[PNOISE] ~ P 0.05
        # f[KE_isv] ~ unif(0.001, 100) 0.1
        f[KE_isv] ~ P 0.03
    ID: |
        r[KE] ~ norm(0, f[KE_isv])
```

Here the generated values of `f[KE]`, `f[PNOISE]` and `f[KE_isv]` are sampled from uniform distributions. The fitting process is initialised with (constant) values in the centre of the uniform distribution, used to sample the generating *fixed effect* values.

24.3 Summary of MTut Results

The *MTut Script* should generate an output folder containing three new scripts:-

```
mtut_example1.pyml_output/
    mtut_example1_mgen.pyml
    mtut_example1_mfit.pyml
    mtut_example1_mcomp.pyml
```

The purpose of each of theses scripts is as follows:-

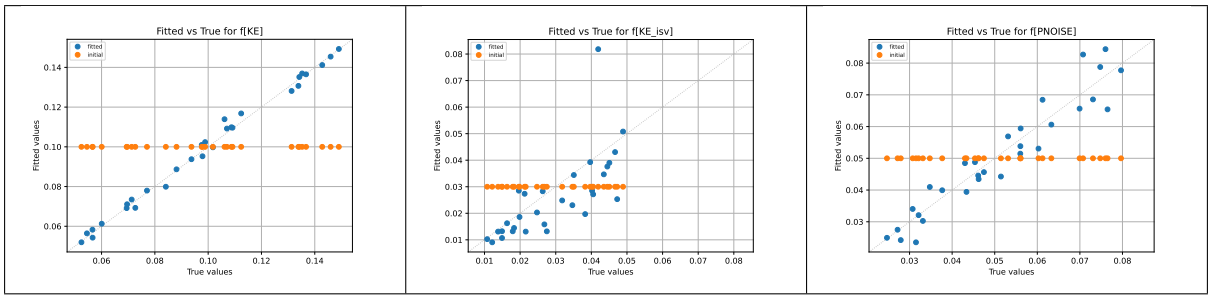
Table 1: Scripts output by a multi tutorial script

Script	Purpose	Documentation
*_mgen.pyml	Generate multiple synthetic data sets from model	*MGen Script*
*_mfit.pyml	Fit model to multiple synthetic data sets	*MFit Script*
*_mcomp.pyml	Compare gen model and fit model `f[X]`	*MComp Script*

The *MGen Script* is very similar to the *Gen Script* described in tut_syntax_simple and the *MFit Script* is very similar to the *Fit Script* described in simple_fit_example. See *Files Generated by MTut Script* for more info.

Here we mainly discuss the *MComp Script* outputs, which processes the results of *MGen Script* and *MFit Script*. The simplest output is a visual comparison of the true and fitted `f[X]` values as shown in Table 2.

Table 2: Fitted model vs true scatter plots for `f[KE]`, `f[KE_isv]` and `f[PNOISE]`



In Table 2 the blue dots are a scatter plot of fitted `f[X]` vs true `f[X]`. The green dots are initial `f[X]` vs true `f[X]`. For example in the case of `f[KE]` the true values are sampled as follows:-

```
f[KE] ~ unif(0.05,0.15)
```

i.e. the true values are uniformly sampled in the range [0.05,0.15]. The fitted `f[KE]` parameters are modelled as follows:-

```
f[KE] ~ P0.1
```

The initial values for `f[KE]` are always 0.1, see green dots in a horizontal line on the left graph in Table 2. The 'P' specifies that the fitting value of `f[KE]` is restricted to positive numbers. The final fitting values are the blue dots on the left graph in Table 2. For `f[KE]` the blue dots are clustered along the black 45 degree line. Hence fitting for `f[KE]` works well, this agrees with the initial findings in simple_fit_example. The blue dots for `f[KE_isv]` and `f[PNOISE]` (centre and right graphs in Table 2) are not as tightly clustered around the 45 degree line, indicating these parameters are harder to identify than `f[KE]`. However both `f[KE_isv]` and `f[PNOISE]` show a reasonable correlation between the true and fitted values.

Note it might well be possible to carry out a more statistical analysis of correlation between the true and fitted `f[X]`, for example finding a line of best fit through the scatter plot data. The *MComp Script* outputs are all saved to .csv files, see *Files Generated by MComp Script*, which could easily be loaded into *R* or other statistical packages for further analysis.

Part VII

PoPy for Nonmem Users

For users who are familiar with *Nonmem* we give an overview for converting *Nonmem* data sets and fitting scripts to *PoPy* format.

CHAPTER

TWENTYFIVE

NONMEM CONTROL FILE TO *POPY* FIT SCRIPT

The Nonmem control file has the same purpose as the *PoPy* *Fit Script*, *i.e.* to estimate the *fixed effects* given a PK/PD model and a *data file*. Table 1 lists the equivalent script file sections.

Table 1: Nonmem to *PoPy* Fitting

Nonmem	*PoPy*	Comment
$PROBLEM	*DESCRIPTION*	Model Description
$DATA	*FILE_PATHS*	Input data file path
$INPUT	*DATA_FIELDS*	Data header information
$SUBROUTINES	*ODE_SOLVER*	*ordinary differential equation* method
$MODEL	N/A	Number of compartments
$THETA	*EFFECTS*	Main *fixed effects*
$OMEGA	*EFFECTS*	Variance of *random effects*
$SIGMA	*EFFECTS*	Variance of measurement noise
$PK	*MODEL_PARAMS*	Model Parameters
$DES	*DERIVATIVES*	*ordinary differential equation* system
$ERROR	*PREDICTIONS*	Likelihood model
$ESTIMATION	*FIT_METHODS*	Fitting algorithms

25.1 $PROBLEM

Nonmem:-

```
$PROBLEM My pkpd model
```

to *PoPy*:-

```
DESCRIPTION: {title: My pkpd model}
```

Note in *PoPy* you can optionally add additional fields ‘name’, ‘author’, ‘abstract’ and ‘keywords’, see *DESCRIPTION* section for examples.

25.2 $DATA

Nonmem:-

```
$DATA my_nm_data.csv
```

to *PoPy*:-

```
FILE_PATHS: {input_data_file: my_popy_data.csv}
```

Here the file 'my_nm_data.csv' is assumed to be in the same directory as the *Nonmem* script file. Similarly 'my_popy_data.csv' will need to be in the same directory as the *PoPy* script.

Note you will need to convert the data file to a valid *PoPy* *data file*, see *Nonmem Data to PoPy Data File*.

25.3 $INPUT

Nonmem:-

```
$INPUT EVID ID TIME CMT AMT DV MDV
```

to *PoPy*:-

```
DATA_FIELDS: {type_field: TYPE, id_field: ID, time_field: TIME}
```

Note that the *Nonmem* '$INPUT' section redefines the header file for the data file. *PoPy* simply uses the header supplied in the .csv data file.

In *PoPy* you only need to specify the following fields in the *DATA_FIELDS* section:-

- *type_field* - equivalent to *Nonmem* EVID
- *id_field* - same as *Nonmem* ID
- *time_field* - same as *Nonmem* TIME

If you miss out the *DATA_FIELDS* section completely. Then you just need the default field names of 'TYPE', 'ID' and 'TIME' to be present in your *data file*.

See *Nonmem Data to PoPy Data File* for a more detailed discussion of the data format differences between *Nonmem* and *PoPy*.

25.4 $SUBROUTINES

25.4.1 ADVAN13 Example

Nonmem:-

```
$SUBROUTINES ADVAN13 TOL=6
```

to *PoPy*:-

```
ODE_SOLVER: {CPPLSODA: {rtol: 1e-06}}
```

In the *Nonmem* script 'ADVAN13' specifies the *LSODA* solver which is used to compute numerical solutions for the *ordinary differential equations* specified in the *Nonmem* $DES block.

The equivalent of this in *PoPy* is to specify 'CPPLSODA' in the *ODE_SOLVER* section, which is a *Python* wrapper around the *LSODA* solver. The 'CPPLSODA' parameters are as follows:-

- rtol - relative tolerance of *LSODA* solver, equivalent of TOL in *Nonmem* ($SUBROUTINES section)
- atol - additive tolerance of *LSODA* solver, equivalent of ATOL in *Nonmem* ($ESTIMATION section)
- max_nsteps - maximum number of steps allowed for *LSODA* solver, an exposed parameter in *PoPy* with a default value of '1e7'. In *Nonmem* this is fixed to '750' and **not** configurable.

See *ODE_SOLVER* section for more information on the 'CPPLSODA' solver and other solvers available in *PoPy*.

25.4.2 ADVAN 1,2,3,4,11,12 Example

Note if an analytic 'ADVAN' model is used in *Nonmem* for example:-

```
$SUBROUTINES  ADVAN2 TRANS2
```

Then this is converted to *PoPy* in a slightly different way, the *ODE_SOLVER* is as follows:-

```
ODE_SOLVER: {ANALYTIC:{}}
```

And the *DERIVATIVES* section is as follows:-

```
DERIVATIVES: |
    s[ABS,CEN] = @dep_one_cmp_cl{
        dose: @bolus{amt:c[AMT]}, KA: m[KA],
        CL: m[CL], V: m[V]}
```

Here the '@dep_one_cmp_cl' compartment function expects the parameters `m[KA]` `m[CL]` `m[V]` to be defined in *MODEL_PARAMS*, similar to how *Nonmem* 'ADVAN2' expects KA, CL and V to be defined in the $PK section. Also *PoPy* expects `c[AMT]` to be defined in the *data file* and *Nonmem* expects the 'AMT' field to exist in the *Nonmem* data file.

Note *PoPy* is explicit about the input parameter names that are required for compartment model functions and you can easily change the name of the input `c[X]` and `m[X]` parameters in your script file. In *Nonmem* you just have to know what the fixed magic words and conventions are. For example, you need to declare the variables 'KA', 'CL' and 'V' carefully in the *Nonmem* $PK section and have an 'AMT' field in your data file.

The *PoPy* equivalents of the various 'ADVAN' analytic compartment models are discussed in the *Nonmem Advan1,2,3,4,11,12 to PoPy analytic compartment models* section below.

25.5 $MODEL

Nonmem:-

```
$MODEL NCOMPARTMENTS=1
```

This does **not** have any equivalent in *PoPy*, you do **not** need to specify the number of compartments in the *DERIVATIVES* section. *PoPy* can count!

25.6 $THETA

Nonmem:-

```
$THETA
    (0.001, 0.05, 1)  ; KE
```

to *PoPy*:-

```
EFFECTS:
    POP: |
        f[KE] ~ unif(0.001, 1) 0.05
```

In *PoPy* the *fixed effect* `f[KE]` is explicitly declared at the **POP** level, *i.e.* there is one value for this parameter over the population.

The limits on `f[KE]` are defined using a *~unif()* distribution, with the initial value added as suffix.

Note that comments have been added to the *Nonmem* notation here to make it more readable. *PoPy* insists on names for all variables.

25.7 $OMEGA

Nonmem:-

```
$OMEGA
    0.1  ; KE_isv
```

to *PoPy*:-

```
EFFECTS:
    POP: |
        f[KE_isv] ~ unif(0.001, +inf) 0.1
```

In *PoPy* the *fixed effect* variance parameter `f[KE_isv]` is explicitly declared at the **POP** level, *i.e.* there is one value for this parameter over the population.

The limits on `f[KE_isv]` are defined using a *~unif()* distribution, with the initial value added as a suffix. In *PoPy* you can use the equivalent shorter notation:-

```
f[KE_isv] ~ P 0.1
```

Note that *PoPy* can deduce automatically that `f[KE_isv]` is a variance by examining the `r[KE] ~norm()` distribution definition from the *ID* level:-

```
EFFECTS:
    ID: |
        r[KE] ~ norm(0, f[KE_isv])
```

25.8 $SIGMA

Nonmem:-

```
$SIGMA
    0.001 FIX ; ANOISE
    0.2 ; PNOISE
```

to *PoPy*:-

```
EFFECTS:
    POP: |
        f[ANOISE] = 0.001
        f[PNOISE] ~ P0.2
```

In *PoPy* the *fixed effect* noise variance parameters `f[ANOISE]` and `f[PNOISE]` are explicitly declared at the *ID* level, *i.e.* there is one value for both parameters over the population.

The '=' sign is used instead of the *Nonmem* 'FIX' keyword. The '~P' notation is a shortcut for defining a *~unif()* distribution, which is equivalent to the following:-

```
f[PNOISE] ~ unif(0.001, +inf) 0.2
```

The initial value 0.2 is added as a suffix after the distribution.

Note that *PoPy* can deduce automatically that `f[ANOISE]` and `f[PNOISE]` are noise parameters (*i.e.* sigma like variables) by examining the `c[DV_CENTRAL] ~norm()` distribution likelihood definition from the *PREDICTIONS* section:-

```
PREDICTIONS: |
    p[DV_CENTRAL] = s[CENTRAL]
    var = m[ANOISE] + m[PNOISE]*p[DV_CENTRAL]**2
    c[DV_CENTRAL] ~ norm(p[DV_CENTRAL], var)
```

Where `m[ANOISE]` and `m[PNOISE]` are copies of the `f[ANOISE]` and `f[PNOISE]` parameters as defined in the *MODEL_PARAMS* section.

25.9 $PK

Nonmem:-

```
$PK
    KE = THETA(1) * exp(ETA(1))
```

to *PoPy*:-

```
MODEL_PARAMS: |
    m[KE] = f[KE] * exp(r[KE])
    m[ANOISE] = f[ANOISE]
    m[PNOISE] = f[PNOISE]
```

The *PoPy* *MODEL_PARAMS* section has exactly the same purpose as the *Nonmem* $PK section. Some differences are:-

- *Nonmem* uses numbered THETA and ETA variables, whilst *PoPy* uses named `f[X]` and `r[X]` variables.
- *Nonmem* uses bare variable names on the **left hand side** of equations in the $PK section and exports **all** of the variables for use in the *$DES* and *$ERROR* sections (just 'KE' in this case). *PoPy* requires explicit use of the `m[X]` syntax to export variables from the *MODEL_PARAMS* section.

You can use the local variable syntax in the *PoPy* *MODEL_PARAMS* section *e.g.*:-

```
MODEL_PARAMS: |
    f = f[KE]
    r = r[KE]
    m = f * exp(r)
    m[KE] = m
```

However only `m[KE]` will be available in the *PoPy* *DERIVATIVES* and *PREDICTIONS* sections **not** 'f', 'r' or 'm'. Some other differences are:-

- *PoPy* requires that the `f[ANOISE]` and `f[PNOISE]` variables (which are sigmas in *Nonmem*) are converted to `m[X]` variables, because `f[X]` variables can **not** be used as inputs to the *DERIVATIVES* and *PREDICTIONS* sections.
- *Nonmem* implements **IOV** using 'if' statements. *PoPy* *MODEL_PARAMS* can handle **IOV** `r[X]` variables without using 'if' statements, see *DDMoRe0238 Conversion Example*.
- *Nonmem* converts the $PK section into a Fortran function, whilst *PoPy* converts the *MODEL_PARAMS* section into executable *C++* code.

Otherwise the $PK and *MODEL_PARAMS* sections are quite similar. They both accept procedural *pseudocode*, *e.g.* 'if' statements *etc.* and compute model variables for each row of the data set when fitting PK/PD models.

25.10 $DES

Nonmem:-

```
$DES
    DADT(1) = -KE*A(1)
```

to *PoPy*:-

```
DERIVATIVES: |
    d[CENTRAL] = @bolus{amt:c[AMT]} - m[KE]*s[CENTRAL]
```

The *PoPy* *DERIVATIVES* section has exactly the same of purpose as the *Nonmem* $DES section. Some differences are:-

- *Nonmem* uses numbered DADT and A variables, whilst *PoPy* uses named `d[X]` and `s[X]` variables.
- *Nonmem* will allow you to use any previously specified variables as input, whereas *PoPy* restricts input variables to be of type `c[X]` and `m[X]`.
- *Nonmem* specifies the dosing time, amount and compartment (either bolus or infusion) entirely from the data file, however *PoPy* has *Dosing Functions* that have explicitly declared input parameters. The dosing compartment is clearly defined by the dose function location in the *DERIVATIVES* section. The time of each dose is still defined in the *data file*. See *Dosing Fields* for more info.
- *Nonmem* uses the magic variable 'T' to specify continuous time in the $DES section, *PoPy* uses the more obviously magic `x[TIME]` variable to do the same thing.
- *Nonmem* converts the $DES section into a Fortran function, whilst *PoPy* converts the *DERIVATIVES* section into a *C++* function, that can be called by a numerical *ordinary differential equation* solver.

Otherwise the $DES and *DERIVATIVES* sections are quite similar. They both accept procedural *pseudocode*, *e.g.* 'if' statements *etc.* and generate code that be executed by a numerical *ordinary differential equation* solver to compute compartment model amount/state variables at time points specified in the data file.

25.11 $ERROR

Nonmem:-

```
$ERROR
    Y = F + EPS(1) + F*EPS(2)
```

to *PoPy*:-

```
PREDICTIONS: |
    p[DV_CENTRAL] = s[CENTRAL]
    var = m[ANOISE] + m[PNOISE] * p[DV_CENTRAL]**2
    c[DV_CENTRAL] ~ norm(p[DV_CENTRAL], var)
```

The *PoPy* *PREDICTIONS* section has the same of purpose as the *Nonmem* $ERROR section. They both compare observations from the *data file* with model predictions. In a fitting script these sections compute a likelihood, when simulating these sections produce a noisy measurement prediction.

Some differences are:-

- The *Nonmem* $error section is much shorter, it relies on the convention that 'Y' is the dependant variable 'DV' column from the data file and 'F' is the model prediction for the compartment specified by the 'CMT' column in the data file.
- The *PoPy* *PREDICTIONS* explicitly creates `p[X]` prediction variables, often based on `s[X]` amounts from the compartment model, which all have human readable names.
- The *Nonmem* likelihood is expressed as a sum of 'EPS' normal distributions, which *Nonmem* combines into a single *~norm()* distribution to compute the likelihood for each data row.
- The *PoPy* likelihood is constructed using the '~' notation and uses named *Probability Distributions*.

In *PoPy* you have to work out how to compute the *~norm()* distribution parameters yourself. Here the mean is simply the model prediction `p[DV_CENTRAL]`. The `m[ANOISE]` and `m[PNOISE]` parameters are separate additive and proportional variances, so have a combined variance as follows:-

```
var = m[ANOISE] + m[PNOISE] * p[DV_CENTRAL]**2
```

Otherwise the $ERROR and *PREDICTIONS* sections are quite similar. They both accept procedural *pseudocode*, *e.g.* 'if' statements *etc.* and generate code that can be used to compute a likelihood value for each row of a data set (when fitting).

25.12 $ESTIMATION

Nonmem:-

```
$ESTIMATION METHOD=ITS INTERACTION PRINT=1 MAXEVALS=100
```

Or

```
$ESTIMATION METHOD=FOCE INTERACTION PRINT=1 MAXEVALS=100
```

to *PoPy*:-

```
FIT_METHODS: [JOE: {max_n_main_iterations: 100}]
```

Note estimation methods can be called sequentially in *Nonmem*:-

```
$ESTIMATION METHOD=ITS INTERACTION PRINT=1 MAXEVALS=100
$ESTIMATION METHOD=FOCE INTERACTION PRINT=1 MAXEVALS=100
```

Similarly the *JOE* fitting method can be called sequentially in *PoPy* as:-

```
FIT_METHODS:
    - JOE: {max_n_main_iterations: 100}
    - FOCE: {max_n_main_iterations: 100}
```

In *PoPy* the *JOE* fitting method, optimises the same *ObjV* as *Nonmem FOCE* and *ITS*, but the fitting algorithm is slightly different.

Note that in *PoPy* v1.0.5 the most accurate, but slower fitting method is *ND* specifed as follows:-

```
FIT_METHODS:
    - ND: {max_n_main_iterations: 100}
```

ND optimises the *FOCE ObjV* so can be compared to *Nonmem FOCE* directly.

CHAPTER

TWENTYSIX

NONMEM ADVAN1,2,3,4,11,12 TO *POPY* ANALYTIC COMPARTMENT MODELS

Table 1 shows how to convert each *Nonmem* inbuilt ADVAN analytic functions to *PoPy* equivalents.

See *ADVAN 1,2,3,4,11,12 Example* for info on *Nonmem $SUBROUTINES* section.

Note to use the analytic compartment models in *PoPy* you should always set *ODE_SOLVER* as follows:-

```
ODE_SOLVER: {ANALYTIC:{}}
```

For more information on *PoPy* analytic functions see *Analytic Compartment Functions*.

Table 1: Nonmem to *PoPy* Analytic ODEs

Nonmem	ADVAN Parameters	*PoPy*	CMP Parameters
ADVAN1	K	*@iv_one_cmp_k*	KE
ADVAN1 TRANS2	CL/V	*@iv_one_cmp_cl*	CL/V
ADVAN2	KA/K	*@dep_one_cmp_k*	KA/KE
ADVAN2 TRANS2	KA/CL/V	*@dep_one_cmp_cl*	KA/CL/V
ADVAN3	K/K12/K21	*@iv_two_cmp_k*	KE/K12/K21
ADVAN3 TRANS4	CL/V1/Q/V2	*@iv_two_cmp_cl*	CL/V1/Q/V2
ADVAN4	KA/K/K23/K32	*@dep_two_cmp_k*	KA/KE/K12/K21
ADVAN4 TRANS4	KA/CL/V1/Q/V2	*@dep_two_cmp_cl*	KA/CL/V1/Q/V2
ADVAN11	K/K12/K21/K13/K31	*@iv_three_cmp_k*	KE/K12/K21/K13/K31
ADVAN11 TRANS4	CL/V1/Q2/V2/Q3/V3	*@iv_three_cmp_cl*	CL/V1/Q2/V2/Q3/V3
ADVAN12	KA/K/K23/K32/K24/K42	*@dep_three_cmp_k*	KA/KE/K12/K21/K13/K31
ADVAN12 TRANS4	KA/CL/V2/Q3/V3/Q4/V4	*@dep_three_cmp_cl*	KA/CL/V1/Q2/V2/Q3/V3

26.1 ADVAN1

In *Nonmem*:-

```
$SUBROUTINES  ADVAN1
```

With following parameters defined in the *Nonmem* $PK section:-

- K = elimination rate

Equivalent *PoPy* *DERIVATIVES* section:-

```
DERIVATIVES: |
    s[CEN] = @iv_one_cmp_k{dose: @bolus{amt:c[AMT]}, KE: m[KE]}
```

The *Nonmem* parameters are mapped to *PoPy* as follows:-

- K -> KE

See *@iv_one_cmp_k*.

26.2 ADVAN1 TRANS2

In *Nonmem*:-

```
$SUBROUTINES  ADVAN1 TRAN2
```

With following parameters defined in the *Nonmem* $PK section:-

- CL = *clearance*
- V = *volume of distribution*

Equivalent *PoPy* *DERIVATIVES* section:-

```
DERIVATIVES: |
    s[CEN] = @iv_one_cmp_cl{dose: @bolus{amt:c[AMT]}, CL: m[CL], V: m[V]}
```

The *Nonmem* parameters are mapped to *PoPy* as follows:-

- CL -> CL
- V -> V

See *@iv_one_cmp_cl*.

26.3 ADVAN2

In *Nonmem*:-

```
$SUBROUTINES  ADVAN2
```

With following parameters defined in the *Nonmem* $PK section:-

- KA = absorption rate
- K = elimination rate

Equivalent *PoPy* *DERIVATIVES* section:-

```
DERIVATIVES: |
    s[DEP,
↪CEN] = @dep_one_cmp_k{dose: @bolus{amt:c[AMT]}, KA: m[KA], KE: m[KE]}
```

The *Nonmem* parameters are mapped to *PoPy* as follows:-

- KA -> KA
- K -> KE

See *@dep_one_cmp_k*.

26.4 ADVAN2 TRANS2

In *Nonmem*:-

```
$SUBROUTINES  ADVAN2 TRAN2
```

With following parameters defined in the *Nonmem* $PK section:-

- KA = absorption rate
- CL = *clearance*
- V = *volume of distribution*

Equivalent *PoPy* *DERIVATIVES* section:-

```
DERIVATIVES: |
    s[DEP,CEN] = @dep_one_cmp_cl{
        dose: @bolus{amt:c[AMT]},
        KA: m[KA], CL: m[CL], V: m[V]
    }
```

The *Nonmem* parameters are mapped to *PoPy* as follows:-

- KA -> KA
- CL -> CL
- V -> V

See *@dep_one_cmp_cl*.

26.5 ADVAN3

In *Nonmem*:-

```
$SUBROUTINES  ADVAN3
```

With following parameters defined in the *Nonmem* $PK section:-

- K = elimination rate from central compartment
- K12 = elimination rate from central to peripheral compartment
- K21 = elimination rate from peripheral to central compartment

Equivalent *PoPy* *DERIVATIVES* section:-

```
DERIVATIVES: |
    s[CEN,PERI] = @iv_two_cmp_k{
        dose: @bolus{amt:c[AMT]},
        KE: m[KE], K12: m[K12], K21: m[K21]}
```

The *Nonmem* parameters are mapped to *PoPy* as follows:-

- K -> KE
- K12 -> K12
- K21 -> K21

See *@iv_two_cmp_k*.

26.6 ADVAN3 TRANS4

In *Nonmem*:-

```
$SUBROUTINES  ADVAN3 TRANS4
```

With following parameters defined in the *Nonmem* $PK section:-

- CL = *clearance* from central compartment
- V1 = *volume of distribution* for central compartment
- Q = *clearance* between central and peripheral compartment
- V2 = *volume of distribution* for peripheral compartment

Equivalent *PoPy DERIVATIVES* section:-

```
DERIVATIVES: |
    s[CEN,PERI] = @iv_two_cmp_cl{
        dose: @bolus{amt:c[AMT]},
        CL: m[CL], V1: m[V1], Q: m[Q], V2: m[V2]}
```

The *Nonmem* parameters are mapped to *PoPy* as follows:-

- CL -> CL
- V1 -> V1
- Q -> Q
- V2 -> V2

See *@iv_two_cmp_cl*.

26.7 ADVAN4

In *Nonmem*:-

```
$SUBROUTINES  ADVAN4
```

With following parameters defined in the *Nonmem* $PK section:-

- KA = absorption rate from depot to central compartment
- K = elimination rate from central compartment
- K23 = elimination rate from central to peripheral compartment
- K32 = elimination rate from peripheral to central compartment

Equivalent *PoPy* *DERIVATIVES* section:-

```
DERIVATIVES: |
    s[DEP,CEN,PERI] = @dep_two_cmp_k{
        dose: @bolus{amt:c[AMT]},
        KA: m[KA], KE: m[KE], K12: m[K12], K21: m[K21]}
```

The *Nonmem* parameters are mapped to *PoPy* as follows:-

- KA -> KA
- K -> KE
- K23 -> K12
- K32 -> K21

Note *PoPy* uses consistent parameter names between *@iv_two_cmp_k* (advan3) and *@dep_two_cmp_k* (advan4), whereas *Nonmem* renumbers the parameters.

See *@dep_two_cmp_k*.

26.8 ADVAN4 TRANS4

In *Nonmem*:-

```
$SUBROUTINES  ADVAN4 TRANS4
```

With following parameters defined in the *Nonmem* $PK section:-

- KA = absorption rate from depot to central compartment
- CL = *clearance* from central compartment
- V2 = *volume of distribution* for central compartment
- Q = *clearance* between central and peripheral compartment
- V3 = *volume of distribution* for peripheral compartment

Equivalent *PoPy* *DERIVATIVES* section:-

```
DERIVATIVES: |
    s[DEP,CEN,PERI] = @dep_two_cmp_cl{
        dose: @bolus{amt:c[AMT]},
        KA: m[KA], CL: m[CL], V1: m[V1],
        Q: m[Q], V2: m[V2]}
```

The *Nonmem* parameters are mapped to *PoPy* as follows:-

- KA -> KA
- CL -> CL
- V2 -> V1
- Q -> Q
- V3 -> V2

Note *PoPy* uses consistent parameter names between *@iv_two_cmp_cl* (advan3 trans 4) and *@dep_two_cmp_cl* (advan4 trans4), whereas *Nonmem* renumbers the parameters.

See *@dep_two_cmp_cl*.

26.9 ADVAN11

In *Nonmem*:-

```
$SUBROUTINES  ADVAN11
```

With following parameters defined in the *Nonmem* $PK section:-

- K = elimination rate from central compartment
- K12 = elimination rate from central to first peripheral compartment
- K21 = elimination rate from first peripheral to central compartment
- K13 = elimination rate from central to second peripheral compartment
- K31 = elimination rate from second peripheral to central compartment

Equivalent *PoPy* *DERIVATIVES* section:-

```
DERIVATIVES: |
    s[CEN,PERI1,PERI2] = @iv_three_cmp_k{
        dose: @bolus{amt:c[AMT]},
        KE: m[KE], K12: m[K12], K21: m[K21],
        K13: m[K13], K31: m[K31]}
```

The *Nonmem* parameters are mapped to *PoPy* as follows:-

- K -> KE
- K12 -> K12
- K21 -> K21

- K13 -> K13
- K31 -> K31

See *@iv_three_cmp_k*.

26.10 ADVAN11 TRANS4

In *Nonmem*:-

```
$SUBROUTINES  ADVAN11 TRANS4
```

With following parameters defined in the *Nonmem* $PK section:-

- CL = *clearance* from central compartment
- V1 = *volume of distribution* for central compartment
- Q2 = *clearance* between central and first peripheral compartment
- V2 = *volume of distribution* for first peripheral compartment
- Q3 = *clearance* between central and first peripheral compartment
- V3 = *volume of distribution* for second peripheral compartment

Equivalent *PoPy* *DERIVATIVES* section:-

```
DERIVATIVES: |
    s[CEN,PERI1,PERI2] = @iv_three_cmp_cl{
        dose: @bolus{amt:c[AMT]},
        CL: m[CL], V1: m[V1], Q2: m[Q2],
        V2: m[V2], Q3: m[Q3], V3: m[V3]}
```

The *Nonmem* parameters are mapped to *PoPy* as follows:-

- CL -> CL
- V1 -> V1
- Q2 -> Q2
- V2 -> V2
- Q3 -> Q3
- V3 -> V3

See *@iv_three_cmp_cl*.

26.11 ADVAN12

In *Nonmem*:-

```
$SUBROUTINES  ADVAN12
```

With following parameters defined in the *Nonmem* $PK section:-

- KA = absorption rate from depot to central compartment
- K = elimination rate from central compartment
- K23 = elimination rate from central to first peripheral compartment
- K32 = elimination rate from first peripheral to central compartment
- K24 = elimination rate from central to second peripheral compartment
- K42 = elimination rate from second peripheral to central compartment

Equivalent *PoPy* *DERIVATIVES* section:-

```
DERIVATIVES: |
    s[DEP,CEN,PERI1,PERI2] = @dep_three_cmp_k{
        dose: @bolus{amt:c[AMT]},
        KA: m[KA], KE: m[KE],
        K12: m[K12], K21: m[K21],
        K13: m[K13], K31: m[K31]}
```

The *Nonmem* parameters are mapped to *PoPy* as follows:-

- KA -> KA
- K -> KE
- K23 -> K12
- K32 -> K21
- K24 -> K13
- K42 -> K31

Note *PoPy* uses consistent parameter names between *@iv_three_cmp_k* (advan11) and *@dep_three_cmp_k* (advan12), whereas *Nonmem* renumbers the parameters.

See *@dep_three_cmp_k*.

26.12 ADVAN12 TRANS4

In *Nonmem*:-

```
$SUBROUTINES  ADVAN12 TRANS4
```

With following parameters defined in the *Nonmem* $PK section:-

- KA = absorption rate from depot to central compartment
- CL = *clearance* from central compartment
- V2 = *volume of distribution* for central compartment
- Q3 = *clearance* between central and first peripheral compartment
- V3 = *volume of distribution* for first peripheral compartment
- Q4 = *clearance* between central and first peripheral compartment
- V4 = *volume of distribution* for second peripheral compartment

Equivalent *PoPy* *DERIVATIVES* section:-

```
DERIVATIVES: |
    s[DEP,CEN,PERI1,PERI2] = @dep_three_cmp_cl{
        dose: @bolus{amt:c[AMT]},
        KA: m[KA],
        CL: m[CL], V1: m[V1],
        Q2: m[Q2], V2: m[V2],
        Q3: m[Q3], V3: m[V3]}
```

The *Nonmem* parameters are mapped to *PoPy* as follows:-

- KA -> KA
- CL -> CL
- V2 -> V1
- Q3 -> Q2
- V3 -> V2
- Q4 -> Q3
- V4 -> V3

Note *PoPy* uses consistent parameter names between *@iv_three_cmp_cl* (advan11 trans 4) and *@dep_three_cmp_cl* (advan12 trans4), whereas *Nonmem* renumbers the parameters.

See *@dep_three_cmp_cl*.

CHAPTER

TWENTYSEVEN

NONMEM DATA TO *POPY* DATA FILE

The *Nonmem* data file has the same purpose as the *PoPy* *data file*. It represents the *observations* at different time points and the dosing regimens for each subject.

For simple data sets you may only need to substitute a few values to create a valid *PoPy* *data file*. If you have multiple dosing regimes and multiple measurements then the conversion may be more difficult.

Table 1 lists the the *PoPy* data fields for each *Nonmem* data field. Each subsection gives a one to one example conversion to *PoPy* format.

Table 1: Nonmem to *PoPy* Data

Nonmem	*PoPy*	Comment
EVID	*TYPE*	Data row property field
ID	*ID*	Identity field
TIME	*TIME*	Time field
CMT	N/A	Compartment field
AMT	AMT	Dose Amount field
DV	*Observation field*	Observations
MDV	*Observation flag field*	Missing observations

Both *PoPy* and *Nonmem* need to load a *data file* when estimating parameters. See *$DATA* for *Nonmem*'s method of specifying the input data file path and how to specify the input data file path in a *PoPy* *Fit Script*.

27.1 EVID

'EVID' is a required field in *Nonmem*. There is an equivalent required *TYPE* field in *PoPy*. The major difference is that *Nonmem* 'EVID' uses integers to define row properties, whereas *PoPy* uses human readable strings as shown in Table 2.

Table 2: Nonmem EVID to *PoPy* TYPE

Nonmem EVID	*PoPy* TYPE	Comment
0	obs	Observation Row
1	dose	Dosing Row
2	pred	Prediction Row
3	reset	Reset Row
4	reset+dose	Reset and Dose Row

Note in *PoPy* the 'dose' *TYPE* entry can have a name suffix using *PoPy*'s ':' notation. See specifying multiple dose types in PoPy for more details.

27.2 ID

'ID' is a required field in *Nonmem* and *PoPy*. The 'ID' column defines the individual for each data row. It is usually **not** necessary to convert the 'ID' column of the data set.

However, note that in *PoPy* the same identifier in the 'ID' field is **always** treated as the same individual. In *Nonmem* only identical identifiers that are in consecutive rows are treated as one individual.

For example in Table 3.

Table 3: Nonmem id example

ID	*Nonmem*	*PoPy*
Bill	New id	New id
Bill	Bill again	Bill again
Sandra	New id	New id
Bill	New id	Bill again
Sandra	New id	Sandra again

Nonmem thinks there are 4 subjects, whilst *PoPy* thinks there are 2.

27.3 TIME

'TIME' is a required field in *Nonmem* and *PoPy*. The 'TIME' column defines the time stamp for each row. It is usually **not** necessary to convert the 'TIME' column of the data set.

In both *Nonmem* and *PoPy* the time field is required to be monotonically increasing, unless a *EVID* = 3 or 4 row is reached. In *PoPy* time is reset when a *TYPE* = 'reset' or 'reset+dose' row is reached.

One complication that can arise is if the *Nonmem* data is split over date and time, for example see Table 4.

Table 4: Nonmem id time

Nonmem date	*Nonmem* time	*PoPy* time
2016-02-12	10:30	0.0
2016-02-13	19:01	32.5167
2016-02-13	19:02	32.5333
2016-02-13	23:39	37.15
2016-02-13	23:42	37.2
2016-02-14	10:06	47.6

Here the data for each individual needs to be converted to the time after the first record (or time since last reset) for use in *PoPy*.

27.4 CMT

The 'CMT' field in *Nonmem* is used to specify the index of the compartment where doses will be administered. *i.e.* rows with EVID=1 or EVID=4, with CMT=x will result in an bolus or infusion dose being administered in the compartment numbered 'x' in the *$DES* section of the Nonmem control file.

PoPy deliberately does **not** specify the dose compartment in the *data file*. Instead the dose compartment is specified in the *DERIVATIVES* section by the location of the *Dosing Functions*.

If the data only contains one type of dose, *e.g.* one drug which is always a bolus or always an infusion, then you can just ignore the 'CMT' field when converting to *PoPy* format.

If the data contains multiple types of dose however then *PoPy* needs a way of distinguishing between the two types (*Nonmem* uses a different CMT integer typically). In *PoPy* you need to give the dose a name, using the ':' notation. An example data conversion with two types of dose is shown in Table 5.

Table 5: Nonmem cmt to *PoPy* dose name

Nonmem evid	*Nonmem* cmt	*PoPy* type	Comment
1	1	dose:first_drug	drug one in first compartment
1	1	dose:first_drug	drug one in first compartment
1	3	dose:second_drug	drug two in third compartment
1	1	dose:first_drug	drug one in first compartment

Note that the *PoPy* format above, leaves the destination compartment of each drug to be determined in the script file (the *DERIVATIVES* section), because the depot compartment for each drug is primarily a modelling decision, which might be changed in later analyses.

27.5 AMT

The *Nonmem* AMT field specifies the amount of each dose. The same field can be used in *PoPy*, so usually the AMT field needs no conversion.

Nonmem only allows a single AMT field to be used. If you have multiple doses, *e.g.* for two different drugs, then *Nonmem* forces you to put all dose amount values in a single column in your data file, even if the amounts are in different units.

In your *PoPy* data file you might want to take the opportunity to use separate columns, *e.g.* 'AMT_DRUG1', 'AMT_DRUG2' as a way of making your *data file* and *script file* clearer.

27.6 DV

The *Nonmem* DV field specifies the observed measurements in a data file. For example plasma drug concentration for a PK study or biomarker data in a PD trial. The same field can be used in *PoPy*, so often the DV field requires no conversion.

However *Nonmem* only allows a single DV field to be used. If you have multiple types of measurement in a study then *Nonmem* forces you to place all measurement values in a single column, even if the values have different units.

In your *PoPy* *data file* you might like to split the DV data into separate columns, *e.g.* 'CONC', and 'MARKER'. This will make your *data file* and *script file* easier to read.

An example data conversion with two types of measurement is shown in Table 6.

Table 6: Nonmem DV to *PoPy* named fields

Nonmem DV	CONC	CONC_FLAG	MARKER	MARKER_FLAG	Comment
5.3	5.3	1	0	0	conc obs
3	0	0	3	1	marker obs
5	0	0	5	1	marker obs
12.1	12.1	1	0	0	conc obs

Note in Table 6 it is necessary to use the '_FLAG' field convention. The '_FLAG' field is similar to the *Nonmem MDV* field, but you can have multiple '_FLAG' fields. In a flag field '1' means use this observation, '0' means ignore. The flag field means you don't have to use 'if statements' in the *PREDICTIONS* section.

27.7 MDV

The *Nonmem* MDV (missing data value) column is used to ignore some observations. It is similar in function to the *PoPy* flag field syntax described in *DV*.

However the MDV indicator contains a double negative, an observation is valid in *Nonmem* if MDV=0, *i.e.* it is **not missing**. The *PoPy* flag field is just yes/no, *i.e.* an observation X is valid if X_FLAG =1.

An example DV/MDV conversion is shown in Table 7.

Table 7: Nonmem DV/MDV to *PoPy* flag fields

Nonmem DV	*Nonmem* MDV	CONC	CONC_FLAG	Comment
5.3	0	5.3	1	valid obs
na	1	0.0	0	invalid obs
0.0	1	0.0	0	invalid obs
2.9	0	2.9	1	valid obs

Here:-

$$FLAG = 1 - MDV$$

Also, in *Nonmem* you are only allowed to have one MDV field, which makes it less useful when you have multiple types of measurement.

CHAPTER

TWENTYEIGHT

DDMORE0061 CONVERSION EXAMPLE

A real world model, based on a gentamicin PK study [Harling2015]. It has a combined *bolus* + *infusion* dose and the original PK model uses ADVAN3 TRANS4 in *Nonmem*. There are some covariate effects on the clearance and volume of distribution of the central compartment parameters. There are 210 individuals and 1949 data rows. See DDMoRe: 0061

Here we describe the manual conversion of the *Nonmem* script file `c:\PoPy\validation\ddmore0061\Executable_gentamicin_pk.mod` to create the *PoPy Fit Script* `c:\PoPy\validation\ddmore0061\popy\fit_script.pyml`.

28.1 Data Source

The *Nonmem* script loads the data as follows:-

```
$DATA Simulated_gentamicin_pk.csv IGNORE=@
```

Whereas the *PoPy* script uses this syntax:-

```
FILE_PATHS: { input_data_file: Simulated_gentamicin_pk.csv }
```

28.2 Data Preprocessing

Because PoPy expects data in a slightly different format to *Nonmem*, we make some modifications to the incoming data using the PREPROCESS block of the input script.

The critical columns for the original *Nonmem* data set looks something like Table 1.

Table 1: Main data fields for Nonmem (first six rows)

ID	TIME	AMT	RATE	DUR	DV	MDV	EVID
1	0	150	0	0	12.376	1	1
1	9	150	0	0	13.156	1	1
1	14.25	0	0	0	2.2347	0	0
1	16.92	150	0	0	13.599	1	1
1	25	150	250	0.5	1.322	1	1
1	33	0	0	0	1.459	0	0

and are missing a `TYPE` column along with a flag column that serves essentially the same purpose as Nonmem's `MDV` column.

We add these missing columns using the *PREPROCESS* section of the fit to modify every row of the incoming data file as it is read:

```
PREPROCESS: |
    # Call this helper function that (a) converts Nonmem's EVID column
    # values into PoPy's TYPE column values and (b) adds a DV_FLAG
    # column that is 1 for observation rows (EVID=0) and 0 everywhere else.
    nonmem2popy()

    # Augment PoPy's new TYPE column value for dose rows (where AMT>0)
    # with information about whether the dose is an infusion (RATE>0)
    # or a bolus (RATE=0).
    if c[AMT] > 0:
        if c[RATE] > 0:
            c[TYPE] += ":DOSE_infrate"
        else:
            c[TYPE] += ":DOSE_bolus"

    # Make a copy of Nonmem's DV column to give it a more descriptive
    # name, DRUG_CONC, and update the corresponding flag.
    # (This is optional and could be left out after changing the
    # PREDICTIONS block to use DV rather than DRUG_CONC.)
    c[DRUG_CONC] = c[DV]
    c[DRUG_CONC_FLAG] = c[DV_FLAG]
```

Technically the *Nonmem* data set above has no CMT field, but CMT defaults to 1 in *Nonmem* if it is not provided. *PoPy* does not assign a compartment in the data file, preferring to assign the compartment in the model definition (where compartments have meaning).

The (critical columns of the) preprocessed dataset used by *PoPy* will now look like that in Table 2.

Table 2: Extra data fields for *PoPy* (first six rows)

DRUG_CONC	DRUG_CONC_FLAG	TYPE
0	0	dose:DOSE_bolus
0	0	dose:DOSE_bolus
2.2347	1	obs
0	0	dose:DOSE_bolus
0	0	dose:DOSE_infrate
1.459	1	obs

Here the new `DRUG_CONC` field will be the target value for the new *PoPy* script. The `DRUG_CONC_FLAG` switches the observations on/off appropriately. Note that `DRUG_CONC_FLAG` is the inverse of *Nonmem*'s' `MDV` flag. The new `TYPE` field is formatted to indicate either *bolus* or *infusion* doses in *PoPy*.

28.3 Fixed/Random Effects Specification

In the *Nonmem* script, the theta/omega/sigma *fixed effect* definitions are:-

```
$THETA
    (0,4.1) ; CL
    (1,8.63) ; V1
    (0,1.21) ; Q
    (0,8.12) ; V2

    (-0.01,0.00982,0.016) ; BCLC
    (-0.409) ; ALB
    (0,0.00683) ; DCLC

$OMEGA
    0.0465 ; CL
    0.376 ; Q

$SIGMA
    0.0297 ; E1
    0.0486 ; E2
```

In *PoPy* these *fixed effect* are defined in *EFFECTS*:-

```
EFFECTS:
    POP: |
        f[CL] ~ P 4.1
        f[V1] ~ unif(1, +inf) 8.63
        f[Q] ~ P 1.21
        f[V2] ~ P 8.12
        f[BCLC_eff] ~ unif(-0.01, 0.016) 0.00982
        f[ALB_eff] = -0.409
        f[DCLC_eff] ~ P 0.00683

        f[CL_isv, Q_isv] ~ diag_matrix() [ [ 0.0465, 0.376 ] ]

        f[PNOISE_var] ~ P 0.0297
        f[ANOISE_var] ~ P 0.0486
```

Note the ranges and initial values of the `f[X]` variables are defined using different syntax from the $THETA variables. Note also that the `f[X]` have explicit names, whereas the *Nonmem* list of thetas has been annotated with comments to make it human readable.

In *PoPy* you can specify a *diagonal matrix*, which can be explicitly used as a covariance matrix input to a *Multivariate Normal Distribution* distributed `r[X]` vector, as follows:-

```
EFFECTS:
    ID: |
        r[CL_isv, Q_isv] ~ mnorm([0,0], f[CL_isv, Q_isv])
```

There `r[CL_isv]` is the equivalent of 'ETA(1)' and `r[Q_isv]` is the equivalent of 'ETA(2)'. In

Nonmem you just have to remember the index values, whereas *PoPy* uses actual real life variable names.

In *PoPy* the `SIGMA` variables are treated as *fixed effects* and also defined in *EFFECTS*.

28.4 PK Model Parameters Specification

The *Nonmem* PK section is as follows:-

```
$PK
    BSA = 71.84*(WT**0.425)*(HT**0.725)
    BSA = BSA/10000
    HT1 = HT/100
    BMI = WT/HT1**2

    IF(NEWIND.NE.2) BCLC = CLC
    DCLC = CLC-BCLC
    IF(NEWIND.NE.2) BBSA = BSA

    TVCL = THETA(1) * (1 + THETA(5)*(BCLC-81) + THETA(7)*DCLC)
    TVV1 = THETA(2) * BBSA*(ALB/34)**THETA(6)
    TVQ  = THETA(3)
    TVV2 = THETA(4)

    CL = TVCL * EXP(ETA(1))
    V1 = TVV1
    Q  = TVQ  * EXP(ETA(2))
    V2 = TVV2

    S1 = V1
```

This maps almost exactly to *PoPy*'s' *MODEL_PARAMS* section:-

```
MODEL_PARAMS: |
    BSA = 71.84 * (c[WT]**0.425) * (c[HT]**0.725)
    BSA = BSA / 10000
    HT1 = c[HT] / 100
    BMI = c[WT] / HT1**2

    BCLC = c[CLC]
    DCLC = c[CLC] - BCLC
    BBSA = BSA

    TVCL = f[CL] * (1 + f[BCLC_eff]*(BCLC-81) + f[DCLC_eff]*DCLC)
    TVV1 = f[V1] * BBSA * (c[ALB]/34)**f[ALB_eff]
    TVQ  = f[Q]
    TVV2 = f[V2]

    m[CL] = TVCL * EXP(r[CL_isv])
    m[V1] = TVV1
```

(continues on next page)

(continued from previous page)

```
    m[Q]  = TVQ  * EXP(r[Q_isv])
    m[V2] = TVV2

    m[PNOISE_var] = f[PNOISE_var]
    m[ANOISE_var] = f[ANOISE_var]
```

Note in the above *Nonmem* specifies 'S1 = V1', to scale the amounts in compartment one. This is not necessary in *PoPy*, as the compartment amounts are converted to concentrations in the *PREDICTIONS* section explicitly.

Also note that *PoPy* requires the lines:-

```
m[PNOISE_var] = f[PNOISE_var]
m[ANOISE_var] = f[ANOISE_var]
```

To propagate the `f[X]` noise *fixed effects* which are similar to the *Nonmem* builtin sigma variables.

Note in this example the *Nonmem* 'NEWIND' keyword is superfluous, as the `c[CLC]` covariate and BSA variable are constant within each individual.

28.5 ODEs Specification

The *Nonmem* control file prescribes the compartment model on this line:-

```
$SUBROUTINE ADVAN3 TRANS4
```

That uses the inbuilt 'ADVAN3' model with 'TRANS4' parametrisations. This uses the analytic solution for a two compartment model that expects the CL, V1, Q, V2 parameters to be defined in the *Nonmem* PK section.

The *PoPy* control file here specifies the compartment model equations explicitly in the *DERIVATIVES* section:-

```
DERIVATIVES: |
    dose[DOSE_bolus] = @bolus{amt:c[AMT], lag:0.0}
    dose[DOSE_infrate] = @inf_rate{amt:c[AMT], lag:0.0, rate:c[RATE]}
    d[CENTRAL] = dose[DOSE_bolus] + dose[DOSE_infrate]␣
↪- s[CENTRAL]*m[Q]/m[V1] + s[PERI]*m[Q]/m[V2] - s[CENTRAL]*m[CL]/m[V1]
    d[PERI]    =          ␣
↪                              s[CENTRAL]*m[Q]/m[V1] - s[PERI]*m[Q]/m[V2]
```

Note that *PoPy* has an equivalent builtin version of 'ADVAN3 TRANS4' as follows:-

```
DERIVATIVES: |
    s[CENTRAL, PERI] = @iv_two_cmp_
↪cl{ dose: @bolus{amt:c[AMT]}, CL: m[CL], V1: m[V1], Q: m[Q], V2: m[V2] }
```

However this function is unable to process two different types of dose (bolus + infusion), unlike the long hand version above. Note that *PoPy* will be using an *ordinary differential equation* solver here (instead of a built in analytic solution), so we also need to specify this *PoPy* field:-

```
ODE_SOLVER: {CPPLSODA: {}}
```

Which tells *PoPy* to use the c++ LSODA solver, the same method as *Nonmem* Advan13.

28.6 Likelihood Error Specification

The *Nonmem* ERROR section is as follows:-

```
$ERROR
    Y = F*EXP(ERR(1)) + ERR(2)
    IPRED=F
    IRES=DV-IPRED
```

The *PoPy* equivalent is shown below in the *PREDICTIONS* section:-

```
PREDICTIONS: |
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
    var = m[PNOISE_var] * p[DV_CENTRAL]**2 + m[ANOISE_var]
    c[DRUG_CONC] ~ norm(p[DV_CENTRAL], var)
```

Note here the line 'Y = F*EXP(ERR(1)) + ERR(2)' defines an proportional exponential normal noise model with an additional additive normal noise component (even though it is not clear - to the *PoPy* developers, at least - what the distribution of a log normal + a normal distribution is). *PoPy* requires a known distribution to calculate a likelihood. If you approximate the exponential (assuming ERR(1) is small) and get 'Y = F*(1 + ERR(1)) + ERR(2)' then this is the standard proportional + additive noise model. In this case, the *PoPy* *Fit Script* return a very similar objective function to the *DDMoRe Nonmem* objective function, indicating that *Nonmem* also makes the same approximation.

28.7 Parameter Fitting Methods Specification

The *Nonmem* control file specifies using FOCE fitting (METHOD=1) below:-

```
$ESTIMATION MAXEVALS=0 SIG=3 PRINT=10 METHOD=1 INTER
```

The *PoPy* equivalent, specifying the *ND* fitting method is:-

```
FIT_METHODS: [ND: {max_n_main_iterations: 0}]
```

Setting 'max_n_main_iterations' to zero, means that *PoPy* will optimise the `r[X]` parameters and leave the `f[X]` unchanged.

CHAPTER

TWENTYNINE

DDMORE0093 CONVERSION EXAMPLE

A real world model, based on a QT study [Cheung2015]. The model incorporates a circadian function with no compartment derivatives. There are 59 individuals and 5790 total data rows. See DDMoRe: 0093

Here we describe the manual conversion of the *Nonmem* script file `c:\PoPy\validation\ddmore0093\Executable_run7b.ctl` to create the *PoPy* *Fit Script* `c:\PoPy\validation\ddmore0093\popy\fit_script.pyml`.

29.1 Data Source & Preprocessing

The *Nonmem* script loads the data as follows:-

```
$DATA Simulated_dataset.csv
   IGNORE = #
   IGNORE = (EVID == 1)
```

Note the 'IGNORE' syntax removes the dosing rows from the data set, effectively preprocessing the dataset at the point of loading.

We achieve the same thing in *PoPy* using the *PREPROCESS* block:

```
FILE_PATHS: { input_data_file: Simulated_dataset.csv }
PREPROCESS: |
    # Call this helper function that (a) converts Nonmem's EVID column
    # values into PoPy's TYPE column values and (b) adds a DV_FLAG
    # column that is 1 for observation rows (EVID=0) and 0 everywhere else.
    nonmem2popy()

    # Suppress warning about having doses we don't use
    # by ignoring dose rows (EVID==1) altogether.
    if c[TYPE] == 'dose': return
```

which loads the *PoPy* data, converts `EVID` values to `TYPE` values and also excludes the dosing rows.

29.2 Fixed/Random Effects Specification

In the *Nonmem* script, the theta/omega/sigma variable definitions are:-

```
$THETA (0, 372.6)       ; 1 Baseline QTcF [ms]
$THETA (0, 0.01844)     ; 2 Amplitude  24h
$THETA (0, 3.62)        ; 3 Peak shift 24h
$THETA (0, 0.01392)     ; 4 Amplitude  12h circadian rhythm
$THETA (0, 1.301)       ; 5 Peak shift 12h circadian rhythm
$THETA (0, 0.003441)    ; 6 Slope (linear effect) parameter

$OMEGA  0.0392          ; 1 Baseline QTcF - (BSV)
$OMEGA  0.3523          ; 2 Amplitude 24h  - BSV
$OMEGA  2.007           ; 3 Peak shift 24h - BSV
$OMEGA  0.3848          ; 4 Amplitude 12h  - BSV
$OMEGA  0.993           ; 5 Peak shift 12h - BSV
$OMEGA  0.0001          ; 6 Slope          - BSV

$SIGMA  0.01802         ; 1 Proportional residual error [ms]
```

The equivalent `f[X]` in the *PoPy* script are:-

```
EFFECTS:
    POP: |
        # THETAs
        f[BASE] ~ P 372.6
        f[AMP24] ~ P 0.01844
        f[SHFT24] ~ P 3.62
        f[AMP12] ~ P 0.01392
        f[SHFT12] ~ P 1.301
        f[SLOPE] ~ P 0.003441

        # OMEGAs
        f[BASE_bsv,
↪AMP24_bsv,SHFT24_bsv,AMP12_bsv,SHFT12_bsv,SLOPE_bsv] ~ diag_matrix() [
            [ 0.0392, 0.3523, 2.007, 0.3848, 0.993, 0.0001 ]
        ]

        # SIGMAs
        f[NOISEVAR] ~ P 0.01802
    ...
```

Additionally *PoPy* has the 'ID' section of the *EFFECTS*:-

```
EFFECTS:
    ...
    ID: |
        r[ BASE, AMP24, SHFT24, AMP12, SHFT12, SLOPE ] ~ mnorm(
            [0,0,0,0,0,0],
            f[BASE_bsv,AMP24_bsv,SHFT24_bsv,AMP12_bsv,SHFT12_bsv,SLOPE_bsv]
```

(continues on next page)

(continued from previous page)

```
        )
```

This section defines `r[BASE]` and `r[AMP24]` etc. These `r[X]` definitions are a more explicit version of *Nonmem* implicitly creating ETA(X) variables for each of the omega variables. With *Nonmem* you have to remember what each of the ETA indices represent. Hence the heavily commented 'PRED' block below, which defined the variables for each individual in the *Nonmem* script.

29.3 PK Model Parameters Specification

```
$PRED
   BASE   = THETA(1) * EXP(ETA(1))
↪                  ; Baseline QTcF with between subject varaibility (BSV)
   AMP24  = THETA(2) * EXP(ETA(2))
↪                     ; The amplitude of the 24h circadian rhythm
   SHFT24 = THETA(3) + ETA(3)
↪                           ; The peak shift of the 24h circadian rhythm
   AMP12  = THETA(4) * EXP(ETA(4))
↪                     ; The amplitude of the 12h circadian rhythm
   SHFT12 = THETA(5) + ETA(5)
↪                           ; The peak shift of the 12h circadian rhythm

   CIRC24
↪= AMP24 * COS(2 * 3.14 * (TIME - SHFT24)/24)   ; 24 hour circadian rhythm
   CIRC12
↪= AMP12 * COS(2 * 3.14 * (TIME - SHFT12)/12)   ; 12 hour circadian rhythm

   RYTM   = BASE * (1 + CIRC24 + CIRC12)
↪          ; Change in baseline QTcF over the day due to circadian rhythm

   SLOPE  = THETA(6) + ETA(6)                              ; Linear effect
   EFF    = SLOPE * CPP                                    ; Linear effect

   ...
```

The *MODEL_PARAMS* section of the *PoPy* script is very similar:-

```
MODEL_PARAMS: |

    BASE   = f[BASE] * exp(r[BASE])
    AMP24  = f[AMP24] * exp(r[AMP24])
    SHFT24 = f[SHFT24] + r[SHFT24]
    AMP12  = f[AMP12] * exp(r[AMP12])
    SHFT12 = f[SHFT12] + r[SHFT12]

    CIRC24 = AMP24 * COS(2 * 3.14 * (c[TIME] - SHFT24)/24)
    CIRC12 = AMP12 * COS(2 * 3.14 * (c[TIME] - SHFT12)/12)
    m[RYTM] = BASE * (1 + CIRC24 + CIRC12)
```

(continues on next page)

(continued from previous page)

```
    SLOPE = f[SLOPE] + r[SLOPE]
    m[EFF] = SLOPE * c[CPP]
    m[NOISEVAR] = f[NOISEVAR]
```

The above is almost identical apart from the line 'm[NOISEVAR] = f[NOISEVAR]'. Note in *MODEL_PARAMS* it is necessary to define `m[X]` variables that you wish to use in subsequent sections, e.g. *DERIVATIVES* or *PREDICTIONS*.

29.4 ODEs Specification

This model does not require or use differential equations so the `$DES` section is absent in the *Nonmem* file and the *DERIVATIVES* section is absent from the *PoPy* file.

29.5 Likelihood Error Specification

The *Nonmem* error model (also here part of the 'PRED' section) is defined as follows:-

```
IPRED  = RYTM  + EFF         ; Linear direct effect model
W      = IPRED * SIGMA(1,1)
IWRES  = (QTCF - IPRED)/W

Y      = IPRED + IPRED*EPS(1)
```

The equivalent in *PoPy* is the *PREDICTIONS* block:-

```
PREDICTIONS: |
    p[CONC_PLASMA] = m[RYTM] + m[EFF]
    var = m[NOISEVAR] * p[CONC_PLASMA]**2
    c[QTCF] ~ norm(p[CONC_PLASMA], var)
```

Note *PoPy* explicitly defines the likelihood by comparing `c[QTCF]` from the data file with the predicition `p[CONC_PLASMA]`, using a normal distribution with proportional variance.

This is more explicit than the *Nonmem* line 'Y = IPRED + IPRED*EPS(1)', which implicitly uses the 'DV' *Nonmem* magic variable for Y. The likelihood is expressed as a function of EPS normal distributions.

29.6 Parameter Fitting Methods Specification

The *Nonmem* control file specifies using FOCE fitting (METHOD=1) below:-

```
$ESTIMATION METHOD=1 MAXEVALS=99999 INTER NOABORT PRINT=5
```

The *PoPy* equivalent, specifying the *ND* fitting method is:-

```
FIT_METHODS: [ND: {max_n_main_iterations: 100}]
```

CHAPTER

THIRTY

DDMORE0238 CONVERSION EXAMPLE

A real world model, based on a gentamicin PK study [Germovsek2017].

A Population PK model with IOV, infusion dosing and custom derivative equations to implement the PK. The dataset consists of 205 individuals and 2788 rows. See DDMoRe: 0238

Here we describe the manual conversion of the *Nonmem* script file, `c:\PoPy\validation\ddmore0238\Executable_run35b_ddm2.ctl` to create the *PoPy* *Fit Script* `c:\PoPy\validation\ddmore0238\popy\fit_script.pyml`.

Here the *PoPy* 'TYPE' field is a more explicit version of the *Nonmem* 'EVID' field. The *PoPy* 'DV_CENTRAL' field is a copy of the *Nonmem* 'DV' field, with an additional 'DV_CENTRAL_FLAG' field to make the true observed values more obvious. Note here *PoPy* could also use the *Nonmem* 'DV' field because it is always defined for all rows with `c[EVID]` =0 or equivalently `c[TYPE]` ='obs'.

30.1 Data Source

The *Nonmem* script loads the data as follows:-

```
$DATA simdataDDM.csv IGNORE=@
```

Whereas the *PoPy* script uses this syntax:-

```
FILE_PATHS:
    # path to input comma separated value file in popy data format
    input_data_file: Simulated_simdataDDM.csv
```

30.2 Data Preprocessing

Because PoPy expects data in a slightly different format to *Nonmem*, we make some modifications to the incoming data using the PREPROCESS block of the input script.

The critical columns for the original *Nonmem* data set looks something like Table 1.

Table 1: Main data fields for Nonmem (first eight rows)

ID	TIME	RATE	EVID	AMT	WT	CREAT	DV	OCC
1001	0	72	1	6	2120	78	0	1
1001	11.5	0	0	0	2120	78	1.109	1
1001	12	72	1	6	2120	78	0	2
1001	12.5	0	0	0	2120	78	7.0181	2
1001	24	48	1	4	2120	78	0	2
1001	36	48	1	4	2120	78	0	2
1001	48	48	1	4	2120	78	0	3
1001	58.5	0	0	0	2120	78	2.2136	3

and are missing a `TYPE` column along with a flag column that serves essentially the same purpose as Nonmem's `MDV` column.

We add these missing columns using the *PREPROCESS* section of the fit to modify every row of the incoming data file as it is read:

```
PREPROCESS: |
    # Call this helper function that (a) converts Nonmem's EVID column
    # values into PoPy's TYPE column values and (b) adds a DV_FLAG
    # column that is 1 for observation rows (EVID=0) and 0 everywhere else.
    nonmem2popy()

    # Augment PoPy's new TYPE column value for dose rows (where AMT>0)
    # with information indicating that the dose is an infusion (RATE>0).
    if c[AMT] > 0 and c[RATE] > 0:
        c[TYPE] += ":DOSE_infrate"

    # Make a copy of Nonmem's DV column to give it a more descriptive
    # name, DV_CENTRAL, and update the corresponding flag.
    # (This is optional and could be left out after changing the
    # PREDICTIONS block to use DV rather than DV_CENTRAL.)
    c[DV_CENTRAL] = c[DV]
    c[DV_CENTRAL_FLAG] = c[DV_FLAG]
```

The (critical columns of the) preprocessed dataset used by *PoPy* will now look like that in Table 2.

Table 2: Extra data fields for *PoPy* (first eight rows)

DV_CENTRAL	DV_CENTRAL_FLAG	TYPE
0	0	dose:DOSE_infrate
1.109	1	obs
0	0	dose:DOSE_infrate
7.0181	1	obs
0	0	dose:DOSE_infrate
0	0	dose:DOSE_infrate
0	0	dose:DOSE_infrate
2.2136	1	obs

30.3 Fixed/Random Effects Specification

In the *Nonmem* script, the theta variable definitions are:-

```
$THETA  (0,6.20684) ; 1. TVCL (lower bound,initial estimate)
$THETA  (0,26.5004) ; 2. TVV1  (lower bound,initial estimate)
$THETA  (0,2.15099) ; 3. TVQ
$THETA  (0,21.151) ; 4. TVV2
$THETA  (0,0.270697) ; 5. TVQ2
$THETA  (0,147.893) ; 6. TVV3
$THETA  55.4 FIX ; 7. T50
$THETA  3.33 FIX ; 8. Hill
$THETA  -0.129934 ; 9. power exponent on creatinine
$THETA  (0,1.70302) ; 10. PNA50
```

In the *PoPy* script the equivalent `f[X]` variables are:-

```
EFFECTS:
    POP: |
        f[TVCL] ~ P 6.20684
        f[TVV1] ~ P 26.5004
        f[TVQ] ~ P 2.15099
        f[TVV2] ~ P 21.151
        f[TVQ2] ~ P 0.270697
        f[TVV3] ~ P 147.893
        f[T50] = 55.4
        f[HILL] = 3.33
        f[CRPWR] = -0.129934
        f[P50] ~ P 1.70302
```

Notice that *PoPy* uses the natural equal sign for `f[X]` that are fixed and the '~' for `f[X]` that need to be estimated.

In the *Nonmem* script, the omega variable definitions are:-

```
$OMEGA  BLOCK(2)
 0.175278  ; variance for ETA(1), initial estimate
 0.115896␣
↪0.112362  ; COvariance ETA(1)-ETA(2), var for ETA(2), initial estimate
$OMEGA  0  FIX
$OMEGA  0.131759
$OMEGA  0  FIX
$OMEGA  0.177214
$OMEGA  BLOCK(1)
 0.0140684  ;  7. IOV_CL
$OMEGA  BLOCK(1) SAME
$OMEGA  BLOCK(1) SAME
$OMEGA  BLOCK(1) SAME
$OMEGA  BLOCK(1) SAME
$OMEGA  BLOCK(1) SAME
```

(continues on next page)

(continued from previous page)

```
$OMEGA  BLOCK(1) SAME
$OMEGA  BLOCK(1) SAME
$OMEGA  BLOCK(1) SAME
$OMEGA  BLOCK(1) SAME
$OMEGA  BLOCK(1) SAME
$OMEGA  BLOCK(1) SAME
$OMEGA  BLOCK(1) SAME
$OMEGA  BLOCK(1) SAME
$OMEGA  BLOCK(1) SAME
$OMEGA  BLOCK(1) SAME
$OMEGA  BLOCK(1) SAME
$OMEGA  BLOCK(1) SAME
$OMEGA  BLOCK(1) SAME
$OMEGA  BLOCK(1) SAME
$OMEGA  BLOCK(1) SAME
$OMEGA  BLOCK(1) SAME
```

The ugly code above defines a 2x2 matrix of omega *fixed effects*, followed by 4 scalar omega *fixed effects*. The '$OMEGA BLOCK(1)' + '$OMEGA BLOCK(1) SAME' lines defines a single **IOV** omega *fixed effects* to be estimated over up to 22 occasions. *Nonmem* creates an ETA normal variable for each individual corresponding to each $OMEGA above. Hence, each individual will have 4 ETAs (with 2 set to zero) and an additional 22 occasion ETAS.

You just have to do all the indexing in your head to use the ETAs in the subsequent PK block. In contrast the *PoPy* script defines the equivalent var `f[X]` variables as:-

```
EFFECTS:
    POP: |
        f[TCVL_isv, TVV1_isv] ~ spd_matrix() [
            [ 0.175278 ],
            [ 0.115896, 0.112362 ]
        ]
        f[TVQ_isv] = 0
        f[TVV2_isv] ~ P 0.131759
        f[TVQ2_isv] = 0
        f[TVV3_isv] ~ P 0.177214
        f[TVCL_iov] ~ P 0.0140684
```

Here the inter-subject variability (isv) variables are just more `f[X]` variables with convenient labels. Note in contrast to *Nonmem* the `f[TVCL_iov]` occasion variance is just defined once here (it being one *fixed effect*).

In *PoPy* the mechanism for creating the appropriate number of `r[X]` variables given the var `f[X]` is as shown in the 'ID' and 'OCC' sections:-

```
EFFECTS:
    ID: |
        r[TVCL_isv, TVV1] ~ mnorm([0,0], f[TCVL_isv, TVV1_isv])
        r[TVQ] ~ norm(0, f[TVQ_isv])
        r[TVV2] ~ norm(0, f[TVV2_isv])
```

(continues on next page)

(continued from previous page)

```
        r[TVQ2] ~ norm(0, f[TVQ2_isv])
        r[TVV3] ~ norm(0, f[TVV3_isv])
    OCC: |
        r[TVCL_iov] ~ norm(0, f[TVCL_iov])
```

This syntax means that each individual has one `r[TVCL_isv, TVV1]` and one `r[TVQ]` instance etc. However the number of instances of `r[TVCL_iov]` are dependent on the number of values of the 'OCC' field for each individual in the *data file*. *i.e.* Each individual will have a different sample of `r[TVCL_iov]` for each occasion. You do not need to know the highest occasion number here (unlike in *Nonmem*).

In the *Nonmem* script, the sigma variable definitions are:-

```
$SIGMA  0.036033  ; variance PROP res error, initial estimate
$SIGMA  0.0164023 ; additional res error, initial estimate
```

In the *PoPy* script the equivalent noise `f[X]` variables are:-

```
EFFECTS:
    POP: |
        f[PNOISE_var] ~ P 0.036033
        f[ANOISE_var] ~ P 0.0164023
```

30.4 PK Model Parameters Specification

The *Nonmem* 'PK' section is defined as follows:-

```
$PK
    ; Three-comp model
    IF(NEWIND.NE.2)OTIM1=0
    IF(NEWIND.NE.2)OCOV1=0
    IF(NEWIND.NE.2)OTIM2=0
    IF(NEWIND.NE.2)OCOV2=0
    ;
    STUDY=0
    IF(ID.LT.2000) STUDY=1                          ;Glasgow, Thomson1988
    IF(ID.GE.2000.AND.ID.LT.3000) STUDY=2           ;Uppsala, Nielsen2009
    IF(ID.GE.3000) STUDY=3                          ;Estonia, unpublished
    ;
    WTKG = WT/1000
    ;
    T50  = THETA(7)
    HILL = THETA(8)
    MF   = PMA**HILL/(PMA**HILL+T50**HILL)
    ;
    CREAT2 = CREAT
    IF(CREAT.
↪LT.0) CREAT2 = TCREA       ; when SCr is NA=-99, it is the typical SCr
```

(continues on next page)

(continued from previous page)

```
    ;OF = (CREAT2/TCREA)**(THETA(9))
    ;
    P50  = THETA(10)
    ;PNAF = PNA/(P50+PNA)
    ;
    CRPWR = THETA(9)
    ;IOV code
    BOVC = 0
    IF(OCC.EQ.1)  BOVC = ETA(7)
    IF(OCC.EQ.2)  BOVC = ETA(8)
    IF(OCC.EQ.3)  BOVC = ETA(9)
    IF(OCC.EQ.4)  BOVC = ETA(10)
    IF(OCC.EQ.5)  BOVC = ETA(11)
    IF(OCC.EQ.6)  BOVC = ETA(12)
    IF(OCC.EQ.7)  BOVC = ETA(13)
    IF(OCC.EQ.8)  BOVC = ETA(14)
    IF(OCC.EQ.9)  BOVC = ETA(15)
    IF(OCC.EQ.10) BOVC = ETA(16)
    IF(OCC.EQ.11) BOVC = ETA(17)
    IF(OCC.EQ.12) BOVC = ETA(18)
    IF(OCC.EQ.13) BOVC = ETA(19)
    IF(OCC.EQ.14) BOVC = ETA(20)
    IF(OCC.EQ.15) BOVC = ETA(21)
    IF(OCC.EQ.16) BOVC = ETA(22)
    IF(OCC.EQ.17) BOVC = ETA(23)
    IF(OCC.EQ.18) BOVC = ETA(24)
    IF(OCC.EQ.19) BOVC = ETA(25)
    IF(OCC.EQ.20) BOVC = ETA(26)
    IF(OCC.EQ.21) BOVC = ETA(27)
    IF(OCC.EQ.22) BOVC = ETA(28)
    ;
    TVCL = THETA(1)*MF*(WTKG/70)**(0.632)    ; typical value of CL
    TVV1 = THETA(2)*(WTKG/70)                        ; typical value of V1
    TVQ  = THETA(3)*(WTKG/
↪70)**(0.75)                ; ty. value of intercompartmental CL
    TVV2 = THETA(4)*(WTKG/70)                        ; ty. value of V2
    TVQ2 = THETA(5)*(WTKG/70)**(0.75)                ; ty value of CL3
    TVV3 = THETA(6)*(WTKG/70)                        ; ty value of V3
    ;
    CL   = TVCL*EXP(ETA(1)+BOVC)  ; individual value of CL
    V1   = TVV1*EXP(ETA(2))
    Q    = TVQ*EXP(ETA(3))
    V2   = TVV2*EXP(ETA(4))
    Q2   = TVQ2*EXP(ETA(5))
    V3   = TVV3*EXP(ETA(6))
    ;
    K    = CL/V1
    K12  = Q/V1
```

(continues on next page)

(continued from previous page)

```
    K21  = Q/V2
    K13  = Q2/V1
    K31  = Q2/V3
    ;
    IF(EVID.EQ.1) TM=TIME
    IF(EVID.EQ.1) TAD=0
    IF(EVID.NE.1) TAD=TIME-TM
    ;
    SL1 = 0
    IF(TIME.GT.OTIM1) SL1 = (PNA-OCOV1)/(TIME-OTIM1)
    A_0(4) = PNA
    ;
    SL2 = 0
    IF(TIME.GT.OTIM2) SL2 = (CREAT2-OCOV2)/(TIME-OTIM2)
    A_0(5) = CREAT2
```

The equivalent in *PoPy* is the *MODEL_PARAMS* section:-

```
MODEL_PARAMS: |

    WTKG = c[WT] / 1000
    MF = c[PMA]**f[HILL]/(c[PMA]**f[HILL] + f[T50]**f[HILL])
    TVCL = f[TVCL] * (WTKG/70)**(0.632) * MF
    TVV1 = f[TVV1] * (WTKG/70)
    TVQ  = f[TVQ]  * (WTKG/70)**(0.75)
    TVV2 = f[TVV2] * (WTKG/70)
    TVQ2 = f[TVQ2] * (WTKG/70)**(0.75)
    TVV3 = f[TVV3] * (WTKG/70)
    BOVC = r[TVCL_iov]
    CL = TVCL * exp(r[TVCL_isv] + BOVC)
    V1 = TVV1 * exp(r[TVV1])
    Q  = TVQ * exp(r[TVQ])
    V2 = TVV2 * exp(r[TVV2])
    Q2 = TVQ2 * exp(r[TVQ2])
    V3 = TVV3 * exp(r[TVV3])
    m[K]   = CL/V1
    m[K12] = Q/V1
    m[K21] = Q/V2
    m[K13] = Q2/V1
    m[K31] = Q2/V3
    m[CRPWR] = f[CRPWR]
    m[P50] = f[P50]
    m[SL1] = 0
    if c[CREAT] < 0:
        m[CREAT2] = c[TCREA]
    else:
        m[CREAT2] = c[CREAT]
    m[SL2] = 0
    m[V1] = V1
```

(continues on next page)

(continued from previous page)

```
    m[PNOISE_var] = f[PNOISE_var]
    m[ANOISE_var] = f[ANOISE_var]
```

The *PoPy* version is shorter, mainly because it does **not** have to write an if statement for all values of the OCC variable *i.e.* 'IF(OCC.EQ.1) BOVC = ETA(7)' etc.

30.5 Initial State Specification

Note that the *Nonmem* $PK section contains the lines:-

```
A_0(4) = PNA
A_0(5) = CREAT2
```

These are necessary to provide initial values for the $DES section. The equivalent in *PoPy* is the *STATES* section:-

```
STATES: |
    s[COVCMT1] = c[PNA]
    s[COVCMT2] = m[CREAT2]
```

which provides `s[X]` values at t=0.0 for the *PoPy* *DERIVATIVES* section. Note by default in both *PoPy* `s[X]` =0.0, unless the *STATES* section says otherwise. A similar convention is used in *Nonmem*.

30.6 ODEs Specification

The *Nonmem* $DES section is as follows:-

```
$DES
    TCOV1 = A(4)
    TCOV2 = A(5)
    PNAF = TCOV1/(P50+TCOV1)
    OF = (TCOV2/TCREA)**CRPWR

    DADT(1) = A(3)*K31+A(2)*K21-A(1)*(K*PNAF*OF+K12+K13)
    DADT(2) = A(1)*K12-A(2)*K21
    DADT(3) = A(1)*K13-A(3)*K31
    DADT(4)= SL1
    DADT(5)= SL2
```

This is very similar to the *PoPy* *DERIVATIVES* section:-

```
DERIVATIVES: |
    TCOV1 = s[COVCMT1]
    TCOV2 = s[COVCMT2]
    PNAF = TCOV1 / (m[P50]+TCOV1)
```

(continues on next page)

(continued from previous page)

```
    OF = (TCOV2/c[TCREA])**m[CRPWR]
    dose[DOSE_infrate] = @inf_rate{ amt: c[AMT], rate: c[RATE], lag: 0 }
    d[CENTRAL] = dose[DOSE_infrate] + m[K31]*s[PERIPH2]␣
↪+ m[K21]*s[PERIPH1] - (m[K]*PNAF*OF+m[K12]+m[K13])*s[CENTRAL]
    d[PERIPH1] = - m[K21]*s[PERIPH1] + m[K12] *s[CENTRAL]
    d[PERIPH2] = - m[K31]*s[PERIPH2] + m[K13] *s[CENTRAL]
    d[COVCMT1] =   m[SL1]
    d[COVCMT2] =   m[SL2]
```

except that in *PoPy* the `dose[DOSE_infrate]` is defined and explicitly placed in the `s[CENTRAL]` compartment. *Nonmem* puts all boluses/infusions in compartment one by convention, unless a CMT field is defined in the data set (it isn't in this case). In *Nonmem* you have to examine the data file to determine the type of dose, however in the *PoPy* syntax above it's pretty clear it's an infusion defined by the `c[AMT]` and `c[RATE]` data entries in each dosing data row.

The *ordinary differential equation* solver parameters for *Nonmem* are set here:-

```
$SUBROUTINE ADVAN6 TOL=6
```

Here 'ADVAN6' is the *Nonmem* ode solver for non-stiff systems. With a relative tolerance of 1e-6. Note this solver is different from the CPPLSODA method employed by *PoPy* which is similar to selecting 'ADVAN13' in *Nonmem*. In this case the *ordinary differential equation* solver method makes little difference, as both *PoPy* and *Nonmem* return similar objective values for this model.

In *PoPy* the *ordinary differential equation* solver parameters are defined as follows:-

```
ODE_SOLVER:
    CPPLSODA:
        # Absolute tolerance of ode solver.
        atol: 1e-12

        # Relative tolerance of ode solver.
        rtol: 1e-12

        # Maximum number of steps allowed in ode solver.
        max_nsteps: 10000000
```

30.7 Likelihood Error Specification

The $ERROR block in the *Nonmem* script defines a proportional + additional noise model:-

```
$ERROR
    IPRED  = A(1)/V1
    Y      = IPRED*(1+EPS(1)) + EPS(2)
```

The *PREDICTIONS* section in the *PoPy* script defines the same error model:-

```
PREDICTIONS: |
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
    var = p[DV_CENTRAL]**2 * m[PNOISE_var] + m[ANOISE_var]
    c[DV_CENTRAL] ~ norm(p[DV_CENTRAL], var)
```

30.8 Parameter Fitting Methods Specification

The *Nonmem* script defines the FOCE fitting method (METHOD=1) here:-

```
$ESTIMATION METHOD=1 INTER MAXEVAL=0 PRINT=1
```

The equivalent in the *PoPy* script is here:-

```
FIT_METHODS: [ND: {max_n_main_iterations: 0}]
```

Note both *Nonmem* and *PoPy* only optimize the `r[X]` in the fitting run as the number of main iterations is set to zero.

Part VIII

PoPy Reference Guide

CHAPTER

THIRTYONE

POPY QUICK REFERENCE GUIDE

See Table 1:-

Table 1: *PoPy* Quick Reference

Commands	*Scripts*	*Sections*
• *popy run* • *popy create* • *popy check* • *popy format* • *popy view* • *popy info* • *popy doc* • *popy validate* • *popy activate* • *popy deactivate*	• *fit* • *gen* • *sim* • *tut* • *comp* • *mfit* • *mgen* • *msim* • *mtut* • *mcomp* • *grph* • *vpc* • *fitsum* • *gensum* • *tutsum*	• *METHOD_OPTIONS* • *DESCRIPTION* • *FILE_PATHS* • *DATA_FIELDS* • *PREPROCESS* • *EFFECTS* • *MODEL_PARAMS* • *STATES* • *DERIVATIVES* • *PREDICTIONS* • *ODE_SOLVER* • *FIT_METHODS* • *COVARIANCE* • *OUTPUT_SCRIPTS* • *OUTPUT_OPTIONS*
Probability Distributions	*Dosing Functions*	*Analytic Solutions*
• *~unif(min,max)init* • *~norm(m,v)* • *~mnorm(m_vec,v_mat)* • *~bernoulli(p)* • *~poisson(p)* • *~negbinomial(p,r)*	• *@bolus* • *@inf_rate* • *@inf_dur* • *@gamma* • *@weibull*	• *@iv_one_cmp_cl* • *@dep_one_cmp_cl* • *@iv_two_cmp_cl* • *@dep_two_cmp_cl* • *@iv_three_cmp_cl* • *@dep_three_cmp_cl*
Variables	• `c[X]`, `f[X]`, `r[X]`, `m[X]`, `w[X]`, `x[NEWIND]` • `d[X]`, `s[X]`, `x[TIME]`, `p[X]`	
TYPE values	• obs, dose, pred, reset, reset+dose	

CHAPTER

THIRTYTWO

COMMAND LINE TOOLS

The *PoPy* tools and their functions are summarised in Table 1:-

Table 1: *PoPy* Command Line Tools

Name	Example	Function
popy run	popy run my_script.pyml	run a *PoPy* script
popy create	popy create fit my_script.pyml	create an example *PoPy* script
popy check	popy check my_script.pyml	check a *PoPy* script
popy format	popy format my_script.pyml	update *PoPy* script format
popy view	popy view my_script.pyml.html	open *PoPy* html output in browser
popy info	popy info	display *PoPy* install details
popy doc	popy doc	open *PoPy* documentation in browser
popy validate	popy validate	run *PoPy Validation*
popy activate	popy activate XXXX	activation_activate using a *product key*
popy deactivate	popy deactivate	*Deactivating a PoPy Licence* a *product key*

32.1 popy run

The most important command in the suite of *Command Line Tools* is *popy run*. It processes *PoPy* scripts.

32.1.1 Running a script

To run a *PoPy* script *PoPy Terminal* and type:-

```
popy run my_script.pyml
```

What happens with the script depends on its *type*. There are many script formats:-

- *fit* - Fit a model to data
- *gen* - Generate data from a model
- *sim* - Simulate PK/PD curves
- *tut* - *gen* data and *fit* and compare the results
- *comp* - Compare *gen* and *fit* results

- *mfit* - Fit a model to multiple data sets
- *mgen* - Generate multiple data from a model
- *msim* - Simulate multiple PK/PD curves
- *mtut* - *mgen* multiple data and *mfit* and compare
- *mcomp* - Compare *mgen* and *mfit* results
- *grph* - Plot graphs
- *vpc* - Plot *VPCs*
- *fitsum* - *HTML* summary of *fit* results
- *gensum* - *HTML* summary of *gen* results
- *tutsum* - *HTML* summary of *tut* results

See *Script File Formats* for more information.

Note a common switch to use with *popy run* is:-

```
popy run -o my_script.pyml
```

Here the '-o' option overwrites previous output automatically. If the script has already been run and you do not use the '-o' option, you will be asked to confirm you want to overwrite any previous output.

For other command line options see *Command line options*.

32.1.2 Running multiple scripts

Note you can also run all scripts in a directory using:-

```
popy run *.pyml
```

This runs all files with the '.pyml' extension in serial.

32.1.3 Log files created by popy_run

When *popy run* processes a script file, it creates a log file, `my_script.pyml.run.main.log` as a record of the output. If a runtime error occurs during processing, a stack trace is sent to `my_script.pyml.run.error.log` as a record of what went wrong.

32.1.4 Command line options

```
usage:␣
↪popy_run [-h] [-t] [-v {noset,info,debug,warning,error,critical}] [-a]
                [-c] [-l] [-s] [-o] [--no_check] [-r] [-w N]
                [--log_folder LOG_FOLDER]
                script_path

Runs a PoPy script.

positional arguments:
  script_path           Required path to configuration file.

options:
  -h, --help            show this help message and exit
  -t,
↪ --timestamp       Optionally included timestamp string in log file name
                        and output folder name.
  -v {noset,info,debug,warning,
↪error,critical}, --verbosity {noset,info,debug,warning,error,critical}
                        verbosity of output in log files
  -a,
↪ --all_config      Optionally output all possible config file entries in
                        output script files. If set to false, the default
                        entries with default values are suppressed for the
                        sake of brevity.
  -c, --comment_scripts
    ␣
↪                   Optionally add explanatory comments to all entries in
                        output script files.
  -l, --line_breaks     Optionally enforce line breaks in the config file.
  ␣
↪                     This increases the length of files, but may improve
   ␣
↪                    clarity. If set to False, short dictionary lines are
                        compacted instead using {} notation.
  -
↪s, --spaces          Optionally add more spaces to the output config file
   ␣
↪                    for greater clarity, but longer config files. Off by
                        default.
  -o, --overwrite       Optionally overwrite existing output files without
                        asking.
  --no_check            Optionally do NOT run the PoPy script checking i.e
  ␣
↪                     silence the warnings and errors output at the start
     ␣
↪                   this option is not recommended, but you can use if you
                        do not believe the warnings/errors and want to run
```

(continues on next page)

(continued from previous page)

```
                        your script regardless.
  -r, --replicate_scripts
    ␣
↪                    Suppress replicating input config files in log files.
  -w N, --worker_index N
     ␣
↪                   Optionally provide a worker index, so get separate log
                        files for each worker
  --log_folder LOG_FOLDER
                        Optional path to worker log folder.
```

32.2 popy create

To help you get started with a new script, we provide a tool called *popy create* that generates an example of one of the *Script File Formats*. You do this by opening a *PoPy Terminal* and typing:-

```
popy create [script_type] my_script.pyml
```

where `[script_type]` is one of the script names (`fit`, `gen`, `tut`, and so on.). See *Script File Formats* for a list of all script types.

Typically, the new script can then be edited using your text editor of choice to suit current requirements or immediately run using *popy run*. See *Creating Your First PoPy Scripts* for more information.

32.2.1 Command line options

```
usage: popy_create [-h] [-t] [-v {noset,info,debug,warning,error,critical}]
                   [-a] [-c] [-l] [-s] [-o]
                   {tut,gen,fit,comp,sim,
↪tutsum,gensum,fitsum,mtut,mgen,mfit,mcomp,msim,grph,vpc,table,val,merge}
                   script_path

Creates an example PoPy script.

positional arguments:
  {tut,gen,fit,comp,sim,
↪tutsum,gensum,fitsum,mtut,mgen,mfit,mcomp,msim,grph,vpc,table,val,merge}
                        type of script to create
  script_path           Required path to configuration file.

options:
  -h, --help            show this help message and exit
  -t,
↪ --timestamp       Optionally included timestamp string in log file name
                        and output folder name.
```

(continues on next page)

(continued from previous page)

```
  -v {noset,info,debug,warning,
↪error,critical}, --verbosity {noset,info,debug,warning,error,critical}
                        verbosity of output in log files
  -a,
↪ --all_config      Optionally output all possible config file entries in
                        output script files. If set to false, the default
                        entries with default values are suppressed for the
                        sake of brevity.
  -c, --comment_scripts
    ␣
↪                   Optionally add explanatory comments to all entries in
                        output script files.
  -l, --line_breaks     Optionally enforce line breaks in the config file.
  ␣
↪                     This increases the length of files, but may improve
   ␣
↪                    clarity. If set to False, short dictionary lines are
                        compacted instead using {} notation.
  -
↪s, --spaces          Optionally add more spaces to the output config file
   ␣
↪                    for greater clarity, but longer config files. Off by
                        default.
  -o, --overwrite       Optionally overwrite existing output files without
                        asking.
```

32.3 popy check

When developing a *PoPy* script file it is sometimes useful to sanity check the input and automatically detect common errors.

Note this checking process also happens when using *popy run*, however the *popy check* enables you to check before you are ready to run a script.

32.3.1 Checking a script

Check a script by opening a *PoPy Terminal* and typing:-

```
$ popy check my_script.pyml
```

Note a common switch to use with *popy check* is:-

```
$ popy check -o my_script.pyml
```

Here the '-o' option overwrites previous output automatically. If the script has already been run and you do not use the '-o' option, you will be asked to confirm you want to overwrite any previous output.

For other command line options see *Command line options*.

32.3.2 Checking multiple scripts

Note you can also check all scripts in a directory using:-

```
popy check *.pyml
```

This checks all files with the '.pyml' extension in serial.

32.3.3 Log files created by popy_check

When *popy check* processes a script file, it creates a log file, `my_script.pyml.check.log` that should enable you to more easily edit and fix a broken script file.

32.3.4 Command line options

```
usage: popy_check [-h] [-t] [-v {noset,info,debug,warning,error,critical}]
                  script_path

Checks PoPy script for consistency and returns errors + warnings. Can use
optionally prior to running script.

positional arguments:
  script_path           Required path to configuration file.

options:
  -h, --help            show this help message and exit
  -t,
↪ --timestamp       Optionally included timestamp string in log file name
                        and output folder name.
  -v {noset,info,debug,warning,
↪error,critical}, --verbosity {noset,info,debug,warning,error,critical}
                        verbosity of output in log files
```

32.4 popy format

Occasionally, we make changes to the *Script File Formats* that render old script files defunct. So that you do not need to rewrite all of your old script files, we provide a tool called *popy format* that reads in the old-style script file and outputs it with the new format.

32.4.1 Format a script

Do this by opening a *PoPy Terminal* and typing:-

```
popy format my_script.pyml
```

which will output a new version of `my_script.pyml` called `my_script.pyml.new`.

You can also edit the script 'inplace' using:-

```
popy format -i my_script.pyml
```

This creates a backup file called 'my_script.pyml.old' and updates the format of 'my_script.pyml' directly. See *Command line options* for other options.

The output of *popy format* may need to be manually edited to obtain desired results (*e.g.* to change from the default value for newly added options).

As usual, a log file will be created to document the formatting process.

32.4.2 Format multiple scripts

Note you can also format all scripts in a directory using:-

```
popy format *.pyml
```

This formats all files with the '.pyml' extension in serial.

32.4.3 Command line options

```
usage: popy_format [-h] [-t] [-v {noset,info,debug,warning,error,critical}]
                   [-a] [-c] [-l] [-s] [-o] [--inplace] [-d]
                   script_path

Tries to reformat an old version of a PoPy script.

positional arguments:
  script_path           Required path to configuration file.

options:
  -h, --help            show this help message and exit
  -t,
↪ --timestamp       Optionally included timestamp string in log file name
                        and output folder name.
  -v {noset,info,debug,warning,
↪error,critical}, --verbosity {noset,info,debug,warning,error,critical}
                        verbosity of output in log files
  -a,
↪ --all_config      Optionally output all possible config file entries in
```

(continues on next page)

(continued from previous page)

```
                        output script files. If set to false, the default
                        entries with default values are suppressed for the
                        sake of brevity.
  -c, --comment_scripts
    ␣
↪                    Optionally add explanatory comments to all entries in
                        output script files.
  -l, --line_breaks     Optionally enforce line breaks in the config file.
  ␣
↪                     This increases the length of files, but may improve
    ␣
↪                    clarity. If set to False, short dictionary lines are
                        compacted instead using {} notation.
  -
↪s, --spaces          Optionally add more spaces to the output config file
    ␣
↪                    for greater clarity, but longer config files. Off by
                        default.
  -o, --overwrite       Optionally overwrite existing output files without
                        asking.
  --
↪inplace             Optionally overwrite original file, i.e fix in place.
  -d,
↪ --delete           Optionally delete extra files, e.g. log + backup file
                        etc.
```

32.5 popy view

Opens a html file in a browser from the command line.

32.5.1 Open a html file

For convenient access to *PoPy*'s html output, *PoPy Terminal* and type:-

```
popy view my_tut.pyml.html
```

This opens a html page in your browser.

It is a fast way of viewing local .html files output by *PoPy*.

32.5.2 Open multiple html files

Note you can also open all html files in a directory using:-

```
popy view *.html
```

This opens all files with the '.html' extension in your default browser.

32.5.3 Command line options

```
usage: popy_view [-h] url

Opens local PoPy html summary output in web browser.

positional arguments:
  url         URL path, you can use *.html.

options:
  -h, --help  show this help message and exit
```

32.6 popy info

A handy tool, *popy info*, displays information about the installation so that you can check at a glance that the installation worked.

PoPy Terminal and type:-

```
popy info
```

and you should see something like this:-

```
INFO - In a PoPy Binary environment
INFO - popy_flavour=binary
INFO - popy_python_path=C:\PoPy\
INFO - popy_release=<X.Y.Z>
INFO - popy_version=academic
INFO - python_version=3.8.6
INFO - windows_version=('10', '10.0.17134', 'SP0', 'Multiprocessor Free')
INFO - machine_name=mickey
INFO - product_key=<XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX>
status=Product key is activated and within licence period
Licence has been running for 3613 days
Licence needs renewing in 1868 days
INFO - name=Phil Tresadern
INFO - email=phil@popypkpd.com
INFO - company=Wright Dose Ltd
INFO - licence_start_date=2009-02-12 00:00:00
```

(continues on next page)

(continued from previous page)

```
INFO - licence_end_date=2024-02-16 00:00:00
INFO - should_run=True
```

The exact details will depend on where you installed *PoPy* and the activation status of *PoPy* on your machine.

32.7 popy doc

For convenient access to *PoPy*'s documentation, *PoPy Terminal* and type:-

```
popy doc
```

This opens the *Population PK/PD with PoPy HTML* documentation page.

32.7.1 Examples

You can access the online documentation as follows (default is locally installed .html files):-

```
popy doc -o
```

Alternatively you can open documentation for a specific tool (e.g., *popy run*) with the **-t** or **–tool** option:-

```
popy doc --tool run
```

You can also open documentation for specific *Script File Formats* (e.g., a *Fit Script*) with the **-s** or **–script** option:-

```
popy doc --script fit
```

32.7.2 Command line options

```
usage: popy_doc [-h] [--online]
                [--script {none,tut,gen,fit,comp,sim,
↪tutsum,gensum,fitsum,mtut,mgen,mfit,mcomp,msim,grph,vpc,table,val,merge}]
                [--tool␣
↪{none,activate,deactivate,info,create,edit,run,check,format,doc,env}]

Opens PoPy html documentation in web browser.

options:
  -h, --help            show this help message and exit
  --online              Optionally access online documentation instead of
                        local.
  --script {none,tut,gen,fit,comp,sim,
↪tutsum,gensum,fitsum,mtut,mgen,mfit,mcomp,msim,grph,vpc,table,val,merge}
```

(continues on next page)

(continued from previous page)

```
  ␣
↪                       Optionally open documentation for a specific script
                         type.
  --
↪tool {none,activate,deactivate,info,create,edit,run,check,format,doc,env}
      ␣
↪                     Optionally open documentation for a specific PoPy tool
```

32.8 popy validate

A tool to *validate popy*. To run *popy validate*, *PoPy Terminal* and type:-

```
popy validate
```

See *PoPy Validation* for more information.

32.9 popy activate

A tool to activation_activate. To run *popy activate*, *PoPy Terminal* and type:-

```
popy activate [product key]
```

where `[product key]` is the *product key* supplied by Wright Dose Ltd.

See activation_activate for more information.

32.9.1 Command line options

```
usage: popy_activate [-h] [product_key]

Activates a PoPy install using a product key.

positional arguments:
  product_key  Your product key, or the path to a file (relative to
 ␣
↪              POPY_PYTHON_PATH) containing only your product key (default:
               licence-key.txt)

options:
  -h, --help   show this help message and exit
```

32.10 popy deactivate

A tool to *Deactivating a PoPy Licence*. To run *popy deactivate*, *PoPy Terminal* and type:-

```
popy deactivate
```

See *Deactivating a PoPy Licence* for more information.

CHAPTER

THIRTYTHREE

SCRIPT FILE FORMATS

The *PoPy* script formats are shown in Table 1:-

Table 1: Script Format

Name	Purpose	Child Scripts
fit	Fit a model to data	*grph*/*sim*/*msim*/*fitsum*
gen	Generate data from a model	*grph*/*sim*/*gensum*
sim	Simulate PK/PD curves	*grph*
tut	Gen data and fit + compare	*grph*/*fit*/*comp*/*tutsum*
comp	Compare gen and fit results	none
mfit	Fit a model to multiple data sets	none
mgen	Generate multiple data from a model	none
msim	Simulate multiple PK/PD curves	*vpc*
mtut	Gen multiple data and fit + compare	*mgen*/*mfit*/*mcomp*
mcomp	Compare mgen and mfit results	none
grph	Plot graphs	none
vpc	Plot *VPCs*	none
fitsum	*HTML* summary of *fit* results	none
gensum	*HTML* summary of *gen* results	none
tutsum	*HTML* summary of *tut* results	none

A *PoPy* script file is a text file that defines how *PoPy* works with a PK/PD model. Each configuration file typically has a '.pyml' file extension. The file suffix .pyml stands for *PoPy YAML* format.

To run a *PoPy* command on a particular script, *PoPy Terminal* in the directory containing the script file and type:-

```
popy XXXX YYYY.pyml
```

Where XXXX is one of the *Command Line Tools*, such as `run`, `create`, `check` or `format`. And YYYY.pyml is the file name of the script itself. For example:-

```
popy run my_fit_file.pyml
```

for running a script to fit PK/PD parameters to a pre-existing data set, or:-

```
popy create gen my_gen_file.pyml
```

for creating a new script file that will generate synthetic PK/PD data.

PoPy will first parse the script to check it is in the correct format. If so, *PoPy* will then process the script.

A *PoPy* script file can be one of multiple formats, as specified at the start of each script, in *METHOD_OPTIONS*. For example a *Fit Script* starts with:-

```
METHOD_OPTIONS: {py_module: fit}
```

And a *Gen Script* starts with:-

```
METHOD_OPTIONS: {py_module: gen}
```

33.1 Fit Script

The fit script is probably the script you will use most often, as it performs the most common form of PK/PD analysis - finding parameter estimates for a PK/PD model given a set of *observations*.

See simple_fit_example or builtin_fit_example for walk through examples.

33.1.1 Main Sections of a Fit Script

- *METHOD_OPTIONS*
- *PARALLEL*
- *DESCRIPTION*
- *FILE_PATHS*
- *DATA_FIELDS*
- *PREPROCESS*
- *EFFECTS*
- *MODEL_PARAMS*
- *STATES*
- *DERIVATIVES*
- *PREDICTIONS*
- *ODE_SOLVER*
- *FIT_METHODS*
- *COVARIANCE*
- *OUTPUT_SCRIPTS*

33.2 Gen Script

Generates example synthetic *observations* from a PopPK/PD model.

Note that running *PoPy* on a *Gen Script* creates similar outputs to running it on a *Sim Script*. However, a *Gen Script* also generates the time point and dose data to simulate the entire data file **not** just the observations.

33.2.1 Main Sections of a Gen Script

- *METHOD_OPTIONS*
- *PARALLEL*
- *DESCRIPTION*
- *FILE_PATHS*
- *DATA_FIELDS*
- *EFFECTS*
- *PREPROCESS*
- *MODEL_PARAMS*
- *STATES*
- *DERIVATIVES*
- *PREDICTIONS*
- *POSTPROCESS*
- *ODE_SOLVER*
- *OUTPUT_SCRIPTS*

33.3 Sim Script

Running *PoPy* on a *Sim Script* generates dense time point simulations of PK/PD curves given a PopPK/PD model and an input data set.

33.3.1 Main Sections of a Sim Script

- *METHOD_OPTIONS*
- *PARALLEL*
- *DESCRIPTION*
- *FILE_PATHS*
- *DATA_FIELDS*

- *PREPROCESS*
- *STATES*
- *DERIVATIVES*
- *PREDICTIONS*
- *ODE_SOLVER*
- *OUTPUT_OPTIONS*
- *OUTPUT_SCRIPTS*

33.4 Tut Script

This script performs data generation using a *Gen Script*, then fits the model using a *Fit Script* and compares the fitting parameters to the true parameters using a *Comp Script*.

See simple_tut_example and *Running Your First tut Script* for walk through examples.

33.4.1 Main Sections of a Tut Script

- *METHOD_OPTIONS*
- *PARALLEL*
- *DESCRIPTION*
- *FILE_PATHS*
- *DATA_FIELDS*
- *GEN_EFFECTS*
- *FIT_EFFECTS*
- *PREPROCESS*
- *MODEL_PARAMS*
- *STATES*
- *DERIVATIVES*
- *PREDICTIONS*
- *POSTPROCESS*
- *ODE_SOLVER*
- *FIT_METHODS*
- *COVARIANCE*
- *OUTPUT_SCRIPTS*

33.5 Comp Script

A comp script compares the outputs of a *Gen Script* and a *Fit Script*. Usually a *Comp Script* is created automatically using a *Tut Script*.

33.6 MFit Script

Fits a PK/PD model to multiple data sets. A multi population version of *Fit Script*.

A *MFit Script* can be created by a *MTut Script* in a similar fashion to how a *Fit Script* can be created by a *Tut Script*.

33.6.1 Main Sections of a MFit Script

- *METHOD_OPTIONS*
- *PARALLEL*
- *DESCRIPTION*
- *FILE_PATHS*
- *DATA_FIELDS*
- *PREPROCESS*
- *EFFECTS*
- *MODEL_PARAMS*
- *STATES*
- *DERIVATIVES*
- *PREDICTIONS*
- *ODE_SOLVER*
- *FIT_METHODS*
- *COVARIANCE*

33.7 MGen Script

Simulates multiple datasets from a PopPK/PD model. A multi population version of *Gen Script*.

Note that running *PoPy* on a *MGen Script* creates similar outputs to running it on a *MSim Script*. However, a *MGen Script* also generates new time points and dose data for each data set.

A *MGen Script* can be created by a *MTut Script* in a similar fashion to how a *Gen Script* can be created by a *Tut Script*.

33.7.1 Main Sections of a MGen Script

- *METHOD_OPTIONS*
- *PARALLEL*
- *DESCRIPTION*
- *FILE_PATHS*
- *DATA_FIELDS*
- *EFFECTS*
- *PREPROCESS*
- *MODEL_PARAMS*
- *STATES*
- *DERIVATIVES*
- *PREDICTIONS*
- *POSTPROCESS*
- *ODE_SOLVER*
- *OUTPUT_OPTIONS*

33.8 MSim Script

Generates multiple simulations of PK/PD curves from given a PopPK/PD model **and** an input data set.

An *MSim Script* often outputs a *Vpc Script*, which generates a *VPC* plot from the multiple population simulations.

See builtin_msim_example for a working example.

33.8.1 Main Sections of a MSim Script

- *METHOD_OPTIONS*
- *PARALLEL*
- *DESCRIPTION*
- *FILE_PATHS*
- *DATA_FIELDS*
- *PREPROCESS*
- *EFFECTS*
- *MODEL_PARAMS*
- *STATES*

- *DERIVATIVES*
- *PREDICTIONS*
- *ODE_SOLVER*
- *OUTPUT_OPTIONS*
- *OUTPUT_SCRIPTS*

33.9 MTut Script

Generates multiple synthetic populations sets from a PK/PD model, then fits the same model to each population. A multi population version of *Tut Script*.

Generates a *MGen Script* and a *MFit Script*. Also creates a *MComp Script* to compare the true parameters from the synthetic populations with the fitted parameters.

See *Generate multiple data sets and Fit using Simple PopPK Model* and builtin_mtut_example for walk through examples.

33.9.1 Main Sections of a MTut Script

- *METHOD_OPTIONS*
- *PARALLEL*
- *DESCRIPTION*
- *FILE_PATHS*
- *DATA_FIELDS*
- *GEN_EFFECTS*
- *FIT_EFFECTS*
- *PREPROCESS*
- *MODEL_PARAMS*
- *STATES*
- *DERIVATIVES*
- *PREDICTIONS*
- *ODE_SOLVER*
- *FIT_METHODS*
- *COVARIANCE*
- *OUTPUT_OPTIONS*
- *OUTPUT_SCRIPTS*

33.10 MComp Script

A *MComp Script* compares the outputs of a *MGen Script* with a *MFit Script*. Usually a *MComp Script* is created automatically using a *MTut Script*.

A *MComp Script* is a multiple population version of a *Comp Script*.

33.11 Grph Script

Outputs basic plots given input data sets and fitting results.

33.12 Vpc Script

Creates *VPC* plots. The visual predictive check is designed to plot a population of PK/PD curves on one graph.

A *Vpc Script* is usually created as a child script of a *MSim Script*.

See simple_msim_example1 and builtin_msim_example for working examples.

33.13 FitSum Script

A *FitSum Script* generates a *HTML* report on the results of a *Fit Script*.

33.14 GenSum Script

A *GenSum Script* generates a *HTML* report on the results of a *Gen Script*.

33.15 TutSum Script

A *TutSum Script* generates a *HTML* report on the results of a *Tut Script*.

CHAPTER

THIRTYFOUR

SCRIPT FILE SECTIONS

A *PoPy* script file is hierarchical and based on *YAML*.

The file is indented with the main sections defined at the start of a line. Subsections are indented within the main sections.

In Table 1 we list the main sections that are common to many *Script File Formats*.

Table 1: Common Script Sections

Section	Purpose
METHOD_OPTIONS	Script type and run options
PARALLEL	Speed up processing by running in parallel
DESCRIPTION	Title and summary of model
FILE_PATHS	Paths to load and output data
DATA_FIELDS	Define type/id/time fields for data
PREPROCESS	Filter or expand the input data
EFFECTS	Define population `f[X]` and `r[X]` variable structure
MODEL_PARAMS	Define `m[X]` for each data row, given `f[X]`, `r[X]` and `c[X]`
STATES	Define initial value `s[X]`, given `m[X]` and `c[X]`
DERIVATIVES	Define `d[X]` wrt time, given `m[X]`, `c[X]` and `s[X]`
PREDICTIONS	Define `p[X]` for each data row, given `m[X]`, `c[X]` and `s[X]`
POSTPROCESS	Filter or expand the generated data
ODE_SOLVER	Define *ordinary differential equation* method and parameters
FIT_METHODS	Define fit methods and parameters
COVARIANCE	Define a method for computing *standard errors*
OUTPUT_SCRIPTS	Declare child scripts for post processing
OUTPUT_OPTIONS	Miscellaneous output parameters

34.1 METHOD_OPTIONS

This details the methods to be used in the script and is a required section.

34.1.1 Example METHOD_OPTIONS from a Fit Script

```
METHOD_OPTIONS:
    # Python module required to process this script file
    py_module: fit

    # Option to set seed to make run result
    # reproducible -e.g. when debugging.
    # rand_seed: 12345
    rand_seed: auto

    # Format string for numerical output
    float_format: default
```

34.1.2 Main METHOD_OPTIONS Fields

py_module

When calling this script using *popy run*, *PoPy* needs to know which *type of script* is being called. This is determined by the 'py_module' field.

In the example above a 'fit' script is specified. Other possible entries are 'fit', 'tut', 'gen', 'mtut' etc. See *Script File Formats* for more information.

rand_seed

There are two types of scripts run by *PoPy*:-

- Deterministic - given the same inputs the same outputs are **always** returned
- Stochastic - given the same inputs the result are dependent on a random number generator

For example a *Gen Script* is inherently stochastic. Whereas a *Fit Script* using the *JOE*, *FOCE* or *ND* fitting method is inherently deterministic.

When running a stochastic method you have two options, this setting:-

```
rand_seed: auto
```

Will ensure that the random number generator is seeded with a different random number, every time the script is run. This will generate different output each time. Alternatively you set the seed explicitly:-

```
rand_seed: 314159
```

This will initialise the pseudorandom number generator with the value '314159', which will then generate the same output every time, given the same inputs. For more information on seeding and pseudorandom number generators, see Wikipedia page:-

http://en.wikipedia.org/wiki/Random_seed

Note: if your script is deterministic then the 'rand_seed' setting will have no effect.

float_format

This field allows you to control how numbers output by *PoPy* are rendered as strings. If you leave this field out it defaults to:-

```
float_format: default
```

This has the effect of outputting float values to 4 decimal places. Table 2 shows both named float formats.

Table 2: float format options

Entry	Format String	Example Input	Example Output
default	.4F	1.0123456	'1.0123'
2 decimal places in exponent format	.2E	1.0123456	'1.01E+00'

You can also specify your own custom format, *e.g.* '.3G' for 3 significant figures in general format or '.6F' for 6 decimal places in float point format or '.4E' for 4 decimal places in exponent format. Examples of different format strings on the *Python* command prompt are:-

```
>>> '{:.4E}'.format(1.0123456)
'1.0123E+00'
>>> '{:.6F}'.format(1.0123456)
'1.012346'
>>> '{:.3G}'.format(1.0123456)
'1.01'
```

You can experiment with your own formatting. See the rather esoteric instructions here:-

https://docs.python.org/3.4/library/string.html#format-specification-mini-language

Setting the output to six decimal places in *PoPy* could be achieved by:-

```
float_format: .6F
```

Note that *PoPy* adds the prefix '{:' and the suffix '}' to your 'float_format' config file entry.

34.2 PARALLEL

An optionally section to run *PoPy* in parallel to increase the speed of the program by separating tasks amongst the different nodes of a computer.

You can add the *PARALLEL* section to the following scripts:-

- *Tut Script*
- *Gen Script*
- *Fit Script*
- *Sim Script*
- *Comp Script*
- *MTut Script*

- *MGen Script*
- *MFit Script*
- *MSim Script*
- *MComp Script*
- *Grph Script*
- *Vpc Script*

i.e. Any script that does a significant amount of processing over all individuals in the population.

If you leave out the parallel section then the script will be executed in serial, *i.e.* using one processor.

34.2.1 Example PARALLEL section

You can make your *PoPy* script use parallel processing by adding the following:-

```
PARALLEL:
    MPI_WORKERS:
        n_workers: 4
```

This section invokes the 'MPI_WORKERS' parallel method, that uses *mpi* to process individuals in parallel where possible.

The 'n_workers' parameter requests 4 separate processors. You can also use:-

```
PARALLEL:
    MPI_WORKERS:
        n_workers: auto
```

To utilise all processors on a given machine.

Note if you leave out the the 'PARALLEL' section from your script, this is equivalent to using the following:-

```
PARALLEL: {SINGLE: {}}
```

That requests using a single processor.

34.3 DESCRIPTION

This optional section is an opportunity to provide notes about the model. The script is given a (short) **name** and a (longer) **title**. The **author** of the model can be named and the **abstract** is used to describe the model. You can define **keywords** to aid future searches.

34.3.1 Example DESCRIPTION

```
DESCRIPTION:
    # Unique name used to distinguish script
    name: builtin_fit_example

    # A longer text string that could serve as a title
    title: First order absorption model with peripheral compartment

    # Author of the model
    author: J.R. Hartley

    # Abstract paragraph describing model
    abstract: |
        A two compartment PK model with bolus dose and
        first order absorption.

    # Keywords list used to categorise models.
    keywords: ['fitting', 'pk', 'advan4', 'dep_two_cmp', 'first order']
```

34.3.2 Main DESCRIPTION Fields

name

The optional **name** entry is used as an identifier for the script. It is also used to form the name of **child** scripts. For example when specifying the name:-

If you do **not** specify a name the field defaults to:-

```
name: none
```

the **child** *Sim Script* will be named 'none_sim.pyml' and the child *FitSum Script* will be named 'none_fitsum.pyml' *etc.*. For this reason it's a good idea to keep the **name** short and only use alpha numeric characters and underscores.

For more info see *Files Generated by Fit Script*.

title

The **title** field is optional, but otherwise appears in the summary output and on graphical plots.

author

The **author** field is optional but allows you to store the name of the person who created the script.

abstract

The **abstract** field is optional, but allows a longer description of the purpose of the script.

This field is *verbatim*, so it is suffixed with a '|' symbol. See:-

```
abstract: |
    A two compartment PK model with bolus dose and
    first order absorption.
```

The abstract is included in the summary output. It can contain any text you like. Note it is indented relative to the 'abstract' field name, like all *verbatim* sections.

keywords

The **keywords** field is optional and contains a list of strings, for example:-

```
keywords: ['fitting', 'pk', 'dep_two_cmp', 'first order','advan4']
```

Here the *Python* notation for a *list* is used, *i.e.* an opening square bracket '[' followed by comma separated strings and a closing square bracket ']'.

Note: it is intended that the keywords will eventually form part of a publicly available index of *PoPy* script files.

34.4 FILE_PATHS

This required field, defines where any input files can be found and where the results of the computation are to be stored.

34.4.1 Example from FILE_PATHS from a Fit Script

```
FILE_PATHS:
    # path to input comma separated value file in popy data format
    input_data_file: builtin_fit_example_data.csv

    # Output folder - results of computation stored here
    output_folder: auto

    # Temp folder - temporary files stored here
    temp_folder: auto
```

(continues on next page)

(continued from previous page)

```
    # Output file extension - determines graphical output file format.
    output_file_ext: ['svg', 'pdf']

    # Solution containing f[X] values from a previous run.
    input_solution_file: none
```

34.4.2 Main FILE_PATH Fields

input_file

Required path to input a .csv file in popy data format:-

```
input_data_file: builtin_fit_example_data.csv
```

Note that this field is required and the .csv file **must** exist else *PoPy* will throw an error message.

output_folder

Optional field that specifies the output folder, where the results of computation are stored.

The default is:-

```
output_folder: auto
```

Using **auto** will specify an output folder based on the script name. E.g if your script is called 'my_fit_script.pyml', the output folder will be `my_fit_script.pyml.output`

Using **auto** is safest, as then it is impossible to over-write the output from other scripts.

temp_folder

Optional field that specifies the temp folder, where temporary functions generated by *PoPy* are stored.

The default is:-

```
temp_folder: auto
```

The default folder name is '_temp' within the *output_folder*. If you have a script named 'my_fit_script.pyml' and miss out the 'output_folder' and 'temp_folder' you will end up with temporary functions in this folder:-

```
my_fit_script.pyml.output
    /fit
        /_temp
```

Note the subfolders within '_temp' have random names, this is to avoid re-using old temporary functions if the same script is run twice (and altered slightly).

output_file_ext

This is an optional list of file types to output when plotting. The default is:-

```
output_file_ext: ['svg']
```

This outputs plots in .svg format, which stands for scalable vector graphics, see:-

https://en.wikipedia.org/wiki/Scalable_Vector_Graphics

.svg is a good format because it can be displayed at any resolution without being pixelated. For example if you used .png or any other raster format.

The input is a *Python list*. This is to enable you to output plots in multiple formats. For example many of the documentation examples use the following:-

```
output_file_ext: ['svg', 'pdf']
```

To generate plots in .svg and also .pdf format. We do this with the documentation so that the *HTML* documentation can display .svg and the pdf version can use .pdf plots.

input_solution_file

This field is only available in the *Fit Script*. It is an optional link to a 'solution.pyml' file to initialise the `f[X]` starting values during a fit. The default is:-

```
input_solution_file: none
```

This does nothing and uses the `f[X]` starting values specified in the *EFFECTS* section of the script.

If a solution file is specified, *e.g.*:-

```
input_solution_file: ./previous_fit_script.pyml_output/solN/solution.pyml
```

Then the 'solution.pyml' file will be loaded, specifically the 'fx_params.csv' part of the solution.

```
FILE_PATHS:
    fx_params_path: fx_params.csv
```

The `f[X]` params are used as new starting values.

You can use the 'input_solution_file' to restart a *Fit Script* from a previous fitting script. This is useful if the previous script fails, for example if the machine is accidentally switched off or some other system failure.

34.5 DATA_FIELDS

An optional section where the names of the *type_field*, *id_field* and *time_field* can be redefined.

34.5.1 Example Data Fields

```
DATA_FIELDS:
  ␣
↪ # Field name in data file that contains row type info, e.g. obs/dose etc
    type_field: TYPE

    # Field name in data␣
↪file that contains identity string for each data row e.g. obs/dose etc
    id_field: ID

    # Field name in data file that contains time or event for each data row
    time_field: TIME
```

34.5.2 Main Fields

type_field

By default this field is:-

```
type_field: TYPE
```

This means the 'TYPE' field in *data file* specifies the type of row *e.g.* 'dose', 'obs', 'reset' etc.

id_field

By default this field is:-

```
id_field: ID
```

This means the 'ID' field in the *data file* specifies the identity of each subject in the population.

time_field

By default this field is:-

```
time_field: TIME
```

This means the 'TIME' field in *data file* specifies the time stamp of each observation, dose etc.

34.6 PREPROCESS

An optional *verbatim* section that creates extra `c[X]` variables after loading in a *input data file* and can also remove some rows from the data. A kind of flexible filter implemented in *Python*.

The *PREPROCESS* is available in the following scripts:-

- *Fit Script*
- *Sim Script*
- *MFit Script*
- *MSim Script*

i.e. where there is a *data file* loaded by the script.

34.6.1 Example PREPROCESS section

```
PREPROCESS: |
    # exclude negative concentrations
    if c[CONC] < 0.0: return
    # create new OCCASION variable
    if c[DAY] <= 3:
        c[OCCASION] = 1
    elif 3 < c[DAY] <= 6:
        c[OCCASION] = 2
    elif 6 < c[DAY] <= 8:
        c[OCCASION] = 3
    else:
        c[OCCASION] = 4
```

The example above shows the two operations a *PREPROCESS* section can perform, namely:-

- Exclude data rows
- Create extra `c[X]` data columns

The line:-

```
if c[CONC] < 0.0: return
```

Removes all rows from the data set with CONC less than zero. The null return is a *PoPy* convention for ignoring a particular row.

The other rows create a new `c[OCCASION]` variable, as follows:-

```
if c[DAY] <= 3:
    c[OCCASION] = 1
elif 3 < c[DAY] <= 6:
    c[OCCASION] = 2
elif 6 < c[DAY] <= 8:
    c[OCCASION] = 3
```

(continues on next page)

(continued from previous page)

```
else:
    c[OCCASION] = 4
```

The simple *Python* assignment to `c[OCCASION]` creates the 'OCCASION' field. The **`if`/`elif`/`else`** statements are standard *Python* syntax and partition the data rows into occasions according to the existing `c[DAY]` data field.

Note that the remaining sections of the script file, *e.g.* *EFFECTS*, *DERIVATIVES* etc are able to use the new `c[OCCASION]` variable as though it already existed in the data file.

The use of *Python* syntax here means the above can be expanded in arbitrary complex ways to add more `c[X]` variables or exclude other rows from the data set.

Note a common usage of the *PREPROCESS* section is to remove an individual from the analysis as follows:-

```
PREPROCESS: |
    # exclude an individual
    if c[ID] == '7': return
```

Or alternatively keep just one individual:-

```
PREPROCESS: |
    # exclude all individuals apart from 7
    if c[ID] != '7': return
```

Or potentially exclude multiple individuals:-

```
PREPROCESS: |
    # exclude multiple individuals
    if c[ID] in ['7','9','41']: return
```

Or retain only a few individuals:-

```
PREPROCESS: |
    # exlude all individuals apart from 1-3
    if c[ID] not in ['1','2','3']: return
```

Note here the 'ID' field is a *Python string* **not** a *float* or *integer*.

If you want to exclude individuals based on a numerical calculation, you can do this:-

```
PREPROCESS: |
    # exlude all individuals apart from 1-3
    if int(c[ID]) > 3: return
```

The code above assumes that all `c[ID]` values can be converted to an integer. This will **not** be the case if one of your individuals has the identifier '3A' for example.

34.6.2 Rules for PREPROCESS section

Like all *verbatim* sections the *PREPROCESS* section of the config file accepts free form pseudo *Python* code, but there are some rules regarding which variables are allowed in a *PREPROCESS* section as follows:-

- **Only** `c[X]` variables and local *Python* variables are allowed
- `c[X]` on the **right hand side** and within `if` statements must be previously defined on the **left hand side** or in the *data file*
- `c[X]` declared on the **left hand side** must **not** already exist in the *data file*
- return must always be **null**

So you can **not** use `m[X]`, `f[X]`, `r[X]`, `d[X]` etc variables in this section.

The *PREPROCESS* function is run once, shortly after loading in the *data file*, so it is efficient to create required `c[X]` variables in this section, as opposed to creating temporary variables in the *MODEL_PARAMS* or *DERIVATIVES* sections.

Like all *verbatim* sections it is possible to introduce syntax errors by writing malformed *Python*. This will hopefully be picked up when *PoPy* attempts to compile or run the *PREPROCESS* function as a temporary .py file.

34.7 EFFECTS

A required *verbatim* section that defines *fixed effects* and *random effects* for use in mixed effects models. The *EFFECTS* section defines a level structure, which dictates the number of instances of the `f[X]` and `r[X]` effect variables.

For example, `f[X]` variables representing *fixed effects* are usually declared at the **POP** level, so there is only one value of each `f[X]`. Whereas `r[X]` variables representing *random effects* are usually declared at the **ID** level, so each individual has a sample from each *random effect*. It is also possible to define further sub levels below the **ID** level, for example within individual occasions which have there own `r[X]` variables, see *Inter-Occasion Variability (IOV)*.

The *EFFECTS* section is required by the following scripts:-

- *Tut Script*
- *Gen Script*
- *Fit Script*
- *MTut Script*
- *MGen Script*
- *MFit Script*
- *MSim Script*

i.e. Any script that defines a mixed effect model. Note that a *Tut Script* or *MTut Script* has two separate *EFFECTS* sections named *GEN_EFFECTS* and *FIT_EFFECTS*.

34.7.1 EFFECTS with two levels from a fit_script

The example below is used in builtin_fit_example.

```
EFFECTS:
    POP: |
        f[KA] ~ P1.0
        f[CL] ~ P1.0
        f[V1] ~ P20
        f[Q] ~ P0.5
        f[V2] ~ P100
        f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv] ~ spd_matrix() [
            [0.05],
            [0.01, 0.05],
            [0.01, 0.01, 0.05],
            [0.01, 0.01, 0.01, 0.05],
            [0.01, 0.01, 0.01, 0.01, 0.05],
        ]
        f[PNOISE] ~ P0.1
    ID: |
        r[KA,␣
↪CL, V1, Q, V2] ~ mnorm([0,0,0,0,0], f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv])
```

The example above defines two levels:-

- **POP** - single value of each `f[X]` variable over whole population
- **ID** - one value per individual for each `r[X]` variable

This example defines 5 mean *fixed effect* parameters *i.e.* `f[KA]`, `f[CL]`, `f[V1]`, `f[Q]`, `f[V2]`, a 5x5 covariance matrix `f[KA_isv, CL_isv, V1_isv, Q_isv, V2_isv]`, a proportional noise variable `f[PNOISE]` and a 5 element vector `r[KA, CL, V1, Q, V2]` of *random effects* defined for each individual.

Every variable declared at the **POP** level has **one** shared value over the whole population. The **ID** level creates a single instance of each `r[X]` distribution for each 'ID' field present in the *data file*. This is similar to the **factor** concept in *R*. The structure of the mixed effects is a tree, see Fig. 1.

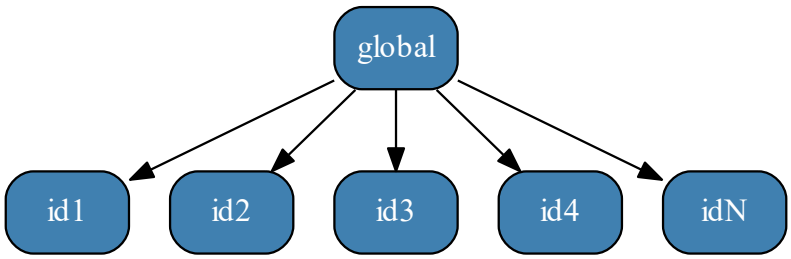


Fig. 1: *EFFECTS* structure with **two** levels

Here the number of `r[X]` values is dependent on the number of individuals in the *data file*.

34.7.2 EFFECTS with three levels from a fit_script

It is possible to add a further level as follows:-

```
EFFECTS:
    POP: |
        f[KA] ~ P1.0
        f[CL] ~ P1.0
        f[V1] ~ P20
        f[Q] ~ P0.5
        f[V2] ~ P100
        f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv] ~ spd_matrix() [
            [0.05],
            [0.01, 0.05],
            [0.01, 0.01, 0.05],
            [0.01, 0.01, 0.01, 0.05],
            [0.01, 0.01, 0.01, 0.01, 0.05],
        ]
        f[KA_iov,CL_iov,V1_iov,Q_iov,V2_iov] ~ spd_matrix() [
            [0.05],
            [0.01, 0.05],
            [0.01, 0.01, 0.05],
            [0.01, 0.01, 0.01, 0.05],
            [0.01, 0.01, 0.01, 0.01, 0.05],
        ]
        f[PNOISE] ~ P0.1
    ID: |
        r[KA, CL, V1, Q, V2] ~ mnorm(
            [0,0,0,0,0],
            f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv]
        )
    IOV: |
        r[KA_iov, CL_iov, V1_iov, Q_iov, V2_iov] ~ mnorm(
            [0,0,0,0,0],
            f[KA_iov, CL_iov, V2_iov, Q_iov, V3_iov]
        )
```

The example above defines three levels:-

- **POP** - single value of each `f[X]` variable
- **ID** - one value per individual for each `r[X]` variable
- **IOV** - one value per iov per individual for each `r[X]` variable

The **IOV** level creates an extra level of `r[KA_iov, CL_iov, V1_iov, Q_iov, V2_iov]` variables. If there are say 2 different values of `c[IOV]` in the *data file* (*i.e.* two occasions) then each individual has a 5 element vector `r[KA, CL, V1, Q, V2]` at the **ID** level and additionally two 5 element vectors `r[KA_iov, CL_iov, V1_iov, Q_iov, V2_iov]` (one for each occasion) at the **IOV** level.

The structure of the mixed effects is now as show in Fig. 2.

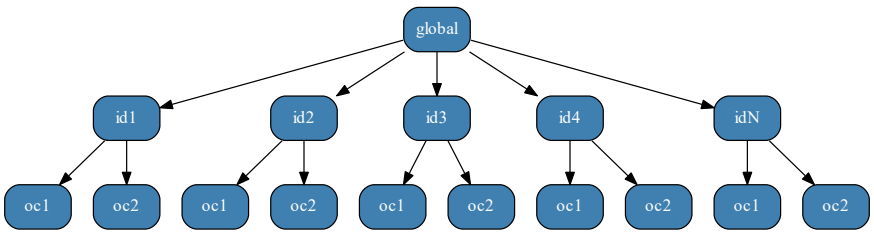


Fig. 2: *EFFECTS* structure with **three** levels

For more information on this topic see *Inter-Occasion Variability (IOV)*.

34.7.3 EFFECTS with two levels from a gen_script

The example below is used in builtin_gen_example.

```
EFFECTS:
    POP: |
        c[AMT] = 100.0
        f[KA] = 0.2
        f[CL] = 2.0
        f[V1] = 50
        f[Q] = 1.0
        f[V2] = 80
        f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv] = [
            [0.1],
            [0.01, 0.03],
            [0.01, -0.01, 0.09],
            [0.01, 0.02, 0.01, 0.07],
            [0.01, 0.02, 0.01, 0.01, 0.05],
        ]
        f[PNOISE] = 0.15
    ID: |
        c[ID] = sequential(50)
        t[DOSE] = 2.0
        t[OBS] ~ unif(1.0, 50.0; 5)
        # t[OBS] = range(1.0, 50.0; 5)
        r[KA,␣
↪CL, V1, Q, V2] ~ mnorm([0,0,0,0,0], f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv])
```

The example above defines two levels and is similar to the *EFFECTS with two levels from a fit_script* section. The *Fit Script* defines `f[X]` and `r[X]` variables. Additionally this *Gen Script* version defines `c[X]` variables and additionally `t[DOSE]` and `t[OBS]` variables that define the dosing and *observation* rows of the generated *data file*.

In the **POP** section:-

```
POP: |
    c[AMT] = 100.0
```

This syntax creates a `c[AMT]` field in the *data file* which is constant over **all** rows. In the **ID** section:-

```
ID: |
    c[ID] = sequential(50)
```

This syntax creates a `c[ID]` field in the *data file* which has values of [1,50]. *i.e.* 50 individuals. Also in the **ID** sections these `t[X]` variables are declared:-

```
ID: |
    t[DOSE] = 2.0
    t[OBS] ~ unif(1.0, 50.0; 5)
```

The `t[DOSE]` creates a dose row at time 2.0 for all individuals. The `t[OBS]` line creates 5 *observation* rows at random time points in the range [1,50.0].

34.7.4 EFFECTS with three levels from a gen_script

It is possible to add a further level to the *Gen Script* EFFECTS as follows:-

```
EFFECTS:
    POP: |
        c[AMT] = 100.0
        f[KA] = 0.2
        f[CL] = 2.0
        f[V1] = 50
        f[Q] = 1.0
        f[V2] = 80
        f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv] = [
            [0.1],
            [0.01, 0.03],
            [0.01, -0.01, 0.09],
            [0.01, 0.02, 0.01, 0.07],
            [0.01, 0.02, 0.01, 0.01, 0.05],
        ]
        f[KA_iov,CL_iov,V1_iov,Q_iov,V2_iov] = [
            [0.1],
            [0.01, 0.03],
            [0.01, -0.01, 0.09],
            [0.01, 0.02, 0.01, 0.07],
            [0.01, 0.02, 0.01, 0.01, 0.05],
        ]
        f[PNOISE] = 0.15


    ID: |
        c[ID] = sequential(50)
```

(continues on next page)

(continued from previous page)

```
        r[KA,␣
↪CL, V1, Q, V2] ~ mnorm([0,0,0,0,0], f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv])

    IOV: |
        c[IOV] = sequential(2)
        t[DOSE] = 2.0
        t[OBS] ~ unif(1.0, 50.0; 5)
        # t[OBS] = range(1.0, 50.0; 5)
        r[KA_iov, CL_iov, V1_iov, Q_iov, V2_iov] ~ mnorm(
            [0,0,0,0,0],
            f[KA_iov, CL_iov, V2_iov, Q_iov, V3_iov]
        )
```

The obvious difference between *EFFECTS with two levels from a gen_script* and the three level version above is the addition of the third section:-

```
IOV: |
    c[IOV] = sequential(2)
    t[DOSE] = 2.0
    t[OBS] ~ unif(1.0, 50.0; 5)
    # t[OBS] = range(1.0, 50.0; 5)
    r[KA_iov, CL_iov, V1_iov, Q_iov, V2_iov] ~ mnorm(
        [0,0,0,0,0],
        f[KA_iov, CL_iov, V2_iov, Q_iov, V3_iov]
    )
```

This denotes an **IOV** level with two occasions for each individual. Note that this line creates two occasions:-

```
IOV: |
    c[IOV] = sequential(2)
```

i.e. rows with `c[IOV]` taking the values [1,2] are created. Within each occasion the `t[DOSE]` and `t[OBS]` create a dosing row and 5 observation rows. Note it is necessary to move the `t[X]` variables from the **ID** level to the **IOV** level. In a *Gen Script* it usually makes sense to place the `t[X]` variables at the lowest level.

34.7.5 Combining EFFECTS in a tut_script

A *Tut Script* combines a *Gen Script* and a *Fit Script*, so has to encode both a generating *EFFECTS* section and a fitting *EFFECTS* section.

GEN_EFFECTS

The generating EFFECTS section in a *Tut Script* is called GEN_EFFECTS. This section is transcribed into the EFFECTS section in the **child** *Gen Script*.

The GEN_EFFECTS can contain `f[X]`, `r[X]`, `t[X]` and `c[X]` variable definitions in each level.

FIT_EFFECTS

The fitting EFFECTS section in a *Tut Script* is called FIT_EFFECTS. This section is transcribed into the EFFECTS section in the **child** *Fit Script*.

The FIT_EFFECTS section can contain `f[X]` and `r[X]` variable definitions in each level.

34.7.6 EFFECTS with two levels from a tut_script

The examples below are used in *Running Your First tut Script*.

```
GEN_EFFECTS:
    POP: |
        c[AMT] = 100.0
        f[KA] = 0.2
        f[CL] = 2.0
        f[V1] = 50
        f[Q] = 1.0
        f[V2] = 80
        f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv] = [
            [0.1],
            [0.01, 0.03],
            [0.01, -0.01, 0.09],
            [0.01, 0.02, 0.01, 0.07],
            [0.01, 0.02, 0.01, 0.01, 0.05],
        ]
        f[PNOISE] = 0.15
    ID: |
        c[ID] = sequential(50)
        t[DOSE] = 2.0
        t[OBS] ~ unif(1.0, 50.0; 5)
        # t[OBS] = range(1.0, 50.0; 5)
        r[KA,␣
↪CL, V1, Q, V2] ~ mnorm([0,0,0,0,0], f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv])
```

```
FIT_EFFECTS:
    POP: |
        f[KA] ~ P1.0
        f[CL] ~ P1.0
        f[V1] ~ P20
        f[Q] ~ P0.5
        f[V2] ~ P100
        f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv] ~ spd_matrix() [
            [0.05],
            [0.01, 0.05],
            [0.01, 0.01, 0.05],
            [0.01, 0.01, 0.01, 0.05],
            [0.01, 0.01, 0.01, 0.01, 0.05],
        ]
        f[PNOISE] ~ P0.1
    ID: |
        r[KA,␣
↪CL, V1, Q, V2] ~ mnorm([0,0,0,0,0], f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv])
```

The *Tut Script* combines the *Gen Script* and *Fit Script* levels.

34.7.7 EFFECTS with three levels from a tut_script

It is possible to add the third level to a *Tut Script* as follows:-

```
GEN_EFFECTS:
    POP: |
        c[AMT] = 100.0
        f[KA] = 0.2
        f[CL] = 2.0
        f[V1] = 50
        f[Q] = 1.0
        f[V2] = 80
        f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv] = [
            [0.1],
            [0.01, 0.03],
            [0.01, -0.01, 0.09],
            [0.01, 0.02, 0.01, 0.07],
            [0.01, 0.02, 0.01, 0.01, 0.05],
        ]
        f[KA_iov,CL_iov,V1_iov,Q_iov,V2_iov] = [
            [0.1],
            [0.01, 0.03],
            [0.01, -0.01, 0.09],
            [0.01, 0.02, 0.01, 0.07],
            [0.01, 0.02, 0.01, 0.01, 0.05],
        ]
        f[PNOISE] = 0.15
```

(continues on next page)

(continued from previous page)

```
    ID: |
        c[ID] = sequential(50)
        r[KA,␣
↪CL, V1, Q, V2] ~ mnorm([0,0,0,0,0], f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv])
    IOV: |
        c[IOV] = sequential(2)
        t[DOSE] = 2.0
        t[OBS] ~ unif(1.0, 50.0; 5)
        # t[OBS] = range(1.0, 50.0; 5)
        r[KA_iov, CL_iov, V1_iov, Q_iov, V2_iov] ~ mnorm(
            [0,0,0,0,0],
            f[KA_iov, CL_iov, V2_iov, Q_iov, V3_iov]
        )

FIT_EFFECTS:
    POP: |
        f[KA] ~ P1.0
        f[CL] ~ P1.0
        f[V1] ~ P20
        f[Q] ~ P0.5
        f[V2] ~ P100
        f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv] ~ spd_matrix() [
            [0.05],
            [0.01, 0.05],
            [0.01, 0.01, 0.05],
            [0.01, 0.01, 0.01, 0.05],
            [0.01, 0.01, 0.01, 0.01, 0.05],
        ]
        f[KA_iov,CL_iov,V1_iov,Q_iov,V2_iov] ~ spd_matrix() [
            [0.05],
            [0.01, 0.05],
            [0.01, 0.01, 0.05],
            [0.01, 0.01, 0.01, 0.05],
            [0.01, 0.01, 0.01, 0.01, 0.05],
        ]
        f[PNOISE] ~ P0.1
    ID: |
        r[KA,␣
↪CL, V1, Q, V2] ~ mnorm([0,0,0,0,0], f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv])
    IOV: |
        r[KA_iov, CL_iov, V1_iov, Q_iov, V2_iov] ~ mnorm(
            [0,0,0,0,0],
            f[KA_iov, CL_iov, V2_iov, Q_iov, V3_iov]
        )
```

This is similar to the *EFFECTS with three levels from a fit_script* and *EFFECTS with three levels from a gen_script* examples. The *Tut Script* copies GEN_EFFECTS into *Gen Script* EFFECTS and FIT_EFFECTS into *Fit Script* EFFECTS.

34.7.8 Rules for each EFFECTS level

Each individual level of the *EFFECTS* is a *verbatim* section, as follows:-

```
EFFECTS:
    <level_name>: |
```

However the level section is limited in what expressions it can accept. For example it is **not** pseudo *Python* code, unlike the *PREPROCESS*, *MODEL_PARAMS* or *DERIVATIVES* sections, which act more like *Python* functions.

Generally only declarative expressions of the type:-

```
x[VAR] = <some definition>
```

Or

```
x[VAR] ~ <some definition>
```

Are allowed. You cannot use `if` statements for example. And all effect levels are declarative and unordered. *i.e.* the order of the statements in a single effects level makes no difference to the mixed effect model.

The variables you are allowed to declare on the **left hand side** depends on which type of script you are running:-

- In a *Fit Script* you are allowed to declare `f[X]` and `r[X]` only.
- In a *Gen Script* you are allowed to declare `f[X]`, `r[X]`, `c[X]` and `t[X]`
- All variables must have a unique name, *i.e.* no duplicate `c[X]` or `f[X]` or `r[X]`
- If a variable is on the **right hand side** of an expression it **must** be defined in a level **above** the current level

You can **not** use `m[X]`, `d[X]`, `p[X]` etc variables in *EFFECTS*.

Like all *verbatim* sections it is possible to introduce syntax errors by writing malformed code in *EFFECTS*. Any errors will be brought to the users attention when *PoPy* attempts to interpret the verbatim sections and form a tree structure to manage the *fixed effect* and *random effect* variables.

34.8 MODEL_PARAMS

A required *verbatim* section that creates `m[X]` variables for each row of the data set. Taking the previously defined `f[X]`, `r[X]` and `c[X]` as input.

The *MODEL_PARAMS* section is used in the following scripts:-

- *Tut Script*
- *Gen Script*
- *Fit Script*
- *MTut Script*

- *MGen Script*
- *MFit Script*
- *MSim Script*

i.e. Any script that defines a mixed effect model.

34.8.1 Example MODEL_PARAMS section

The example below is used in builtin_fit_example.

```
MODEL_PARAMS: |
    m[KA] = f[KA] * exp(r[KA])
    m[CL] = f[CL] * exp(r[CL])
    m[V1] = f[V1] * exp(r[V1])
    m[Q] = f[Q] * exp(r[Q])
    m[V2] = f[V2] * exp(r[V2])
    m[ANOISE] = 0.001
    m[PNOISE] = f[PNOISE]
```

In the example above the main parameters are defined with the following structure:-

```
MODEL_PARAMS: |
    m[X] = f[X] * exp(r[X])
```

This form ensures that the `m[X]` have a log normal distribution and are therefore constrained to be positive, assuming the `f[X]` input is positive. An alternative formulation is:-

```
MODEL_PARAMS: |
    m[X] = f[X] + r[X]
```

Which means the `m[X]` values have a *~norm()* distribution. However if the variance of the `r[X]` is large (relative to `f[X]`) this can result in `m[X]` having negative values sometimes. Often negative values are **not** physically sensible in PK/PD models (*e.g.* a negative *clearance*).

Parameters can also be defined here in terms of `c[X]` values from the *data file*. For example, if an individual's weight has an effect on the volume of distribution, it could be defined as:-

```
MODEL_PARAMS: |
    m[V] = (f[V] + c[WT]*f[WT_EFF]) * exp(r[V])
```

Here the `f[WT_EFF]` is an extra parameter to be estimated by *PoPy* that needs to be defined in the *EFFECTS* section.

Another common pattern in *MODEL_PARAMS* is the incorporation of **IOV** `r[X]` variables. This can be done using declarations like:-

```
MODEL_PARAMS: |
    m[X] = f[X] * exp(r[X] + r[X_iov])
```

Where in *EFFECTS*:-

- `f[X]` is typically defined in the first **POP** level
- `r[X]` is typically defined in the second *ID* level
- `r[X_iov]` is typically defined in the third **IOV** level

This is a common pattern in *PoPy*. It allows the simple combination of `r[X]` from different levels, without using cumbersome and error prone `if` statements. Although you can use `if` statements if you wish. For example a more complex *covariate* example:-

```
if c[GENDER] == "male":
    m[Y] = f[Y_male] * exp(r[Y])
else:
    m[Y] = f[Y_female] * exp(r[Y])
```

Here the `c[GENDER]` field from the *data file* is used as a switch to decide whether to use `f[Y_male]` or `f[Y_female]` for each row as appropriate.

34.8.2 Static Workspace Variables in MODEL_PARAMS

The *MODEL_PARAMS* function operates **independently** on each row of the *data file* and relies on the *EFFECTS* to arrange the `f[X]` and `r[X]` correctly in each data row (which *PoPy* handles for you).

However, there may be times when you want a variable's value to persist from one row to the next. In the *MODEL_PARAMS* you can explicitly define **static** workspace variables using the notation `w[X]`. For example:-

```
if c[TIME] < 0.0:
    w[PERSISTENT] = 1.0
else:
    w[PERSISTENT] *= 1.1
```

This fairly artificial example keeps updating the variable `w[PERSISTENT]` between rows. Here the *Python* notation '*=' multiplies the variable `w[PERSISTENT]` by itself and the number 1.1. Note by default all variables in *PoPy* are local, so a naive attempt to do this say:-

```
if c[TIME] < 0.0:
    PERSISTENT = 1.0
else:
    PERSISTENT *= 1.1
```

Would fail with a *Python* error, because in the line 'PERSISTENT *= 1.1', the local variable 'PERSISTENT' has **not** been previously defined, so can **not** multiply itself by 1.1.

The `w[X]` notation provides a means of remembering variable values between rows. Note having all variables as static by default is a bad idea as it makes it easier to introduce hard to find coding errors.

34.8.3 Rules for MODEL_PARAMS section

Like all *verbatim* sections the *MODEL_PARAMS* section of the config file accepts free form *Python pseudocode*, but there are some rules regarding which variables are allowed in a *MODEL_PARAMS* section as follows:-

- **Only** new `m[X]` variables and local *Python* variables can be defined on the **left hand side**
- `f[X]`, `r[X]`, `c[X]` and `m[X]` are allowed on the **right hand side** of expressions
- `c[X]` on the **right hand side** and within `if` statements must be in the *data file* or defined in the *PREPROCESS* section, in a *Fit Script*
- `c[X]` on the **right hand side** must be defined within the *EFFECTS* in a *Gen Script* or *Tut Script*.
- newly declared `m[X]` on the **left hand side** must have unique names

So you can **not** use `d[X]`, `p[X]`, `s[X]` or `t[X]` etc variables in this section.

Like all *verbatim* sections it is possible to introduce syntax errors by writing malformed *Python*. Any coding errors will be reported by *PoPy* when attempting to compile or run the *MODEL_PARAMS* function as a temporary .py file.

34.9 STATES

An optional *verbatim* section that defines initial `s[X]` variables for the *DERIVATIVES* block. The *ordinary differential equation* model in *PoPy* is typically a *initial value problem*. The *STATES* section defines the **initial values**.

Note the *STATES* section is optional. If it is not provided, then by default all `s[X]` variables are initialised to zero. In PK problems this is often sufficient, but a *STATES* section is often required for PD models.

The *STATES* section takes as input the previously defined `m[X]` and `c[X]` variables and computes the initial `s[X]` variables. The *STATES* function is run on the first row of each individual and for every **reset** row in the *data file*. The *STATES* section is available in the following scripts:-

- *Tut Script*
- *Gen Script*
- *Fit Script*
- *Sim Script*
- *MTut Script*
- *MGen Script*
- *MFit Script*
- *MSim Script*

i.e. Any script that contains a *DERIVATIVES ordinary differential equation* model.

34.9.1 Example STATES for PK Model

In builtin_fit_example, a null states section is used:-

```
STATES: |
```

This is equivalent to just removing the *STATES* section altogether. It is also equivalent to writing this:-

```
STATES: |
    s[DEPOT] = 0.0
    s[CENTRAL] = 0.0
    s[PERI] = 0.0
```

The form of the *STATES* section is very dependent on the *DERIVATIVES* section that it is initialising. The `s[X]` variables that are defined in *STATES* **must** exist in the *DERIVATIVES* section.

```
DERIVATIVES: |
    # s[DEPOT,CENTRAL,PERI] = @dep_two_cmp_cl{dose:@bolus{amt:c[AMT]}}
    d[DEPOT] = @bolus{amt:c[AMT]} - m[KA]*s[DEPOT]
    d[CENTRAL] = m[KA]*s[DEPOT]␣
↪- s[CENTRAL]*m[CL]/m[V1] - s[CENTRAL]*m[Q]/m[V1]  + s[PERI]*m[Q]/m[V2]
    d[PERI] = s[CENTRAL]*m[Q]/m[V1] - s[PERI]*m[Q]/m[V2]
```

Setting all amounts to zero in the PK compartments is fairly common. As before any dose is administered no drug is expected to be present in the body.

The steady state is therefore zero. The amounts in each compartment only become positive after a dose is administered. As the drug is excreted from the body the amount of drug in each compartment converges back to zero.

Note a **reset** row in the *data file* short cuts the wash out process by removing all of the drug from the body, by calling the null *STATES* function above.

34.9.2 Example STATES for PD Model

See *Direct PD Model* for example *Tut Script*, using the following *STATES* and *DERIVATIVES* sections for a PD model:-

```
STATES: |
    s[CENTRAL] = 0.0
    s[MARKER] = m[BASE]
```

```
DERIVATIVES: |
    d[CENTRAL] = @bolus{amt:c[AMT]} - s[CENTRAL]*c[CL]/c[V]
    d[MARKER] = m[BASE]*m[KOUT] - (1+s[CENTRAL]/c[V])*m[KOUT]*s[MARKER]
```

In this example the `s[MARKER]` initial value is set to `m[BASE]`. This is in order to make sure that the 'MARKER' PD compartment is at equilibrium, when the 'CENTRAL' PK compartment is zero.

You can usually deduce the equilibrium conditions for a PD compartment by setting the `d[X]` variables to zero and solving for `s[X]`, with the PK compartments set to zero.

For example in this case, setting `d[CENTRAL]` to zero:-

```
0.0 = -s[CENTRAL]*c[CL]/c[V]
```

implies:-

s[CENTRAL] = 0.0

Then setting `d[MARKER]` to zero:-

```
0.0 = m[BASE]*m[KOUT] - (1+0.0)*m[KOUT]*s[MARKER]
```

implies:-

```
s[MARKER] = m[BASE]
```

These **equilibrium** conditions ensures that the amounts in 'CENTRAL' and 'MARKER' are fixed until the system is disrupted by a drug dose being administered. As the drug is washed out of the system, the system should smoothly return back to equilibrium.

Alternatively if a **reset** row is encountered in the *data file* then the system is jolted back to the equilibrium by running the *STATES* function.

The non-zero equilibrium value of `s[MARKER]` is meant to model the endogenous amount of a substance within the body. *i.e.* a substance that is naturally present without drug intervention. This biological fact makes PD models inherently more complex than PK models and they usually require a more complex *STATES* section.

34.9.3 Rules for STATES section

Like all *verbatim* sections the *STATES* section of the config file accepts free form *Python pseudocode*, but there are some rules regarding which variables are allowed in a *STATES* section as follows:-

- **Only** `s[X]` and local variables can be defined on the **left hand side**
- `s[X]` variables defined on the **left hand side must** also exist in the *DERIVATIVES* section
- `s[X]` variables present in *DERIVATIVES* but **not** in *STATES* are initialised to zero.
- `c[X]` and `m[X]` are only allowed on the **right hand side** of expressions
- `c[X]` must be defined in the *data file* or created in the *PREPROCESS* section in a *Fit Script*
- `c[X]` must be defined within the *EFFECTS* in a *Gen Script* or the *GEN_EFFECTS* in a *Tut Script*.
- `m[X]` must be defined in the *MODEL_PARAMS*

You can **not** use `d[X]`, `p[X]`, `r[X]`, `f[X]` or `t[X]` *etc.* variables at all in this section.

Like all *verbatim* sections it is possible to introduce syntax errors by writing malformed *Python*. Coding errors will be reported when *PoPy* attempts to compile or run the *STATES* function as a temporary .py file.

34.10 DERIVATIVES

An optional *verbatim* section that defines an *ordinary differential equation* model in a *PoPy* script.

The *DERIVATIVES* section computes `s[X]` state amounts at multiple time points using *ordinary differential equations* given initial `s[X]` from the *STATES* section and previously computed `m[X]` and `c[X]` parameters for each row in the *data file*.

Note the *DERIVATIVES* section is optional. It is possible to **not** have a compartment model in your script. For example you can just directly use individual `m[X]` variables in the *PREDICTIONS* section instead of generating `s[X]` values.

The *DERIVATIVES* section is available in the following scripts:-

- *Tut Script*
- *Gen Script*
- *Fit Script*
- *Sim Script*
- *MTut Script*
- *MGen Script*
- *MFit Script*
- *MSim Script*

i.e. Any script that processes PK/PD models.

34.10.1 Example DERIVATIVES for PK Model

The example below is used in builtin_fit_example.

```
DERIVATIVES: |
    # s[DEPOT,CENTRAL,PERI] = @dep_two_cmp_cl{dose:@bolus{amt:c[AMT]}}
    d[DEPOT] = @bolus{amt:c[AMT]} - m[KA]*s[DEPOT]
    d[CENTRAL] = m[KA]*s[DEPOT]␣
↪- s[CENTRAL]*m[CL]/m[V1] - s[CENTRAL]*m[Q]/m[V1]  + s[PERI]*m[Q]/m[V2]
    d[PERI] = s[CENTRAL]*m[Q]/m[V1] - s[PERI]*m[Q]/m[V2]
```

The example above defines a two compartment PK model with a **Depot** compartment. The same model can be expressed as:-

```
DERIVATIVES: |
    s[DEPOT,CENTRAL,PERI] = @dep_two_cmp_cl{
        dose:@bolus{amt:c[AMT]},
        KA: m[KA],
        CL: m[CL],
        V1: m[V1],
        Q: m[Q],
```

(continues on next page)

(continued from previous page)

```
        V2: m[V2],
    }
```

Where *@dep_two_cmp_cl* is a *analytic compartment function*. That uses the analytic solution to the *ordinary differential equation* instead of using a numerical *ODE_SOLVER*.

Note you can also write:-

```
DERIVATIVES: |
    s[DEPOT,CENTRAL,PERI] = @dep_two_cmp_cl{ dose:@bolus{amt:c[AMT]} }
```

As the default values for 'KA', 'CL' etc are the equivalent `m[X]` variables. You still need to have all the appropriate `m[X]` variables defined in *MODEL_PARAMS* and `c[AMT]` defined in your *data file*.

PoPy also provides multiple alternative ways of formulating the `d[X]` equations as follows:-

```
DERIVATIVES: |

    d[DEPOT] = @bolus{amt:c[AMT]}
    d[CENTRAL] = 0.0
    d[PERI] = 0.0

    d[DEPOT] -= m[KA]*s[DEPOT]
    d[CENTRAL] += m[KA]*s[DEPOT]

    d[CENTRAL] -= s[CENTRAL]*m[CL]/m[V1]

    d[CENTRAL] -= s[CENTRAL]*m[Q]/m[V1]
    d[PERI] += s[CENTRAL]*m[Q]/m[V1]

    d[CENTRAL] += s[PERI]*m[Q]/m[V2]
    d[PERI] -= s[PERI]*m[Q]/m[V2]
```

The syntax above is standard *Python*, but makes the inputs and outputs of each compartment clear. You can also use this **flow** based syntax:-

```
DERIVATIVES: |

    d[DEPOT] = @bolus{amt:c[AMT]}
    d[CENTRAL] = 0.0
    d[PERI] = 0.0

    d[DEPOT->CENTRAL] += m[KA]*s[DEPOT]

    d[CENTRAL] -= s[CENTRAL]*m[CL]/m[V1]

    d[CENTRAL->PERI] += s[CENTRAL]*m[Q]/m[V1]
    d[PERI->CENTRAL] += s[PERI]*m[Q]/m[V2]
```

This expresses the flows between compartments explicitly and avoids having to manually pair up the positive and negative flows. Note an unintentional mis-match between inputs and outputs in compartment

models usually leads to *mass balance* issues and models that are difficult to interpret and fit.

Note a sanity check on the *DERIVATIVES* section is to view the compartment diagram that is automatically created by *PoPy*, see Fig. 3.

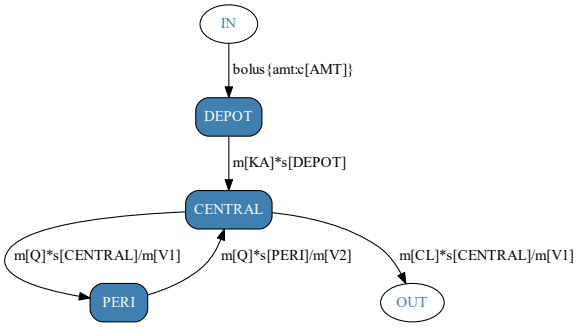


Fig. 3: Two compartment model with depot dosing, computed automatically from *DERIVATIVES* section.

All the example *DERIVATIVES* sections presented here generate the same diagram. If you make a *mass balance* error your diagram may well **not** look like you expect. This is a useful safety feature built into *PoPy*.

34.10.2 Example DERIVATIVES for PD Model

See *Direct PD Model* for example *Tut Script*, using the following *DERIVATIVES* section for a PD model:-

```
DERIVATIVES: |
    d[CENTRAL] = @bolus{amt:c[AMT]} - s[CENTRAL]*c[CL]/c[V]
    d[MARKER] = m[BASE]*m[KOUT] - (1+s[CENTRAL]/c[V])*m[KOUT]*s[MARKER]
```

As discussed in *Example STATES for PD Model*. The following *STATES* section is required to initialise the derivative model in an equilibrium state:-

```
STATES: |
    s[CENTRAL] = 0.0
    s[MARKER] = m[BASE]
```

Note it's possible to split this *DERIVATIVES* model up into separate flows:-

```
DERIVATIVES: |

    d[CENTRAL] = @bolus{amt:c[AMT]}
    d[MARKER] = 0.0

    d[CENTRAL] -= s[CENTRAL]*c[CL]/c[V]

    d[MARKER] += m[BASE]*m[KOUT]
    d[MARKER] -= (1+s[CENTRAL]/c[V])*m[KOUT]*s[MARKER]
```

It's also possible to mix and match `s[X]` and `d[X]` variables, by using a *analytic compartment function* for the PK compartment:-

```
DERIVATIVES: |
    s[CENTRAL] = @iv_one_cmp_cl{ dose:@bolus{amt:c[AMT]}, CL:c[CL], V:c[V]}
    d[MARKER] = m[BASE]*m[KOUT] - (1+s[CENTRAL]/c[V])*m[KOUT]*s[MARKER]
```

Note in the above if you use a *analytic compartment function* then you need `s[X]` on the **left hand side** (see **Central** compartment). If you use numerical equations then you need `d[X]` on the **left hand side** and `s[X]` variables on the **right hand side** (see **Marker** compartment).

All of the above *DERIVATIVES* sections should result in the same compartment diagram as shown in Fig. 4.

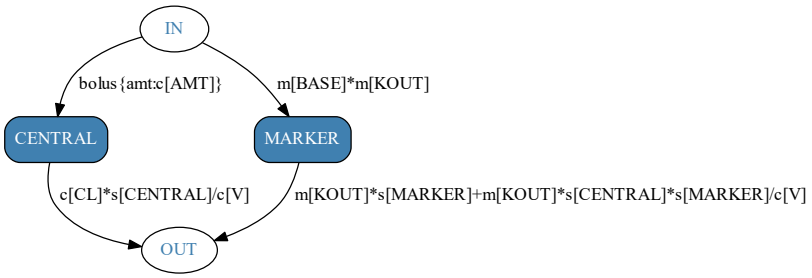


Fig. 4: direct PD model, computed automatically from *DERIVATIVES* section.

34.10.3 Example DERIVATIVES using x[TIME]

See *Sine circadian model* for example *Tut Script*, using the following *DERIVATIVES* section:-

```
DERIVATIVES: |
    conc_central = s[CENTRAL]/c[V]
    circ = exp(m[AMP] * sin(2*pi*(x[TIME]-m[INT])/12))
    d[CENTRAL] = @bolus{lag:0, amt:c[AMT]} - c[CL]*conc_central   # PK
    d[MARKER] =  m[KIN]*conc_central + circ - m[KOUT]*s[MARKER]   # PD
```

This PK model has a circadian input to the 'MARKER' compartment, so is in a equilibrium oscillating about a mean value, until a dose is administered in the PK compartment, which then causes an effect in the PD compartment through the 'conc_central' variable.

The main point of interest in this model is that the variable `x[TIME]` is used within the *DERIVATIVES* section. `x[TIME]` is a special variable that uses the continuous time when solving the *ordinary differential equations*. This is distinct from using `c[TIME]`, which is constant at time points between data rows.

Note you can use `c[TIME]` as an approximation if you have lots of data rows, but generally it's better to use `x[TIME]` to model explicit time dependent components such as circadian variation within the *DERIVATIVES* section.

The compartment diagram for this PK/PD model is as shown in Fig. 5.

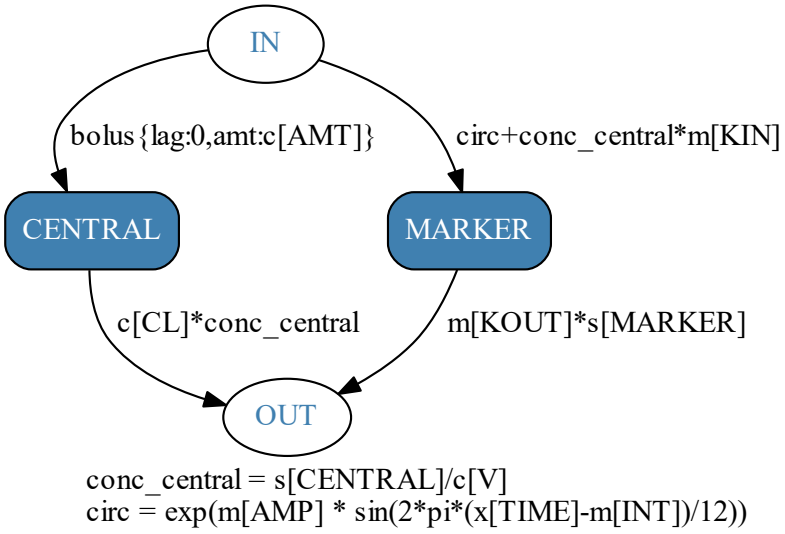


Fig. 5: circadian PD model, computed automatically from *DERIVATIVES* section.

34.10.4 Rules for DERIVATIVES section

Like all *verbatim* sections the *DERIVATIVES* section of the config file accepts free form *Python pseudocode*, but there are some rules regarding which variables are allowed in a *DERIVATIVES* section as follows:-

- **Only** `d[X]` and local variables can be defined on the **left hand side** without using *Analytic Compartment Functions*.
- `s[X]` variables are usually on the **right hand side** and **must** have a corresponding `d[X]` variable
- `s[X]` variables can only be defined on the **left hand side** if you use a *analytic compartment function*
- `s[X]` variables present in *DERIVATIVES* but **not** in *STATES* are initialised to zero.
- `c[X]` and `m[X]` are only allowed on the **right hand side** of expressions
- `c[X]` must be defined in the *data file* or *PREPROCESS* section in a *Fit Script*
- `c[X]` must be defined within the *EFFECTS* section in a *Gen Script* or *GEN_EFFECTS* section in a *Tut Script*.
- `m[X]` must be defined in the *MODEL_PARAMS*
- `x[TIME]` can be used to model continuous time, as opposed to discrete `c[TIME]`

You can **not** use `p[X]`, `r[X]`, `f[X]` or `t[X]` *etc.* variables at all in this section.

Like all *verbatim* sections it is possible to introduce syntax errors by writing malformed *Python*. Any coding errors will be reported when *PoPy* attempts to compile or run the *DERIVATIVES* function as a temporary .py file.

34.11 PREDICTIONS

A required *verbatim* section that defines model `p[X]` prediction variables, but also compares `p[X]` variables to `c[X]` data and evaluates likelihoods or samples new `c[X]` data.

The *PREDICTIONS* section takes as input `c[X]`, `m[X]` and `s[X]` variables and outputs `p[X]` data. The *PREDICTIONS* section is required in the following scripts:-

- *Tut Script*
- *Gen Script*
- *Fit Script*
- *Sim Script*
- *MTut Script*
- *MGen Script*
- *MFit Script*
- *MSim Script*

i.e. Any *Gen Script*/*Sim Script* that samples new data points or any *Fit Script* that evaluates likelihoods requires a *PREDICTIONS* section.

34.11.1 Example PREDICTIONS for PK Model

The example below is used in builtin_fit_example.

```
PREDICTIONS: |
    p[CEN] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[CEN]**2
    c[DV_CENTRAL] ~ norm(p[CEN], var)
```

In a *Fit Script* the *PREDICTIONS* section above computes a likelihood by comparing the `p[DV_CENTRAL]` predictions computed by the model and the `c[DV_CENTRAL]` values found in the *data file*.

The noise model is a combined additive and proportional noise model. See *Residual Error Model* for more details on types of noise model.

Note that in builtin_gen_example, the *PREDICTIONS* section is exactly the same as above, however the interpretation of this line is different:-

```
c[DV_CENTRAL] ~ norm(p[DV_CENTRAL], var)
```

In a *Gen Script* this line is now used to sample new values of `c[DV_CENTRAL]` when creating a new data set.

The dual nature of the '~' symbol allows the same *PREDICTIONS* section to be re-used throughout multiple scripts.

34.11.2 Example PREDICTIONS for PD Model

See *Direct PD Model* for an example *Tut Script*, using the following *PREDICTIONS* section for a PD model:-

```
PREDICTIONS: |
    clabel[TIME] = "Time (minutes)"
    plabel[MARKER] = "Biomarker concentration (mg/L)"
    p[MARKER] = s[MARKER]
    var = m[ANOISE]**2
    c[MARKER] ~ norm(p[MARKER], var)
```

In the above example the structure of the *PREDICTIONS* section is the same as the *Example PREDICTIONS for PK Model*, however this time the likelihood/sampling is between the `c[MARKER]` data and `p[MARKER]` predictions, using an additive noise model.

A more complex *PREDICTIONS* section is shown below. See *Direct PD Model Simultaneous PK/PD Parameter fit*:-

```
PREDICTIONS: |
    clabel[TIME] = "Time (minutes)"
    plabel[MARKER] = "Biomarker concentration (mg/L)"
    plabel[CENTRAL] = "Drug concentration (mg/L)"
    p[CENTRAL] = s[CENTRAL]/m[V]
    c[CENTRAL] ~ norm(p[CENTRAL], m[PK_ANOISE]**2)
    p[MARKER] = s[MARKER]
    c[MARKER] ~ norm(p[MARKER], m[PD_ANOISE]**2)
```

If the *PREDICTIONS* block occurs in a *Gen Script* then both the `c[CENTRAL]` and `c[MARKER]` fields are sampled. If this *PREDICTIONS* section is included in a *Fit Script* then there are two different data fields that contribute to the likelihood, namely 'CENTRAL' and 'MARKER'.

In the *Fit Script* case the likelihood is evaluated for a particular '~' expression for every *observation* row in the data set. The contribution to the likelihood of a particular `c[X]` variable can also be controlled by the `c[X_FLAG]` field, if present in the *data file*. This mechanism avoids erroneously computing likelihoods against null values.

Each *PoPy* `c[X]` variable is allowed to have it's own `c[X_FLAG]`, which enables *PoPy* to handles any number of fields elegantly without changing the natural *data file* structure. This is especially useful when performing a simultaneous fit with more than one measured variable because there is no requirement to mangle the data set by concatenating different fields into one column.

34.11.3 Example PREDICTIONS for BLQ Observations

If your *data file* contains some observations which are recorded as **BLQ**, *i.e.* they are below the **LLQ** of the assay used to measure drug concentrations. Then to include these data points in your analysis you can use a *~rectnorm()* distribution, see below:-

```
PREDICTIONS:
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
```

(continues on next page)

(continued from previous page)

```
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[DV_CENTRAL]**2
    c[DV_CENTRAL] ~ rectnorm(p[DV_CENTRAL], var, LLQ=2.0)
```

This formulation will model observations `c[DV_CENTRAL]` below **LLQ** as the likelihood of being in the range [-inf,LLQ] and observations greater than **LLQ** using a standard *~norm()* distribution likelihood. See *~rectnorm()* distribution for more information on this topic.

34.11.4 Rules for PREDICTIONS section

Like all *verbatim* sections, the *PREDICTIONS* section of the configuration file accepts free form *Python pseudocode*, but there are some rules regarding which variables are allowed in a *PREDICTIONS* section as follows:-

- **Only** `p[X]` and local variables can be defined on the **left hand side** using '='
- **Only** `c[X]` can be defined on the **left hand side** using '~'
- `m[X]` and `s[X]` are only allowed on the **right hand side** of expressions
- `c[X]` must be in the *data file* or defined in the *PREPROCESS* section in a *Fit Script*
- `c[X]` must be defined within the *EFFECTS* in a *Gen Script* or *GEN_EFFECTS* in a *Tut Script*.
- `m[X]` must be defined in the *MODEL_PARAMS* section
- `s[X]` must be defined in the *DERIVATIVES* section

You can **not** use `f[X]`, `r[X]` or `t[X]` *etc.* variables at all in this section.

Like all *verbatim* sections it is possible to introduce syntax errors by writing malformed *Python*. Such errors will be reported when *PoPy* attempts to compile or run the *PREDICTIONS* function as a temporary .py file.

34.12 POSTPROCESS

An optional *verbatim* section that post processes the `c[X]` variables after running the *PREDICTIONS* section. This section can be used to alter the final `c[X]` variables, or possibly remove some data, *e.g.* negative concentrations.

A kind of flexible post generation filter implemented in *Python*, similar in functionality to *PREPROCESS*.

The *POSTPROCESS* is available in the following scripts:-

- *Tut Script*
- *MTut Script*
- *Gen Script*
- *MGen Script*

i.e. any script that generates a population data set from a model.

34.12.1 Example POSTPROCESS section

Some simple examples of using a *POSTPROCESS* section. Use the 'return' syntax to remove observation rows with negative concentrations:-

```
POSTPROCESS: |
    if c[CONC] < 0.0 and c[TYPE] == 'obs':
        return
```

Setting negative concentrations to zero for observation rows:-

```
POSTPROCESS: |
    if c[CONC] < 0.0 and c[TYPE] == 'obs':
        c[CONC] = 0.0
```

Enforcing a below quantification limit for observation rows:-

```
POSTPROCESS: |
    bql_limit = 5.0
    if c[CONC] < bql_limit and c[TYPE] == 'obs':
        c[CONC] = bql_limit
```

Creating some derived data for observation rows:-

```
POSTPROCESS: |
    if c[CONC] > 100 and c[TYPE] == 'obs':
        c[HIGH_CONC_FLAG] = 1.0
    else:
        c[HIGH_CONC_FLAG] = 0.0
```

Like the *PREPROCESS* section each row of the data is processed separately, but otherwise any valid *Python* function can be used to create new `c[X]` data or remove rows using the "return" syntax.

34.12.2 Rules for POSTPROCESS section

Like all *verbatim* sections the *POSTPROCESS* section of the config file accepts free form pseudo *Python* code, but there are some rules regarding which variables are allowed in a *POSTPROCESS* section as follows:-

- **Only** `c[X]` variables and local *Python* variables are allowed
- `c[X]` on the **right hand side** and within `if` statements must be previously defined on the **left hand side** or in the *data file*
- `c[X]` declared on the **left hand side** must **not** already exist in the *data file*
- return must always be **null**

So you can **not** use `m[X]`, `f[X]`, `r[X]`, `d[X]` etc variables in this section.

Like all *verbatim* sections it is possible to introduce syntax errors by writing malformed *Python*. This will hopefully be picked up when *PoPy* attempts to compile or run the *POSTPROCESS* function as a temporary .py file.

34.13 ODE_SOLVER

An optional section that defines how an *PoPy* will solve the *ordinary differential equations* in the *DERIVATIVES* section.

Note the *ODE_SOLVER* section is optional, if you have no *DERIVATIVES* section then you do not need to have an *ODE_SOLVER* section.

The *ODE_SOLVER* section is available in the following scripts:-

- *Tut Script*
- *Gen Script*
- *Fit Script*
- *Sim Script*
- *MTut Script*
- *MGen Script*
- *MFit Script*
- *MSim Script*

i.e. Any script that processes PK/PD models and may have a *DERIVATIVES* section, may also have an *ODE_SOLVER* section.

34.13.1 ODE_SOLVER Options

There are three principle *ODE_SOLVER* options available:-

- *NO_SOLVER* - use if there is no *DERIVATIVES* section
- *ANALYTIC* - use if *DERIVATIVES* contains only *analytic compartment functions*
- *CPPLSODA* - numerical solver - use if *DERIVATIVES* contains `d[X]` equations
- *SCIPY_ODEINT* - numerical solver - deprecated, slower, SciPy version of *CPPLSODA*.

The options above are tied to the nature of the *DERIVATIVES* section. Some examples of *DERIVATIVES* sections and appropriate *ODE_SOLVER* settings are shown below.

34.13.2 Example ODE_SOLVER using NO_SOLVER

In the case where the *PoPy* script contains no *DERIVATIVES* section or the *DERIVATIVES* section is null:-

```
DERIVATIVES: |
```

Then the *PoPy* script should contain no *ODE_SOLVER* section or the *ODE_SOLVER* should also be null:-

```
ODE_SOLVER:
    NO_SOLVER: {}
```

If the *DERIVATIVES* section is null and *ODE_SOLVER* is set to 'ANALYTIC' or 'SCIPY_ODEINT', then *PoPy* should issue a warning, but the *ODE_SOLVER* section will otherwise have no effect.

Note, the simplest solution if you have **no** compartment model is to just remove the *DERIVATIVES* and *ODE_SOLVER* sections from the script completely.

34.13.3 Example ODE_SOLVER using ANALYTIC

In the case where the *PoPy* script contains a *DERIVATIVES* section consisting of only an *analytic compartment function*:-

```
DERIVATIVES: |
    s[DEPOT,CENTRAL,PERI] = @dep_two_cmp_cl{dose:@bolus{amt:c[AMT]}
```

Then the *ODE_SOLVER* section should be:-

```
ODE_SOLVER:
    ANALYTIC: {}
```

If the *DERIVATIVES* section contains only an *analytic compartment function* and *ODE_SOLVER* is set to 'NO_SOLVER' or 'SCIPY_ODEINT', then *PoPy* should issue a warning, but the *ODE_SOLVER* section will otherwise **automatically** switch to 'ANALYTIC'.

34.13.4 Example ODE_SOLVER using CPPLSODA

In the case where the *PoPy* script contains a *DERIVATIVES* section consisting of *ordinary differential equations* with `d[X]` variables on the **left hand side** and `s[X]` on the **right hand side**, for example:-

```
DERIVATIVES: |
    # s[DEPOT,CENTRAL,PERI] = @dep_two_cmp_cl{dose:@bolus{amt:c[AMT]}}
    d[DEPOT] = @bolus{amt:c[AMT]} - m[KA]*s[DEPOT]
    d[CENTRAL] = m[KA]*s[DEPOT]␣
↪- s[CENTRAL]*m[CL]/m[V1] - s[CENTRAL]*m[Q]/m[V1]  + s[PERI]*m[Q]/m[V2]
    d[PERI] = s[CENTRAL]*m[Q]/m[V1] - s[PERI]*m[Q]/m[V2]
```

Then the *ODE_SOLVER* section should use 'CPPLSODA', for example:-

```
ODE_SOLVER:
    CPPLSODA:
        atol: 1e-06
        rtol: 1e-06
        max_nsteps: 10000000
```

If the *DERIVATIVES* section contains `d[X]` equations and *ODE_SOLVER* is set to 'NO_SOLVER' or 'ANALYTIC', then *PoPy* should issue an error when checking the script, because a numerical solver is required.

CPPLSODA is a *C++* version of the Fortran *LSODA* solver [Radhakrishnan1994]. The same *ordinary differential equation* solver method used by the *Nonmem* ADVAN13 routine.

Note that CPPLSODA exposes the following parameters of *LSODA*:-

- atol - Additive tolerance of the *LSODA* error control
- rtol - Relative tolerance of the *LSODA* error control
- max_nsteps - Maximum number of steps for *LSODA* integrator

All of the parameters above are optional with the following default values:-

- atol: 1e-12
- rtol: 1e-12
- max_nsteps: 10000000

Hence the following *ODE_SOLVER* section:-

```
ODE_SOLVER: {CPPLSODA: {}}
```

Is equivalent to:-

```
ODE_SOLVER:
    CPPLSODA:
        atol: 1e-12
        rtol: 1e-12
        max_nsteps: 10000000
```

34.13.5 Example ODE_SOLVER using SCIPY_ODEINT

Note 'SCIPY_ODEINT' is deprecated in favour of *CPPLSODA* above.

SCIPY_ODEINT is a wrapper around the following *Python* numerical *ordinary differential equation* solver from the SciPy package:-

```
scipy.integrate.odeint
```

Which in turn is a wrapper around the Fortran *LSODA* solver [Radhakrishnan1994]. The same *ordinary differential equation* solver that is used by the *Nonmem* ADVAN13 routine. You can read more about this integrator here in the SciPy documentation:-

https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html

Note that *PoPy* exposes the following parameters of *LSODA*:-

- atol - Additive tolerance of the *LSODA* error control
- rtol - Relative tolerance of the *LSODA* error control
- max_nsteps - Maximum number of steps for *LSODA* integrator (called 'mxstep' in link above)

All of the parameters above are optional with the following default values:-

- atol: 1e-12
- rtol: 1e-12
- max_nsteps: 10000000

Hence the following *ODE_SOLVER* section:-

```
ODE_SOLVER: {SCIPY_ODEINT: {}}
```

Is equivalent to:-

```
ODE_SOLVER:
    SCIPY_ODEINT:
        atol: 1e-12
        rtol: 1e-12
        max_nsteps: 10000000
```

Note it is recommended to use *CPPLSODA* instead, which is faster and will give very similar results.

34.14 FIT_METHODS

A required section that defines how *PoPy* estimates the `f[X]` and `r[X]` model parameters given a *data file*.

The *FIT_METHODS* section is required in the following scripts:-

- *Tut Script*
- *Fit Script*
- *MTut Script*
- *MFit Script*

i.e. Any script that estimates PK/PD model parameters is required to have a *FIT_METHODS* section.

PoPy currently has the following fitting methods:-

- *JOE Fitting Method*
- *FOCE Fitting Method*
- *ND Fitting Method*

Which can be combined using a simple list, see *FIT_METHODS Structure*.

34.14.1 FIT_METHODS Structure

The *FIT_METHODS* section is a list of fitting methods to be applied in order to estimate the `f[X]` and `r[X]` variables of a PK/PD model, for example:-

```
FIT_METHODS:
    - <fitter1>: {}
    - <fitter2>: {}
    - <fitter3>: {}
```

This allows in this case <fitter1> to initialise the `f[X]` for <fitter2> and for <fitter2> to then initialise the `f[X]` for <fitter3> *etc.*.

For example it is recommended that with *PoPy*, you first apply the *JOE* fitter method, followed by the *FOCE* method.

34.14.2 JOE Fitting Method

JOE stands for Joint Optimisation and Estimation and is *PoPy*'s approximate equivalent of *Nonmem*'s *ITS* (Iterative Two Stage) method. The methods differ in how they update the `f[X]` provided in a *Fit Script* by the modeller to estimate the final optimised `f[X]`.

However both *JOE* and *ITS* both attempt to separately optimise the main `f[X]`, variance `f[X]`, noise `f[X]` and `r[X]` parameters to arrive at a reasonable overall fitted `f[X]` vector. This is in contrast to the *FOCE* approach which applies a quasi-newton optimisor to the combined `f[X]` vector.

In general *JOE* is more capable of finding sensible fitted `f[X]` parameters when initialised further away from the true `f[X]` compared to *FOCE*. However, *FOCE* is often more accurate when started close to the true solution.

34.14.3 FOCE Fitting Method

The *FOCE* and *JOE* methods both use a first order version of the Laplace approximation to make the computation of the likelihood function tractable. See [Wang2007] for details. We refer to this likelihood as the *FOCE ObjV*, which was first described in [Lindstrom1990], but is now most commonly associated with the popular *FOCE* fitting method in *Nonmem*.

The *FOCE* method uses a quasi-newton optimisor inter-leaved with a `r[X]` optimiser for each individual. It is the most common fitting method used in PK/PD today. The *PoPy* implementation of *FOCE* is similar to *Nonmem*, but it is not 100% the same.

Note, given the same PK/PD model, *data file* and same `f[X]` and `r[X]` variables. *JOE* and *FOCE* will return the same *FOCE ObjV* in *PoPy*. The fitting results are only likely to agree exactly for very simple PK models, but should be similar for more complex cases.

Note *FOCE* is generally more accurate than *JOE* when initialised close to the true minima. However it is also capable of falling into false minima away from the global minima. For this reason we recommend using *JOE* then *FOCE* when using *PoPy*, see *FIT_METHODS Examples*.

The type of optimisation methods used by *PoPy* are described in references such as [DennisSchnabel1987] [NocedalWright2006].

34.14.4 ND Fitting Method

The recommended fitting method for *PoPy* version 1.0.5 is the *ND* method.

The *ND* method uses the same *ObjV* as *FOCE*. This new approach uses *FOCE* to estimate the `f[X]` parameters. However when the gradient based *FOCE* algorithm converges *ND* switches to using the non-gradient *BOBYQA* optimisation method. When *BOBYQA* converges it switches back to the *FOCE*, until both methods have converged. The *ND* method is therefore very robust as it utilises two different fitting strategies.

However the *ND* method will always be slower than *FOCE*, so for simple cases, or where a quick result, but not necessarily optimal solution is required, it may be preferable to use *JOE* or *FOCE* in preference to *ND*.

In the more common case, where accuracy is required it is recommended to use *ND*.

34.14.5 Other Fitting Methods

Note *JOE*, *ITS* and *FOCE* fitting methods are deterministic and fundamentally different from the stochastic sampling methods such as *SAEM* and *IMP*. *SAEM* and *IMP* that also estimate `f[X]` for PK/PD models, but the stochastic *Objective Value* is **not** based on the Laplace approximation used in the *FOCE ObjV*.

34.14.6 FIT_METHODS Examples

Fit using ND

The recommended fitting method for *PoPy* version 1.0.5 is the *ND* method.

The *ND* method can be implemented as follows:-

```
FIT_METHODS: [ND:{}]
```

Or alternatively:-

```
FIT_METHODS:
    - ND: {max_n_main_iterations: 30}
```

Fit using JOE and FOCE

Another way to fit PK/PD models in *PoPy* is to use *JOE* followed by *FOCE* in a *Fit Script* is as follows:-

```
FIT_METHODS: [JOE:{}, FOCE:{}]
```

This single line, runs the *JOE* fitting method followed by *FOCE* with the default settings. Note that the square bracket notation '[]' is required because 'FIT_METHODS' expects a list. Or equivalently:-

```
FIT_METHODS:
    - JOE: {}
    - FOCE: {}
```

Where the '-' is *YAML* notation for a list item.

One of the simplest *JOE* and *FOCE* parameters is 'max_n_main_iterations':-

```
FIT_METHODS:
    - JOE: {max_n_main_iterations: 30}
    - FOCE: {max_n_main_iterations: 30}
```

Which sets the number of times the `f[X]` parameters are updated. The *JOE* fitting proceeds iteratively with interleaved `f[X]` and `r[X]` updates, followed by *FOCE* starting from the final `f[X]` result returned by *JOE*. Setting this limit forces *PoPy* to stop after a finite number of iterations.

If you miss out 'max_n_main_iterations' it defaults to 50. Another option is as follows:-

```
FIT_METHODS:
    - JOE: {max_n_main_iterations: 0}
```

This only updates the `r[X]` parameters and leaves the `f[X]` fixed.

34.14.7 Fitting Convergence Criteria

Another common setting is the 'CONVERGER' field. There are three possible settings:-

- NONE - no convergence termination - estimation stops when 'max_n_main_iterations' is reached
- OBJ_INC - terminate convergence if the *ObjV* increases
- OBJ_SIM - terminate convergence if the *ObjV* is similar between iterations

These settings decide when a *Fit Script* will return the final `f[X]` estimates.

The default setting for *JOE* and *FOCE* fitting is 'OBJ_INC', hence:-

```
FIT_METHODS:
    - JOE:
        max_n_main_iterations: 100
    - FOCE:
        max_n_main_iterations: 100
```

Is the same as:-

```
FIT_METHODS:
    - JOE:
        max_n_main_iterations: 100
        CONVERGER: {OBJ_INC: {}}
    - FOCE:
        max_n_main_iterations: 100
        CONVERGER: {OBJ_INC: {}}
```

And the fit estimation will stop as soon as any iteration increases the *ObjV*. To allow the *ObjV* to increase during fitting, *i.e.* if the likelihood gets worse between iterations, but you wish the search to continue, use:-

```
FIT_METHODS:
    - JOE:
        max_n_main_iterations: 100
        CONVERGER: {OBJ_SIM: {}}
    - FOCE:
        max_n_main_iterations: 100
        CONVERGER: {OBJ_SIM: {}}
```

Here 'OBJ_SIM' will terminate the estimation when two consecutive iterations return similar objective values (within a relative tolerance of 1e-06).

Note that when fitting complex models the *ObjV* usually decreases at each iteration, but occasionally increases. This is due to fact that `f[X]` and `r[X]` are optimised separately. Separate optimisation is necessary for computational efficiency. However optimising components of the likelihood independently

can mean the combined *ObjV* increases. The 'OBJ_SIM' option above is designed to allow the search to continue in these cases.

34.15 COVARIANCE

An optional section that requests *PoPy* to compute *Standard Errors* of the `f[X]` variables, after running a *Fit Script*.

The *COVARIANCE* option is only available in a *Fit Script* or *Tut Script*. The *Tut Script* simply passes the *COVARIANCE* section to the *Fit Script*. The *MFit Script* and *MTut Script* do not currently include this option.

34.15.1 COVARIANCE Example

If you miss out the 'COVARIANCE' field, that is equivalent to putting:-

```
COVARIANCE: {NO_COVARIANCE: {}}
```

In the *Fit Script* and no standard errors are computed.

If you use this setting:-

```
COVARIANCE: {SANDWICH: {}}
```

After the *Fit Script* outputs the final `f[X]` estimates, *PoPy* will compute the standard errors utilising a sandwich operator. See *Standard Errors* for more details.

By default *PoPy* computes standard errors using only the main `f[X]` estimates and excludes the variance `f[X]`.

You can include the variance `f[X]` using the following setting:-

```
COVARIANCE: {SANDWICH: {var_covariance_flag: True}}
```

This will compute standard error estimates for all `f[X]` however it will take a long time if you have large variance `f[X]` matrices in your model. Also note that the standard error estimates will be different if the variances are included in the computation.

34.16 OUTPUT_SCRIPTS

An optional section that controls the **child** scripts that are created and possibly run after a **parent** script has finished running.

The *OUTPUT_SCRIPTS* section is available in the following scripts:-

- *Tut Script*
- *Gen Script*
- *Fit Script*

- *Sim Script*
- *MSim Script*
- *MTut Script*

i.e. Any script that can generate **child** scripts may have an *OUTPUT_SCRIPTS* section.

34.16.1 Structure of OUTPUT_SCRIPTS

The *OUTPUT_SCRIPTS* in all *Script File Formats* share a common format, as follows:-

```
OUTPUT_SCRIPTS:
    <child_script1>: {output_mode: run}
    <child_script2>: {output_mode: create}
    <child_script3>: {output_mode: none}
```

The *OUTPUT_SCRIPTS* section is a *dict* record. i.e it is a *Python* dictionary with child script names as keys.

All child script sections share the 'output_mode' field, which can take the following values:-

- run - create the child script and run it
- create - create the child script, but do **not** run it
- none - do **not** create the child script

Generally if a child script is not mentioned in the parent script the 'output_mode' defaults to 'none'.

The available child scripts are dependent on the parent *script type*. Also there maybe additional options for each child script, see examples below for more concrete details.

34.16.2 Example OUTPUT_SCRIPTS for tut_script

The example below is used in *Running Your First tut Script*.

```
OUTPUT_SCRIPTS:
    GEN: {output_mode: run, sim_time_step: 1.0}
    FIT: {output_mode: run, sim_time_step: 1.0}
    COMP: {output_mode: run}
    TUTSUM: {output_mode: run}
```

This specifies that a child *Gen Script*, *Fit Script*, *Comp Script* and *TutSum Script* are all run.

Note it's possible to miss out the *OUTPUT_SCRIPTS* section from a *Tut Script*, then the default settings are run, which is equivalent to:-

```
OUTPUT_SCRIPTS:
  ␣
↪ GEN: {output_mode: run, sim_time_step: -1.0, grph_list: ["OBS_vs_TIME"]}
  ␣
↪ FIT: {output_mode: run, sim_time_step: -1.0, grph_list: ["OBS_vs_TIME"]}
```

(continues on next page)

(continued from previous page)

```
    COMP: {output_mode: none}
    TUTSUM: {output_mode: run}
```

This results in more limited tutorial outputs compared to the previous example.

34.16.3 Example OUTPUT_SCRIPTS for gen_script

The example below is used in builtin_gen_example.

```
OUTPUT_SCRIPTS:
    SIM: {output_mode: run, sim_time_step: 1.0}
    GENSUM: {output_mode: run}
```

This specifies that a child *Sim Script* and *GenSum Script* are run.

Note it's possible to miss out the *OUTPUT_SCRIPTS* section from a *Gen Script*, then the default settings are run, which is equivalent to:-

```
OUTPUT_SCRIPTS:
    GRPH: {output_mode: none, grph_list: ["OBS_vs_TIME"]}
    SIM: {output_mode: none, grph_list: ["OBS_vs_TIME"], sim_time_step: -1.0}
    GENSUM: {output_mode: none}
```

i.e. if you miss out the *OUTPUT_SCRIPTS* then you just get the *Gen Script* output and no further processing, as you might expect.

34.16.4 Example OUTPUT_SCRIPTS for fit_script

The example below is used in builtin_fit_example.

```
OUTPUT_SCRIPTS:
    SIM: {output_mode: run, sim_time_step: 1.0}
    MSIM: {output_mode: create}
    FITSUM: {output_mode: run}
```

This specifies that a child *Sim Script* and *FitSum Script* are run and a *MSim Script* is created on disk but **not** run automatically.

Note it's possible to miss out the *OUTPUT_SCRIPTS* section from a *Fit Script*, then the default settings are run, which is equivalent to:-

```
OUTPUT_SCRIPTS:
    GRPH: {output_mode: none, grph_list: ["OBS_vs_TIME"]}
    SIM: {output_mode: none, grph_list: ["OBS_vs_TIME"], sim_time_step: 1.0}
    MSIM: {output_mode: none, vpc_list: ["OBS_vs_TIME_VPC"]}
    FITSUM: {output_mode: none}
```

i.e. if you miss out the *OUTPUT_SCRIPTS* then you just get the *Fit Script* output and no further processing, as you might expect.

34.16.5 Example OUTPUT_SCRIPTS for sim_script

The example below is from a *Sim Script* generated by a *Fit Script* in builtin_fit_example.

```
OUTPUT_SCRIPTS: {TABLE: {}}
```

The single 'GRPH' child script above, is used to plot the results of the *Sim Script* simulated PK curves.

You can edit the 'GRPH' options to control the graphical output.

34.16.6 Example OUTPUT_SCRIPTS for msim_script

The example below is from a *MSim Script* generated by a *Fit Script* in builtin_fit_example.

```
OUTPUT_SCRIPTS:
    VPC:
        output_mode: run
        vpc_list: ["COMB_QUANT_SIM_VPC"]
```

The single 'VPC' child script above, is used to plot a *VPC* using the results of the *MSim Script* multi population simulated PK curves.

You can edit the 'VPC' options to control the graphical output.

34.16.7 Example OUTPUT_SCRIPTS for mtut_script

The example below is used in builtin_mtut_example.

```
OUTPUT_SCRIPTS:
    MGEN: {output_mode: run}
    MFIT: {output_mode: run}
    MCOMP: {output_mode: run, dot_size: 12}
```

This *MTut Script* outputs and runs a *MGen Script*, *MFit Script* and *MComp Script*.

Notice you can optionally alter the size of the dots on the *MComp Script* scatter plots using the 'dot_size' field.

34.17 OUTPUT_OPTIONS

An optional section that controls the output from a script.

The *OUTPUT_OPTIONS* section is available in the following scripts:-

- *Sim Script*
- *MSim Script*
- *MGen Script*
- *MTut Script*

The *OUTPUT_OPTIONS* contain a *script type* specific set of extra options.

34.17.1 Example OUTPUT_OPTIONS for sim_script

The *OUTPUT_OPTIONS* in a *Sim Script* contain a single field 'sim_time_step'. For example the following:-

```
OUTPUT_OPTIONS:
    sim_time_step: 0.5
```

means that *PoPy* will simulate a `p[X]` value at every 0.5 time units when simulating from a PK/PD model.

An alternative (and default setting) is:-

```
OUTPUT_OPTIONS:
    sim_time_step: -1.0
```

Which only simulates `p[X]` model predictions for time points present in the *data file*, *i.e.* dense time point sampling is switched off by using a negative 'sim_time_step'.

34.17.2 Example OUTPUT_OPTIONS for msim_script

The *OUTPUT_OPTIONS* in a *MSim Script* contain the 'sim_time_step' and 'n_pop_samples' fields. For example the following:-

```
OUTPUT_OPTIONS:
    sim_time_step: -1.0
    n_pop_samples: 100
```

In a *MSim Script* it is simpler to keep 'sim_time_step' set to -1.0. As you usually want the noise `c[X]` samples to be at the same time points as the original *data file* when creating a *VPC*.

The 'n_pop_samples' allows you to vary the number of population samples you collect for your *VPC*. Generally the more the better, but you have to wait longer for more samples.

34.17.3 Example OUTPUT_OPTIONS for mtut_script and mgen_script

The *OUTPUT_OPTIONS* in a *MTut Script* and *MGen Script* allow you to control the number of population samples, for example:-

```
OUTPUT_OPTIONS: {n_pop_samples: 30}
```

The more population samples the more comprehensive your data, but the longer the computation will take.

CHAPTER

THIRTYFIVE

SCRIPT FILE ELEMENTS

See Table 1 for more detail on various script file elements.

Table 1: Script File Elements

Name	Purpose
Variable Types	*PoPy* variables and their usage
Probability Distributions	Probability functions for *verbatim* sections
Matrices	Matrix functions for *verbatim* sections
Dosing Functions	Dosing functions in *DERIVATIVES* section
Analytic Compartment Functions	Analytic compartment models in *DERIVATIVES* section
Plot Types	Preset plots for use in *PoPy*
Script Nodes	Elements of a *YAML* file

35.1 Variable Types

Table 2 shows the different type of script variables that are available in the *verbatim* sections of *PoPy* scripts:-

Table 2: *PoPy* Variable Types

Type	Description	Defined	Used
`c[X]` (fit script)	*covariates* (fit)	*data file*/*PREPROCESS*	main sections
`c[X]` (gen script)	*covariates* (gen)	*EFFECTS*	main sections
`c[X]`	*observations*	*PREDICTIONS* (fit)	*PREDICTIONS* (gen)
`f[X]`	*fixed effects*	*EFFECTS*	*MODEL_PARAMS*
`r[X]`	*random effects*	*EFFECTS*	*MODEL_PARAMS*
`m[X]`	Model parameters	*MODEL_PARAMS*	main sections
`w[X]`	Workspace Variables	*MODEL_PARAMS*	*MODEL_PARAMS*
`x[NEWIND]`	First row for individual	N/A	*MODEL_PARAMS*
`d[X]`	Derivatives wrt time	*DERIVATIVES*	N/A
`s[X]`	States	*DERIVATIVES*/*STATES*	*DERIVATIVES*/*PREDICTIONS*
`x[TIME]`	Continuous time	N/A	*DERIVATIVES*
`p[X]`	Predictions	*PREDICTIONS*	*PREDICTIONS*

In Table 2, the 'Defined' column shows where a variable of particular type is first declared, typically on the **left hand side** of an '=' or '~' operator. The 'Used' column shows where a variable may be used, typically on the **right hand side** of an '=' or '~' operator.

Note the entry 'main sections' in table Table 2 above means the following sections - *MODEL_PARAMS*/*STATES*/*DERIVATIVES*/*PREDICTIONS*.

35.1.1 Parameterized Variables

There are circumstances where many compartments in the model follow the same basic structure. When using delay compartments, for example, there is typically only a flow in from the previous compartment and a flow out to the next one:

```
DERIVATIVES: |
    d[DEPOT]   = @bolus{amt:c[AMT]} - m[K]*s[DEPOT]
    d[DELAY1]  = m[K]*s[DEPOT]      - m[K]*s[DELAY1]
    d[DELAY2]  = m[K]*s[DELAY1]     - m[K]*s[DELAY2]
    d[DELAY3]  = m[K]*s[DELAY2]     - m[K]*s[DELAY3]
    d[DELAY4]  = m[K]*s[DELAY3]     - m[K]*s[DELAY4]
    d[CENTRAL] = m[K]*s[DELAY4]     - m[KE]*s[CENTRAL]
```

Expressing these models can be time-consuming and error-prone, so *PoPy* provides a means to define multiple compartments with parameters in the definitions.

To define a sequence of numbered compartments (DELAY1, DELAY2, DELAY3, and so on) use braces on the left hand side to define the index variable (e.g., "i") and the range of numbers it should take on (e.g., "1..4" to denote 1, 2, 3 and 4). This line will be replicated for every value of the index, i, in the specified range, and the braces (along with their contents) will be replaced with the corresponding value of i.

On the right hand side, you may use expressions with simple functions of the index variable (also within braces) to refer to a parameterised compartment. When i=2, for example, c[DELAY{i-1}] refers to c[DELAY1] and so on. The model above then becomes:

```
DERIVATIVES: |
    d[DEPOT]          = @bolus{amt:c[AMT]} - m[K]*s[DEPOT]
    d[DELAY1]         = m[K]*s[DEPOT]      - m[K]*s[DELAY1]

    # this line...
    d[DELAY{i:2..4}] = m[K]*s[DELAY{i-1}] - m[K]*s[DELAY{i}]

    # ...expands to, and is therefore equivalent to, these lines:
    # d[DELAY2]       = m[K]*s[DELAY1]     - m[K]*s[DELAY2]  # i = 2
    # d[DELAY3]       = m[K]*s[DELAY2]     - m[K]*s[DELAY3]  # i = 3
    # d[DELAY4]       = m[K]*s[DELAY3]     - m[K]*s[DELAY4]  # i = 4

    d[CENTRAL]        = m[K]*s[DELAY4]     - m[KE]*s[CENTRAL]
```

It is also possible to use parameterized variables in the same way within a sum() function for the sake of simplicity:

```
PREDICTIONS: |
    p[TOTAL_CONC] = sum(s[CMT{i:1..4}])
    ...
```

35.2 Probability Distributions

The distributions available for use in *PoPy* models are shown in Table 3:-

Table 3: Probability Distributions

Name	Syntax
Uniform	x ~ unif(min_x, max_x) init_x
Normal	y ~ norm(mean, var)
Censored Normal	y ~ cennorm(mean, var, LLQ=llq, ULQ=ulq)
Rectified Normal	y ~ rectnorm(mean, var, LLQ=llq, ULQ=ulq)
Truncated Normal	y ~ truncnorm(mean, var, MIN=min, MAX=max)
Truncated Censored Normal	y ~ trunccennorm(mean, var, MIN=min, LLQ=llq, ULQ=ulq, MAX=max)
Truncated Rectified Normal	y ~ truncrectnorm(mean, var, MIN=min, LLQ=llq, ULQ=ulq, MAX=max)
Multi Normal	y_vec ~ mnorm(mean_vec, var_mat)
Bernoulli	y ~ bernoulli(p)
Poisson	y ~ poisson(p)
Binomial	y ~ binomial(p, n)
Negative Binomial	y ~ negbinomial(p, n)

35.2.1 Uniform Distribution

Uniform is a continuous univariate distribution, written as:-

```
x ~ unif(min_x, max_x) init_x
```

The uniform distribution is used to define a range of values for an unknown scalar that you wish *PoPy* to estimate.

The input parameters are:-

- min_x - the **minimum** value that variable 'x' is allowed to take during estimation.
- max_x - the **maximum** value that variable 'x' is allowed to take during estimation.
- init_x - the **initial** value that variable 'x' takes at the start of estimation.

The output 'x' and inputs 'min_x', 'max_x', 'init_x' are all continuous values.

For more information see Uniform Distribution on Wikipedia.

Uniform Distribution Examples

You use the *Uniform Distribution* in the *EFFECTS* section of a *PoPy Fit Script* as follows:-

```
f[KE] ~ unif(0.1, 100) 0.05
```

The above expressions limits the `f[KE]` variable to the range [0.1, 100] with an initial starting value of 0.05.

Alternatively you can use some convenient shortcuts, for example:-

```
f[KE] ~ P 0.05
```

Where 'P' stands for +ve. The equivalent long form is:-

```
f[KE] ~ unif(1e-06, +inf) 0.05
```

Which limits `f[KE]` to be greater than 1e-06.

You can also have an unconstrained variable as follows:-

```
f[KE] ~ U 0.05
```

Where 'U' stands for unlimited. The equivalent long form is:-

```
f[KE] ~ unif(-inf, +inf) 0.05
```

35.2.2 Normal Distribution

The Normal distribution is used for continuous variables and written in *PoPy* as:-

```
x ~ norm(mean, var)
```

The Normal models a Gaussian distribution with two parameters 'mean' and 'var'.

The input parameters are:-

- mean - the expected value of the Normal
- var - the variance of the Normal

The output 'x' and inputs 'mean', 'var' are all continuous values

For more information see Normal Distribution on Wikipedia.

Normal Random Effect Example

You can use the *Normal Distribution* in the *EFFECTS* section of a *PoPy* script, to define a `r[X]` *random effect* variable as follows:-

```
EFFECTS:
    ID: |
        r[KE] ~ norm(0, f[KE_isv])
```

Here the `r[KE]` scalar variable is defined as a normal with mean zero and positive scalar variance `f[KE_isv]`.

`r[KE]` is defined at the 'ID' level, so each individual in the population has an independent sample of this normal distribution.

Normal Likelihood Example

You can use the *Normal Distribution* in the *PREDICTIONS* section of a *PoPy Fit Script* as follows:-

```
PREDICTIONS:
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[DV_CENTRAL]**2
    c[DV_CENTRAL] ~ norm(p[DV_CENTRAL], var)
```

The above syntax in a *Fit Script* specifies the likelihood of the observed `c[DV_CENTRAL]` observation from the *data file*, when modelled as a Normal variable, with mean p[DV_CENTRAL] and variance 'var'.

35.2.3 Censored Normal Distribution

The *~cennorm()* distribution is used to model whether the output of a Normal random variable lies within a particular range and is written in *PoPy* as:-

```
x ~ cennorm(mean, var, LLQ=llq, ULQ=ulq)
```

The *~cennorm()* distribution models a Censored Gaussian distribution with two parameters 'mean' and 'var' and two limit parameters 'llq' and 'ulq'.

The input parameters are:-

- mean - the expected value of the Normal
- var - the variance of the Normal
- llq - lower limit of quantification - optional parameter - default value is -inf
- ulq - upper limit of quantification - optional parameter - default value is +inf

The inputs 'mean', 'var', 'llq', 'ulq' are all continuous values. The default values above imply that the following:-

```
x ~ cennorm(mean, var)
```

Is the same as this:-

```
x ~ cennorm(mean, var, LLQ=-inf, ULQ=+inf)
```

Which is completely uninformative, as for any value of x the likelihood is one and the log likelihood contribution zero.

The 'llq' and 'ulq' values define 3 adjacent regions in the range [-inf, +inf]. When sampling from a Censored Normal, the output can take one of three values:-

- llq - represents a sample in the range [-inf, llq]
- (llq + ulq)/2 - represents a sample in the range [llq, ulq]
- ulq - represents a sample in the range [ulq, +inf]

The probability of each value is computed using the cumulative normal distribution for each range. The sum of all three range probabilities will sum to one.

For more information see Cumulative Normal Distribution on Wikipedia.

Censored Normal Likelihood Example

You can use the *~cennorm()* distribution in the *PREDICTIONS* section of a *PoPy Fit Script* to model below level of quantification (**BLQ**) data, *i.e.* observations that are **not** observed directly, but are known to be below a certain lower limit of quantification (**LLQ**), as follows:-

```
PREDICTIONS:
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[DV_CENTRAL]**2
    llq = 2.0
    if c[DV_CENTRAL] <= llq:
        c[DV_CENTRAL] ~ cennorm(p[DV_CENTRAL], var, LLQ=llq)
    else:
        c[DV_CENTRAL] ~ norm(p[DV_CENTRAL], var)
```

The above syntax in a *Fit Script* specifies the likelihood of the observed `c[DV_CENTRAL]` observation from the *data file*. `c[DV_CENTRAL]` observations greater than **LLQ** are modelled as a Standard Normal variable, with mean p[DV_CENTRAL] and proportional variance 'var'. `c[DV_CENTRAL]` observations less than **LLQ** are modelled as a cumulative normal distribution with the same mean and variance lying within the range [-inf, llq]. This **BLQ** data model is referred to as method 'M3' in [Beal2001].

Note that any value for `c[DV_CENTRAL]` in the data set less than or equal to llq is treated as a **BLQ** observation by this model.

Also note that *PoPy* requires the keyword syntax 'LLQ=llq' here to clarify the purpose of the third *~cennorm()* distribution parameter. It is also possible to model above level of quantification (**ALQ**) observations, as follows:-

```
PREDICTIONS:
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[DV_CENTRAL]**2
    ulq = 100.0
    if c[DV_CENTRAL] >= ulq:
```

(continues on next page)

(continued from previous page)

```
        c[DV_CENTRAL] ~ cennorm(p[DV_CENTRAL], var, ULQ=ulq)
    else:
        c[DV_CENTRAL] ~ norm(p[DV_CENTRAL], var)
```

Or potentially both **BLQ** and **ALQ** observations:-

```
PREDICTIONS:
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[DV_CENTRAL]**2
    llq = 2.0
    ulq = 100.0
    if c[DV_CENTRAL] <= llq or c[DV_CENTRAL] >= ulq:
        c[DV_CENTRAL] ~ cennorm(p[DV_CENTRAL], var, LLQ=llq, ULQ=ulq)
    else:
        c[DV_CENTRAL] ~ norm(p[DV_CENTRAL], var)
```

The 'if' statement above, makes it reasonably clear how *PoPy* models **BLQ** and **ALQ** data, when fitting a model, however these formulae are quite long winded and difficult to sample from, so in practice it is recommended to use a *Rectified Normal Distribution* instead, see below.

35.2.4 Rectified Normal Distribution

The *~rectnorm()* distribution combines a *~cennorm()* distribution and a *~norm()* distribution. Its primary purpose is modelling of **BLQ** and **ALQ** observations. It is written in *PoPy* as:-

```
x ~ rectnorm(mean, var, LLQ=llq, ULQ=ulq)
```

The *~rectnorm()* distribution models **BLQ** and **ALQ** *observations* using a *~cennorm()* distribution and a *~norm()* distribution for fully observed data, with shared parameters 'mean' and 'var' over the following ranges:-

- [-inf, llq] - cennorm(mean, var, LLQ=-inf, ULQ=llq)
- [llq, ulq] - norm(mean, var)
- [ulq, +inf] - cennorm(mean, var, LLQ=ulq, ULQ=+inf)

The input parameters are:-

- mean - the expected value of the Normal
- var - the variance of the Normal
- llq - lower limit of quantification - optional parameter - default value is -inf
- ulq - upper limit of quantification - optional parameter - default value is +inf

The inputs 'mean', 'var', 'llq', 'ulq' are all continuous values. The default values above imply that the following:-

```
x ~ rectnorm(mean, var)
```

Is the same as this:-

```
x ~ rectnorm(mean, var, LLQ=-inf, ULQ=+inf)
```

Which is the same as a *~norm()* distribution:-

```
x ~ norm(mean, var)
```

The 'llq' and 'ulq' values define 3 adjacent regions in the range [-inf, +inf]. When sampling from a Rectified Normal, the output can take one of three types of value:-

- llq - represents a sample in the range [-inf, llq]
- [llq, ulq] - a standard Normal sample in the range [llq, ulq]
- ulq - represents a sample in the range [ulq, +inf]

The discrete probability of a value of **LLQ** or less is computed using the cumulative normal distribution over the range [-inf, llq]. The discrete probability of a value of **ULQ** or more is computed using the cumulative normal distribution over the range [ulq, +inf]. The continous probability density function (pdf) in the range [llq, ulq] is computed from the standard Normal distribution. The area under the pdf in the range [llq, ulq] added to the **BLQ** and **ULQ** discrete probabilities sums to one.

For more information see Rectified Gaussian Distribution on Wikipedia.

Rectified Normal Likelihood Example

You can use the *~rectnorm()* distribution in the *PREDICTIONS* section of a *PoPy Fit Script* to model below level of quantification (**BLQ**) data, *i.e.* observations that are **not** observed directly, but are known to be below a certain lower limit of quantification (**LLQ**), as follows:-

```
PREDICTIONS:
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[DV_CENTRAL]**2
    llq = 2.0
    c[DV_CENTRAL] ~ rectnorm(p[DV_CENTRAL], var, LLQ=llq)
```

The above syntax in a *Fit Script* specifies the likelihood of the observed `c[DV_CENTRAL]` observation from the *data file*. `c[DV_CENTRAL]` observations greater than **LLQ** are modelled as a Standard Normal variable, with mean p[DV_CENTRAL] and proportional variance 'var'. `c[DV_CENTRAL]` observations less than **LLQ** are modelled as a cumulative normal distribution with the same mean and variance lying within the range [-inf, llq]. This **BLQ** data model is referred to as method 'M3' in [Beal2001] and recommended by [Ahn2008].

Note that any value for `c[DV_CENTRAL]` in the data set less than or equal to llq is treated as a **BLQ** observation by this model. You can vary the **LLQ** limit for each observation by specifying the limit as a separate field in the *data file*:-

```
PREDICTIONS:
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[DV_CENTRAL]**2
    c[DV_CENTRAL] ~ rectnorm(p[DV_CENTRAL], var, LLQ=c[LLQ])
```

You can then remove the **BLQ** limit for selected observations by setting `c[LLQ]` to zero or a large negative number. Sometimes a **BLQ** observation is recorded in the *data file* using a separate flag field

and the `c[DV_CENTRAL]` value itself is then the **LLQ**. In this case you could use the *PREDICTIONS* section above and *PREPROCESS* the data to compute a suitable `c[LLQ]` field as follows:-

```
PREPROCESS:
    if c[BLQ_FLAG] > 0.5:
        c[LLQ] = c[DV_CENTRAL]
    else:
        c[LLQ] = -inf
```

Also note that *PoPy* requires the keyword syntax 'LLQ=llq' here to clarify the purpose of the third *~rectnorm()* distribution parameter. It is also possible to model above level of quantification (**ALQ**) observations, as follows:-

```
PREDICTIONS:
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[DV_CENTRAL]**2
    ulq = 100.0
    c[DV_CENTRAL] ~ rectnorm(p[DV_CENTRAL], var, ULQ=ulq)
```

Or potentially both **BLQ** and **ALQ** observations:-

```
PREDICTIONS:
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[DV_CENTRAL]**2
    llq = 2.0
    ulq = 100.0
    c[DV_CENTRAL] ~ rectnorm(p[DV_CENTRAL], var, LLQ=llq, ULQ=ulq)
```

The functionality above is the same as combining a *~cennorm()* distribution and a *~norm()* distribution using an 'if' statement, see *Censored Normal Likelihood Example*. However using *~rectnorm()* distribution is recommended as it is more compact and also more flexible. For example the syntax above works in the context of a *Gen Script*, *Sim Script* or *Tut Script* as well as a *Fit Script*. *i.e.* you can sample from a *~rectnorm()* distribution easily.

The *~rectnorm()* distribution is the principle way that *PoPy* modellers are encouraged to deal with **BLQ** and **ALQ** data.

35.2.5 Truncated Normal Distribution

The *~truncnorm()* distribution is based on the *~norm()* distribution, but with the domain of the distribution limited to a range [min,max]. It can be used to restrict a *~norm()* distribution to say all positive values. It is written in *PoPy* as:-

```
x ~ truncnorm(mean, var, MIN=min, MAX=max)
```

The input parameters are:-

- mean - the expected value of the Normal
- var - the variance of the Normal
- min - minimum value of truncated range - optional parameter - default value is -inf

- max - maximum value of truncated range - optional parameter - default value is +inf

The inputs 'mean', 'var', 'min', 'max' are all continuous values. The default values above imply that the following:-

```
x ~ truncnorm(mean, var)
```

Is the same as this:-

```
x ~ truncnorm(mean, var, MIN=-inf, MAX=+inf)
```

Which is the same as a *~norm()* distribution:-

```
x ~ norm(mean, var)
```

Note the *~truncnorm()* distribution is different from the *~rectnorm()* distribution. A *~truncnorm()* distribution rescales its probability density function, so that the area under the curve in the domain [min, max] is one. There is zero probability mass outside of the [min, max] range. A *~rectnorm()* distribution keeps the same probability density function as the *~norm()* distribution within the range [llq, ulq], but includes the cumulative probability outside this region to achieve a total probality of one.

For more information see Truncated Normal Distribution on Wikipedia.

Truncated Normal Likelihood Example

You can use the *~truncnorm()* distribution in the *PREDICTIONS* section of a *PoPy Fit Script* to model data that is known to occur in a certain range, *e.g.* all positive data:-

```
PREDICTIONS:
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[DV_CENTRAL]**2
    c[DV_CENTRAL] ~ truncnorm(p[DV_CENTRAL], var, MIN=0)
```

The above syntax in a *Fit Script* specifies the likelihood of the observed `c[DV_CENTRAL]` observation from the *data file*.

Also note that *PoPy* requires the keyword syntax 'MIN=min' here to clarify the purpose of the third *~truncnorm()* distribution parameter. It is also possible to model observations with a known upper limit, *e.g.* data that is known to be negative, as follows:-

```
PREDICTIONS:
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[DV_CENTRAL]**2
    c[DV_CENTRAL] ~ truncnorm(p[DV_CENTRAL], var, MAX=0)
```

Or potentially observations that are known to be within a single standard deviation:-

```
PREDICTIONS:
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[DV_CENTRAL]**2
    std = sqrt(var)
    min = p[DV_CENTRAL] - std
```

(continues on next page)

(continued from previous page)

```
    max = p[DV_CENTRAL] + std
    c[DV_CENTRAL] ~ truncnorm(p[DV_CENTRAL], var, MIN=min, MAX=max)
```

Note that if values of `c[DV_CENTRAL]` lie outside the range [min, max] then this model will make little sense, as the likelihood of such observations are zero and the loglikelihood is -inf.

You might wish to use *~truncnorm()* distribution to generate synthetic positive only observations from a model. The alternative, is possibly to use *~rectnorm()* distribution and generate sythetic data with a small positive **LLQ**.

35.2.6 Truncated Censored Normal Distribution

The *~trunccennorm()* distribution is based on the *~cennorm()* distribution, but with the domain of the distribution limited to a range [min,max]. It can be used to restrict a *~cennorm()* distribution to say all positive values. It is written in *PoPy* as:-

```
x ~ trunccennorm(mean, var, MIN=min, LLQ=llq, ULQ=ulq, MAX=max)
```

The input parameters are:-

- mean - the expected value of the Normal
- var - the variance of the Normal
- min - minimum value of truncated range - optional parameter - default value is -inf
- llq - lower limit of quantification - optional parameter - default value is -inf
- ulq - upper limit of quantification - optional parameter - default value is +inf
- max - maximum value of truncated range - optional parameter - default value is +inf

The inputs 'mean', 'var', 'min', 'llq', 'ulq', 'max' are all continuous values. The default values above imply that the following:-

```
x ~ trunccennorm(mean, var)
```

Is the same as this:-

```
x ~ trunccennorm(mean, var, MIN=-inf, LLQ=-inf, ULQ=+inf, MAX=+inf)
```

Which is completely uninformative, as for any value of x the likelihood is one and the log likelihood contribution zero.

Note the *~trunccennorm()* distribution rescales a *~cennorm()* distribution, so that the area under the curve in the domain [min, max] is one. There is zero probability mass outside of the [min, max] range. Effectively the range [-inf,+inf] is split into 5 sub ranges:-

- [-inf, min] - zero probability
- [min, llq] - cumulative normal probability
- [llq, ulq] - cumulative normal probability
- [ulq, max] - cumulative normal probability

- [max, +inf] - zero probability

Truncated Censored Normal Likelihood Example

You can use the *~trunccennorm()* distribution in the *PREDICTIONS* section of a *PoPy Fit Script* to model data that is known to occur in a certain range, *e.g.* all positive data:-

```
PREDICTIONS:
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[DV_CENTRAL]**2
    if c[DV_CENTRAL] <= llq:
        c[DV_CENTRAL] ~ trunccennorm(p[DV_CENTRAL], var, MIN=0, LLQ=2.0)
    else:
        c[DV_CENTRAL] ~ truncnorm(p[DV_CENTRAL], var, MIN=0)
```

The above syntax in a *Fit Script* specifies the likelihood of the `c[DV_CENTRAL]` known positive observation from the *data file*, with a **LLQ** of 2.0.

The model above shows how you might implement the 'M4' method described in [Beal2001], which conditions on the **BLQ** data being positive. However a more convenient notation for doing this is described in *~truncrectnorm()* distribution below.

35.2.7 Truncated Rectified Normal Distribution

The *~truncrectnorm()* distribution is based on the *~rectnorm()* distribution, but with the domain of the distribution limited to a range [min,max]. It can be used to restrict a *~rectnorm()* distribution to say all positive values. It is written in *PoPy* as:-

```
x ~ truncrectnorm(mean, var, MIN=min, LLQ=llq, ULQ=ulq, MAX=max)
```

The input parameters are:-

- mean - the expected value of the Normal
- var - the variance of the Normal
- min - minimum value of truncated range - optional parameter - default value is -inf
- llq - lower limit of quantification - optional parameter - default value is -inf
- ulq - upper limit of quantification - optional parameter - default value is +inf
- max - maximum value of truncated range - optional parameter - default value is +inf

The inputs 'mean', 'var', 'min', 'llq', 'ulq', 'max' are all continuous values. The default values above imply that the following:-

```
x ~ truncrectnorm(mean, var)
```

Is the same as this:-

```
x ~ truncrectnorm(mean, var, MIN=-inf, LLQ=-inf, ULQ=+inf, MAX=+inf)
```

Which is the same as a *~norm()* distribution:-

```
x ~ norm(mean, var)
```

Note the *~truncrectnorm()* distribution rescales a *~rectnorm()* distribution, so that the area under the curve in the domain [min, max] is one. There is zero probability mass outside of the [min, max] range. Effectively the range [-inf,+inf] is split into 5 sub ranges:-

- [-inf, min] - zero probability
- [min, llq] - cumulative normal probability
- [llq, ulq] - standard normal probability
- [ulq, max] - cumulative normal probability
- [max, +inf] - zero probability

Truncated Rectified Normal Likelihood Example

You can use the *~truncrectnorm()* distribution in the *PREDICTIONS* section of a *PoPy Fit Script* to model data that is known to occur in a certain range, *e.g.* all positive data:-

```
PREDICTIONS:
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[DV_CENTRAL]**2
    c[DV_CENTRAL] ~ truncrectnorm(p[DV_CENTRAL], var, MIN=0, LLQ=2.0)
```

The above syntax in a *Fit Script* specifies the likelihood of the `c[DV_CENTRAL]` known positive observation from the *data file*, with a **LLQ** of 2.0.

The model above shows the recommend way for *PoPy* modellers to implement the 'M4' method described in [Beal2001], which conditions on the **BLQ** data being positive.

The *~truncrectnorm()* distribution is easier to sample from and therefore use in a *Gen Script*, *Tut Script* and *Sim Script* compared to the 'if' statment method show in *Truncated Censored Normal Likelihood Example*.

Note in many cases it may be easier and more appropriate to use the 'M3' method and the *~rectnorm()* distribution shown in *Rectified Normal Likelihood Example*.

35.2.8 Multivariate Normal Distribution

Multivariate-Normal distribution is used for vectors of continuous variables and written like this:-

```
output_vector ~ mnorm(mean_vector, covariance_matrix)
```

The Multivariate Normal is a generalisation of the *Normal Distribution* with two parameters 'mean_vector' and 'covariance_matrix', as follows:-

- mean_vector - the mean of the 'output_vector'
- covariance_matrix - the covariance of the 'output_vector' elements

The 'output_vector' must have the same number of dimensions as the 'mean_vector'. Also the 'covariance_matrix' needs to be *symmetric positive definite* with a matching dimensionality. See *Matrices* for examples of how to define the covariance matrix.

For more information see Multivariate Normal Distribution on Wikipedia.

Multivariate Normal Random Effect Example

You can use the *Multivariate Normal Distribution* in the *EFFECTS* section of a *PoPy* script, to define a vector of `r[X]` *random effects* variables as follows:-

```
EFFECTS:
    ID: |
        r[KA,CL,V] ~ mnorm([0, 0, 0], f[KA_isv,CL_isv,V_isv])
```

Here the `r[KA,CL,V]` variable is defined as a 3 element vector with mean zero. `[0,0,0]` is a 3 element 'mean_vector' and `f[KA_isv,CL_isv,V_isv]` is a 3x3 'covariance_matrix'. The `f[KA_isv,CL_isv,V_isv]` matrix can be a diagonal or square symmetric matrix, see *Matrices*.

The `r[KA,CL,V]` is defined at the 'ID' level, so each individual in the population has an independent sample of this multivariate normal distribution.

35.2.9 Bernoulli Distribution

The Bernoulli is univariate discrete distribution used to model binary variables, and written in *PoPy* as:-

```
y ~ bernoulli(prob_success)
```

The Bernoulli models the distribution of a single Bernoulli trial.

The input parameters are:-

- prob_success - probability of success of the bernouilli trial

The output 'y' is a binary value, *i.e.* either 1 for success or 0 for failure. 'prob_success' is a real valued number in the range [0,1].

For more information see Bernoulli Distribution on Wikipedia.

Bernoulli Likelihood Example

You can use the *Bernoulli Distribution* in the *PREDICTIONS* section of a *PoPy Fit Script* as follows:-

```
PREDICTIONS:
    conc = s[X]/m[V]
    p[DV_BERN] = 1.0 / (1.0+ exp(-conc))
    c[DV_BERN] ~ bernoulli(p[DV_BERN])
```

The above syntax in a *Fit Script* specifies the likelihood of the observed `c[DV_BERN]` binary observation from the *data file*, when modelled as a Bernoulli variable, with success rate dependent on 'conc' via a logistic transform.

35.2.10 Poisson Distribution

The Poisson is a discrete univariate distribution, to model discrete count variables, written in *PoPy* as:-

```
y ~ poisson(lambda)
```

The Poisson models the distribution of the number of events occurring within a fixed time interval, if each individual event occurs independently and at constant rate 'lambda'.

The input parameters are:-

- lambda - the expected number of occurrences within the time interval

The output 'y' is the observed count, *i.e.* a non-negative integer value. 'lambda' is a positive real valued number, which represents the mean rate of event occurrence.

For more information see Poisson Distribution on Wikipedia.

Poisson Likelihood Example

You can use the *Poisson Distribution* in the *PREDICTIONS* section of a *PoPy Fit Script* as follows:-

```
PREDICTIONS:
    c[COUNT] ~ poisson(m[LAMBDA])
```

The above syntax in a *Fit Script* specifies the likelihood of the observed `c[COUNT]` count observations from the *data file*, when modelled as a Poisson process with estimated rate parameter `m[LAMBDA]`.

35.2.11 Binomial Distribution

The binomial is a univarite discrete distribution, written in *PoPy* as:-

```
num_successes ~ binomial(prob_success, num_trials)
```

The binomial models the distribution of the number of successes given a fixed number of independent *Bernoulli* trials.

The input parameters are:-

- prob_success - probability of success of each bernouilli trial
- num_trials - number of bernouilli trials

Here the output 'num_successes' is an integer. 'num_trials' is also an integer and 'prob_success' is a real valued number in the range [0,1].

For more information see Binomial Distribution on Wikipedia.

Binomial Likelihood Example

You can use the *Binomial Distribution* in *PREDICTIONS* section of a *PoPy Fit Script* as follows:-

```
PREDICTIONS:
    conc = s[X]/m[V]
    p[DV_B] = 1.0 / (1.0 + exp(-conc))
    c[DV_B] ~ binomial(p[DV_B], c[N_OBS])
```

The above syntax in a *Fit Script* specifies the likelihood of the observed `c[DV_B]` count data from the *data file* when modelled as the number of successes of `c[N_OBS]` trials performed. Here success rate is dependent on 'conc' via a logistic transform.

35.2.12 Negative Binomial Distribution

The negative binomial is a univarite discrete distribution, written in *PoPy* as:-

```
num_fails ~ negbinomial(prob_success, num_of_successes)
```

The negative binomial models the distribution of the number of failures for a series of independent *Bernoulli* trials until the success count reaches 'num_of_successes'.

The input parameters are:-

- prob_success - probability of success of each bernouilli trial
- num_of_successes - number of successful bernouilli trials before num_fails output

Here the output 'num_fails' is an integer. 'num_of_successes' is also an integer and 'prob_success' is a real valued number in the range [0,1].

For more information see Negative Binomial Distribution on Wikipedia. However, at the time of writing, the wikipedia page inverts the definition of success/failure. In practice there are many ways of parameterising the negative binomial parameterisation, *PoPy* uses the SciPy parameterisation described here:-

https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.nbinom.html

Negative Binomial Likelihood Example

You can use the *Negative Binomial Distribution* in *PREDICTIONS* section of a *PoPy Fit Script* as follows:-

```
PREDICTIONS:
    conc = s[X]/m[V]
    p[DV_NB] = 1.0 / (1.0 + exp(-conc))
    c[DV_NB] ~ negbinomial(p[DV_NB], 1)
```

The above syntax in a *Fit Script* specifies the likelihood of the observed `c[DV_NB]` count data from the *data file* when modelled as the number of failures of a Bernoulli variable (with success rate dependent on 'conc' via a logistic transform) until the occurrence of the first success.

35.3 Matrices

The matrices available for use in *PoPy* models are shown in Table 4:-

Table 4: Matrices

Name	Syntax
Constant Diagonal Matrix	y_mat = [[x(1,1), x(2,2)]]
Constant Square Matrix	y_mat = [[x(1,1)], [x(2,1), x(2,2)]]
SPD Diagonal Matrix	y_mat ~ diag_matrix() [[x(1,1), x(2,2)]]
SPD Square Matrix	y_mat ~ spd_matrix() [[x(1,1)], [x(1,2), x(2,2)]]

35.3.1 Constant Diagonal Matrix

A constant nxn diagonal matrix is specified using the following syntax:-

```
y_mat = [[x(1,1), x(2,2), ... , x(n,n)]]
```

Where:-

- y_mat is a multi-element `f[X]` matrix label
- x(1,1) is the first element of the diagonal
- x(n,n) is the last element of the diagonal

Note: A diagonal matrix definition in *PoPy* starts with **double** opening square brackets '[[' and ends with matching **double** closing square brackets ']]'.

This is to distinguish from a *list* which uses **single** square brackets.

An example diagonal matrix is shown below:-

```
f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv] = [
        [ 0.1, 0.03, 0.09, 0.07, 0.05]
    ]
```

Here a 5x5 matrix called `f[KA_isv, CL_isv, V1_isv, Q_isv, V2_isv]` is created with the following values:-

$$\begin{pmatrix} 0.1 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.03 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.09 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.07 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.05 \end{pmatrix}$$

In the definition above the number of `f[X]` labels on the **left hand side** must correspond to the number of list elements along the diagonal on the **right hand side**. *i.e.* in this case they both contain 5 elements.

35.3.2 Constant Square Symmetric Matrix

A constant nxn square symmetric matrix is specified using the following syntax:-

```
y_mat = [
    [x(1,1)],
    [x(2,1), x(2,2)],
    ...
    [x(n,1), x(n,2) ... , x(n,n)]
]
```

Where:-

- y_mat is a multi-element `f[X]` matrix label
- x(1,1) is the first element of the diagonal
- x(i,j) defines the element of the ith row and jth column
- x(n,n) is the last element of the diagonal

The definition above just requires the lower triangular elements to be defined, because the matrix is assumed to be square and symmetric.

An example of defining a square symmetric matrix is shown below:-

```
f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv] = [
        [0.1],
        [0.01, 0.03],
        [0.01, -0.01, 0.09],
        [0.01, 0.02, 0.01, 0.07],
        [0.01, 0.02, 0.01, 0.01, 0.05],
    ]
```

The number of elements in each row of the matrix definition need to be [1,2,3,4,5], because the `f[X]` variable contains 5 elements.

Therefore a 5x5 matrix called `f[KA_isv, CL_isv, V1_isv, Q_isv, V2_isv]` is created with the following values:-

$$\begin{pmatrix} 0.1 & 0.01 & 0.01 & 0.01 & 0.01 \\ 0.01 & 0.03 & -0.01 & 0.02 & 0.02 \\ 0.01 & -0.01 & 0.09 & 0.01 & 0.01 \\ 0.01 & 0.02 & 0.01 & 0.07 & 0.01 \\ 0.01 & 0.02 & 0.01 & 0.01 & 0.05 \end{pmatrix}$$

Note you can also define the same matrix as follows:-

```
f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv] = [
        [0.1 , 0.01,  0.01, 0.01, 0.01],
        [0.01, 0.03, -0.01, 0.02, 0.02],
        [0.01, -0.01, 0.09, 0.01, 0.01],
        [0.01, 0.02,  0.01, 0.07, 0.01],
        [0.01, 0.02,  0.01, 0.01, 0.05],
    ]
```

However this format risks accidentally creating a non-symmetric matrix.

In *PoPy*, you can only specify square symmetric matrices. If the input matrix is non-square *PoPy* will report an error. If the input matrix is square, but non-symmetric then *PoPy* will average the upper and lower triangles and output a warning.

35.3.3 Positive Definite Diagonal Matrix

A positive definite nxn diagonal matrix for estimation is specified using the following syntax:-

```
y_mat ~ diag_matrix() [[x(1,1), x(2,2), ... , x(n,n)]]
```

Where:-

- y_mat is a multi-element `f[X]` matrix label
- x(1,1) is the first element of the diagonal
- x(n,n) is the last element of the diagonal

Note: The '~' notation means that 'y_mat' is a diagonal matrix to be estimated, and only initialised with the diagonal matrix specified in square brackets.

An example diagonal matrix for estimation in a *Fit Script* is shown below:-

```
f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv] ~ diag_matrix() [
        [ 0.1, 0.03, 0.09, 0.07, 0.05]
    ]
```

Here a 5x5 matrix variable `f[KA_isv, CL_isv, V1_isv, Q_isv, V2_isv]` is initialised with the following values:-

$$\begin{pmatrix} 0.1 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.03 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.09 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.07 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.05 \end{pmatrix}$$

In the fitting process the diagonal elements are constrained to be positive and the off diagonal terms are fixed to zero.

Note: This matrix is defined using the '~' notation, however it is **not** a true distribution in the sense that if you sample from the matrix distribution, only the current value is returned. This fact is only relevant for a *Gen Script* or *MGen Script*.

35.3.4 Symmetric Positive Definite Matrix

A *symmetric positive definite* nxn square symmetric matrix for estimation is specified using the following syntax:-

```
y_mat ~ spd_matrix() [
    [x(1,1)],
    [x(2,1), x(2,2)],
    ...
    [x(n,1), x(n,2) ... , x(n,n)]
]
```

Where:-

- y_mat is a multi-element `f[X]` matrix label
- x(1,1) is the first element of the diagonal
- x(i,j) defines the element of the ith row and jth column
- x(n,n) is the last element of the diagonal

An example of defining a square symmetric matrix to be estimated is shown below:-

```
f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv] ~ spd_matrix() [
        [0.1],
        [0.01, 0.03],
        [0.01, -0.01, 0.09],
        [0.01, 0.02, 0.01, 0.07],
        [0.01, 0.02, 0.01, 0.01, 0.05],
    ]
```

Here we are only defining the lower triangle elements, as the upper triangle elements are the same in a symmetric matrix. The number of elements in each row of the initial matrix definition needs to be [1,2,3,4,5], because the `f[X]` variable contains 5 elements.

We are declaring a 5x5 matrix called `f[KA_isv, CL_isv, V1_isv, Q_isv, V2_isv]`, that will be estimated by a *Fit Script* and is constrained to be *symmetric positive definite*. The matrix is initialised with the following values:-

$$\begin{pmatrix} 0.1 & 0.01 & 0.01 & 0.01 & 0.01 \\ 0.01 & 0.03 & -0.01 & 0.02 & 0.02 \\ 0.01 & -0.01 & 0.09 & 0.01 & 0.01 \\ 0.01 & 0.02 & 0.01 & 0.07 & 0.01 \\ 0.01 & 0.02 & 0.01 & 0.01 & 0.05 \end{pmatrix}$$

Note that the symmetric *~spd_matrix()* distribution is initialised here by defining just the lower triangular elements (without loss of generality), but it can also be initialised by carefully defining all 5*5 elements.

35.3.5 Covariance Matrix Usage

The most common use for matrices as defined above is to define a covariance matrix for a *Multivariate Normal Distribution*. For example in a *EFFECTS* section:-

```
EFFECTS:
    ID: |
        r[KA,CL,V1,Q,V2] ~ mnorm(
            [0, 0, 0, 0, 0], f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv] )
```

Here the matrix `f[KA_isv, CL_isv, V1_isv, Q_isv, V2_isv]`, can be defined in any of the ways specified above, *i.e.* constant/variable or square/diagonal.

In the definition of `r[KA, CL, V1, Q, V2]`, the size of the variance matrix must agree with the size of the random vector and the mean vector of the *~mnorm()* distribution. See *Multivariate Normal Random Effect Example* for more information.

35.4 Dosing Functions

Table 5 shows the dosing functions that are available in the *DERIVATIVES* section and their parameters:-

Table 5: Dosing Functions

Name	Parameters
@bolus	amt/lag
@inf_rate	amt/lag/rate
@inf_dur	amt/lag/dur
@gamma	amt/lag/alpha/beta
@weibull	amt/lag/lambda/kappa

35.4.1 @bolus

Example of *Bolus dosing*:-

```
DERIVATIVES: |
    d[DEPOT] = @bolus{amt: c[AMT], lag: m[LAG]} - m[KE] * s[DEPOT]
```

This bolus dose will be activated by the 'dose' entry in the *TYPE* field of the *PoPy data file*. Note it is also possible to specify a **named** bolus dose as follows:-

```
DERIVATIVES: |
    dose[my_bolus] = @bolus{amt: c[AMT], lag: m[LAG]}
    d[DEPOT] = dose[my_bolus] * s[DEPOT]
```

This bolus dose will be activated by the 'dose:my_bolus' entry in the *TYPE* field of the *PoPy data file*.

35.4.2 @inf_rate

Example of *infusion rate* dosing:-

```
DERIVATIVES: |
    d[DEPOT] = (
        @inf_rate{amt: c[AMT], lag: m[LAG], rate: c[RATE]}
        - m[KE] * s[DEPOT])
```

This infusion rate dose will be activated by the 'dose' entry in the *TYPE* field of the *PoPy data file*. Note it is also possible to specify a **named** infusion rate dose as follows:-

```
DERIVATIVES: |
    dose[my_inf_rate] = @inf_rate{amt: c[AMT], lag: m[LAG], rate: c[RATE]}
    d[DEPOT] = dose[my_inf_rate] * s[DEPOT]
```

This infusion rate dose will be activated by the 'dose:my_inf_rate' entry in the *TYPE* field of the *PoPy data file*.

35.4.3 @inf_dur

Example of *infusion duration* dosing:-

```
DERIVATIVES: |
    d[DEPOT] = (
        @inf_dur{amt: c[AMT], lag: m[LAG], dur: c[DUR]}
        - m[KE] * s[DEPOT])
```

This infusion duration dose will be activated by the 'dose' entry in the *TYPE* field of the *PoPy data file*. Note it is also possible to specify a **named** infusion duration dose as follows:-

```
DERIVATIVES: |
    dose[my_inf_dur] = @inf_dur{
        amt: c[AMT], lag: m[LAG], dur: c[DUR]}
    d[DEPOT] = dose[my_inf_dur] * s[DEPOT]
```

This infusion duration dose will be activated by the 'dose:my_inf_dur' entry in the *TYPE* field of the *PoPy data file*.

35.4.4 @gamma

Example of *gamma* dosing:-

```
DERIVATIVES: |
    d[DEPOT] = (
        @gamma{amt: c[AMT], lag: m[LAG],
            alpha: m[ALPHA], beta: m[BETA]}
        - m[KE] * s[DEPOT]
    )
```

This gamma dose will be activated by the 'dose' entry in the *TYPE* field of the *PoPy data file*. Note it is also possible to specify a **named** gamma dose as follows:-

```
DERIVATIVES: |
    dose[my_gamma] = @gamma{
        amt: c[AMT], lag: m[LAG],
        alpha: m[ALPHA], beta: m[BETA]}
    d[DEPOT] = dose[my_gamma] * s[DEPOT]
```

This gamma dose will be activated by the 'dose:my_gamma' entry in the *TYPE* field of the *PoPy data file*.

35.4.5 @weibull

Example of *weibull* dosing:-

```
DERIVATIVES: |
    d[DEPOT] = (
        @weibull{
            amt: c[AMT], lag: m[LAG],
            lambda: m[LAMBDA], kappa: m[KAPPA]}
        - m[KE] * s[DEPOT]
    )
```

This weibull dose will be activated by the 'dose' entry in the *TYPE* field of the *PoPy data file*. Note it is also possible to specify a **named** weibull dose as follows:-

```
DERIVATIVES: |
    dose[my_weibull] = @weibull{
        amt: c[AMT], lag: m[LAG],
        lambda: m[LAMBDA], kappa: m[KAPPA]}
    d[DEPOT] = dose[my_weibull] * s[DEPOT]
```

This weibull dose will be activated by the 'dose:my_weibull' entry in the *TYPE* field of the *PoPy data file*.

35.5 Analytic Compartment Functions

Table 6 shows the inbuilt compartment functions that are available in the *DERIVATIVES* section using the '_cl' suffix:-

Table 6: Compartment Model Functions using '_cl'

Name	Parameters
@iv_one_cmp_cl	dose/CL/V
@dep_one_cmp_cl	dose/KA/CL/V
@iv_two_cmp_cl	dose/CL/V1/Q/V2
@dep_two_cmp_cl	dose/KA/CL/V1/Q/V2
@iv_three_cmp_cl	dose/CL/V1/Q2/V2/Q3/V3
@dep_three_cmp_cl	dose/KA/CL/V1/Q2/V2/Q3/V3

The '_cl' suffix means that elimination rates between compartments (K) are generally parameterised as follows:-

$$K = CL/V$$

i.e. the ratio of the *clearance* and *volume of distribution*.

35.5.1 @iv_one_cmp_cl

Intra-venous one compartment model:-

```
DERIVATIVES: |
    s[CEN] = @iv_one_cmp_cl{
        dose: @bolus{amt:c[AMT]},
        CL: m[CL], V: m[V]}
```

This analytic model is equivalent to solving **numerically**:-

```
DERIVATIVES: |
    d[CEN] = @bolus{amt:c[AMT]} - s[CEN]*m[CL]/m[V]
```

See *One Compartment Model with Intravenous Dosing* for full example.

35.5.2 @dep_one_cmp_cl

Depot and one compartment model:-

```
DERIVATIVES: |
    s[DEP,CEN] = @dep_one_cmp_cl{
        dose: @bolus{amt:c[AMT]},
        KA: m[KA], CL: m[CL], V: m[V]}
```

Numerical *ordinary differential equation* equivalent is:-

```
DERIVATIVES: |
    d[DEP] = @bolus{amt:c[AMT]} - s[DEP]*m[KA]
    d[CEN] = s[DEP]*m[KA] - s[CEN]*m[CL]/m[V]
```

See *One Compartment Model with Absorption* for full example.

35.5.3 @iv_two_cmp_cl

Intra-venous two compartment model:-

```
DERIVATIVES: |
    s[CEN,PERI] = @iv_two_cmp_cl{
        dose: @bolus{amt:c[AMT]},
        CL: m[CL], V1: m[V1],
        Q: m[Q], V2: m[V2]}
```

Numerical *ordinary differential equation* equivalent is:-

```
DERIVATIVES: |
    d[CEN] = (
        @bolus{amt:c[AMT]} - s[CEN]*m[CL]/m[V1]
        - s[CEN]*m[Q]/m[V1] + s[PERI]*m[Q]/m[V2]
    )
    d[PERI] = s[CEN]*m[Q]/m[V1] - s[PERI]*m[Q]/m[V2]
```

See *Two Compartment Model with Intravenous Dosing* for full example.

35.5.4 @dep_two_cmp_cl

Depot and two compartment model:-

```
DERIVATIVES: |
    s[DEP,CEN,PERI] = @dep_two_cmp_cl{
        dose: @bolus{amt:c[AMT]},
        KA: m[KA],
        CL: m[CL], V1: m[V1],
        Q: m[Q], V2: m[V2]}
```

Numerical *ordinary differential equation* equivalent is:-

```
DERIVATIVES: |
    d[DEP] = @bolus{amt:c[AMT]} - s[DEP]*m[KA]
    d[CEN] = (
        s[DEP]*m[KA] - s[CEN]*m[CL]/m[V1]
        - s[CEN]*m[Q]/m[V1] + s[PERI]*m[Q]/m[V2]
    )
    d[PERI] = s[CEN]*m[Q]/m[V1] - s[PERI]*m[Q]/m[V2]
```

See *Two Compartment Model with Absorption* for full example.

35.5.5 @iv_three_cmp_cl

Intra-venous three compartment model:-

```
DERIVATIVES: |
    s[CEN,PERI1,PERI2] = @iv_three_cmp_cl{
        dose: @bolus{amt:c[AMT]},
        CL: m[CL], V1: m[V1],
        Q2: m[Q2], V2: m[V2],
        Q3: m[Q3], V3: m[V3]
    }
```

Numerical *ordinary differential equation* equivalent is:-

```
DERIVATIVES: |
    d[CEN] = (
        @bolus{amt:c[AMT]}  - s[CEN]*m[CL]/m[V1]
        - s[CEN]*m[Q2]/m[V1] + s[PERI1]*m[Q2]/m[V2]
        - s[CEN]*m[Q3]/m[V1] + s[PERI2]*m[Q3]/m[V3]
    )
    d[PERI1] = s[CEN]*m[Q2]/m[V1] - s[PERI1]*m[Q2]/m[V2]
    d[PERI2] = s[CEN]*m[Q3]/m[V1] - s[PERI2]*m[Q3]/m[V3]
```

See *Three Compartment Model with Intravenous Dosing* for full example.

35.5.6 @dep_three_cmp_cl

Depot and three compartment model:-

```
DERIVATIVES: |
    s[DEP,CEN,PERI1,PERI2] = @dep_three_cmp_cl{
        dose: @bolus{amt:c[AMT]},
        KA: m[KA],
        CL: m[CL], V1: m[V1],
        Q2: m[Q2], V2: m[V2],
        Q3: m[Q3], V3: m[V3]
    }
```

Numerical *ordinary differential equation* equivalent is:-

```
DERIVATIVES: |
    d[DEP] = @bolus{amt:c[AMT]} - s[DEP]*m[KA]
    d[CEN] = (
        s[DEP]*m[KA]  - s[CEN]*m[CL]/m[V1]
        - s[CEN]*m[Q2]/m[V1] + s[PERI1]*m[Q2]/m[V2]
        - s[CEN]*m[Q3]/m[V1] + s[PERI2]*m[Q3]/m[V3]
    )
    d[PERI1] = s[CEN]*m[Q2]/m[V1] - s[PERI1]*m[Q2]/m[V2]
    d[PERI2] = s[CEN]*m[Q3]/m[V1] - s[PERI2]*m[Q3]/m[V3]
```

See *Three Compartment Model with Absorption* for full example.

It is also possible to parametrise the rates directly using the '_k' suffix, see Table 7:-

Table 7: Compartment Model Functions using '_k'

Name	Parameters
@iv_one_cmp_k	dose/KE
@dep_one_cmp_k	dose/KA/KE
@iv_two_cmp_k	dose/KE/K12/K21
@dep_two_cmp_k	dose/KA/KE/K12/K21
@iv_three_cmp_k	dose/KE/K12/K21/K13/K31
@dep_three_cmp_cl	dose/KA/KE/K12/K21/K13/K31

35.5.7 @iv_one_cmp_k

Intra-venous one compartment model:-

```
DERIVATIVES: |
    s[CEN] = @iv_one_cmp_k{
        dose: @bolus{amt:c[AMT]},
        KE: m[KE]}
```

Numerical *ordinary differential equation* equivalent is:-

```
DERIVATIVES: |
    d[CEN] = @bolus{amt:c[AMT]} - s[CEN]*m[KE]
```

35.5.8 @dep_one_cmp_k

Depot and one compartment model:-

```
DERIVATIVES: |
    s[DEP,CEN] = @dep_one_cmp_k{
        dose: @bolus{amt:c[AMT]},
        KA: m[KA], KE: m[KE]}
```

Numerical *ordinary differential equation* equivalent is:-

```
DERIVATIVES: |
    d[DEP] = @bolus{amt:c[AMT]} - s[DEP]*m[KA]
    d[CEN] = s[DEP]*m[KA] - s[CEN]*m[KE]
```

35.5.9 @iv_two_cmp_k

Intra-venous two compartment model:-

```
DERIVATIVES: |
    s[CEN,PERI] = @iv_two_cmp_k{
        dose: @bolus{amt:c[AMT]},
        KE: m[KE], K12: m[K12], K21: m[K21]}
```

Numerical *ordinary differential equation* equivalent is:-

```
DERIVATIVES: |
    d[CEN] = (
        @bolus{amt:c[AMT]} - s[CEN]*m[KE]
        - s[CEN]*m[K12] + s[PERI]*m[K21]
    )
    d[PERI] = s[CEN]*m[K12] - s[PERI]*m[K21]
```

35.5.10 @dep_two_cmp_k

Depot and two compartment model:-

```
DERIVATIVES: |
    s[DEP,CEN,PERI] = @dep_two_cmp_k{
        dose: @bolus{amt:c[AMT]},
        KA: m[KA], KE: m[KE],
        K12: m[K12], K21: m[K21]}
```

Numerical *ordinary differential equation* equivalent is:-

```
DERIVATIVES: |
    d[DEP] = @bolus{amt:c[AMT]} - s[DEP]*m[KA]
    d[CEN] = (
        s[DEP]*m[KA] - s[CEN]*m[KE]
        - s[CEN]*m[K12] + s[PERI]*m[K21]
    )
    d[PERI] = s[CEN]*m[K12] - s[PERI]*m[K21]
```

35.5.11 @iv_three_cmp_k

Intra-venous three compartment model:-

```
DERIVATIVES: |
    s[CEN,PERI1,PERI2] = @iv_three_cmp_k{
        dose: @bolus{amt:c[AMT]},
        KE: m[KE],
        K12: m[K12], K21: m[K21],
        K13: m[K13], K31: m[K31]}
```

Numerical *ordinary differential equation* equivalent is:-

```
DERIVATIVES: |
    d[CEN] = (
        @bolus{amt:c[AMT]} - s[CEN]*m[KE]
        - s[CEN]*m[K12] + s[PERI1]*m[K21]
        - s[CEN]*m[K13] + s[PERI2]*m[K31]
    )
    d[PERI1] = s[CEN]*m[K12] - s[PERI1]*m[K21]
    d[PERI2] = s[CEN]*m[K13] - s[PERI2]*m[K31]
```

35.5.12 @dep_three_cmp_k

Depot and three compartment model:-

```
DERIVATIVES: |
    s[DEP,CEN,PERI1,PERI2] = @dep_three_cmp_k{
        dose: @bolus{amt:c[AMT]},
        KA: m[KA], KE: m[KE],
        K12: m[K12], K21: m[K21],
        K13: m[K13], K31: m[K31]}
```

Numerical *ordinary differential equation* equivalent is:-

```
DERIVATIVES: |
    d[DEP] = @bolus{amt:c[AMT]} - s[DEP]*m[KA]
    d[CEN] = (
        s[DEP]*m[KA] - s[CEN]*m[KE]
        - s[CEN]*m[K12] + s[PERI1]*m[K21]
        - s[CEN]*m[K13] + s[PERI2]*m[K31]
    )
    d[PERI1] = s[CEN]*m[K12] - s[PERI1]*m[K21]
    d[PERI2] = s[CEN]*m[K13] - s[PERI2]*m[K31]
```

35.6 Plot Types

PoPy's grph and vpc are highly configurable, but for everyday use *PoPy* comes with a variety of preset plots that will be familiar to those working in PK/PD.

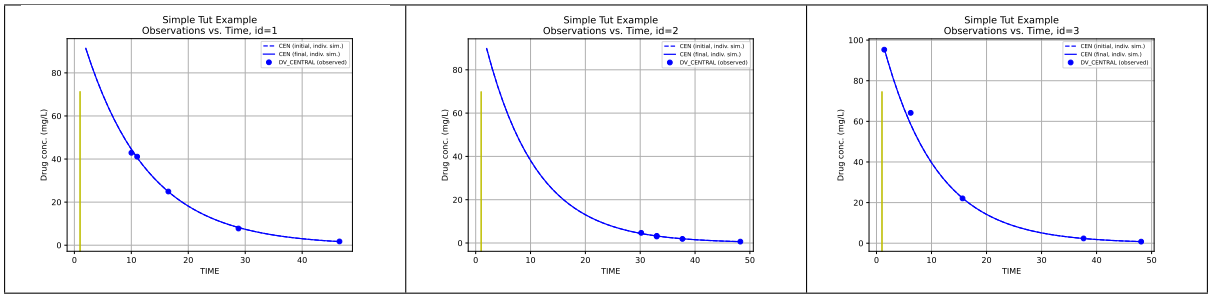
These presets are added in the grph_list option (for single populations) and vpc_list option (for simulated datasets of many populations).

35.6.1 Individual Observations vs. Time

Adding "OBS_vs_TIME" to grph_list creates a plot of predictions (line on the y-axis) and observations (points on the y-axis) against Time (on the x-axis) for each individual in the population.

```
GRPH:
    grph_list: ["OBS_vs_TIME"]
```

Table 8: Individual Observations vs Time for three individuals from a population



35.6.2 Population Observations vs. Time

Adding "{OBS}_vs_TIME" to grph_list creates a single plot of predictions (line on the y-axis) and observations (points on the y-axis) against Time (on the x-axis) for the whole population, overlaying individual data on the same graph.

```
GRPH:
    grph_list: ["{OBS}_vs_TIME"]
```

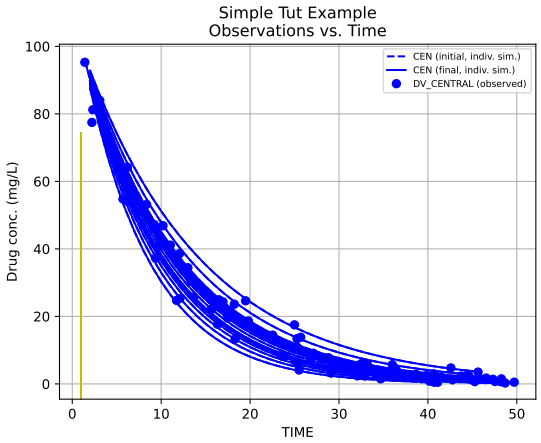


Fig. 1: Population Observations vs Time for a whole population

35.6.3 Population Observations vs. Individual Predictions

Adding "{OBS}_vs_IPRED" to grph_list creates a scatter plot for the whole population with Observations on the y-axis against Individualised (i.e. incorporating *random effects*) Predictions on the x-axis.

```
GRPH:
    grph_list: ["{OBS}_vs_IPRED"]
```

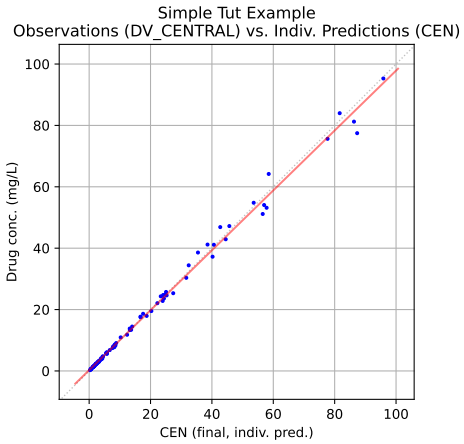


Fig. 2: Population Observations vs Individual Predictions for a whole population

35.6.4 Population Observations vs. Population Predictions

Adding "{OBS}_vs_PRED" to grph_list creates a scatter plot for the whole population with Observations on the y-axis against Population (i.e. using only fixed effects) Predictions on the x-axis.

```
GRPH:
    grph_list: ["{OBS}_vs_PRED"]
```

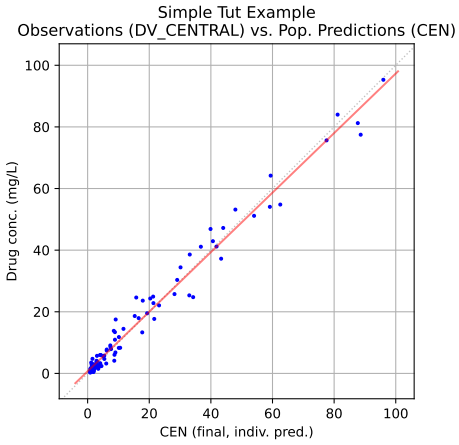


Fig. 3: Population Observations vs Population Predictions for a whole population

35.6.5 Individual Residuals vs. Individual Predictions

Adding "{IRES}_vs_IPRED" to grph_list creates a scatter plot for the whole population with Individualised (i.e. incorporating random effects) Residuals on the y-axis against Individualised Predictions on the x-axis.

```
GRPH:
    grph_list: ["{IRES}_vs_IPRED"]
```

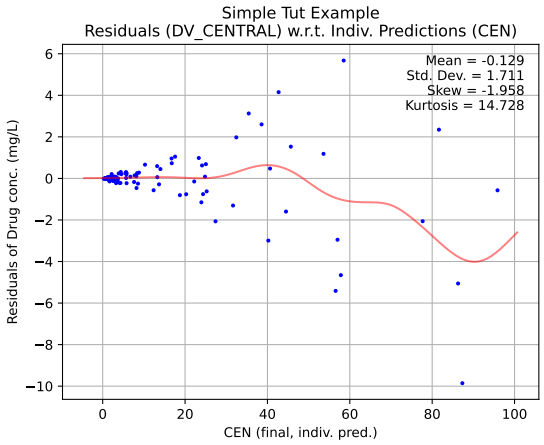


Fig. 4: Individual Residuals vs. Individual Predictions for a whole population

35.6.6 Individual Weighted Residuals vs. Individual Predictions

Adding "{IWRES}_vs_IPRED" to grph_list creates a scatter plot for the whole population with Individualised (i.e. incorporating random effects) Weighted (i.e. normalized with respect to the variance) Residuals on the y-axis against Individualised Predictions on the x-axis.

```
GRPH:
    grph_list: ["{IWRES}_vs_IPRED"]
```

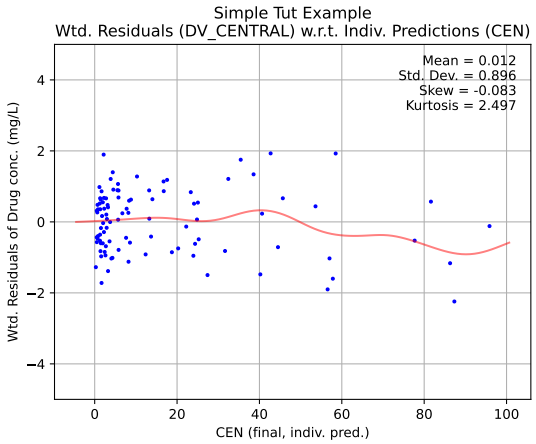


Fig. 5: Individual Weighted Residuals vs. Individual Predictions for a whole population

35.6.7 Individual Observations vs Time (VPC)

Adding "OBS_vs_TIME_VPC" to vpc_list creates a Visual Predictive Check plot that shows the percentiles of predictions and observations, and gives you some indication of whether the estimated model parameters represent a distribution that is close to the observed data.

```
VPC:
    vpc_list: ["OBS_vs_TIME_VPC"]
```

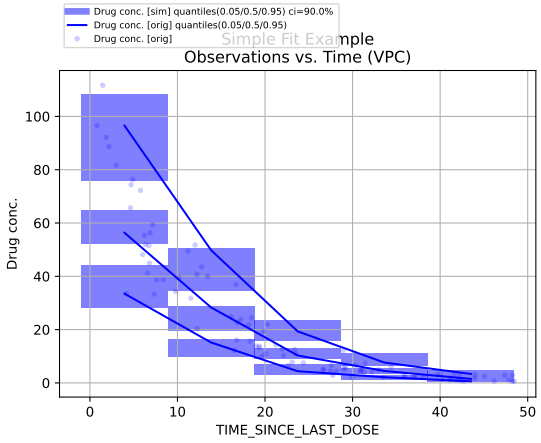


Fig. 6: Individual Observations vs Time VPC for 100 populations

35.7 Script Nodes

A *PoPy* script file is in *YAML* format, which consists of nested dictionaries that form a tree structure.

Each node of the tree is either:-

- a dictionary (*dict*) with sub nodes
- or a leave node

A leave node has to be one of the script node types listed below.

auto
: The word "auto", indicating a preference for default behaviour.

bool
: A boolean: "true"/"yes"/"y"/"1" or "False"/"no"/"n"/"0"

dict
: A dictionary of unspecified structure - the user chooses the keys.

  Many of the elements of a script file are known as dictionaries (also known as mappings or associative arrays). These are key:value pairs that can be written as follows using curly brackets:-

  ```
  my_dictionary1: { key1: value1, key2: value2 }
  ```

  or alternatively using spacing:-

```
my_dictionary2:
    key1: value1
    key2: value2
```

Whichever you use is a matter of convenience and space.

dict_record
A dictionary with a pre-determined structure (i.e. key names and corresponding types)

float
Any number

input_file
The path to a file that already exists. (An error occurs if it does not.)

input_file_as_glob
A pattern (e.g. `*.csv`) that has only a single match. (An error occurs if there are multiple matching filenames.)

input_files_as_glob
A pattern (e.g. `"*.csv"`) that has at least one match. (An error occurs only if there are no matching filenames.)

input_folder
The path to a folder that already exists. (An error occurs if it does not.)

int
An integer (whole number)

list
A list of values of unspecified type.

list_of
A list of one or more strings taken from the list of options in brackets.

list_record
A list

none
The word "none", indicating a file that does not exist.

one_of
A string, limited to one of the options given in brackets.

one_of_record
One from a selection of dict_records

output_file
The path to a file that may not yet exist (in contrast to *input_file*).

output_folder
The path to a folder that may not yet exist (in contrast to *input_folder*).

pseudocode
verbatim sections of the script file accept *Python* code with extra *PoPy* syntax added. For example special variable names such as `f[X]`, `r[X]` etc. We refer to this code as 'pseudocode'.

Pseudocode is automatically translated into runnable *Python* functions that are executed by *PoPy* to process your script.

repeat_dict_record
One or more of a selection of dict_records

repeat_verb_record
One or more of a selection of verbatim records

star
The "*" character, shorthand for "all possibilities".

str
A string.

verbatim
Several blocks in the script file (namely *MODEL_PARAMS*, *STATES*, *DERIVATIVES* and *PREDICTIONS*) are reserved for *pseudocode* of a relatively unstructured kind. This is translated into executable code.

CHAPTER

THIRTYSIX

SCRIPT FILE OUTPUTS

The files generated by *PoPy* scripts are described in Table 1:-

Table 1: Script Outputs

Script Name	Outputs
fit	*Files Generated by Fit Script*
gen	*Files Generated by Gen Script*
sim	*Files Generated by Sim Script*
tut	*Files Generated by Tut Script*
comp	*Files Generated by Comp Script*
mfit	*Files Generated by MFit Script*
mgen	*Files Generated by MGen Script*
msim	*Files Generated by MSim Script*
mtut	*Files Generated by MTut Script*
mcomp	*Files Generated by MComp Script*
fitsum	*Files Generated by Fitsum Script*
gensum	*Files Generated by Gensum Script*
tutsum	*Files Generated by Tutsum Script*

36.1 Files Generated by Fit Script

You can examine the fit_script log file and the html output. However it is useful to also understand the files output to disk by a fitting script.

The files generated in the same folder as 'fit_example1.pyml' are:-

```
fit_example1.pyml.html
fit_example1.pyml.run.main.log
```

Here 'fit_example1.pyml.html' is a shortcut to the web page output. 'fit_example1.pyml.run.main.log', contains a copy of the text output to the console whilst running *popy run*. This text file acts as an audit trail if you want to review the output of running the fit script later, or text search the console output for example.

The main file outputs from 'fit_example1.pyml' are contained in the folder called `fit_example1.pyml_output`

The default convention for most *PoPy* scripts is to generate an output folder for each individual script as follows:-

```
script_name + '_output'
```

This simple convention has the useful facility of guaranteeing a unique output folder for each *PoPy* script file, so you can run multiple *.pyml files in the same folder without worrying about over-writing output from other scripts (something other PopPK/PD systems struggle with).

Note if you attempt to run the same *.pyml file twice than *PoPy* will ask you if you want to over-write the previous output folder. You can force *PoPy* to over-write existing folders using:-

```
popy run -o fit_example1.pyml
```

See *popy run* for more command line switch options.

The contents of the 'fit_example1.pyml_output' are determined by the *OUTPUT_SCRIPTS* section of the 'fit_example1.pyml' script file:-

```
OUTPUT_SCRIPTS:
    SIM: {output_mode: run, sim_time_step: 1.0}
    MSIM: {output_mode: run}
    FITSUM: {output_mode: run}
```

This section requests that a *Sim Script*, *MSim Script* and *FitSum Script* child script are created after the main *Fit Script* has finished. Note the *Sim Script* and *FitSum Script* will also be run automatically. So you end up with output folders on disk with this structure:-

```
fit_example1.pyml_output/
    fit/
    msim/
    sim/
    sum/
```

in which

- **fit** contains the results of running the *Fit Script*
- **msim** is a *MSim Script* that can be run later to generate a *visual predictive check*
- **sim** is the results of running *Sim Script* to create smoother profile curves from the fitted results
- **sum** is a html summary of the fitted model

This system of *PoPy* automatically generating new **child** scripts to process the results of an original **parent** script (which is called using *popy run*) is key concept in how *PoPy* works, see typical_workflows. It is hierarchical scripting of hierarchical PopPK/PD modelling!

Note the *OUTPUT_SCRIPTS* section is entirely optional. If you remove the *OUTPUT_SCRIPTS* section from the 'fit_example1.pyml', then you just end up with a single output folder as follows:-

```
fit_example1.pyml_output/fit
```

We will now look at the outputs of the individual scripts in each subfolder.

36.1.1 Fit Script Outputs

The 'fit' subfolder has the following structure:-

```
fit_example1.pyml_output/
    fit/
        _temp
        observed_data
        sol0
        sol00
        sol1
        solN
        compartment_diagram.dot
        compartment_diagram.svg
        OBJV_vs_time.csv
```

In this folder, the files are as follows:-

- 'compartment_diagram.dot' - dot *Graphviz* file derived from *DERIVATIVES*
- 'compartment_diagram.svg' - scalable vector graphics file displaying the compartment structure, derived from the .dot file
- 'OBJV_vs_time.csv' - table showing objective value vs time, see OBJV_vs_time

The subfolders are:-

- _temp - Temporary folder used by *PoPy* to create *Python* functions
- observed_data - Contains filtered copy of 'fit_example1_data.csv' input data
- solX - Where X is one of [0,00,1,N]

You can examine the '_temp' folder when debugging, if one of the '.py' python functions derived from the 'fit_example1.pyml' script has not compiled properly. Or just for the curious.

The 'observed data' folder is a copy of the input data, which contains the original data set but may have been filtered using an optional *PREPROCESS*.

The solX folders each contain the current *solution* at each stage of processing. As follows:-

- sol00 - Solution using initial `f[X]` and all `r[X]` =0.0
- sol0 - Solution using initial `f[X]` and fitted `r[X]`
- sol1 - Solution for first fitting method
- sol2 - Solution for second fitting method (not present for 'fit_example1.pyml')
- solXXX - Solution for further fitting methods ...
- solN - Final Solution (copy of sol1 folder for 'fit_example1.pyml')

Each solution folder contains multiple files the principle files being:-

- cur_fx_params.txt - `f[X]` parameters in human readable format
- cur_rx_params.txt - `r[X]` parameters in human readable format
- cur_obj_value.txt - current objective value

There are also various .csv files which are more verbose, but easier to load in to software programs, e.g. *PoPy* or *R* for example.

Note sol00, sol0 and solN each contain a single *solution*. However here the 'sol1' folder contains the results of applying the *JOE*,|foce| or *ND* fitting method and has a slightly different structure:-

```
fit_example1.pyml_output/
    fit/
        solXXX/
            itYYY/
```

Where XXX is the fitting method and YYY is the results of single iteration of the 'JOE' fitting algorithm.

The final solution for the *Fit Script* are at this location on disk:

```
fit_example1.pyml_output/
    fit/
        solN/
```

36.1.2 Sim Script Outputs

The *Fit Script* creates the child *Sim Script* within the 'sim' subfolder:-

```
fit_example1.pyml_output/
    sim/
        fit_example1_sim.pyml
```

The 'fit_example1_sim.pyml' script, is run automatically after the 'fit_example1.pyml' script has finished. This *Sim Script* creates **dense** profile plots of the fitted model predicted values at time steps of 1.0, which can then be compared with the original simulated data for each individual.

The full set of dense data plots for all individuals are located in this folder on disk:-

```
fit_example1.pyml_output/
    sim/
        dense/
            DV_CENTRAL,DV_CENTRAL,DV_CENTRAL_wrt_TIME_spag_graphs
```

For more information see *Files Generated by Sim Script*.

36.1.3 MSim Script Outputs

The *Fit Script* creates the child *MSim Script* within the 'msim' subfolder:-

```
fit_example1.pyml_output/
    msim/
        fit_example1_msim.pyml
```

This *MSim Script* is generated by the original fit_script but **not** run automatically due to this line:-

```
MSIM: {output_mode: create}
```

The purpose of the msim script is to generate a *visual predictive check* by simulating from the fitted model and comparing the simulated curves to the original data set. Running the 'msim' script is described in simple_msim_example1.

For more information on files output once the *MSim Script* is run see *Files Generated by MSim Script*.

36.1.4 FitSum Outputs

The *Fit Script* creates the child *FitSum Script* within the 'sum' subfolder:-

```
fit_example1.pyml_output/
    sum/
        fit_example1_fitsum.pyml
```

The purpose of this script is to summarise the fit/sim output using automatically generated web pages.

This **sum** tool displays the output from running a *Fit Script* and the child *Sim Script* on disk in a convenient format. For more information see *Files Generated by Fitsum Script*.

The output from **sum** tools is also used to automate large parts of this documentation.

36.2 Files Generated by Gen Script

This section discusses how the *Gen Script* saves files to disk, you can see an example in builtin_gen_example. After running a gen script, named 'my_gen_script.pyml', using the command:-

```
popy run my_gen_script.pyml
```

the output folder 'my_gen_script.pyml_output' will be created as follows:-

```
my_gen_script.pyml_output/
    gen/
    sim/
    sum/
```

where

- **gen** contains the results of running the *Gen Script*
- **sim** contains the results of running *Sim Script* to create smoother profile curves and
- **sum** is a html summary of the generated data.

The *Gen Script* also creates a log file **my_gen_script.pyml.run.main.log** that stores the output to the command prompt.

36.2.1 Gen Folder Output

The gen folder has the following contents:-

```
builtin_gen_example_gen.pyml_output/
    gen/
        _temp/
        clean_sol/
        noisy_sol/
        compartment_diagram.dot
        compartment_diagram.svg
        dupe_covars_data.csv
        synthetic_data.csv
```

Here '_temp' is a temporary folder used by *PoPy*. The file 'dupe_covars_data.csv' is an intermediate file.

A compartment diagram is created in .svg format. The .dot file is the *Graphviz* file, used to create the .svg file. You can see an example compartment diagram in `fig_two_comp_depot_diagram_gen`.

The 'clean_sol' folder contains a *solution* with no noise added. The 'noisy_sol' contains as *solution* with measurement noise added. For example the 'noisy_sol' folder contains these files:-

```
builtin_gen_example_gen.pyml_output/
    gen/
        noisy_sol/
            fx_params.csv
            mx_params.csv
            rx_params.csv
            solution.pyml
            sx_params.csv
            synthetic_data.csv
```

Here the key file is 'synthetic_data.csv', which is the full synthetic data file generated for this population. The other files represent intermediate variables for each time point, see *Files Generated by Sim Script* for more information.

Note the 'synthetic_data.csv' file is also output to this location, for easier access:-

```
builtin_gen_example_gen.pyml_output/
    gen/
        synthetic_data.csv
```

It is the main data file created by the *Gen Script*.

36.2.2 Naming of child scripts

Each child script file name is based on the following entry, in the parent *Gen Script*:-

```
DESCRIPTION: {name: <gen_name>}
```

This naming convention for output folders and generated scripts is followed by all *Script File Formats* in *PoPy*.

36.2.3 Gen Sim Folder Output

The *Gen Script* creates the following *Sim Script*:-

```
my_gen_script.pyml_output/
    sim/
        <gen_name>_sim.pyml
```

This *Sim Script* plots smooth PK curves to visualise the `f[X]` parameters sampled by the *Gen Script*. See *Files Generated by Sim Script* for more details.

36.2.4 Gen Sum Folder Output

The *Gen Script* creates the following *GenSum Script*:-

```
my_gen_script.pyml_output/
    sum/
        <gen_name>_gensum.pyml
```

The *GenSum Script* creates html output that summaries the **gen** and **sim** folders. See *Files Generated by Gensum Script* for more details.

36.3 Files Generated by Sim Script

This section discusses how the *Sim Script* saves files to disk. Typically a *Sim Script* is created in a sub folder as a child script of a *Fit Script* or *Gen Script*.

Here we assume that the *Sim Script* was created by a *Fit Script* parent. Similar to the *Fit Script* described in *Files Generated by Fit Script*.

When a *Sim Script* is run, folders will be created on disk (within the same folder as the script) as follows:-

```
_temp
dense
observed_data
```

Here '_temp' is a folder used by *PoPy* to write temporary python functions and 'dense' contains the final plots.

'observed_data' is a *solution* folder that only contains original input data from the *Sim Script*, for example, as specified in the 'input_data_file' field:-

```
FILE_PATHS:
    input_data_file: ..\gen\synthetic_data.csv
```

Here, the 'observed_data' folder contains a copy of the 'synthetic_data.csv' generated by the *Gen Script*. The other folders generated by the *Sim Script* are named after the *solutions* loaded in by the 'solutions' section of the *Sim Script* section, for example:-

```
FILE_PATHS:
    solutions:
        initial: ..\fit\sol00\solution.pyml
        final: ..\fit\solN\solution.pyml
```

Therefore *solution* folders 'initial' and 'final' will be created and will contain simulated PK data using the original *solution* folders from the **fit** output above.

The *Sim Script* generates `s[X]` state values and noiseless `p[X]` predictions at regular, dense time intervals (1.0 hours). These files are also saved as comma separated value .csv files (one per individual) in:-

```
initial/
    pred_1.csv
    ...
    pred_50.csv
    sx_1.csv
    ...
    sx_50.csv
```

And similarly in the 'final' *solution* folder. Here the *Sim Script* is effectively re-creating the original initial `f[X]` solution and final `f[X]` solution, but sampling more time points, to generate smoother PK curves.

The sim script also outputs a *Grph Script* that plots the 'initial', 'final' and 'observed_data' solutions on one graph, with one plot per individual in the 'dense' folder:-

```
dense/
    DV_CENTRAL,DV_CENTRAL,DV_CENTRAL_wrt_TIME_spag_graphs/
        000000.svg
        ...
        000049.svg
```

36.4 Files Generated by Tut Script

This section describes the output files from the *Tut Script*, you can see examples in simple_tut_example and *Running Your First tut Script*.

After running a tutorial script, named 'my_tut_script.pyml', as follows:-

```
popy run my_tut_script.pyml
```

the output folder will contain four new scripts:-

```
my_tut_script.pyml_output/
    <tut_name>_gen.pyml
    <tut_name>_fit.pyml
    <tut_name>_comp.pyml
    <tut_name>_tutsum.pyml
```

Note the tutorial output folder name **my_tut_script.pyml_output** is derived from the tutorial script filename. However the generated script file names are based on the following entry:-

```
DESCRIPTION: {name: <tut_name>}
```

This naming convention for output folders and generated scripts is followed by all *Script File Formats* in *PoPy*.

Scripts and description names should be chosen with this in mind, *i.e.* short names without spaces are recommended.

For a *Tut Script* the four subscripts are run automatically, see `table_tut_script_sub_outputs` for links to the files generated by the tut subscripts, each in their own sub directories.

Script	Outputs
<tut_name>_gen.pyml	*Files Generated by Gen Script*
<tut_name>_fit.pyml	*Files Generated by Fit Script*
<tut_name>_comp.pyml	*Files Generated by Comp Script*
<tut_name>_tutsum.pyml	*Files Generated by Tutsum Script*

If all four scripts execute then you will see a file structure like:-

```
my_tut_script.pyml_output/
    <tut_name>_gen.pyml_output/
    <tut_name>_fit.pyml_output/
    <tut_name>_comp.pyml_output/
    <tut_name>_tutsum.pyml_output/
```

36.5 Files Generated by Comp Script

This section discusses how the comp script saves files to disk, if you want to see a full *Comp Script* example using a *Tut Script* try *Running Your First tut Script*.

Output from an *Comp Script* called 'builtin_tut_example_comp.pyml' are as follows:-

```
builtin_tut_example_comp.pyml_output/
    _temp
    dense
    re_fit
    re_gen
    builtin_tut_example_dense_grph.pyml
    builtin_tut_example_dense_grph.pyml.run.main.log
```

The '_temp' folder contains the temporary functions generated by *PoPy*.

The 'dense' folder contains the PK curve plots of fitted vs true `f[X]` and the synthetic data. These plots are generated by the 'builtin_tut_example_dense_grph.pyml' *Grph Script*.

The 're_fit' folder contains the output from optimising the `r[X]` when computing the fitted `f[X]` objective function.

The 're_gen' folder contains the output from optimising the `r[X]` when computing the true `f[X]` objective function.

36.6 Files Generated by MFit Script

This section discusses how the mfit script saves files to disk, if you want to see a full *MFit Script* example using a *MTut Script* try builtin_mtut_example.

Outputs from an *MFit Script* called 'builtin_mtut_example_mfit.pyml' are as follows:-

```
builtin_mtut_example_mfit.pyml_output/
    _temp
    fitpop_*
    compartment_diagram.dot
    compartment_diagram.svg
```

The '_temp' folder contains the temporary functions generated by *PoPy*. The 'compartment_diagram.dot' file is a *Graphviz* file used to generate the 'compartment_diagram.svg' image file.

The main output of *MFit Script* are the multiple 'fitpop_*' folders. Each of these folders contains the results of fitting the model to a separate synthetic data set.

The input data files are determined by the *FILE_PATHS* section of the *MFit Script*. For example the entry:-

```
FILE_PATHS:
    input_data_files_glob: builtin_
↪mtut_example_mgen.pyml_output/genpop_*/noisy_sol/synthetic_data.csv
```

Notice the '*' in the 'input_data_files_glob' field, which uses a glob pattern to load multiple data files. Essentially any files matching this pattern, usually generated by a *MGen Script*, see *Files Generated by MGen Script*.

The structure of each 'fitpop_*' folder is as follows:-

```
builtin_mtut_example_mfit.pyml_output/
    fitpop_*/
        sol00/
        sol0/
        sol1/
        solN/
        OBJV_vs_time.csv
```

This folder structure is very similar to the output of a *Fit Script*. See *Fit Script Outputs* for a detailed explanation. The main result of each fit are the final `f[X]` parameter estimates which are store here:-

```
builtin_mtut_example_mfit.pyml_output/
    fitpop_*/
        solN/
            fx_params.csv
```

The 'fx_params.csv' contains the fitted `f[X]` which can be compared with the equivalent true `f[X]` values generated by *MGen Script*, usually located here (see *Files Generated by MGen Script*):-

```
builtin_mtut_example_mgen.pyml_output/
    genpop_*/
        noisy_sol/
            fx_params.csv
```

The fitted vs true `f[X]` comparison is performed using the *MComp Script*, see (see *Files Generated by MComp Script*):

Note the *MFit Script* has **no** *OUTPUT_SCRIPTS* section, so does not generate any child scripts unlike a *Fit Script* see *Files Generated by Fit Script*.

36.7 Files Generated by MGen Script

This section discusses how the mgen script saves files to disk, if you want to see a full *MGen Script* example using a *MTut Script* try builtin_mtut_example.

Outputs from an *MGen Script* called 'builtin_mtut_example_mgen.pyml' are as follows:-

```
builtin_mtut_example_mgen.pyml_output/
    _temp
    genpop_*
    compartment_diagram.dot
    compartment_diagram.svg
```

The '_temp' folder contains the temporary functions generated by *PoPy*. The 'compartment_diagram.dot' file is a *Graphviz* file used to generate the 'compartment_diagram.svg' image file.

The main output of *MGen Script* are the multiple 'genpop_*' folders. Each of these folders contains the current sampled `f[X]` values and the synthetic data generated from the `f[X]` and the model.

Note the number of 'genpop_*' folders created by the *MGen Script* is determined by the 'n_pop_samples' in the *OUTPUT_OPTIONS* section:-

```
OUTPUT_OPTIONS: {n_pop_samples: 30}
```

The structure of each 'genpop_*' folder is as follows:-

```
builtin_mtut_example_mgen.pyml_output/
    genpop_*/
        clean_sol/
        noisy_sol/
        dupe_covars_data.csv
        gen_fx_params.txt
```

Here the 'gen_fx_params.txt' is a human readable text file containing the `f[X]` values for this population. The 'dupe_covars_data.csv' is an intermediate file used by *PoPy* when constructing a new data set. The 'clean_sol' folder contains a *solution* with no noise added. The 'noisy_sol' contains as *solution* with measurement noise added. For example the 'noisy_sol' folder contains these files:-

```
builtin_mtut_example_mgen.pyml_output/
    genpop_*/
        noisy_sol/
            fx_params.csv
            mx_params.csv
            rx_params.csv
            solution.pyml
            sx_params.csv
            synthetic_data.csv
```

Here the key file is 'synthetic_data.csv', which is the full synthetic data file generated for this population.

The other files represent intermediate variables for each time point, see *Files Generated by Sim Script* for more information.

Note the *MGen Script* has **no** *OUTPUT_SCRIPTS* section, so does not generate any child scripts unlike a *Gen Script* see *Files Generated by Gen Script*.

36.8 Files Generated by MSim Script

This section discusses how the *MSim Script* saves files to disk, if you want to see an example of a *MSim Script* generated by a *Fit Script* then see builtin_fit_example.

Outputs from an msim script called 'fit_example1_msim.pyml' are:-

```
fit_example1_msim.pyml_output/
    _temp
    msol
    fit_example1_vpc.pyml
```

The '_temp' folder contains the temporary functions generated by *PoPy* and the 'msol' folder contains the new simulated populations. There is one .csv file per simulated population, that all have the same number of rows as the original 'fit_example1_data.csv'.

The *MSim Script* also outputs the 'fit_example1_vpc.pyml' *Vpc Script* and runs it automatically after finishing the multi population simulation. The vpc_script is responsible for loading in the data from the 'msol' folder and creating a vpc graph.

The output of the 'fit_example1_vpc.pyml' script is here:-

```
fit_example1_msim.pyml_output/
    DV_CENTRAL_sim,DV_CENTRAL_wrt_TIME_SINCE_LAST_DOSE_comb_quant_sim_vpc/
        000000.svg
```

The parameters of the vpc graph are used in the folder name, so it ends up being quite long.

36.9 Files Generated by MTut Script

This section discusses how the *MTut Script* saves files to disk. After running a multi-tutorial script, named 'my_mtut_script.pyml', as follows:-

```
popy run my_mtut_script.pyml
```

the output folder will contain three new scripts:-

```
my_mtut_script.pyml_output/
    <mtut_name>_mgen.pyml
    <mtut_name>_mfit.pyml
    <mtut_name>_mcomp.pyml
```

Note the multi-tutorial output folder name **my_mtut_script.pyml_output** is derived from the multi-tutorial script filename. However the generated script file names are based on the following entry:-

```
DESCRIPTION: {name: <mtut_name>}
```

This naming convention for output folders and generated scripts is followed by all *Script File Formats* in *PoPy*.

It is worth naming your scripts and description names with this in mind, i.e., short names without spaces are recommended.

For a *MTut Script* the three subscripts are usually run automatically, see `table_mtut_script_sub_outputs` for links to the files generated by each of the subscripts, in their own sub folders.

Script	Outputs
<mtut_name>_mgen.pyml	*mgen outputs*
<mtut_name>_mfit.pyml	*mfit outputs*
<mtut_name>_mcomp.pyml	*mcomp outputs*

These three scripts are run in order. When all three scripts are run then you end up with a file structure like:-

```
my_mtut_script.pyml_output/
    <mtut_name>_mgen.pyml_output/
    <mtut_name>_mfit.pyml_output/
    <mtut_name>_mcomp.pyml_output/
```

36.10 Files Generated by MComp Script

This section discusses how the mcomp script saves files to disk, if you want to see a full *MComp Script* example using a *MTut Script* try builtin_mtut_example.

Output from an *MComp Script* called 'builtin_mtut_example_mcomp.pyml' are as follows:-

```
builtin_mtut_example_mcomp.pyml_output/
    _temp
    fx_scatter
    mcomp_sections
    mfit_sections
    mgen_sections
    final_fx.csv
    gt_fx.csv
    init_fx.csv
    type_fx.csv
```

The '_temp' folder contains the temporary functions generated by *PoPy*. The 'fx_scatter' folder contains the scatter plots of fitted vs true `f[X]`. For example files like:-

```
builtin_mtut_example_mcomp.pyml_output/
    fx_scatter/
        fitted_vs_true_for_f[cl].svg
        fitted_vs_true_for_f[ka].svg
        fitted_vs_true_for_f[pnoise].svg
        ..
```

The folders named '*_sections', just contain the original .pyml scripts in sections for use in the *PoPy* documentation.

The .csv files are as follows:-

Table 2: .csv file output by MComp Script

Filename	Contents
final_fx.csv	Each row contains `f[X]` estimates for one population
gt_fx.csv	Each row contains true `f[X]` values for one population
init_fx.csv	Each row contains starting `f[X]` values for one population
type_fx.csv	Each row contains `f[X]` type (e.g. main or var `f[X]`)

Note each of the *_fx.csv files has the same number of rows and columns, so you could easily load this data into *R* for further analysis if you like.

36.11 Files Generated by Fitsum Script

This section discusses how the *FitSum Script* saves files to disk, if you want to see an example of a *FitSum Script* created by a *Fit Script*, see builtin_fit_example.

If you run a *FitSum Script* using the following command:-

```
popy run my_fitsum_script.pyml
```

files and folders will be created on disk (within the same folder as the script) as follows:-

```
build
src
my_fitsum_script.pyml.html
my_fitsum_script.pyml.run.main.log
sphinx.log
```

Here 'src' contains files copied from the output of the parent *Fit Script* and some automatically generated .rst files.

The 'build' folder is the output of running *Sphinx* on the 'src' files. Sphinx is a utility for processing .rst files into *HTML* and optionally other formats. The main output of running sphinx is here:-

```
build/
    html/
        index.html
```

Here 'sphinx.log' is the log file for *Sphinx* and 'my_fitsum_script.pyml.run.main.log' is the log file for the *FitSum Script*.

You can view the summary of the fit/sim output to *HTML* files by typing:-

```
popy view my_fitsum_script.pyml.html
```

Which should look something like sum_link_builtin_fit_example_fit.

36.12 Files Generated by Gensum Script

This section discusses how the *GenSum Script* saves files to disk, if you want to see an example of a *GenSum Script* created by a *Gen Script*, see builtin_gen_example.

If you run a *GenSum Script* using the following command:-

```
popy run my_gensum_script.pyml
```

folders will be created on disk (within the same folder as the script) as follows:-

```
build
src
my_gensum_script.pyml.html
```

(continues on next page)

(continued from previous page)

```
my_gensum_script.pyml.run.main.log
sphinx.log
```

Here 'src' contains files copied from the output of the parent *Gen Script* and some automatically generated .rst files.

The 'build' folder is the output of running *Sphinx* on the 'src' files. Sphinx is a utility for processing .rst files into *HTML* and optionally other formats. The main output of running sphinx is here:-

```
build/
    html/
        index.html
```

Here 'sphinx.log' is the log file for *Sphinx* and 'my_gensum_script.pyml.run.main.log' is the log file for the *GenSum Script*.

You can view the summary of the gen/sim output to *HTML* files by typing:-

```
popy view my_gensum_script.pyml.html
```

Which should look something like sum_link_builtin_gen_example_gen.

36.13 Files Generated by Tutsum Script

This section describes the output of the *TutSum Script*. If you want to see an example of a *TutSum Script* created by a *Tut Script*, see simple_tut_example.

If you run a *TutSum Script* using the following command:-

```
popy run my_tutsum_script.pyml
```

folders will be created on disk (within the same folder as the script) as follows:-

```
build
src
my_tutsum_script.pyml.html
my_tutsum_script.pyml.run.main.log
sphinx.log
```

Here 'src' contains files copied from the output of the parent *Tut Script* and some automatically generated .rst files.

The 'build' folder is the output of running *Sphinx* on the 'src' files. Sphinx is a utility for processing .rst files into *HTML* and optionally other formats. The main output of running Sphinx is here:-

```
build/
    html/
        index.html
```

Here 'sphinx.log' is the log file for *Sphinx* and 'my_tutsum_script.pyml.run.main.log' is the log file for the *TutSum Script*.

You can view the summary of the gen/fit/comp/sim output to *HTML* files by typing:-

```
popy view my_tutsum_script.pyml.html
```

Which should look something like *First order absorption model with peripheral compartment*.

CHAPTER

THIRTYSEVEN

GLOSSARY

Akaike information criterion
: A method of comparing two similar models by penalising models with a larger number of parameters. See Akaike information criterion on Wikipedia

basin of convergence
: A set of initial points that lead to the same local minimum under a given iterative algorithm.

bobyqa optimisation
: BOBYQA= Bounded Optimisation by Quadratic Approximation, a none derivative based optimisation method [Powell2009] used by *ND* method.

C++
: C++ is a low level programming language which is automatically used by *PoPy* for some time critical operations C++ on Wikipedia

categorical covariates
: Covariates that indicates membership in one of a set of unordered categories, such as race.

clearance
: The *volume* of the fluid presented to the *eliminating* organ (extractor) that is effectively completely cleared of drug per unit time. (Definition from [RowlandTozer2012]), also see Clearance on Wikipedia

Compartment Diagram
: A graphical visualisation of the compartment model, using nodes for compartments and edges for flows between compartments

confidence intervals
: Ranges in which we can be X% confident that a parameter lies.

covariance matrix
: A measure of spread for multiple random variables that may be correlated. See Covariance on Wikipedia

covariates
: Measured or observed quantities that are read in from the input data file. Signified by a `c[X]` in the model specification file (which could also be thought of as an abbreviation of "column"). They include information such as ID, time, weight, and also measurements such as drug concentration.

Cython
: Cython is an superset of *Python* that compiles to *C++*. *PoPy* uses *Cython* extensively to process the user config file. See Cython on Wikipedia

DDMoRe
: An organization founded in April 2016 with the aim to maintain and enhance the published open-source DDMoRe standards and tools to deliver and improve the quality, efficiency and cost effectiveness of Model-Informed Drug Discovery & Development (MID3) and Therapeutic Use. See DDMoRe Website

dos prompt
: The dos prompt command line in Microsoft Windows. This is the older Windows shell, by default with a black background.

elimination
: The removal of a drug from the body.

excretion
: The removal of waste substances from the body including unchanged drugs and metabolic products. Most drug products are eliminated through the kidneys.

first order conditional estimation
: *FOCE* is a fitting method in *Nonmem* that uses a first order approximation of the *objective function* conditioned on optimised *random effects* for each individual in the population. For a description of *PoPy*'s implementation of *FOCE* see *FOCE Fitting Method*.

first pass effect
: A reduction in the amount of drug entering circulation due to it being metabolised by the liver or gut on its way to the blood system. First pass effect on Wikipedia

fixed effects
: A population-level parameters (usually means) that describe an average from which individuals deviate in a random way, though where the nature of the randomness is known. Signified by `f[X]` in the model specifications file.

Graphviz
: Graphviz is open source software used to create *Compartment Diagram* in *PoPy*. See Graphviz on Wikipedia

hessian
: The $n \times n$ matrix of second derivatives of a function of n variables. Contains information that describes the shape of the surface at a given point (the minimum, for example).

HTML
: Hyper Text Markup Language used on the web and by *PoPy* to generate summary output. See HTML on Wikipedia

importance sampling
: A method of sampling from a complex distribution by first sampling from a simpler distribution and re-weighting with the ratio of the complex and simpler Wikipedia: Probability_density_function. See Importance Sampling on Wikipedia.

initial value problem
: The *ordinary differential equations* typically solve a dynamic system which has a defined input state and then the system evolves over time according to the *ordinary differential equation* system. This type of integration problem, typical in PK/PD, is known as a Initial Value Problem.

iterative two stage
: *ITS* is a fitting method in *Nonmem* that optimises the *objective function* by switching between optimising the *fixed effects* and *random effects*.

joint optimisation and estimation
: *JOE* is *PoPy*'s original fitting method see *JOE Fitting Method*, it optimises the same *objective function* as *FOCE* and *ITS* and is most similar to *ITS* in terms of fitting performance.

Laplace approximation
: A method of approximating integrals. See Laplace method on Wikipedia. This *objective function* is used by *LAPLACE* and an approximation is used by *JOE*, *FOCE* and *ITS* fitting methods.

laplace fitting method
: A fitting method that uses the *Laplace approximation* as an *objective function*. Note *JOE*, *FOCE* and *ITS* use a related, but less computationally expensive *objective function*.

Likelihood
: The conditional probability, $p(D|M)$, of observing data D given a hypothesized model M. This expresses the *plausibility* of model M given data D, but is a probability distribution over D rather than M. As a result, it cannot be used to compare different models, only different parameter values for the same model. Likelihood on Wikipedia

LSODA
: Numerical *ordinary differential equation* solver [Radhakrishnan1994] available in *PoPy*, see *Example ODE_SOLVER using CPPLSODA*.

mass balance
: The principal that matter cannot be created or destroyed within a compartment model, apart from deliberate inputs (e.g doses) and sink compartments that model excretion from the body. See Mass Balance on Wikipedia.

metabolism
: Process by which drug is chemically transformed into another substance. Takes place primarily in the liver.

Microsoft Windows
: A popular operating system for personal computers.

mixed effect model
: A structural model that uses both *fixed effects* and *random effects* to model population parameters. In practise, all models contain at least one *fixed effect*, so the key feature is the use of *random effects* to allow parameters to vary between subjects in the population.

model parameters
: Person-specific PK/PD parameters, usually defined as a function of the fixed effects, random effects and measured covariates. Signified by `m[X]` in the model specification file.

Monolix
: Matlab based PK/PD modelling software. See *http://lixoft.com/products/monolix/*

MPI
: Message Passing Interface. A protocol for parallelising software by passing information between processors. See Message Passing Interface on Wikipedia

noise
: Random displacements added to a signal. See Signal Processing Noise on Wikipedia

none derivative estimation
: *ND* is *PoPy*'s newest fitting method see *ND Fitting Method*, it optimises the same *objective function* as *FOCE* and uses the derivative based *FOCE* fitting method, but also utilises the none derivative *BOBYQA* algorithm.

Nonmem
: Nonmem (NONlinear Mixed Effect Modelling) is a Fortran based system for PK/PD modelling. [Bauer2009]

objective function
: The *fixed effects* and *random effects* of a model are estimated by minimising the *objective function*, which is equivalent to maximising the likelihood of the model given the *observations*.

observations
: The observed values to be modelled, also known as the dependent variable. These measurements (either synthetic or real) are signified by `c[X]` in the *PREDICTIONS* section of a *PoPy script file*.

ordinal covariates
: Covariates derived from a discretisation of a continuum such that values have a definite order, such as the East Coast Oncology Group status that ranges from 0 (normal) to 4 (most severe).

ordinary differential equations
: Multiple differential equations, each with one independent variable. See Ordinary differential equation on Wikipedia

powershell prompt
: The powershell prompt command line in Microsoft Windows. This is the newer Windows shell, by default with a blue background.

practically identifiable
: A parameter of a model is **practically identifiable** or estimable, if the true value can be estimated from a finite amount of data. See Identifiability Analysis on Wikipedia

practically unidentifiable
: A parameter that is **not** *practically identifiable*

predictions
: The value the model calculates for a given observations, usually a conversion to concentrations via division by the volumes of the compartments. Signified by `p[X]` in the model specification file.

product key
: The *PoPy* product key is the unique key that identifies the the current licence. It has a form like 'XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX'. See licences.

Python
: Python is a general purpose programming language used in *PoPy* scripts and to implement *PoPy* itself. See Python on Wikipedia

R
: R is open source statistical software used extensively in the PK/PD community. See R on Wikipedia

random effects
: Deviation from the population-level fixed parameters, with defined distribution parameters. Signified by `r[X]` in the model specification file.

shrinkage
: The tendency to for *random effects* to shrink towards the mean value when data are sparse.

solutions
: Solutions are defined by a .pyml file containing links to .csv files that determine a set of `f[X]`, `r[X]`, `m[X]`, `s[X]`, `p[X]` variables that represent a candidate solution to a PK/PD model fitting problem.

Sphinx
: Documentation system used by *PoPy* and many other *Python* projects to generate .html and .pdf files. See Sphinx on Wikipedia

state parameters
: The amount - not concentration - of drug in each compartment of the compartment model. Signified by `s[X]` in the model specification file.

stochastic approximation expectation maximisation
: *SAEM* is a probabilistic fitting method originally implemented in *Monolix* and also available in *Nonmem*.

structurally identifiable
: A parameter of a model is **structurally identifiable**, if given an infinite amount of data the true underlying parameter value is recoverable. See Identifiability Analysis on Wikipedia

structurally unidentifiable
: A parameter that is **not** *structurally identifiable*

symmetric positive definite
: A symmetric positive definite matrix is a matrix whose eigenvalues are all positive. It is the matrix equivalent of having a real valued square root. In PK/PD models a population covariance matrix is required to be symmetric positive definite. See Matrix Definiteness on Wikipedia

variance
: A measure of spread for a random variable. See Variance on Wikipedia

visual predictive check
: Given a set of `f[X]` values and a model, new `p[X]` values are simulated which can then be compared with original `c[X]` data on a graph.

volume of distribution
: The volume (or volume of distribution) is the theoretical volume that a compartment would need to have to give the concentration of drug found in the blood plasma. See Volume of Distribution on Wikipedia

YAML
: A simple markup language used by *PoPy* *Script File Formats*. See YAML on Wikipedia

CHAPTER

THIRTYEIGHT

HTML SUMMARY LINKS

PoPy outputs *HTML* summaries of *fit_scripts*, *gen_scripts* and *tut_scripts*.

This page lists the summary outputs for **all** example scripts used in this documentation. Browse this list to see the variety of PK/PD modelling available in *PoPy*.

Note, each summary contains a link to the original script. *e.g.* A tut summary contains a link to the original *Tut Script*, so you can download each script and re-run all of the examples on this page using your own installation of *PoPy*. You can also adapt each example script to your own PK/PD modelling requirements.

38.1 Example Summaries

38.1.1 Simple Fit Example

Used in simple_fit_example to demonstrate running a *Fit Script*.

Note: Online Example (v1.3.1): quick_start/fit_example1

38.1.2 Simple Tut Example

Used in simple_tut_example to demonstrate running a *Tut Script*.

Note: Online Example (v1.3.1): quick_start/tut_example1

38.1.3 First order absorption model with peripheral compartment

Used in *Running Your First tut Script* to demonstrate running a *Tut Script*.

Note: Online Example (v1.3.1): quick_start/builtin_tut_example

Note: Example Data: https://product.popypkpd.com/docs/en/1.3.1//case-studies/quick_start/builtin_tut_example/builtin_tut_example_gen.pyml_output/synthetic_data.csv

38.2 Individual Model Summaries

38.2.1 Elimination Example with KE parameter

Used in *Elimination, Clearance and Volume of Distribution* to demonstrate elimination with the elimination rate constant, *KE*.

Note: Online Example (v1.3.1): indiv_examples/elimination/elim_ke_example

38.2.2 Elimination Example with Volume of Distribution

Used in *Volume of Distribution* to demonstrate elimination with the apparent volume of distribution, *V*.

Note: Online Example (v1.3.1): indiv_examples/elimination/elim_v_example

38.2.3 Elimination Example with Clearance

Used in *Clearance* to demonstrate elimination with clearance, *CL*.

Note: Online Example (v1.3.1): indiv_examples/elimination/elim_cl_example

38.2.4 One Compartment Model with Intravenous Dosing

Used in *One Compartment Model with Intravenous Dosing* to demonstrate a one compartment model with intravenous dosing.

Note: Online Example (v1.3.1): indiv_examples/compartment_models/odes/iv_one_cmp_cl

38.2.5 One Compartment Model with Absorption

Used in *One Compartment Model with Absorption* to demonstrate a one compartment model with absorption.

Note: Online Example (v1.3.1): indiv_examples/compartment_models/odes/dep_one_cmp_cl

38.2.6 Two Compartment Model with Intravenous Dosing

Used in *Two Compartment Model with Intravenous Dosing* to demonstrate a two compartment model with intravenous dosing.

Note: Online Example (v1.3.1): indiv_examples/compartment_models/odes/iv_two_cmp_cl

38.2.7 Two Compartment Model with Absorption

Used in *Two Compartment Model with Absorption* to demonstrate a two compartment model with absorption.

Note: Online Example (v1.3.1): indiv_examples/compartment_models/odes/dep_two_cmp_cl

38.2.8 Three Compartment Model with Intravenous Dosing

Used in *Three Compartment Model with Intravenous Dosing* to demonstrate a three compartment model with intravenous dosing.

Note: Online Example (v1.3.1): indiv_examples/compartment_models/odes/iv_three_cmp_cl

38.2.9 Three Compartment Model with Absorption

Used in *Three Compartment Model with Absorption* to demonstrate a three compartment model with absorption.

Note: Online Example (v1.3.1): indiv_examples/compartment_models/odes/dep_three_cmp_cl

38.2.10 Bolus Dose with no elimination.

Used in *Bolus Dose* to demonstrate a single bolus dose with no elimination.

Note: Online Example (v1.3.1): indiv_examples/dosing/bolus_tut

38.2.11 Infusion Duration Dose with no elimination.

Used in *Infusion Duration* to demonstrate a single infusion dose parametrised by duration, with no elimination.

Note: Online Example (v1.3.1): indiv_examples/dosing/inf_dur_tut

38.2.12 Infusion Rate Dose with no elimination.

Used in *Infusion Rate* to demonstrate a single infusion dose parametrised by rate, with no elimination.

Note: Online Example (v1.3.1): indiv_examples/dosing/inf_rate_tut

38.2.13 Gamma Dose with no elimination.

Used in *Gamma Dose* to demonstrate a single gamma dose with no elimination.

Note: Online Example (v1.3.1): indiv_examples/dosing/gamma_tut

38.2.14 Weibull Dose with no elimination.

Used in *Weibull Dose* to demonstrate a single weibull dose with no elimination.

Note: Online Example (v1.3.1): indiv_examples/dosing/weibull_tut

38.2.15 Repeated Bolus Dose with first order elimination.

Used in repeated_dosing to demonstrate a repeated bolus dose with first order elimination.

Note: Online Example (v1.3.1): indiv_examples/dosing/bolus_repeated_tut

38.2.16 Repeated Infusion Duration Dose with first order elimination.

Used in repeated_dosing to demonstrate a repeated infusion duration dose with first order elimination.

Note: Online Example (v1.3.1): indiv_examples/dosing/inf_dur_repeated_tut

38.2.17 Repeated Infusion Rate Dose with first order elimination.

Used in repeated_dosing to demonstrate a repeated infusion rate dose with first order elimination.

Note: Online Example (v1.3.1): indiv_examples/dosing/inf_rate_repeated_tut

38.2.18 Repeated Gamma Dose with first order elimination.

Used in repeated_dosing to demonstrate a repeated gamma dose with first order elimination.

Note: Online Example (v1.3.1): indiv_examples/dosing/gamma_repeated_tut

38.2.19 Repeated Weibull Dose with first order elimination.

Used in repeated_dosing to demonstrate a repeated weibull dose with first order elimination.

Note: Online Example (v1.3.1): indiv_examples/dosing/weibull_repeated_tut

38.2.20 Model containing additive error only and additive error only input data

Tut script used in *Residual Error Model* to demonstrate additive noise only model.

Note: Online Example (v1.3.1): indiv_examples/error_models/ao_tut

38.2.21 Model containing proportional error only, with proportional only data

Tut script used in *Residual Error Model* to demonstrate proportional noise only model.

Note: Online Example (v1.3.1): indiv_examples/error_models/po_tut

38.2.22 Model containing both proportional and additive error

Tut script used in *Residual Error Model* to demonstrate proportional and additive noise model.

Note: Online Example (v1.3.1): indiv_examples/error_models/pa_tut

38.2.23 Mixed error model fitted to mixed error data, but with incorrect variance definition

Fit script used in *Residual Error Model* to demonstrate fitting mis-specified proportional and additive noise model to proportional and additive noise synthetic data.

Note: Online Example (v1.3.1): indiv_examples/error_models/pa_gen_pa_fit_badvar

38.2.24 Sine circadian model

Used in *Example DERIVATIVES using x[TIME]* to demonstrate a PK/PD model with a circadian input function for a single individual.

Note: Online Example (v1.3.1): indiv_examples/pd_models/circ_sin

38.2.25 Direct PD Model

Used in *Example DERIVATIVES for PD Model* to demonstrate an individual PK/PD model with a bolus dose, one compartment PK and single PD compartment.

Note: Online Example (v1.3.1): indiv_examples/pd_models/direct_pd

38.2.26 Direct PD Model Simultaneous PK/PD Parameter fit

Used in *Example PREDICTIONS for PD Model* to demonstrate an individual PK/PD model with a bolus dose, one compartment PK and single PD compartment.

Note: Online Example (v1.3.1): indiv_examples/pd_models/direct_pd_simul

38.2.27 One Compartment Model with Absorption estimating KA

Used in *Uncertainty and Standard Errors* to show how we estimate confidence in a single parameter problem.

Note: Online Example (v1.3.1): indiv_examples/uncertainty/d1cmp_ka_stderr

38.2.28 One Compartment Model with Absorption estimating KA and V

Used in *Uncertainty and Standard Errors* to show how we estimate confidence in a two parameter problem.

Note: Online Example (v1.3.1): indiv_examples/uncertainty/d1cmp_ka_v_stderr

38.2.29 One Compartment Model with Absorption estimating KA and CL

Used in *Uncertainty and Standard Errors* to show how we estimate confidence in a two parameter problem.

Note: Online Example (v1.3.1): indiv_examples/uncertainty/d1cmp_ka_cl_stderr

38.2.30 One Compartment Model with Absorption estimating V and CL

Used in *Uncertainty and Standard Errors* to show how we estimate confidence in a two parameter problem.

Note: Online Example (v1.3.1): indiv_examples/uncertainty/d1cmp_v_cl_stderr

38.2.31 One Compartment Model with Absorption estimating KA, V and CL

Used in *Uncertainty and Standard Errors* to show how we estimate confidence in a three parameter problem.

Note: Online Example (v1.3.1): indiv_examples/uncertainty/d1cmp_ka_v_cl_stderr

38.3 Population Model Summaries

38.3.1 One Compartment Model with Absorption and Inter-subject Variance f[CL_isv]=0.2

Used in *Inter-Subject Variability (ISV)* to demonstrate inter-subject (or between-subject) variability.

Note: Online Example (v1.3.1): pop_examples/compartment_models/d1cmp_cl_isv

38.3.2 One Compartment Model with Absorption and Inter-subject Variance f[CL_isv]=0.01

Used in *Inter-Subject Variability (ISV)* to demonstrate inter-subject (or between-subject) variability.

Note: Online Example (v1.3.1): pop_examples/compartment_models/d1cmp_cl_isv_01

38.3.3 One Compartment Model with Absorption and Inter-subject Variance f[CL_isv]=0.5

Used in *Inter-Subject Variability (ISV)* to demonstrate inter-subject (or between-subject) variability.

Note: Online Example (v1.3.1): pop_examples/compartment_models/d1cmp_cl_isv_05

38.3.4 One Compartment Model with Absorption and no inter-subject Variance f[CL_isv]=0

Used in *Inter-Subject Variability (ISV)* to demonstrate inter-subject (or between-subject) variability.

Note: Online Example (v1.3.1): pop_examples/compartment_models/d1cmp_cl_isv_naive

38.3.5 One Compartment Model with Absorption and Inter-occasion Variance f[CL_isv]=0.2

Used in *Inter-Occasion Variability (IOV)* to demonstrate inter-occasion (or between-occasion) variability.

Note: Online Example (v1.3.1): pop_examples/compartment_models/d1cmp_cl_iov

38.3.6 One Compartment Model with Absorption and Inter-occasion Variance f[CL_isv]=0.5

Used in *Inter-Occasion Variability (IOV)* to demonstrate inter-occasion (or between-occasion) variability.

Note: Online Example (v1.3.1): pop_examples/compartment_models/d1cmp_cl_iov_05

38.3.7 One Compartment Model with Absorption and no inter-occasion Variance f[CL_iov]=0

Used in *Inter-Occasion Variability (IOV)* to demonstrate inter-occasion (or between-occasion) variability.

Note: Online Example (v1.3.1): pop_examples/compartment_models/d1cmp_cl_iov_naive

38.3.8 Diagonal matrix generation diagonal matrix fit using separate univariate normals

Used in *Modelling Correlation in Random Effects* to demonstrate correlation between random effects.

Note: Online Example (v1.3.1): pop_examples/re_covariance/gen_indep_fit_indep

38.3.9 Diagonal matrix generation diagonal matrix fit

Used in *Modelling Correlation in Random Effects* to demonstrate correlation between random effects.

Note: Online Example (v1.3.1): pop_examples/re_covariance/gen_diag_fit_diag

38.3.10 Diagonal matrix generation full matrix fit

Used in *Modelling Correlation in Random Effects* to demonstrate correlation between random effects.

Note: Online Example (v1.3.1): pop_examples/re_covariance/gen_diag_fit_full

38.3.11 Full matrix generation diagonal matrix fit

Used in *Modelling Correlation in Random Effects* to demonstrate correlation between random effects.

Note: Online Example (v1.3.1): pop_examples/re_covariance/gen_full_fit_diag

38.3.12 Full matrix generation full matrix fit

Used in *Modelling Correlation in Random Effects* to demonstrate correlation between random effects.

Note: Online Example (v1.3.1): pop_examples/re_covariance/gen_full_fit_full

38.3.13 Body Weight Covariate

Used in *Covariates* to demonstrate using weight as a covariate.

Note: Online Example (v1.3.1): pop_examples/covariates/weight_covariate

38.3.14 Depot + One compartment PK with BLQ

Used in *Generate BLQ observations and fit different error models* to demonstrate using *~rectnorm()* distribution to model observations below LLQ.

Note: Online Example (v1.3.1): pop_examples/blq/blq_pk_tut

38.3.15 Depot One Comp PK with BLQ observations set to LLQ

Used in *Generate BLQ observations and fit different error models* to demonstrate replacing **BLQ** observations with **LLQ**.

Note: Online Example (v1.3.1): pop_examples/blq/blq_pk_norm_fit

38.3.16 Depot One Comp PK with BLQ observations set to 0.5*LLQ

Used in *Generate BLQ observations and fit different error models* to demonstrate replacing **BLQ** observations with 0.5*|llq|.

Note: Online Example (v1.3.1): pop_examples/blq/blq_pk_norm_fit_half

38.3.17 Depot One Comp PK ignoring BLQ observations.

Used in *Generate BLQ observations and fit different error models* to demonstrate removing **BLQ** observations from data set.

Note: Online Example (v1.3.1): pop_examples/blq/blq_pk_norm_fit_ignore

CHAPTER

THIRTYNINE

RELEASE NOTES

This page lists the various *PoPy* releases to date:-

39.1 *PoPy* v1.3.0

Release date = 20 March 2026

39.1.1 Improvements

- Changes across the board that make processing much more efficient. Validation now runs in around 65% of the time compared with v1.2.3.
- Made PoPy faster to start up by postponing imports that may not be necessary.
- Moved temporary build files closer to the source so that they can more easily be deleted and do not clutter the user's home directory.

39.1.2 Bug Fixes

- Numerical errors were creeping into finite-differencing differentiation on some models that could result in instability of outputs not only between machines but on the same machine when using serial processing vs. parallel processing.
- Made PoPy more robust when installed to a Windows path that is long or that includes spaces.

39.2 *PoPy* v1.2.3

Release date = 2 Apr 2025

39.2.1 Improvements

- Option to specify the delimiter of the data file (e.g., a semicolon).
- Option to display a progress bar when iterating over individuals.
- Shortcut functions for the PREPROCESS block, making it easier to read Nonmem data files and to debug.
- Access to alternative Levenberg-Marquardt methods (e.g., the Trust Region Reflective algorithm) that may be more stable.
- Templating facility for config files whereby you can use ${ENVVAR} and PoPy will automatically substitute environment variable ENVVAR into the script.
- New "PRED_vs_TIME" and "{PRED}_vs_TIME" plot types that plot only predictions, without observation points.
- Faster "leastsq" r[X] optimization.
- A fail() function to the PREPROCESS to stop PoPy on demand during preprocessing.
- Access to initial states within the DERIVATIVES block via the s0[X] notation.
- Added output of random effect covariances as a correlation matrix.
- The *= operator is now accepted in DERIVATIVES.

39.2.2 Bug Fixes

- Initializing a state within a condition without an "else:" clause no longer leaves the state uninitialized (and unrecognized entirely).
- A zero bioavailability no longer causes the code to crash when there are multiple doses.
- The *= compound operator is now accepted in the DERIVATIVES for a single compartment (but not for directed flows between two compartments).
- TABLE now has access to the c[X] input data and supports the PREPROCESS block so that non-standard input data doesn't fail.
- FITSUM now processes non-standard input data without needing a PREPROCESS block.
- Expressions evaluating to log(0) now give an error to the user rather than generating unrunnable code.
- Bug in Solution file's "fx_opt_types_path" corrected so Solutions can be copied directly between folders.
- Closing the console window in Windows 7 now forcibly terminates all processes in the current PoPy run.
- You can now have an empty list for some covariates (e.g., t[DOSE]).
- Pseudocode that evaluates to log(0) now raises an exception rather then generating unrunnable code.
- Data files with "." or "-" as aliases for zero are now converted much earlier to avoid problems downstream.

- PREPROCESS is now propagated to all scripts where it might be necessary/useful.

39.3 *PoPy* v1.2.2

Release date = 21 Feb 2024

39.3.1 Improvements

- Better checking of scripts and reporting of errors found during checking.
- Better handling of long file names/paths.
- Better error reporting when trying to write to a file that is open elsewhere (e.g., Excel).

39.3.2 Bug Fixes

- Correct initialization of some random variables (e.g., Bernoulli)
- Copied DATA_FIELDS to solutions such that non-standard columns are recognized rather than looking for default column names.
- Passed PREPROCESS code to Grph scripts so that preprocessed input observations/covariates are loaded correctly.
- Removed erroneous point at dose events in spaghetti plots.
- Corrected the way that string columns are processed in auto-generated code so that conditions on string fields are evaluated correctly.

39.4 *PoPy* v1.2.1

Release date = 15 Dec 2023

39.4.1 Improvements

- Simplified the launch interface - e.g., to "popy run …" rather than "popy_run" - to reduce the number of additional scripts bundled with PoPy.
- Retired the "imconv" and "edit" tools, trusting the user to use their preferred tools for image conversion and text file editing.
- Retired the CPPODE solver in favour of the superior CPPLSODA solver.

39.4.2 Bug Fixes

- Added measures to make PoPy more tolerant of long path names that exceed the Windows limit of 260 characters.
- Fixed errors when a single epoch contains more than one occasion without a reset separating occasions.

39.5 *PoPy* v1.2.0

Release date = 6 Nov 2023

39.5.1 Improvements

- Scatter plots implemented for rapid diagnostics of model fits.
- Tabular outputs now available (similar to Nonmem's $TABLE directive).
- Conditional Weighted Residuals added as a diagnostics measure.
- Sequential variables can now be defined as an arbitrary list.
- Likelihoods exported on a per-row basis for detailed analysis.
- Improvements to efficiency and reliability when using parallel processing.
- Simplifications to the output data structures, with fewer subfolders and better consistency in file locations.
- Added option to specify parallelism by % of CPU load rather than an absolute number of processors.

39.5.2 Bug Fixes

- Random effects with a fixed variance of zero are now correctly treated as constants, and any associated fixed effects treated as unassociated fixed effects.
- Tabulated fixed effects were previously incorrect at times, but are now outputted correctly.
- Fixed an indexing bug when updating individuals' parameter values.
- Clearer numbering of populations in multi-population simulations to avoid confusion by both users and computers.
- Fixed bug in normal distribution objects after reseeding with a known value.
- Corrected the way that flows of the form "d[X] -= x + y" are evaluated.
- Improved the way variable substitutions are handled in the PREDICTIONS block.
- Fixed bugs arising when trying to write files to folders that don't (yet) exist.
- Corrected the way that analytic solvers deal with parameter sensitivity.

39.6 *PoPy* v1.1.2

Release date = 11 Mar 2022

39.6.1 Improvements

- Faster processing for models with lots (tens of thousands) of observation rows that have a normally distributed error model.
- Parameterized compartment definitions so that many compartments can be defined in a single line.
- Better feedback when calculations result in a not-a-number (NaN) that derails the fitting process.
- Assumes that any uninitialized d[X] are initialized to zero, so you can add to flows without having to explicitly initialize them first.
- Faster evaluation of normally-distributed likelihoods.
- Extended sequential() distribution now allows arbitrary splitting of tree levels, e.g., c[AMT] = sequential([0, 100, 200]).
- Better code parsing flags up errors in the script earlier in the process.

39.6.2 Bug Fixes

- Covariance matrix (OMEGA) values that were defined as being constant were being modified by the JOE fitter.
- Models containing fixed effects that are constant would sometimes crash due to an indexing bug in the subset of all fixed effects.
- Incorrect calculation in ds/dm when using mixed models (with both a closed-form compartmental model and ODEs).
- IMP under parallel processing tried to recompute proposal distributions on every process rather than just on the master process.
- IMP used an approximate Lambda when it should have been using an exact one (and vice versa).
- Parsing the PREPROCESS block would sometimes alter its contents, causing faults downstream.
- Failure due to long filenames avoided by using symbolic links and raising warnings rather than errors under recoverable conditions.
- Covariances (OMEGAs) were being updated when they were defined as constants in some methods (e.g. JOE).
- Constant fixed effects were causing indexing problems in some fitting algorithms.
- Output folders created whenever they are needed (which wasn't always the case).
- Sim failed when a solution file had a different DATA_FIELDS content from the corresponding Gen file.

- Fixed effects "associated" with an r[X] that is effectively constant (e.g., mean=0 and variance=0) were not being updated at all, rather than being updated using alternative (e.g., gradient-based) methods.
- Incorrect covariance matrix when the model included no random effects.
- Tabulated f[X] values were all the same.
- Tutsum failed to copy the compartment diagram from Gensum.

39.7 *PoPy* v1.1.1

Release date = 22 Oct 2021

39.7.1 Improvements

- More stable estimation when using forward sensitivity equations.
- Improvements to VPC generation
- Some improvements in speed during fitting, especially for situations where there are lots of bolus doses per subject.
- Better memory management when fitting in parallel.
- Quotes in column headers of the data file are now permitted.

39.7.2 Bug Fixes

- Fixed various bugs with fitting in parallel

39.8 *PoPy* v1.1.0

Release date = 22 Sep 2021

39.8.1 Improvements

- Reduced RAM usage when running *PoPy* in parallel using *MPI*. Previous version shared all subject data on all processors. New parallel code spreads subject data across all processors and avoids unnecessary duplication. Improves performance on large multi-core machines.
- Increased number of parameters that the *ND* fitting method and *BOBYQA* optimiser can process, previously restricted to 35, now increased to 200.
- Change the colour of the dosing lines in individual subject plots to yellow (formerly black), this enables more informative graphs for subjects who have a lot of separate doses.
- Improved the speed and accuracy of the `r[X]` optimisation using more analytic gradient results.

- Added ability to create prediction corrected (pred_corr) and prediction variance corrected (predvar_corr) *VPC* output via the "norm_method" parameter of a *Vpc Script*.
- Added ability to create stratified *VPC* output via the "split_field" parameter of a *Vpc Script*. A separate vpc is output for each unique value in a specified `c[X]` data variable.
- Made the *MSim Script* and *Vpc Script* code more RAM efficient.
- Made models with a large number of dosing events per subject run more efficiently.

39.8.2 Bug Fixes

- Fixed bug when passing string data into *Cython* functions.

39.9 *PoPy* v1.0.5

Release date = 13 May 2021

39.9.1 Improvements

- Added new *ND* fitting method, the latest and most robust *PoPy* estimation approach.
- Improved fitting and lower *ObjV* using *ND* method.
- Upgraded base *Python* distribution of *PoPy* from version 3.5 to version 3.8. This allows us access to more up-to-date *Python* packages and functionality.
- Improved automatic compartment diagram generation with "in" and "out" nodes.
- Improved automatic checking of input scripts.

39.9.2 Bug Fixes

- Improvements in error catching and reporting when running *PoPy* in parallel using *MPI*.
- Fixed bug when using backslashes in *DERIVATIVES* section.

39.10 *PoPy* v1.0.4

Release date = 23 Dec 2020

39.10.1 Improvements

- Added *CPPLSODA C++* based implementation of *LSODA ODE_SOLVER*, which runs faster than *SCIPY_ODEINT* within *PoPy* and returns similar results.
- Improved *PARALLEL* performance of *FOCE* algorithm.
- Improved fitting of *FOCE* method to achieve lower *ObjV* in many models.
- Revised documentation.

39.10.2 Bug Fixes

- Bugs in gradient computation due to variable `c[X]` within individual data fixed.
- Fixed bugs in *standard error* computation, now more reliable.

39.11 *PoPy* v1.0.3

Release date = 05 Sep 2020

39.11.1 Improvements

- Added implementation of an *FOCE Fitting Method* to complement the previous *JOE Fitting Method*. Note we now recommend running the *JOE* fitter, then the *FOCE* fitter in serial to get the best parameter fitting results in *PoPy*.
- New *~rectnorm()* distribution for fitting to **BLQ** data, using *JOE* or *FOCE* methods without using the more computationally expensive *LAPLACE* objective value. See *Generate BLQ observations and fit different error models*.
- Also new **BLQ** related *~cennorm()* distribution and *~truncnorm()* distribution.
- New *~binomial()* distribution likelihood.
- Improved the speed and quality of the noise `f[X]` parameter estimates for *JOE* fitting.
- New plots show the final `f[X]` population fit (same for all individuals, with `r[X]` zero) and the individual fit (*i.e.* empirical bayes estimates for `r[X]` optimised for each individual). Previously we plotted only the initial unoptimised `f[X]` fit and the final `f[X]` individual fits.
- The temporary functions generated by *PoPy* (based on the config file) are now written into a single module on disk. This single module compiles quicker using *Cython*, compared to having each temporary function in it's own module.
- Added *PREPROCESS* section to multiple scripts, for example a *Fit Script*. This allows the user to easily filter and alter the data set, using simple *Python* code, before fitting a model, without changing the *data file* on disk.
- Added *POSTPROCESS* section for multiple scripts, for example a *Gen Script*. This allows the user to easily alter or filter a synthetically generated data set, before saving to disk.

39.11.2 Bug Fixes

- Fixed bug in *PREDICTIONS* section that did not handle multiple observations correctly in some instances, when some rows should be ignored due to *Python* 'if' statements.
- Fixed interpretation of '+inf' and '-inf' labels in *EFFECTS*.

39.12 *PoPy* v1.0.2

Release date = 18 Sep 2019

39.12.1 Improvements

- Added ability to process scripts in *PARALLEL*. This speeds up computation time for models with a large number of individuals and takes advantage of multi-core computers.
- Created *POSTPROCESS* section to allow filtering of data create by a *PoPy* model. For example with a *Gen Script*
- Added ability to process multiple files in a directory with one command using '*' syntax in order to *Running multiple scripts*, *Checking multiple scripts*, *Format multiple scripts*, edit multiple scripts and *Open multiple html files*
- Speed improvements to core *JOE* algorithm, *Standard Errors* computation and plotting using *Grph Script*.
- *MFit Script* now outputs a summary of the fitting results for all populations to a single .csv file.
- Displaying the diagonal elements of `f[X]` variance matrices at each iteration. The off diagonal `f[X]` elements are only displayed at the end of fitting, due to space considerations.
- Simplified the structure of old 'LEVEL_PARAMS', replaced with *EFFECTS* which removes the redundant 'split_field' and 'split_dict' attributes.
- Renamed the 'output_mode' values in *OUTPUT_SCRIPTS*, using shorter names 'gen_script_and_run' -> 'run', 'gen_script_only' -> 'create', 'no_output' -> 'none'.
- Renamed the 'FINITE_DIFF' method in *COVARIANCE* to 'SANDWICH'.

Note scripts in v1.0.1 and v1.0.0 format can be automatically updated to v1.0.2 using the *popy format* tool.

39.12.2 Bug Fixes

- Made *float_format* change format of all float values in output scripts and summary html outputs. Only applied to subset before.
- Fixed bug in iteration number when displaying *ObjV* over time obj_vs_time_fit_example1

39.13 *PoPy* v1.0.1

Release date = 28 Jun 2019

This is mainly a bug fix release of *PoPy*

39.13.1 Improvements

- The *Standard Errors* are now output in an easy to read format, instead of the whole hessian matrix.
- Changed the configuration file entry required to compute standard errors see *COVARIANCE*.
- Log files apart from the main log file are now stored in a '_log' sub directory.
- The *popy run* tool now asks the user if they want to proceed if the main log file already exist for a given script. Previously only the existence of the output folder was checked. The can still skip this check using the '-o' option.

39.13.2 Bug Fixes

- Licence manager was giving a date error in North American time zones.
- Bug in configuration file parser for *DERIVATIVES* section when encountering round brackets (as opposed to expected curly brackets) in *Dosing Functions*.
- Documentation contents page indexing in PDF was incorrect.

39.14 *PoPy* v1.0.0

Release date = 24 May 2019

This is the first version of *PoPy*, so there are no bug fixes or new features only the initial implementation.

Or in other words, all the features and bugs are new!

CHAPTER

FORTY

NOTEPAD++

Notepad++ is a text editor that uses tabs and context highlighting, both of which are helpful when editing *PoPy* script files.

40.1 Install *Notepad++*

Download the installer from https://notepad-plus-plus.org/download/ and run it.

If you install the 64-bit binary installer the default install directory is `C:\Program Files\Notepad++` but you may install *Notepad++* at another location.

40.2 Configure *Notepad++* Tabs for Indentation

PoPy is written in *Python*, and *Python* does **not** like tabs for indentation. It is essential that you set this from within Notepad++:-

```
Settings->Preferences->Indentation->Indent Using: Space character(s)
```

The default tab size is 4, which is a sensible choice.

Note this means that when you hit the tab key you will get 4 space characters instead of a tab character. To check that this is the case do:-

```
View->Show Symbol->Show Space and Tab
```

Then *Notepad++* will give you a faint orange dot for a space character and an arrow for a tab. You are advised to delete any tabs you have in your *.pyml files.

Note that it is still possible to introduce tabs into your text file accidentally using cut and paste.

Python using spaces instead of the more conventional curly brackets is an endearing, perhaps controversial, language feature. (See https://jayconrod.com/posts/101/how-python-parses-white-space for a fairly balanced discussion of the pros and cons of white spacing.)

However *PoPy* uses *Python*, so it is white space for us.

40.3 Configure *Notepad++* Colouring

It is highly advisable to load in the PoPy xml syntax highlighting file. You do this by opening *Notepad++* and doing:-

```
Language->User Defined Language->Define your language...
Import...
```

Then selecting the `c:\PoPy\conf\notepadplusplus_popy.xml` file.

You should then be able to open any *.pyml file and see the variables coloured like this:-

```
DERIVATIVES: |
    # s[DEPOT,CENTRAL,PERI] = @dep_two_cmp_cl{dose:@bolus{amt:c[AMT]}}
    d[DEPOT] = @bolus{amt:c[AMT]} - m[KA]*s[DEPOT]
    d[CENTRAL] = (
        m[KA]*s[DEPOT] - s[CENTRAL]*m[CL]/m[V1]
        - s[CENTRAL]*m[Q]/m[V1]  + s[PERI]*m[Q]/m[V2]
    )
    d[PERI] = s[CENTRAL]*m[Q]/m[V1] - s[PERI]*m[Q]/m[V2]
PREDICTIONS: |
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[DV_CENTRAL]**2
    c[DV_CENTRAL] ~ norm(p[DV_CENTRAL], var)
```

As opposed to the default (plain) *Notepad++* text display, like this:-

```
DERIVATIVES: |
    # s[DEPOT,CENTRAL,PERI] = @dep_two_cmp_cl{dose:@bolus{amt:c[AMT]}}
    d[DEPOT] = @bolus{amt:c[AMT]} - m[KA]*s[DEPOT]
    d[CENTRAL] = (
        m[KA]*s[DEPOT] - s[CENTRAL]*m[CL]/m[V1]
        - s[CENTRAL]*m[Q]/m[V1]  + s[PERI]*m[Q]/m[V2]
    )
    d[PERI] = s[CENTRAL]*m[Q]/m[V1] - s[PERI]*m[Q]/m[V2]
PREDICTIONS: |
    p[DV_CENTRAL] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[DV_CENTRAL]**2
    c[DV_CENTRAL] ~ norm(p[DV_CENTRAL], var)
```

Note that it might be necessary to restart *Notepad++* to get the colouring file to work.

We find that the variable colouring, makes model editing easier and less error prone. For example, if you misspell a section header e.g "DERVIATIVES", then you will notice because the section header will not appear in bold.

The colouring file also just makes PK/PD models look nicer.

CHAPTER

FORTYONE

VALIDATION EXAMPLES

The validation examples are often taken from the publicly available *DDMoRe* repository. Typically these are a Nonmem control file and Nonmem data file that together illustrate fitting a PK/PD model.

You can see the validation examples supplied with *PoPy* in the `c:\PoPy\validation` folder.

Each subdirectory contains a 'readme.txt' file that gives a brief overview of the example and how it was created from the *DDMoRe* equivalent.

Table 1 lists each of the models in this folder:-

Table 1: Validation Models

Name	Summary	Link	Citation
ddmore0061	Combined *bolus* + *infusion* PK model.	DDMoRe: 0061	[Harling2015]
ddmore0093	Circadian function with covariates.	DDMoRe: 0093	[Cheung2015]
ddmore0215	Markov and dropout hazard.	DDMoRe: 0215	[Girard2012]
ddmore0238	PopPK with *infusion* dosing + **IOV**.	DDMoRe: 0238	[Germovsek2017]

CHAPTER

FORTYTWO

TROUBLESHOOTING

42.1 Win10 Increase max path length

Note if you use *PoPy* with deeply nested folder structures it is recommended that you increase the max file path length from the default Windows 10 limit of 260 characters, see here:-

https://docs.microsoft.com/en-us/windows/win32/fileio/maximum-file-path-limitation

For your convenience we have supplied the registry editor file `conf/win10_allow_long_paths.reg`.

If you are using Windows 10 and have admin privileges, you can simply click on this file to remove the 260 character limit.

BIBLIOGRAPHY

[HookerEtAl2007] Hooker AC, Staatz CE, Karlsson MO. "Conditional weighted residuals (CWRES): a model diagnostic for the FOCE method." *Pharm Res.* 2007 Dec; 24(12):2187-97. doi: 10.1007/s11095-007-9361-x. Epub 2007 Jul 6. PMID: 17612795.

[Ahn2008] Ahn J.E., Karlsson M.O., Dunne A., Ludden T.M. Likelihood based approaches to handling data below the quantification limit using NONMEM VI. Journal of Pharmacokinetics and Pharmacodynamics. 2008 Aug;35(4):401-21.

[Bauer2009] Beal, S.L. , Sheiner, L.B., Boeckmann, A., & Bauer, R.J. NONMEM User's Guides. (1989-2009), Icon Development Solutions, Ellicott City, MD, USA, 2009.

[Beal2001] Beal, S.L Ways to Fit a PK Model with Some Data Below the Quantification Limit Journal of Pharmacokinetics and Pharmacodynamics volume 28, pages481–504(2001)

[Cheung2015] Amy Cheung, Predictive QT project, DDMODEL 0000093, 2015, (repository.ddmore.eu/model/DDMODEL00000093).

[Christensen1980] Finn Norring Christensen, Flemming Yssing Hansen and Helle Bechgaard Physical interpretation of parameters in the Rosin-Rammler-Sperling-Weibull distribution for drug release from controlled release dosage forms Journal of Pharmacy and Pharmacology, Volume 32, Issue 1, September 1980, pp 580-582

[DennisSchnabel1987] J. E. Dennis, Jnr. and R. B. Schnabel Numerical Methods for Unconstrained Optimization and Nonlinear Equations Society for Industrial and Applied Mathematics ISBN-10: 0898713641

[Germovsek2017] Eva Germovsek, Population PK model for gentamicin, DDMODEL 0000238, 2017, (repository.ddmore.eu/model/DDMODEL00000238).

[Girard2012] P. Girard, Simultaneous ocular adverse event and dropout model of pimasertib, DDMODEL 0000215, 2012, (repository.ddmore.eu/model/DDMODEL00000215).

[Harling2015] Kajsa Harling Population PK of gentamicin in cancer patients with time-varying covariates, DDMODEL 0000061, 2015, (repository.ddmore.eu/model/DDMODEL00000061).

[Holford1996] N. H. Holford, A size standard for pharmacokinetics. Clin Pharmacokinet. 1996 May; 30(5):329-32.

[KarlssonSheiner1993] Karlsson MO and Sheiner LB The importance of modeling interoccasion variability in population pharmacokinetic analyses. J Pharmacokinet Biopharm. 1993 Dec; 21(6):735-50.

[Lindstrom1990] L Lindstrom, M & Bates, Mark. (1990). Nonlinear Mixed Effects Models for Repeated Measures Data. Biometrics. 46. 673-87. 10.2307/2532087.

[Millar2011] Russell B. Millar Maximum Likelihood Estimation and Inference - With Examples in R, SAS, and ADMB John Wiley & Sons, 2011

[MouldUpton2012] D. R. Mould and R. N. Upton Basic Concepts in population Modeling, Simulation, and Model-Based drug development CPT Pharmacometrics Syst Pharmacol. 2012 Sep; 1(9): e6.

[MouldUpton2013] D. R. Mould and R. N. Upton Basic concepts in population modeling, simulation, and model-based drug development-part 2: introduction to pharmacokinetic modeling methods. CPT Pharmacometrics Syst Pharmacol. 2013 Apr 17; 2: e38.

[NocedalWright2006] J. Nocedal and S. Wright Numerical Optimization Springer ISBN-10: 0387303030

[Piotrovskii1987] Vladimir K. Piotrovskii The use of Weibull distribution to describe thein vivo absorption kinetics Journal of Pharmacokinetics and Biopharmaceutics, Volume 15, Issue 6, December 1987, pp 681–686

[Powell2009] M. J. D. Powell The BOBYQA algorithm for bound constrained optimization without derivatives Report No. DAMTP 2009/NA06, Centre for Mathematical Sciences, Cambridge, UK.

[Radhakrishnan1994] Radhakrishnan, Krishnan & C. Hindmarsh, Alan. (1994). Description and Use of LSODE, the Livermore Solver for Ordinary Differential Equations.

[RowlandTozer2012] M. Rowland and T. N. Tozer Clinical Pharmacokinetics and Pharmacodynamics: Concepts and Applications Lippincott Williams & Wilkins, 2012

[Sheiner1980] Sheiner L.B., Beal S.L. Evaluation of methods for estimating population pharmacokinetics parameters. I. Michaelis-Menten model: routine clinical pharmacokinetic data. J. Pharmacokinet. Biopharm 86553–571.1980

[UptonMould2013] R. N. Upton and D. R. Mould Basic Concepts in Population Modeling, Simulation, and Model-Based Drug Development: Part 3—Introduction to Pharmacodynamic Modeling Methods CPT Pharmacometrics Syst Pharmacol. 2014 Jan; 3(1): e88.

[Wang2007] Wang, Yaning. (2007). Derivation of various NONMEM estimation methods. Journal of pharmacokinetics and pharmacodynamics. 35. 249. 10.1007/s10928-008-9083-7.

INDEX

N

O

P

R

S

T

V

W

Y