The PoPy Manual

Release 1.1.2

David Cristinacce
Andrew Cristinacce
Phil Tresadern
James Wright

Mar 09, 2022

CONTENTS

1 Getting Started Guide **5**
1.1 Introducing PoPy . . . 5
1.2 Install PoPy . . . 6
1.3 Configure Editor . . . 8
1.4 Fitting a Simple PopPK Model using PoPy . . . 10
1.5 PoPy Data Format . . . 17
1.6 Generate data and Fit using Simple PopPK Model . . . 21
1.7 Typical Workflows . . . 28

2 Principles of Pharmacokinetics **31**
2.1 Elimination, Clearance and Volume of Distribution . . . 31
2.2 Compartment Models . . . 33
2.3 Dose Administration . . . 43
2.4 Residual Error Model . . . 50
2.5 Estimating Model Parameters . . . 53
2.6 Uncertainty and Standard Errors . . . 56

3 Population Models in PoPy **61**
3.1 Inter-Subject Variation (ISV) . . . 61
3.2 Inter-Occasion Variation (IOV) . . . 65
3.3 Modelling Correlation in Random Effects . . . 68
3.4 Covariates . . . 74

4 PoPy Example Models **79**
4.1 Creating Example Scripts . . . 79
4.2 Fitting a Two Compartment PopPK Model . . . 80
4.3 Generate a Two Compartment PopPK Data Set . . . 88
4.4 Generate data and Fit using a Two Compartment Model . . . 93
4.5 Generate multiple data sets and Fit using a Two Compartment Model . . . 98
4.6 Generate BLQ observations and fit different error models . . . 104

5 PoPy for Nonmem Users **110**
5.1 Nonmem Data to PoPy Data File . . . 110
5.2 Nonmem to PoPy Data conversions using P2NDAT and N2PDAT Scripts . . . 113
5.3 Nonmem control file to PoPy Fit Script . . . 120
5.4 Nonmem Advan1,2,3,4,11,12 to PoPy analytic compartment models . . . 127
5.5 DDMoRe0061 Conversion Example . . . 135
5.6 DDMoRe0093 Conversion Example . . . 139
5.7 DDMoRe0238 Conversion Example . . . 143

6 PoPy Reference Guide **151**

6.1 Open a PoPy Command Prompt . . . 151
6.2 PoPy Activation . . . 153
6.3 Configure PoPy . . . 156
6.4 PoPy Website . . . 157
6.5 Validate PoPy . . . 159
6.6 Command Line Tools . . . 161
6.7 Script File Formats . . . 171
6.8 Script File Sections . . . 178
6.9 Script File Elements . . . 219
6.10 Script File Outputs . . . 245
6.11 PoPy Quick Reference Guide . . . 258

7 Appendices **260**
7.1 Glossary . . . 260
7.2 HTML Summary Links . . . 263
7.3 Troubleshooting . . . 272
7.4 Bug Reporting . . . 276
7.5 Credits . . . 278
7.6 Release Notes . . . 278

Bibliography **284**

LIST OF FIGURES

1.1 One compartment model with bolus dosing for *Fit Script*. Here `c[AMT]` is the amount of the bolus dose. `m[KE]` is the elimination rate and `s[CENTRAL]` is the current amount in the central compartment. See *Variable Types* for a summary of the prefixes used in PoPy. 10
1.2 Visual Predictive Check for Simple PopPK model. 15
1.3 One compartment model with bolus dosing for *Tut Script*. Here `c[AMT]` is the amount of the bolus dose. `m[KE]` is the elimination rate and `s[CENTRAL]` is the current amount in the central compartment. 22
1.4 Hierarchy of child scripts for a parent *Fit Script*. 29
1.5 Hierarchy of child scripts for a parent *Tut Script*. 29
1.6 Hierarchy of child scripts for a parent *MTut Script*. 30

2.1 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.) . . . 32
2.2 Concentration vs Time curve for an intravenously administered 100 mg bolus dose with first order elimination 33
2.3 Compartment diagram for a one compartment model with intravenous dosing. 34
2.4 Concentration vs Time curve for a bolus dose, intravenously applied to the **Central** compartment, with first order elimination. 35
2.5 Compartment diagram for a one compartment model with absorption. 37
2.6 Concentration vs Time curve for a one compartment model with absorption 38
2.7 Compartment diagram for a two compartment model with intravenous dosing. 39
2.8 Concentration vs Time curve for a two compartment model with intravenous dosing 39
2.9 Compartment diagram for a two compartment model with absorption 40
2.10 Concentration vs Time for a two compartment model with absorption 41
2.11 Compartment diagram for a three compartment model with intravenous dosing 41
2.12 Amount vs. Time for a three compartment model with intravenous dosing 42
2.13 Compartment diagram for a three compartment model with absorption 43
2.14 Concentration vs Time for a three compartment model with absorption 43
2.15 Cumulative amount following a bolus dose with no elimination 45
2.16 Cumulative amount following a infusion dose of 95 mg over a duration of 19 min with no elimination 46
2.17 Cumulative amount following a infusion dose of 95 mg at a rate of 5 mg/min with no elimination 47
2.18 Cumulative amount following a Gamma dose with no elimination 48
2.19 Cumulative amount following a Weibull dose with no elimination 49
2.20 Concentration vs Time for a one compartment model with absorption, without measurement error (noise). 51
2.21 Concentration vs Time for a one compartment model with absorption, plus additive noise. . 52
2.22 Concentration vs Time for a one compartment model with absorption, plus proportional noise. 52
2.23 Concentration vs Time for a one compartment model with absorption, plus both proportional and additive noise 53

2.24 *CL* vs *V* surface plot of the Objective Function Value (*ObjV*) for a bolus dose, administered to a one compartment model with absorption . . . 54
2.25 CL vs V surface plot overlaid with the path taken by the optimization algorithm to the minimum. '+' shows starting values, square shows minimum values. . . . 55
2.26 *ObjV* vs *KA* with other parameters fixed at their true values . . . 56
2.27 *ObjV* vs *KA* with other parameters fixed at their true values, plus the quadratic approximation at the minimum . . . 57
2.28 Likelihood vs *KA* with other parameters fixed at their true values, plus the gaussian approximation at the maximum . . . 57
2.29 *ObjV* surface with population parameters sampled from the approximating gaussian and 90% confidence intervals in red for *KA* and *CL* . . . 59
2.30 *ObjV* surface with population parameters sampled from the approximating gaussian and 90% confidence intervals in red for log(*KA*) and log(*CL*) . . . 59

3.1 Simulated prediction curves for a population with independent inter-subject variability on parameters CL and V . . . 69
3.2 Simulated prediction curves for a population with correlated inter-subject variability on parameters CL and V. . . . 72
3.3 Simulated prediction curves for a population of 40 individuals with weight-dependent clearance, CL, and volume of distribution, V . . . 75

4.1 Two compartment model with depot dosing for *Fit Script*. Here `c[AMT]` is the size of the bolus dose specified in the *data file*. `m[KA]`, `m[CL]`, `m[Q]`, `m[V1]`, `m[V2]` are all *MODEL_PARAMS* to be estimated for each individual. . . . 81
4.2 Visual Predictive Check for Complex PopPK model. . . . 87
4.3 Two compartment model with depot dosing for *Gen Script*. This is the same model as Fig. 4.1. 89
4.4 Two compartment model with depot dosing for *Tut Script*. . . . 93
4.5 Two compartment model with depot dosing for *MTut Script*. . . . 99
4.6 One compartment model with depot dosing used to generate and fit **BLQ** PK data. . . . 104

6.1 PoPy dos prompt . . . 151
6.2 Plain command prompt . . . 152
6.3 PoPy command prompt . . . 152
6.4 PoPy command prompt in the 'c:\Users\david\my_work' folder. . . . 153
6.5 *EFFECTS* structure with **two** levels . . . 189
6.6 *EFFECTS* structure with **three** levels . . . 191
6.7 Two compartment model with depot dosing, computed automatically from *DERIVATIVES* section. 202
6.8 direct PD model, computed automatically from *DERIVATIVES* section. . . . 203
6.9 circadian PD model, computed automatically from *DERIVATIVES* section. . . . 204

LIST OF TABLES

1.1 PoPy Documentation Structure 5
1.2 System Requirements for PoPy 6
1.3 Model predictions vs original data points for first three individuals 12
1.4 Objective function at each iteration for simple PopPK example 14
1.5 PoPy data fields 17
1.6 PoPy time reset example 18
1.7 PoPy single dose type example 18
1.8 PoPy multi dose type example 19
1.9 PoPy single observed field example 20
1.10 PoPy single observed field missing data example 20
1.11 PoPy multiple observed fields 21
1.12 Scripts output by a tutorial script 24
1.13 Fitted model PK curves vs true model PK curves for first three individuals 25
1.14 Comparison of initial, fitted and true `f[KE]` values 25
1.15 Comparison of initial, fitted and true `f[KE_isv]` values 25
1.16 Comparison of initial, fitted and true `f[PNOISE]` values 25
1.17 Scripts output by a multi tutorial script 27
1.18 Fitted model vs true scatter plots for `f[KE]`, `f[KE_isv]` and `f[PNOISE]` 28

2.1 Repeated dosing, administered to a one compartment model with first order elimination . . . 50
2.2 Comparison of ObjV surface (left) with quadratic approximation (right) 58

3.1 Population graphs with increasing ISV: (left) CL_isv=0.01; (centre) CL_isv=0.2; (right) CL_isv=0.5 63
3.2 Population graphs with increasing ISV: (left) CL_isv=0.2, CL_iov=0.1; (right) CL_isv=0.5, CL_iov=0.01 66

4.1 Model predictions vs original data points for first three individuals 82
4.2 Objective values vs iteration number and time 85
4.3 Synthetic data plots for first three individuals 90
4.4 First 10 rows of 'synthetic_data.csv' file 92
4.5 Fitted model PK curves vs true model PK curves for first three individuals 95
4.6 Comparison of main initial, fitted and true `f[X]` values 96
4.7 Comparison of fitted and true proportional noise `f[X]` values 96
4.8 Comparison of fitted and true isv variance diagonal `f[X]` values 96
4.9 Scripts output by a multi tutorial script 101
4.10 Fitted model vs true scatter plots for `f[KA]` and `f[CL]` 102
4.11 Fitted model vs true scatter plots for `f[Q]` and `f[V1]` and `f[PNOISE]` 102
4.12 Generated observations + true underlying model PK curves for first three individuals 104
4.13 Fitted model PK curves vs true model PK curves for first three individuals 105
4.14 Comparison of initial, fitted and true `f[X]` main values 105
4.15 Comparison of initial, fitted and true `f[X]` noise values 105

4.16 Fitted model ~norm() distribution error curves vs **LLQ** model PK curves for first three individuals 106
4.17 Fitted model ~norm() distribution error curves vs half **LLQ** model PK curves for first three individuals . . . 107
4.18 Fitted model ~norm() distribution error curves vs ignore **BLQ** model PK curves for first three individuals . . . 108

5.1 Nonmem to PoPy Data . . . 110
5.2 Nonmem EVID to PoPy TYPE . . . 111
5.3 Nonmem id example . . . 111
5.4 Nonmem id time . . . 111
5.5 Nonmem cmt to PoPy dose name . . . 112
5.6 Nonmem DV to PoPy named fields . . . 113
5.7 Nonmem DV/MDV to PoPy flag fields . . . 113
5.8 Original data in PoPy format (first ten rows) . . . 114
5.9 Output data in Nonmem format (first ten rows) . . . 115
5.10 Comparing PoPy 'fit_example1_data.csv' and Nonmem 'fit_example1_nm_data.csv' . . . 115
5.11 Output data in Nonmem format (first ten rows) . . . 118
5.12 Comparing Nonmem 'fit_example1_nm_data.csv' and PoPy 'fit_example1_data_v2.csv' . . . 118
5.13 Nonmem to PoPy Fitting . . . 121
5.14 Nonmem to PoPy Analytic ODEs . . . 128
5.15 Main data fields for Nonmem (first six rows) . . . 135
5.16 Extra data fields for PoPy (first six rows) . . . 136
5.17 Main data fields for Nonmem and PoPy . . . 140
5.18 Main data fields for Nonmem (first 8 rows) . . . 143
5.19 Extra data fields for PoPy (first 8 rows) . . . 144

6.1 Validation Models . . . 160
6.2 PoPy Command Line Tools . . . 161
6.3 Script Format . . . 172
6.4 Common Script Sections . . . 179
6.5 float format options . . . 180
6.6 Script File Elements . . . 219
6.7 PoPy Variable Types . . . 220
6.8 Probability Distributions . . . 221
6.9 Matrices . . . 233
6.10 Dosing Functions . . . 237
6.11 Compartment Model Functions using '_cl' . . . 239
6.12 Compartment Model Functions using '_k' . . . 242
6.13 Script Outputs . . . 245
6.14 .csv file output by MComp Script . . . 256
6.15 PoPy Quick Reference . . . 259

7.1 Solutions to common PoPy issues . . . 272

CHAPTER

ONE

GETTING STARTED GUIDE

1.1 Introducing PoPy

1.1.1 Overview

PoPy (pronounced pop-eye, as in the sailor man) is new software to support population modelling of PK/PD (pharmacokinetic/pharmacodynamic) data using the *Python* Programming Language.

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

PoPy's features include:-

- Nonlinear *mixed-effect* models
- *Bolus* and *infusion* dosing regimens
- *Inter-occasion variability*
- *Continuous* and *categorical* likelihoods
- *PK* and *PD* models

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

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

1.1.2 Documentation Structure

See Table 1.1 for layout of this documentation:-

Table 1.1: PoPy Documentation Structure

Section	Contents
Getting Started Guide	Read this first to get familiar with PoPy
Principles of Pharmacokinetics	A summary of **individual** PK/PD models using PoPy syntax
Population Models in PoPy	A summary of **population** PK models using PoPy syntax
PoPy Example Models	Examples of using PoPy for PK/PD analysis
PoPy for Nonmem Users	Guidance for converting *Nonmem* examples to PoPy
PoPy Reference Guide	A comprehensive reference on PoPy tools and scripts
Appendices	Links to PK/PD terms used throughout this guide.

1.1.3 System Requirements

PoPy is currently available as a 64 bit *Microsoft Windows* binary. See Table 1.2:-

Table 1.2: System Requirements for PoPy

Operating System	*Microsoft Windows* 7.0/8.0/10.0
Disk Space Required	1.5Gb
Bit Size	64
Processor	Intel Core i3 and better recommended
Binary Installer Size	~250Mb

1.1.4 Abbreviations

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*

1.2 Install PoPy

1.2.1 Downloading PoPy

First, download the latest binary from:-

https://product.popypkpd.com/releases/

To access the url above you need to *Obtain a website account* for the PoPy product site, then login using your email address and a password.

We currently provide a **64-bit version** of PoPy for Windows 7/8/10. The download file is approximately 300Mb which will take around 6 minutes to download using a broadband connection.

Note: If you have a previous version of PoPy installed, please *Uninstall PoPy* before installing the new version.

1.2.2 Installing PoPy

Double-click on the PoPy installer, popy-1.1.2-win64-installer.exe

The installer places the software in a subdirectory called 'PoPy'. We recommend PoPy is installed in the root directory of the main drive, typically:-

```
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 will uncompress the PoPy files, just click on the .exe installer and follow the prompts. You will need approx 1.5Gb of disk space, and the install time may take 5 minutes or so depending on the speed of your computer.

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

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

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.

Multi User Installation

The installer sets up PoPy for the current user only. To share a PoPy installation with another user, simply add the PoPy directory to their system path, so they can access the 'popy_env.bat' batch file. Windows start menu shortcuts can also be created manually, if desired.

1.2.3 Checking your new PoPy Installation

PoPy runs from the command line, typically called *command prompt* under Windows 7-9 or *PowerShell* under Windows 10. To access PoPy commands in the current command prompt, type:-

```
$ popy_env
```

This scripts sets up the environment variables so you can access PoPy commands. See *Open a PoPy Command Prompt*.

To verify that PoPy is installed, execute the command:-

```
$ popy_info
```

If PoPy has successfully installed, output similar to that below will appear, providing detailed information on the installation:-

```
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 will vary depending on the *PoPy Activation* status of PoPy.

If you want to run more rigorous tests then see *Validate PoPy*.

1.2.4 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 following 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. Unfortunately this registry edit does **not** work on Windows 7.

1.2.5 Troubleshooting

If you have any problems with this installation process, and cannot find the answer on the *Troubleshooting* page, then email us at info@popypkpd.com.

1.3 Configure Editor

PoPy is script driven, so it is very important to have a suitable editor installed on your system.

We recommended that you use *Notepad++* to edit PoPy text files.

Note: It is especially important that you follow the advice in *Configure Notepad++ Tabs*.

1.3.1 Notepad++

Notepad++ is a text editor that uses tabs and context highlighting, both of which are helpful when editing PoPy script files.

Install Notepad++

Download it from:-

https://notepad-plus-plus.org/download/

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.

Configure Notepad++ Tabs

PoPy is written in *Python*, and *Python* does **not** like tabs. It is essential that you set this from within Notepad++:-

```
Settings->Preferences->Language->Replace by space
```

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 white 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 (but perhaps controversial) language feature. See this blog post, for a fairly balanced discussion of the pros and cons of white spacing:-

https://jayconrod.com/posts/101/how-python-parses-white-space

However PoPy uses *Python*, so it is white space for us.

Configure Notepad++ Path

Note that to use Notepad++ with *popy_edit* you need to make sure your *PoPy Config File* contains this entry:-

```
text_editor_path: "C:/Program Files/Notepad++/notepad++.exe"
```

Which points to where you have installed Notepad++. Note with the path above set correctly you can now *Open a PoPy Command Prompt* and do:-

```
$ popy_edit my_script.pyml
```

The file 'my_script.pyml' should then open within Notepad++. See *popy_edit*.

Configure Notepad++ Colouring

It is highly advisable to load in the PoPy xml colouring file. You do this by opening Notepad++ and doing:-

```
Language->Define your language->Import..
```

Then selecting:-

```
c:\PoPy\conf\notepadplusplus_popy.xml
```

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.

1.4 Fitting a Simple PopPK Model using PoPy

Here we are going to work with the simplest possible single compartment model and bolus dose, see Fig. 1.1:-

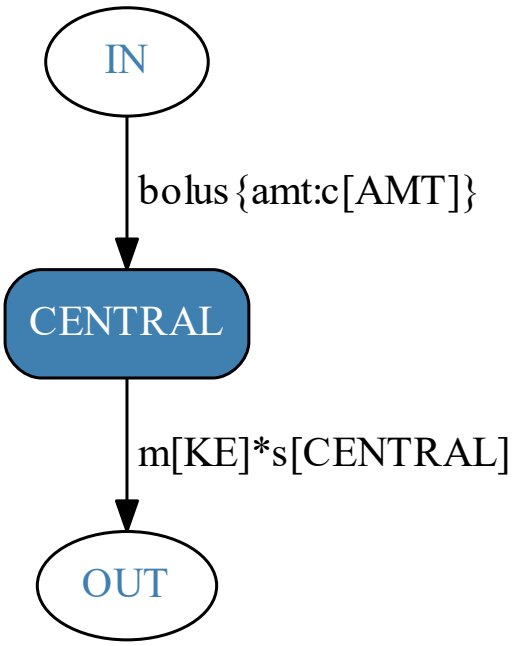


Fig. 1.1: One compartment model with bolus dosing for *Fit Script*. Here `c[AMT]` is the amount of the bolus dose. `m[KE]` is the elimination rate and `s[CENTRAL]` is the current amount in the central compartment. See *Variable Types* for a summary of the prefixes used in PoPy.

In this example, we will walk through fitting the one compartment model shown in Fig. 1.1 to a pre-existing *data file* using PoPy, explaining the commands, input files and output files at each step.

Note: See the *Simple Fit Example* obtained by the PoPy developers for this example, including input script and input data file.

1.4.1 Run the Fit Script

To fit a model in PoPy, you need a model file ending in .pyml and a *data file* in comma separated value format (.csv). See files in your PoPy 'examples' sub directory:-

```
c:\PoPy\examples\fit_example1.pyml
                 fit_example1_data.csv
```

Open a PoPy Command Prompt to setup the PoPy environment in this folder:-

```
c:\PoPy\examples\
```

With the PoPy environment enabled, open the script using:-

```
$ popy_edit fit_example1.pyml
```

then call *popy_run* on the *Fit Script* from the command line:-

```
$ popy_run fit_example1.pyml
```

While the script runs, you will see informative text regarding the progress of the fitting process.

You can observe the fitting process proceed through the text outputs in the command window. When completed, you can view the output using:-

```
$ popy_view fit_example1.pyml.html
```

Note the extra '.html' extension in the above command. The command *popy_view* opens a local .html file in your web browser to summarise the result of the fitting.

You can compare your local html output with the pre-computed documentation output, see *Simple Fit Example*. You should expect some minor numerical differences when comparing results with the documentation. If you are concerned by any differences in results relative to the official PoPy documentation see *Validate PoPy*.

1.4.2 Summary of Fit Results

The results of running the fitting script are PoPy's best estimate for the presumed unknown *fixed effects* variables:-

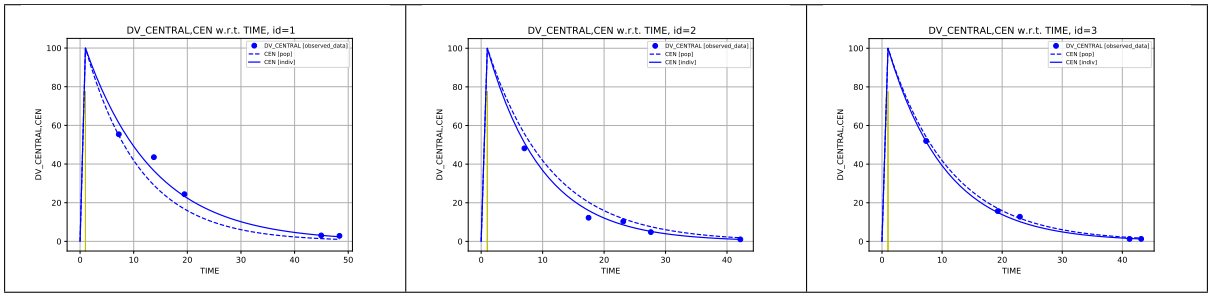
```
f[KE] = 0.0968
f[PNOISE] = 0.2136
f[KE_isv] = 0.0160
```

In PoPy *fixed effects* are denoted using the `f[X]` notation, where 'X' is the name of the *fixed effect*.

The purpose of a *Fit Script* is to optimise the *fixed effects* and *random effects* by maximizing the likelihood of observing the input data given the model structure defined in 'fit_example1.pyml'. The input data in this case, is the `c[DV_CENTRAL]` column in 'fit_example1_data.csv', which contains 20 individuals each with 5 observations at random time points following a bolus dose event.

You can visually compare the PK curves using the fitted `f[X]` outputs with the input data in Table 1.3.

Table 1.3: Model predictions vs original data points for first three individuals



In the graphs above the blue dots represent the observed data points. The solid blue line represents the model individual predictions based on the final `f[X]` parameters and fitted `r[X]` values for each individual. The dashed blue lines represent the model population predictions based on fitted `f[X]` parameters and `r[X]` values set to zero.

Note in this model a bolus dose is received by all individuals at time 1.0. Then the amount of dose follows a first order exponential decay curve as the drug is eliminated from the body over time.

The graphs illustrate how PoPy has optimized the `f[X]` and `r[X]` parameters to maximize the likelihood of the data under this model.

1.4.3 Syntax of Fit Script

This section explains the fitting script notation to represent the components of a mathematical model, such as fixed and random effects and the equation relating the parameters to the observed data. In this section, we will look more closely at how the model file works.

The data file included in this example is simulated from a first order PK model of the same form described in 'fit_example1.pyml'. The population structure is defined in the *EFFECTS* section as follows:-

```
EFFECTS:
    POP: |
        f[KE] ~ unif(0.001, 100) 0.05
        f[PNOISE] ~ unif(0.001, 100) 0.1
        f[KE_isv] ~ unif(0.001, 100) 0.1
    ID: |
        r[KE] ~ norm(0, f[KE_isv])
```

There are three population *fixed effects* `f[X]` parameters to be estimated and one `r[X]` which can take a different value for each individual, sampled from the population distribution. There are 20 individuals in the data set, therefore this model is attempting to estimate 23 parameters in total (i.e 3 f[X] + 20 r[X]). The *fixed effects* are defined as follows:-

```
f[X] ~ unif(min_x, max_x) start_x
```

Here a uniform distribution is used to define a range of allowed values [min_x, max_x], as a kind of prior. Currently in PoPy, `f[X]` are restricted to having a *~unif()* distribution prior.

Note, it is quite common to require PK/PD model parameters be non-negative, in order to make physical sense. The 'start_x' value is the initial value for `f[X]` used in the optimisation, which is usually an initial guess by the modeller. The `r[X]` are here sampled from a zero-mean, univariate normal distribution with a variance `f[KE_isv]` that is optimized for the population:-

```
r[KE] ~ norm(0, f[KE_isv])
```

Each individual has a unique set of `r[X]` values, because the *random effects* are defined at the *ID* level. This has the effect of creating a single `r[KE]` sample for each identity in the data file. For more info on the syntax above see *EFFECTS*.

The mapping from `f[X]` and `r[X]` to the `m[X]` for each individual is defined in the *MODEL_PARAMS* section:-

```
MODEL_PARAMS: |
    m[KE] = f[KE] * exp(r[KE])
    m[PNOISE] = f[PNOISE]
    m[ANOISE] = 0.001
```

This models the `m[KE]` elimination rate for each individual as a log normally distribution with a median value of `f[KE]` and variance parametrised by `f[KE_isv]`. There is a shared proportional noise parameter `f[PNOISE]` for all individuals. For more info on the syntax above see *MODEL_PARAMS*.

The *DERIVATIVES* section defines how the parameters and dosing history relate to the observed data. In this case, we have simple bolus dosing and first-order elimination:-

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

The amount of the bolus dose is `c[AMT]`, which is taken from the data file for each individual. In this example it is always 100 units and occurs at time point 1.0 for every individual. The `m[KE]` elimination rate parameter is first order with respect to `s[CENTRAL]`. Here `s[CENTRAL]` is the amount in the single compartment. For more info on the syntax above see *DERIVATIVES*.

For each row of the data set, `c[X]` values are compared with `p[X]` variables predicted by the model, as defined below:-

```
PREDICTIONS: |
    p[CEN] = s[CENTRAL]
    var = (p[CEN]*m[PNOISE])**2 + m[ANOISE]**2
    c[DV_CENTRAL] ~ norm(p[CEN], var)
```

This section shows that we are comparing model prediction `p[CEN]` with `c[DV_CENTRAL]` using a proportional noise model, where the standard deviation of the proportional noise is `m[PNOISE]`. Here `m[ANOISE]` is fixed to a small positive constant, in order to avoid zero variances when `p[CEN]` is close to zero. For more detailed information on the syntax above see *PREDICTIONS*.

PoPy finds the best combination of the estimated parameters:-

- `f[KE]` - the median elimination rate - which roughly makes sure that the PK curves are of the correct shape to find the data.
- `f[KE_isv]` - the magnitude of the variability in `m[KE]` between individuals
- `f[PNOISE]` - the proportional noise not explained by the model in the `c[DV_CENTRAL]` data

The unexplained noise `f[PNOISE]` and between subject variance `f[KE_isv]` compete with each other to explain the data. For example, do measurements vary from the average model prediction due to measurements lacking precision (or some unknown mechanism) or because subjects just vary a lot in their physiology? This dual explanation for noisy data makes population mixed-effects models difficult to fit. However the population as a whole contains enough data to solve this problem using maximum likelihood *[Sheiner1980]*.

In PoPy the likelihood is optimised iteratively, with the `f[X]` and `r[X]` being updated at each iteration. In this case, the likelihood (or objective function) progressed as shown in Table 1.4

Table 1.4: Objective function at each iteration for simple PopPK example

Iteration	Time	OBJV
0.	0.00	4497.841579275584
0.	0.14	564.0664527231055
1.	0.70	564.0664527231055
1.1	0.84	337.93702802800635
1.2	6.82	326.8404441663747
1.3	10.30	324.2624160122567
1.4	14.32	311.35327301459006
1.5	18.64	307.33975507427624
1.6	21.40	303.48857198389396
1.7	24.18	303.13023189966407
1.8	26.93	303.0605737485126
1.9	29.74	303.0554364772976
1.10	32.92	303.0547922394584
1.11	50.61	303.05478687234404
1.12	66.70	303.05476960838354
1.13	82.50	303.05476960838354

Note that the objective function is defined as -2 * the log likelihood (ignoring fixed proportionality constants). Therefore the lower the value of the objective function the better the estimated parameters fit the observed data. By default PoPy stops the fitting algorithm once the objective function has stopped decreasing.

1.4.4 Visual Predictive Check for Simple PopPK Model

Given the estimated parameter values, *i.e.* the optimised `f[X]` variables, we can check whether the model and its estimate parameters are a good description of the observed data using a *visual predictive check* (VPC).

Running the MSim Script

When you run a PoPy *Fit Script*, it automatically generates several other scripts, including a 'msim' simulation script. For the simple model which we have already fitted, this script can be found in:-

```
fit_example1.pyml_output/
    msim/
        fit_example1_msim.pyml
```

To view or edit the *MSim script*, which runs simulation, navigate to:-

```
fit_example1.pyml_output/
    msim/
```

Open a PoPy Command Prompt in the 'msim' sub folder then do:-

```
$ popy_edit fit_example1_msim.pyml
```

To view the *MSim Script*.

Then you can run the script with the following command:-

```
$ popy_run fit_example1_msim.pyml
```

Running the 'fit_example1_msim.pyml' script creates the following .svg file in the output directory:-

```
fit_example1_msim.pyml_output/
    fit_example1_vpc.pyml_output/
        comb_quant_sim_vpc/
            vpc.svg
```

This graphic should look something like Fig. 1.2:-

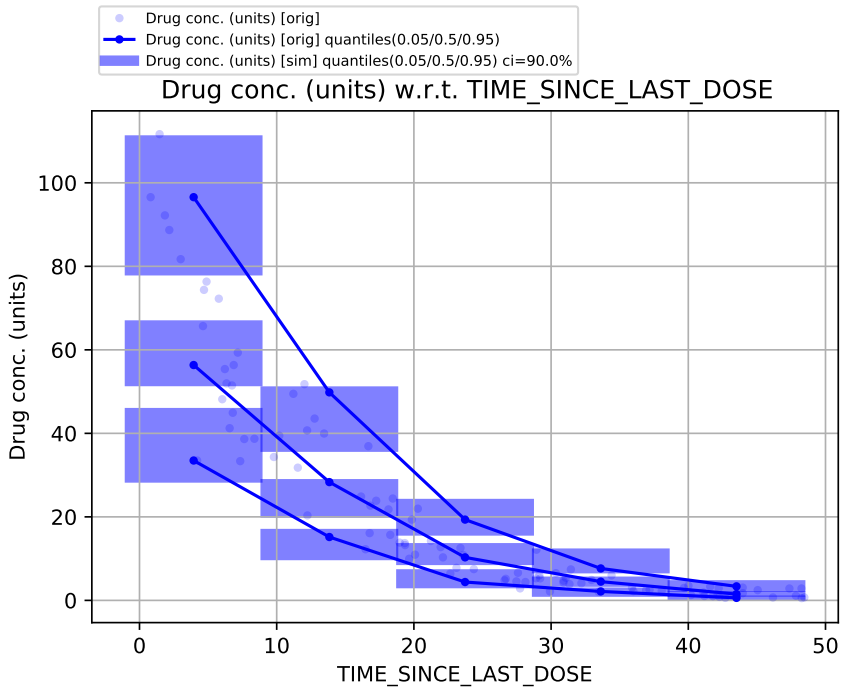


Fig. 1.2: Visual Predictive Check for Simple PopPK model.

In the vpc graph the y axis is the amount in the single compartment and the x axis is the time since the last dose (*TSLD*). It's common to use TSLD in a plot that combines all individuals, as different individuals may have been administered doses at different times in a real life analysis, so absolute times are not comparable.

The TSLD values are grouped into 5 bins along the x axis. Note you need a minimum number of data points in each bin and there are only 100 data points in this simple example, hence the small number of bins.

In Fig. 1.2, the pale blue dots represent the original data points. In each bin the 5%,50% and 95% quantiles are plotted for the original data set (see solid blue lines). Also in each bin, the same quantiles are computed for each of the 100 synthetic data samples. The 90% confidence interval for each of these quantiles, calculated across the 100 simulated data sets, is depicted by the shaded blue region.

The key result from the vpc graph is that the solid blue line (*i.e.* quantiles from the original data set) mostly lie within the shaded blue region (quantile ranges from the synthetic data sets). Since this is the case here, the model performs adequately on the *VPC*. Note that the solid blue line should be within the shaded region approx 90% of the time, because the synthetic quantile ranges are constructed as a 90% confidence interval. You can change the number of simulated data sets in the *MSim Script*. The quantiles of interest and the confidence intervals for those quantiles are specified in the *Vpc Script*.

The blue dots (original data) are mainly shown to give some visual corroboration of the quantiles (solid blue

line). In this graph because there are only 5 time axis bins and therefore each time bin is quite wide, the data points on the left side of each time bin tend to have higher concentrations. This within-bin sample distortion is quite common. Only more bins, which in turn require more data, can address this issue.

Syntax of MSim Script

The *MSim Script* processes these three elements:-

- A data set
- The model - as automatically defined in the *MSim Script* file
- The estimated model parameters

For each individual in the original data set, new synthetic data sets are created by sampling new *random effects* `r[X]` variables for each individual and new measurement noise for all data rows. *i.e.* The synthetic populations vary due to sampling the `r[X]` for each individual here:-

```
EFFECTS:
    ID: |
        r[KE] ~ norm(0, f[KE_isv])
```

And adding measurement noise here:-

```
PREDICTIONS: |
    p[CEN_sim] = s[CENTRAL]
    var = (p[CEN_sim]*m[PNOISE])**2 + m[ANOISE]**2
    c[DV_CENTRAL_sim] ~ norm(p[CEN_sim], var)
```

Note the simulated data c[DV_CENTRAL_sim] has a slightly different name from the original data set field `c[DV_CENTRAL]`, in order to avoid name clashes when constructing graphs.

The **~** notation in the *PREDICTIONS* section of a PoPy script has two slightly different interpretations in fitting versus simulation scripts, in terms of how the operator compares the left hand side (lhs) and right hand side (rhs) of the expression:-

1. In simulation scripts **~** means **sample** the lhs from the distribution on the rhs
2. In fitting scripts **~** means evaluate the **likelihood** of the rhs given the lhs

In a *MSim Script* the former sampling definition is used. In a *Fit Script* the latter likelihood definition is employed.

This procedure creates a set of N new data sets, which can be compared with the original data set. In this case N=100 is defined in the *OUTPUT_OPTIONS* section:-

```
OUTPUT_OPTIONS:
    n_pop_samples: 100
```

You can increase the number of samples, in order to estimate the percentiles, and their confidence intervals, more accurately. If the PK/PD model contains more parameters or the *data file* is more structured, you probably need 500-1000 population samples.

Conceptually, if the model is sensible and the fitted `f[X]` parameters are well estimated then the artificial data sets generated by sampling the random variables should generate PK/PD curves that resemble the observed data PK/PD curves.

If your VPC curves do **not** look like the original data it may be possible to improve upon your model. The pattern of differences between your VPC predictions and the original data set, may give you some clues in how to improve your model.

Note that the VPC says nothing about your models ability to **generalise**, it only compares the model with the original data. For example, if you want to predict the response to much higher doses, than those present in the your original data set, the VPC provides no guarantee that predictions will be accurate.

1.5 PoPy Data Format

The PoPy data file records *observation* and dosing regimens for each individual in a study.

The columns or fields in the data file are split into four main types in Table 1.5:-

Table 1.5: PoPy data fields

Field	Comment
Required Fields	TYPE/ID/TIME
Dosing Fields	dosing regime data
Observation Fields	observed measurements
Extra Fields	extra co-variate information

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

1.5.1 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. You can use other field names if you choose to redefine them in the *script file DATA_FIELDS* section.

TYPE

The 'TYPE' field specifies the event that is happening in each row of the data file. The different types of row are as follows:-

- 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' combined with a 'dose' event.

Typically a drug trial data set mainly consists mainly of 'obs' and 'dose' rows with a few 'reset' rows, per subject.

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.

TIME

The 'TIME' field defines the time stamp for each row.

The time field is required to be monotonically increasing, unless a *TYPE* = 'reset' or 'reset+dose' row is reached. 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 1.6.

Table 1.6: 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 1.6 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.

1.5.2 Dosing Fields

Dosing events are created in the data file using 'dose' values in the *TYPE* field.

There are two methods of associating data dose rows with the *DERIVATIVES* section in the PoPy *script file*, as follows:-

- *Single Dose Type*
- *Multiple Dose Types*

The first involves using just the 'dose' value, the second involves defining dose type names.

The amount of each dose is usually specified in an AMT field.

Note in PoPy AMT is **not** a keyword. It is just the conventional name for the dose amount field used in this documentation.

Single Dose Type

The simplest way to create doses at a set of fixed times is shown in Table 1.7.

Table 1.7: 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

Note that this creates 3 doses at times [1.0, 2.0, 3.0]. The script file loading this data set should have a *DERIVATIVES* section something like:-

```
DERIVATIVES: |
    d[DEPOT] = @bolus{amt: c[AMT]} - m[KE] * s[DEPOT]
```

Note that the *@bolus* dose has no name associated with it.

Multiple Dose Types

If you have multiple types of dose in your analysis, *e.g.* two different drugs being prescribed, then you need to give each dose type a name, as shown in Table 1.8.

Table 1.8: PoPy multi dose type example

TYPE	*TIME*	AMT_DRUG1	AMT_DRUG2	comment
dose:drug1	1.0	100	0	100 units of drug1
dose:drug2	2.0	0	200	200 units of drug2
dose:drug1	3.0	50	0	50 units of drug1

The data file above creates 2 doses of drug1 and 1 dose of drug2. The script file loading this data set should have a *DERIVATIVES* section something like:-

```
DERIVATIVES: |
    dose[drug1] = @bolus{amt: c[AMT_DRUG1]}
    dose[drug2] = @bolus{amt: c[AMT_DRUG2]}
    d[DEPOT1] = dose[drug1] - m[KE1] * s[DEPOT1]
    d[DEPOT2] = dose[drug2] - m[KE2] * s[DEPOT2]
```

The important aspect here is that the *@bolus* doses are defined with names 'drug1' and 'drug2'. These names also appear in the *TYPE* field in the data set as 'dose:drug1' and 'dose:drug2'.

An alternative naming syntax is as follows:-

```
DERIVATIVES: |
    d[DEPOT1] = @bolus{amt: c[AMT_DRUG1], name: 'drug1'} - m[KE1] * s[DEPOT1]
    d[DEPOT2] = @bolus{amt: c[AMT_DRUG2], name: 'drug2'} - m[KE2] * s[DEPOT2]
```

Note that when creating a PoPy data set, you only need to specify a name for each type of dose. You can leave the modelling decision of where each dose appears in the compartment model to a later time.

1.5.3 Observation Fields

Another important set of fields in the data file are the columns that define observed measurements. Observation rows are defined by setting *TYPE* = 'obs'.

This section shows examples of the following:-

- *Single Observed Field*
- *Observed Field with missing data*
- *Multiple Observed Fields*

Note in each case the *PREDICTIONS* section of the PoPy *script file* is associated with observation fields in the data file in order to compute the likelihood correctly.

Single Observed Field

An example of a single observed field is shown in Table 1.9.

Table 1.9: PoPy single observed field example

TYPE	DRUG_CONC
obs	10.5
obs	15.5
obs	2.0

In this simple case the *PREDICTIONS* section may look something like:-

```
PREDICTIONS: |
    p[DRUG_CONC] = s[CEN]/m[V]
    c[DRUG_CONC] ~ norm(p[DRUG_CONC], m[ANOISE_var])
```

Note that the `c[DRUG_CONC]` references the 'DRUG_CONC' field of the data set. Here the likelihood is computed by comparing the model prediction `p[DRUG_CONC]` and the data file observation `c[DRUG_CONC]` for **all** rows of the data set, where *TYPE* = 'obs'.

Therefore all values of the data column 'DRUG_CONC' have to be valid observations. If you have missing values then you need to use the data structure in *Observed Field with missing data*.

Observed Field with missing data

An example of a single observed field, with some **missing** data is shown in Table 1.10.

Table 1.10: PoPy single observed field missing data example

TYPE	DRUG_CONC	DRUG_CONC_FLAG	comment
obs	10.5	1	DRUG_CONC valid
obs	0.0	0	DRUG_CONC invalid
obs	-5.0	0	DRUG_CONC invalid
obs	2.0	1	DRUG_CONC valid

In this case the *PREDICTIONS* section may still look something like:-

```
PREDICTIONS: |
    p[DRUG_CONC] = s[CEN]/m[V]
    c[DRUG_CONC] ~ norm(p[DRUG_CONC], m[ANOISE_var])
```

However not all the *TYPE* = 'obs' rows contribute to the likelihood in this case. Only the rows that have *TYPE* = 'obs' **and** DRUG_CONC_FLAG = 1.

It is similar to having the following 'if' statement in your *PREDICTIONS* section:-

```
PREDICTIONS: |
    p[DRUG_CONC] = s[CEN]/m[V]
    if c[DRUG_CONC_FLAG] > 0.5:
        c[DRUG_CONC] ~ norm(p[DRUG_CONC], m[ANOISE_var])
```

You can include the 'if' statement in your *PREDICTIONS* section if you like, but it is not required (or encouraged).

Note also that missing out the 'DRUG_CONC_FLAG' field from your data set, has a similar effect to creating a 'DRUG_CONC_FLAG' field and setting all the values to 1. *i.e.* Flags default to 1 in PoPy.

If you have multiple observation types in your data set then flag fields become more important, see the example data structure in *Multiple Observed Fields*.

Multiple Observed Fields

An example of multiple observed fields, is shown in Table 1.11.

Table 1.11: PoPy multiple observed fields

TYPE	DRUG1	DRUG1_FLAG	DRUG2	DRUG2_FLAG	comment
obs	10.5	1	0.2	1	Both drugs valid
obs	10.5	1	0.0	0	only drug1 valid
obs	-4.1	0	0.0	0	both drugs invalid
obs	-4.1	0	0.5	1	only drug2 valid

In this case the *PREDICTIONS* section may look something like:-

```
PREDICTIONS: |
    p[DRUG1] = s[CEN1]/m[V1]
    c[DRUG1] ~ norm(p[DRUG1], m[ANOISE_var1])
    p[DRUG2] = s[CEN2]/m[V2]
    c[DRUG2] ~ norm(p[DRUG2], m[ANOISE_var2])
```

Here PoPy uses the 'DRUG1_FLAG' and 'DRUG2_FLAG' fields from the data set to only compute the likelihood from valid observations. You don't have to use 'if' statements in the *PREDICTIONS* section to achieve this.

1.5.4 Extra Fields

The other columns of the PoPy data file are available to use in the following *verbatim* sections:-

- *MODEL_PARAMS*
- *STATES*
- *DERIVATIVES*
- *PREDICTIONS*

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 usage 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.

1.6 Generate data and Fit using Simple PopPK Model

PoPy provides a method to simulate, analyse and compare results in a single script, which is ideal for generating tutorials or illustrative examples. Here we will demonstrate a *Tut Script* using the same compartment model as used in *Fitting a Simple PopPK Model using PoPy*, see Figure Fig. 1.3:-

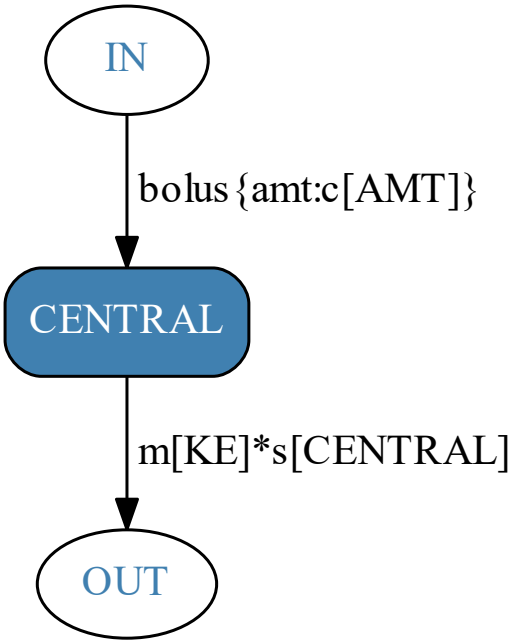


Fig. 1.3: One compartment model with bolus dosing for *Tut Script*. Here `c[AMT]` is the amount of the bolus dose. `m[KE]` is the elimination rate and `s[CENTRAL]` is the current amount in the central compartment.

Note: See the *Simple Tut Example* obtained by the PoPy developers for this example, including input script and output data file.

A *Tut Script* can be used as a theoretical tool to investigate identifiability of PK/PD models, because the true `f[X]` parameters and underlying structure of the data are known. Unfortunately this is never the case in a real life analysis.

This documentation makes extensive use of *tut_scripts* to create examples to illustrate different *Principles of Pharmacokinetics*.

1.6.1 Running the Tutorial Script

This tutorial example requires a single input file:-

```
c:\PoPy\examples\tut_example1.pyml
```

Open a PoPy Command Prompt to set up the PoPy environment in this folder:-

```
c:\PoPy\examples\
```

With the PoPy environment enabled, you can open the script using:-

```
$ popy_edit tut_example1.pyml
```

Again, with the PoPy environment enabled, call *popy_run* on the *Tut Script* from the command line:-

```
$ popy_run tut_example1.pyml
```

When the tut script has completed, you can view the output of the fit using *popy_view*, by typing the following command:-

```
$ popy_view tut_example1.pyml.html
```

Note the extra '.html' extension in the above command. This command opens a local .html file in your web browser to summarise the result of the generating process.

You can compare your local html output with the pre-computed documentation output, see *Simple Tut Example*. You should expect some minor numerical differences when comparing results with the documentation.

1.6.2 Syntax of Tut Script

The major structural difference between a *Gen Script* or *Fit Script* and a *Tut Script* is that the tut_script has separate *GEN_EFFECTS* and *FIT_EFFECTS* sections to describe both the generating and fitting effects. The *GEN_EFFECTS* section for this tutorial example is as follows:-

```
GEN_EFFECTS:
    POP: |
        c[AMT] = 100.0
        f[KE] = 0.1
        f[PNOISE] = 0.05
        f[KE_isv] = 0.03
    ID: |
        c[ID] = sequential(20)
        t[DOSE] = 1.0
        t[OBS] ~ unif(1.0, 50.0; 5)
        r[KE] ~ norm(0, f[KE_isv])
```

The *FIT_EFFECTS* section for this tutorial example is as follows:-

```
FIT_EFFECTS:
    POP: |
        f[KE] ~ unif(0.001, 100) 0.05
        f[PNOISE] ~ unif(0.001, 100) 0.1
        f[KE_isv] ~ unif(0.001, 100) 0.1
    ID: |
        r[KE] ~ norm(0, f[KE_isv])
```

The *GEN_EFFECTS* get copied into the *Gen Script* and renamed *EFFECTS*. Similarly the *FIT_EFFECTS* get copied into the *Fit Script* and also renamed *EFFECTS*. From the examples above you can see that the GEN_EFFECTS->POP section has:-

```
f[X] = true_value
```

Whereas the FIT_EFFECTS->POP section has:-

```
f[X] ~ unif(0.001, 100) starting_value
```

Reflecting the fact that the `f[X]` are known constants for a *Gen Script*, but are unknown values to be estimated in a *Fit Script*, with lower and upper limits of [0.001, 100]. Note in PoPy you can also use 'P' meaning positive, as a shortcut for '~unif(0.0,+inf)'.

The GEN_EFFECTS->POP level has this extra line:-

```
c[AMT] = 100.0
```

This sets the dose amount to be 100.0 units for all individuals.

The GEN_EFFECTS->ID level also contains these extra lines:-

```
c[ID] = sequential(20)
t[DOSE] = 1.0
t[OBS] ~ unif(1.0, 50.0; 5)
```

These lines are passed to the *Gen Script* and generate 20 individuals all with a dose at time 1.0 and 5 observations uniformally sampled in the time interval [1.0, 50.0]. See *EFFECTS with two levels from a gen_script* for more information on the *Gen Script* syntax within a *EFFECTS* section.

1.6.3 Summary of Tut Results

See *Simple Tut Example* for example *HTML* outputs generated by the PoPy developers.

On your local machine, the *Tut Script* generates an output folder containing four new scripts:-

```
simple_tut_example.pyml_output/
    simple_tut_example_gen.pyml
    simple_tut_example_fit.pyml
    simple_tut_example_comp.pyml
    simple_tut_example_tutsum.pyml
```

See *Files Generated by Tut Script* for more info. The purpose of each of theses scripts is as follows:-

Table 1.12: Scripts output by a tutorial script

Script	Purpose	Documentation
*_gen.pyml	Generate synthetic data set from model	*Gen Script*
*_fit.pyml	Fit model to synthetic data set	*Fit Script*
*_comp.pyml	Compare gen model and fit model to synthetic data set	*Comp Script*
*_tutsum.pyml	Summary of generating and fitting and comparison results	*TutSum Script*

These four scripts are run in order.

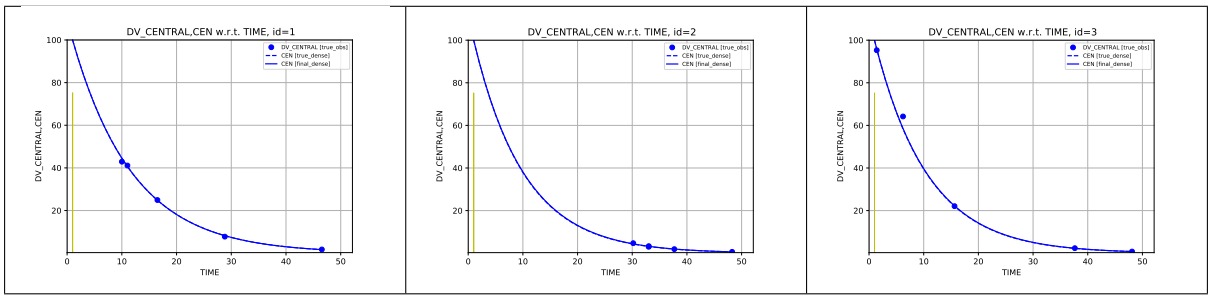
The *Gen Script* uses a *EFFECTS* structure similar to a *Fit Script*, but with some extra commands to generate new data rows, see *Syntax of Tut Script* above. Alternatively see *Generate a Two Compartment PopPK Data Set*, for a longer explanation of how a *Gen Script* works.

The *Fit Script* here is very similar to the PK/PD model described in *Fitting a Simple PopPK Model using PoPy*. Therefore here we will focus on the *Comp Script* outputs. To generate the comp output, you need this entry in your *Tut Script OUTPUT_SCRIPTS* section:-

```
OUTPUT_SCRIPTS:
    COMP: {output_mode: run}
```

Otherwise the *TutSum Script* will have **no** comp output to summarise. The comp outputs are PK curves from the fitted and generated `f[X]` parameters and the associated *objective function* values. The simplest *Comp Script* output is a visual comparison of the true and fitted `f[X]` PK curves and the synthetic generated data, see Table 1.13.

Table 1.13: Fitted model PK curves vs true model PK curves for first three individuals



The solid blue lines in Table 1.13 show the predicted PK curves for the fitted model `f[X]` values. The dotted blue lines show the PK curves for the true `f[X]` values that were used to generated the data set (in the *Gen Script*). The blue dots are the target `c[DV_CENTRAL]` values from the data file.

The target `c[DV_CENTRAL]` values have measurement noise added, so blue dot data points do **not** lie exactly on the true `f[X]` curves. The graphs show that the PK curves for the fitted `f[X]` are almost identical to the true `f[X]` curves, this is to be expected as the model only contains a single model parameter `m[KE]` and we have 5 observations per individual.

If the *Comp Script* has been run, the *TutSum Script* outputs convenient tables to compare the initial, fitted and true `f[X]` values, see Table 1.14, Table 1.15 and Table 1.16.

Table 1.14: Comparison of initial, fitted and true `f[KE]` values

Name	Initial	Fitted	True	Prop. Error	Abs. Error
f[KE]	0.05	0.0999	0.1	0.13%	1.30e-04

Table 1.15: Comparison of initial, fitted and true `f[KE_isv]` values

Name	Initial	Fitted	True	Prop. Error	Abs. Error
f[KE_isv]	0.1	0.0183	0.03	39.00%	1.17e-02

Table 1.16: Comparison of initial, fitted and true `f[PNOISE]` values

Name	Initial	Fitted	True	Prop. Error	Abs. Error
f[PNOISE]	0.1	0.0503	0.05	0.67%	3.36e-04

Table 1.14 shows that the `f[KE]` parameter is recovered reasonably well, in the sense that the fitted value 0.106 is close to the true generating value 0.1 starting from an initial value of 0.05. Similarly the fitted `f[KE_isv]` and `f[PNOISE]` parameters are close to the true generating values.

The objective function *Comp Script* computes the *objective function* given the synthetic data and the true generating `f[X]` parameters (the `r[X]` are re-optimised). In this case the true `f[X]` *ObjV* is:-

```
-44.20
```

The *Comp Script* also computes he *ObjV* for the fitted `f[X]` and optimised `r[X]`, which is as follows:-

```
-48.43
```

The lower objective value for the fitted `f[X]` is quite common, because the fitted `f[X]` can take advantage of noise in the generated synthetic data set. If the size of the synthetic data set is increased, then it is likely that the `f[KE]`, `f[KE_isv]` and `f[PNOISE]` parameters will move closer to the true generating values and the *ObjVs* will also converge.

In this simple example the parameters are very easy to identify. For a more challenging example see *Generate data and Fit using a Two Compartment Model*.

1.6.4 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. 1.3.

Running the MTut Script

This multi tutorial example makes use of a single script file:-

```
c:\PoPy\examples\mtut_example1.pyml
```

Open a PoPy Command Prompt to setup the PoPy environment in this folder:-

```
c:\PoPy\examples\
```

With the PoPy environment enabled, you can open the script using:-

```
$ popy_edit mtut_example1.pyml
```

Again, with the PoPy environment enabled, call *popy_run* on the *MTut Script* from the command line:-

```
$ popy_run mtut_example1.pyml
```

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.

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.

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:-

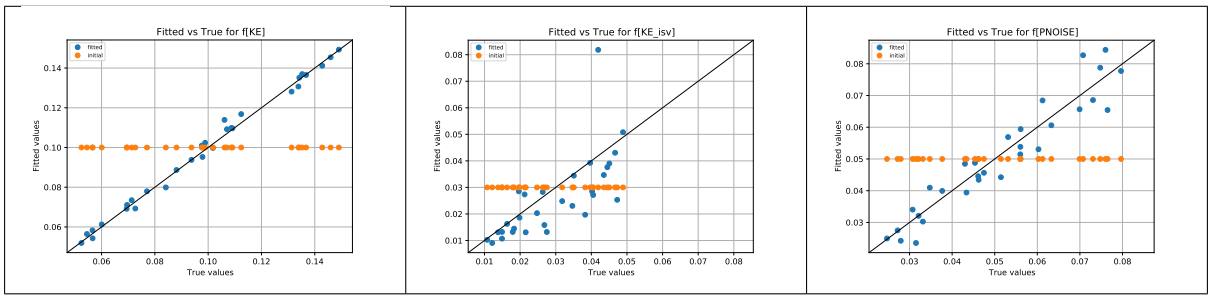
Table 1.17: Scripts output by a multi tutorial script

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 *Syntax of Tut Script* and the *MFit Script* is very similar to the *Fit Script* described in *Fitting a Simple PopPK Model using PoPy*. 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 1.18.

Table 1.18: Fitted model vs true scatter plots for `f[KE]`, `f[KE_isv]` and `f[PNOISE]`



In Table 1.18 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 1.18. 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 1.18. 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 *Fitting a Simple PopPK Model using PoPy*. The blue dots for `f[KE_isv]` and `f[PNOISE]` (centre and right graphs in Table 1.18) 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.

1.7 Typical Workflows

1.7.1 Developing a PK/PD model for a real life Data Set

A typical task for a PK/PD modeller is deducing a model given a data set collected from patients. It is usually best to take pre-existing knowledge about the PK and PD of the drug and use this as a starting point to create an initial model. You can then explore more complex models by adding new parameters and comparing the fitted `f[X]` and objective value obtained with the new model, to prior models. In PoPy you can achieve this by developing a series of *fit scripts*, running each script and examining the outputs.

To help with this process the potential child scripts of an individual *Fit Script* are summarised in Fig. 1.4.

The child scripts are automatically generated by PoPy in order to facilitate other tasks you may wish to perform. Each child script is optionally generated by an entry in the *OUTPUT_SCRIPTS* section of the *Fit Script*.

For example a *Sim Script* will plot a dense PK or PD curve for each individual and you can see where your current model may not be matching the data set and where the model is extrapolating.

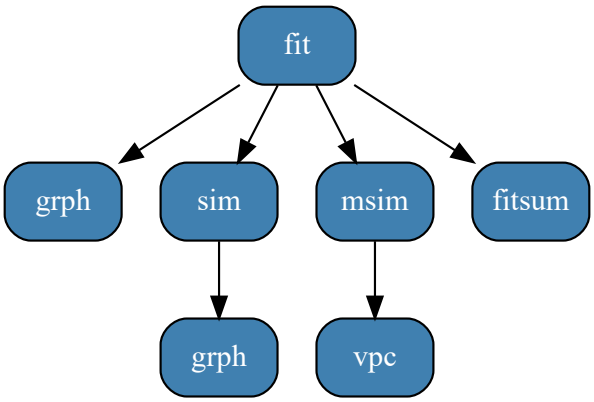


Fig. 1.4: Hierarchy of child scripts for a parent *Fit Script*.

An *MSim Script* allows you to easily generate a *VPC* plot, which allow you to see how your current model compares to the data set for the whole population. As a *VPC* tests the distributional assumptions of the model as well as the fit to each individual, it is a rigorous test for a population model.

1.7.2 Investigate a PK/PD model using synthetic data

If you do **not** have a real life data set or just wish to carry out a more theoretical investigation you can use PoPy's tools to generate artificial data. This approach has the benefit of allowing you to create data sets with known properties and test hypotheses about the identifiability of parameters. The primary tool to do this in PoPy is the *Tut Script*, see *Generate data and Fit using Simple PopPK Model* for a run through of using a tut script. The potential child scripts of a *Tut Script* are shown in Fig. 1.5.

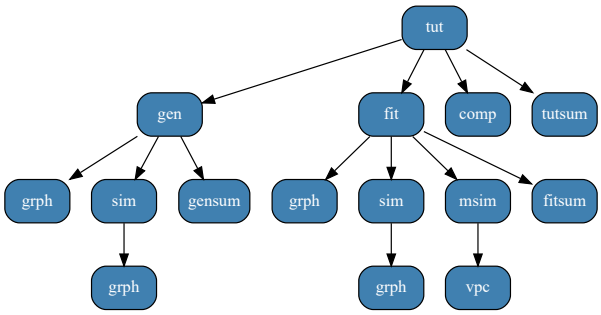


Fig. 1.5: Hierarchy of child scripts for a parent *Tut Script*.

Another tool for investigating models using artificial data in PoPy is the *MTut Script*. See *Generate multiple data sets and Fit using Simple PopPK Model* for a run through of using a MTut script. The potential child scripts of a *MTut Script* are shown in Fig. 1.6.

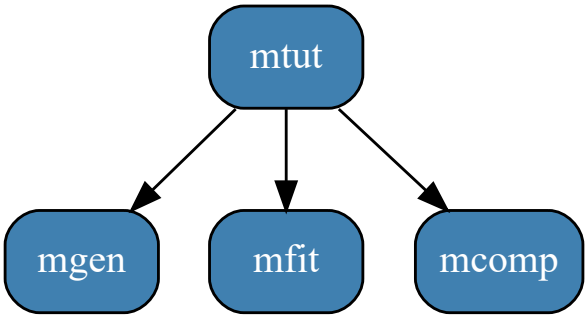


Fig. 1.6: Hierarchy of child scripts for a parent *MTut Script*.

CHAPTER

TWO

PRINCIPLES OF PHARMACOKINETICS

Pharmacokinetics is the study of the time profile of drugs and metabolites in a physiological organism. It is often described as "what the body does to the drug". Generally, our goal is to predict a time course of drug concentration that resembles the observations we have collected from an individual. (Observations from a population are covered in *Population Models in PoPy*.)

To make these predictions we build a mathematical model that, for an individual, has two main components:

1. A deterministic structural model with parameters that define the shape of the time course
2. A stochastic residual error model that specifies how observations deviate from the deterministic predictions

- *[MouldUpton2012]*
- *[MouldUpton2013]*
- *[UptonMould2013]*
- *[RowlandTozer2012]*

This part of the book provides a detailed description of how various pharmacokinetic (PK) models are implemented in PoPy. It assumes no prior knowledge of PK, hence experienced modellers may be tempted to skip this section. It is, however, probably valuable to look at the syntax for absorption models, particular the Weibull model, which is not available in other population modelling software.

2.1 Elimination, Clearance and Volume of Distribution

Pharmacokinetics is the study of "what the body does to the drug." 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 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. 2.1).

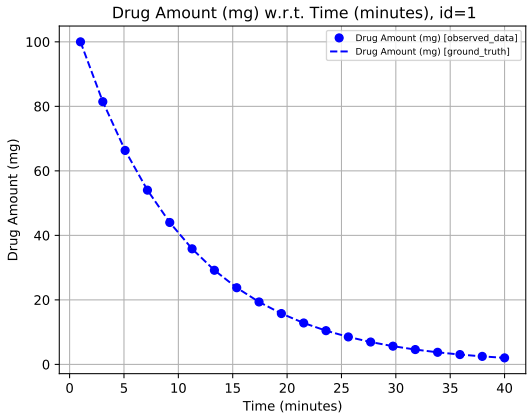


Fig. 2.1: Amount Administered vs Time curve for an intravenously administered 100 mg bolus dose at time *t0=1* with first order elimination. (Observations are noiseless in this example.)

2.1.1 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. 2.2; note the scale of the *y*-axis).

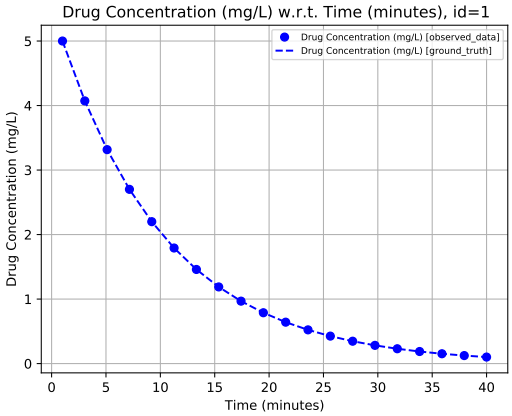


Fig. 2.2: Concentration vs Time curve for an intravenously administered 100 mg bolus dose with first order elimination

2.1.2 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. 2.2) is unchanged.

2.2 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.

2.2.1 One Compartment Model

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. 2.3).

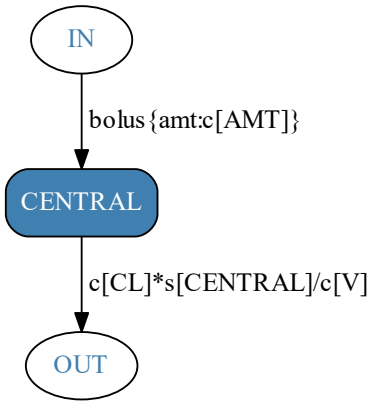


Fig. 2.3: Compartment diagram for a one compartment model with intravenous dosing.

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. 2.4).

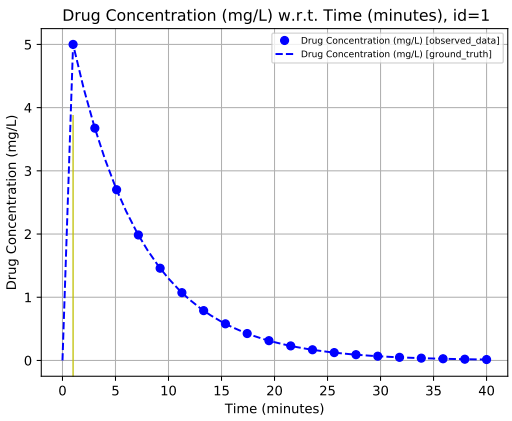


Fig. 2.4: Concentration vs Time curve for a bolus dose, intravenously applied to the **Central** compartment, with first order elimination.

(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).

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:

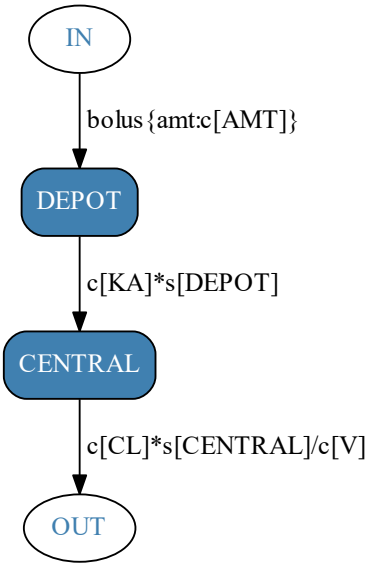


Fig. 2.5: Compartment diagram for a one compartment model with absorption.

- 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:

```
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.

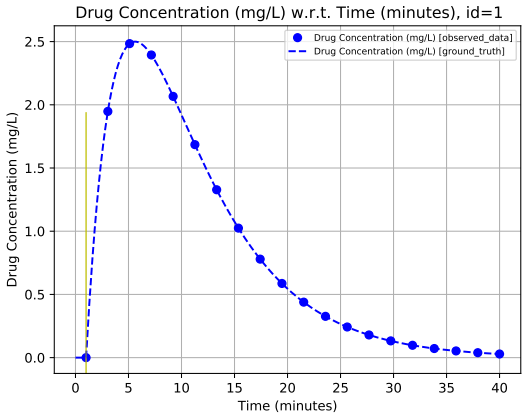


Fig. 2.6: Concentration vs Time curve for a one compartment model with absorption

Regardless of which formulation is used, the resulting Concentration vs Time curve is the same (Fig. 2.6). 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.

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.

2.2.2 Two Compartment Model

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:

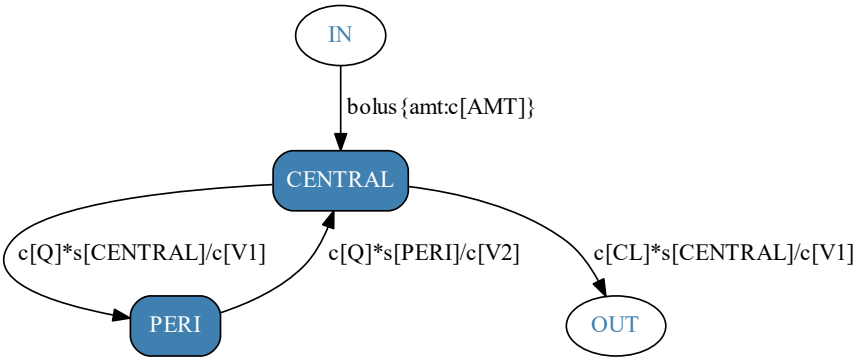


Fig. 2.7: Compartment diagram for a two compartment model with intravenous dosing.

The corresponding *DERIVATIVES* section is:

```
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. 2.8) is the same regardless of whether we use flows or the closed form solution because they are equivalent.

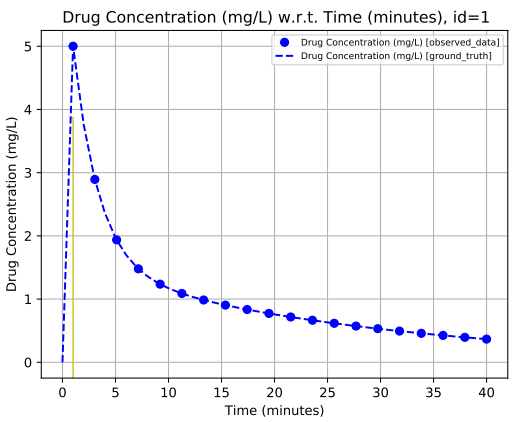


Fig. 2.8: Concentration vs Time curve for a two compartment model with intravenous dosing

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.

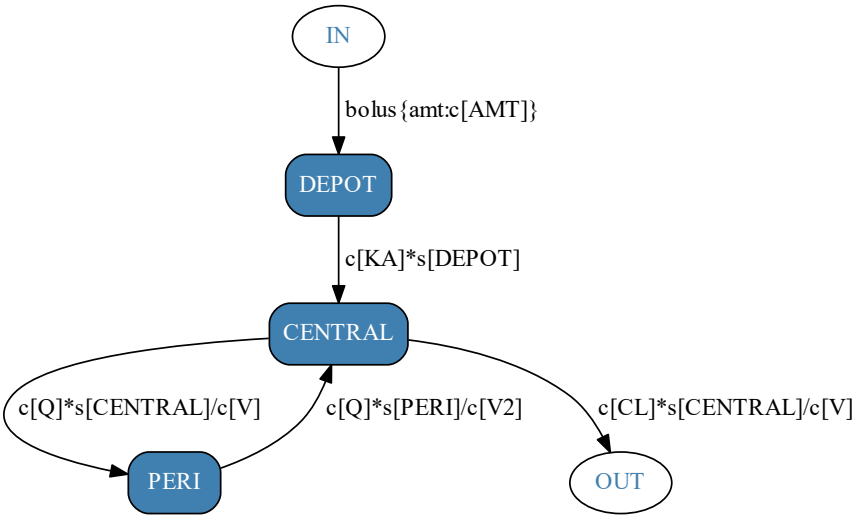


Fig. 2.9: Compartment diagram for a two compartment model with absorption

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. 2.10) we see a combination of the behaviours exhibited in the two earlier examples (iv_two_cmp_cl_tut and dep_one_cmp_cl_tut).

2.2.3 Three Compartment Model

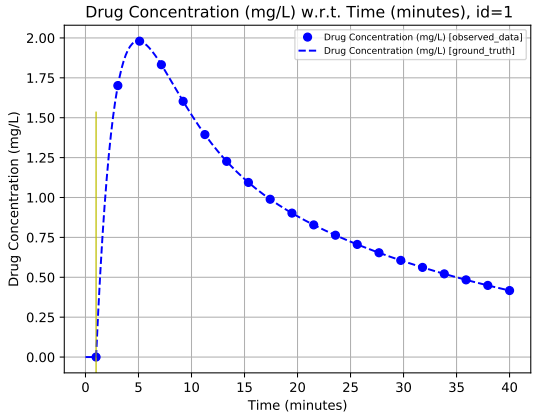


Fig. 2.10: Concentration vs Time for a two compartment model with absorption

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*.

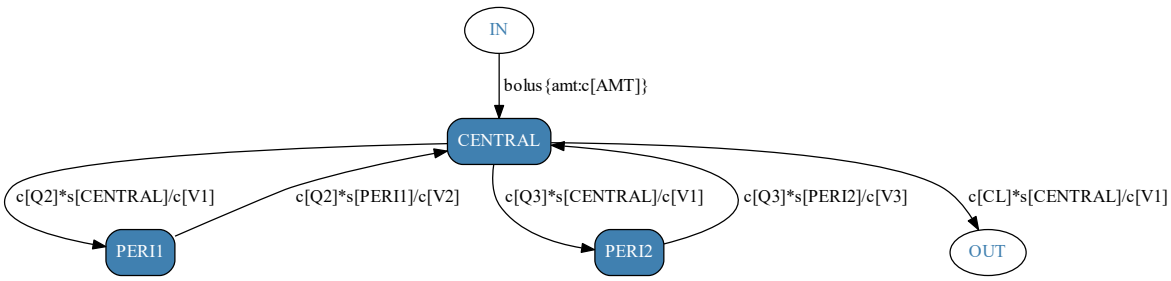


Fig. 2.11: Compartment diagram for a three compartment model with intravenous dosing

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

```
DERIVATIVES: |
    # initialize
    d[CENTRAL] = 0.0
    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. 2.12), 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.

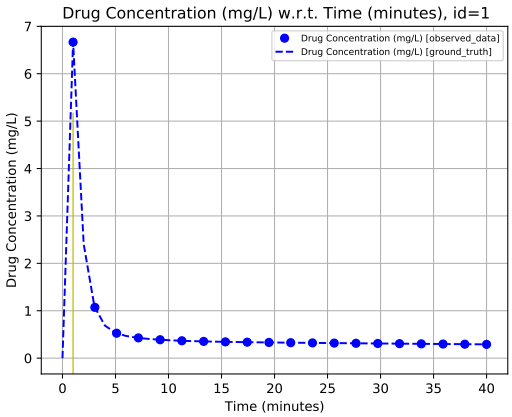


Fig. 2.12: Amount vs. Time for a three compartment model with intravenous dosing

Three Compartment Model with Absorption

The final model we consider in this chapter is a three compartment model with first order absorption (Fig. 2.13), 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.

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]
```

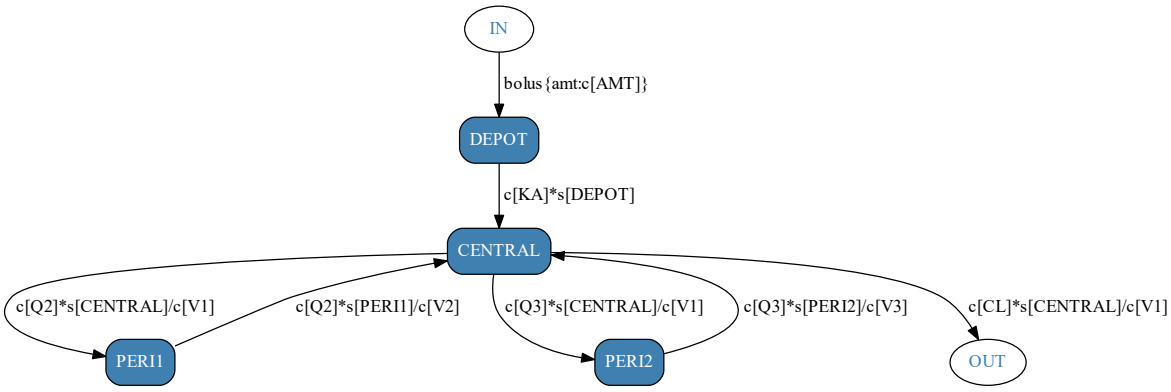


Fig. 2.13: Compartment diagram for a three compartment model with absorption

```
    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. 2.14).

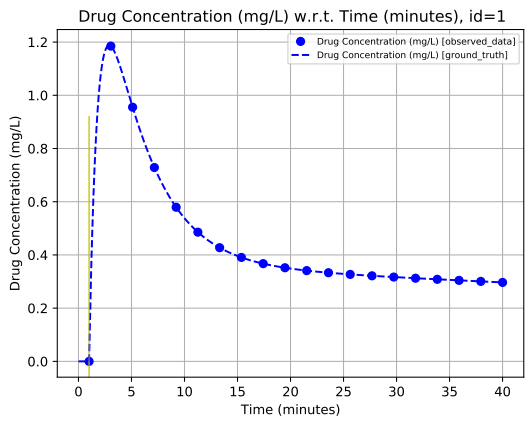


Fig. 2.14: Concentration vs Time for a three compartment model with absorption

2.3 Dose Administration

So far we have considered only a bolus dose - a single amount administered instantaneously to a given compartment, either intravenously (*e.g.* an injection) or via absorption (*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.

2.3.1 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:-

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

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. 2.15):

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]
```

2.3.2 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:

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

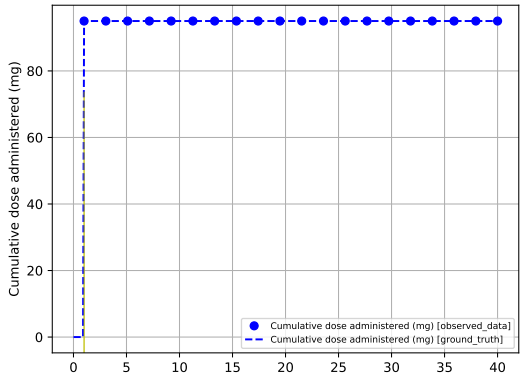


Fig. 2.15: Cumulative amount following a bolus dose with no elimination

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.

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. 2.16).

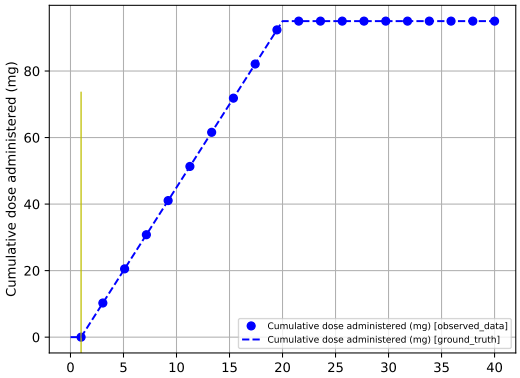


Fig. 2.16: Cumulative amount following a infusion dose of 95 mg over a duration of 19 min with no elimination

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.

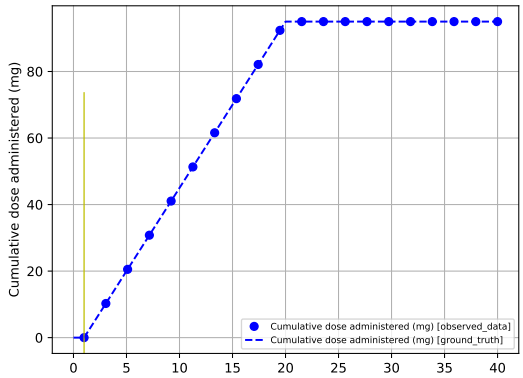


Fig. 2.17: Cumulative amount following a infusion dose of 95 mg at a rate of 5 mg/min with no elimination

2.3.3 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	Symbol
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]
```

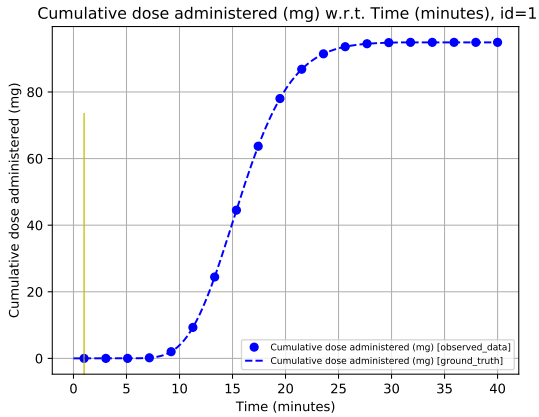


Fig. 2.18: Cumulative amount following a Gamma dose with no elimination

2.3.4 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]}
```

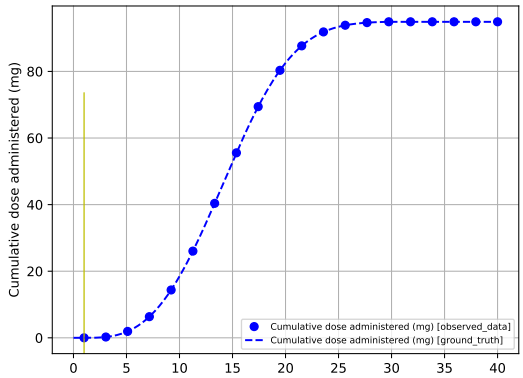
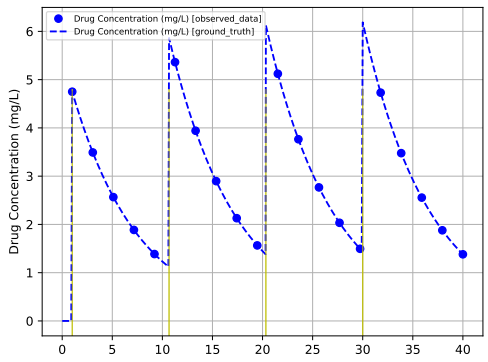
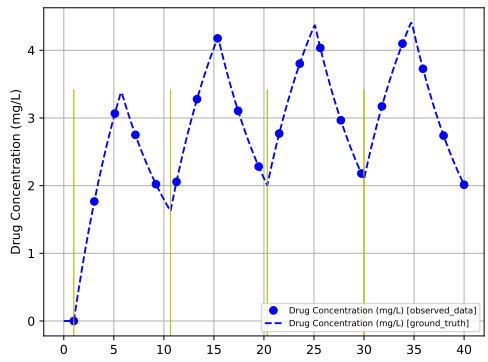
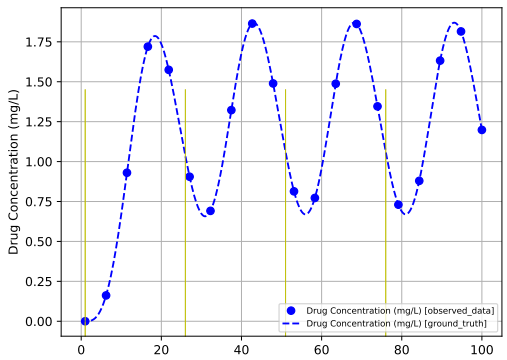
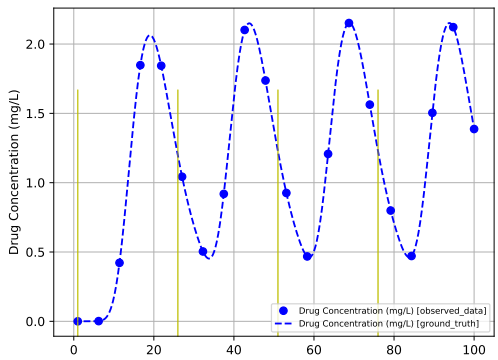


Fig. 2.19: Cumulative amount following a Weibull dose with no elimination

2.3.5 Repeated Dosing

There need not be only one dose administered to the subject for a given experiment – several doses may be given, for example to maintain drug concentration at a steady state over a period of time. PoPy handles this by allowing multiple dose lines in the *data file* and superimposing the resulting curves, see Table 2.1.

Table 2.1: Repeated dosing, administered to a one compartment model with first order elimination

Repeated Bolus Dosing	Repeated Infusion Dosing
	
Repeated Gamma Dosing	**Repeated Weibull Dosing**
	

Note: See *Repeated Bolus Dose with first order elimination.*, *Repeated Infusion Rate Dose with first order elimination.*, *Repeated Gamma Dose with first order elimination.* and *Repeated Weibull Dose with first order elimination.* for the *tutorial scripts* used to generate results in this section.

Note PoPy can superimpose consecutive complex *Dosing Functions* (*e.g.* *Weibull* and *Gamma* functions) that can cause numerical instability in other PK/PD packages.

2.4 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. 2.20).

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

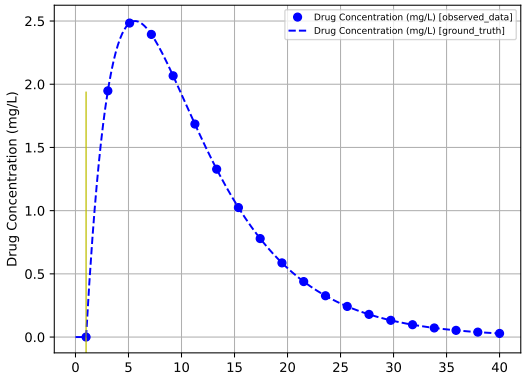


Fig. 2.20: Concentration vs Time for a one compartment model with absorption, without measurement error (noise).

incorporate this into a fitted model.

2.4.1 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).

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

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. 2.21).

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.

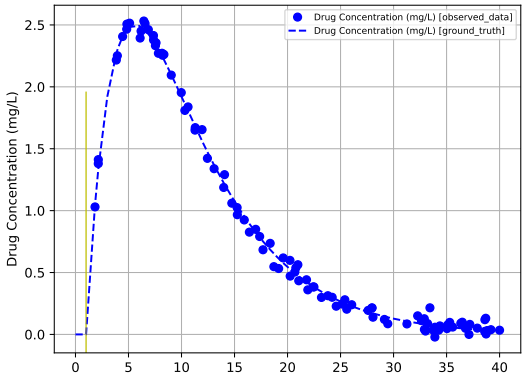


Fig. 2.21: Concentration vs Time for a one compartment model with absorption, plus additive noise.

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. 2.22)

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

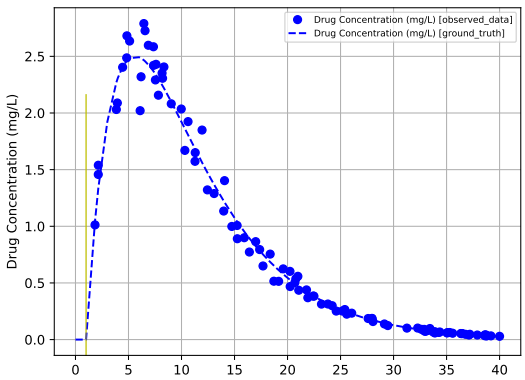


Fig. 2.22: Concentration vs Time for a one compartment model with absorption, plus proportional noise.

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. 2.23).

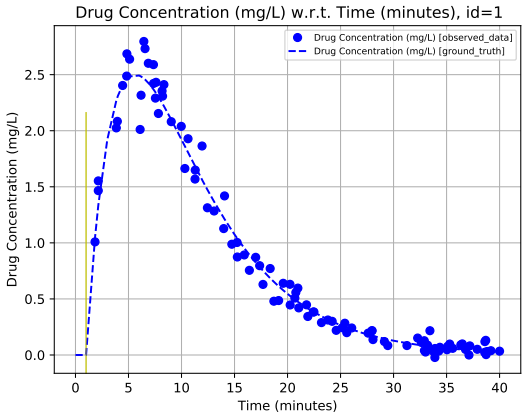


Fig. 2.23: Concentration vs Time for a one compartment model with absorption, plus both proportional and additive noise

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: |
    plabel[DV_CENTRAL] = "Drug Concentration (mg/L)"
    clabel[TIME] = "Time (minutes)"
    p[DV_CENTRAL] = s[CENTRAL]/m[V]
    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)
```

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*.

2.5 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.

2.5.1 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. 2.24).

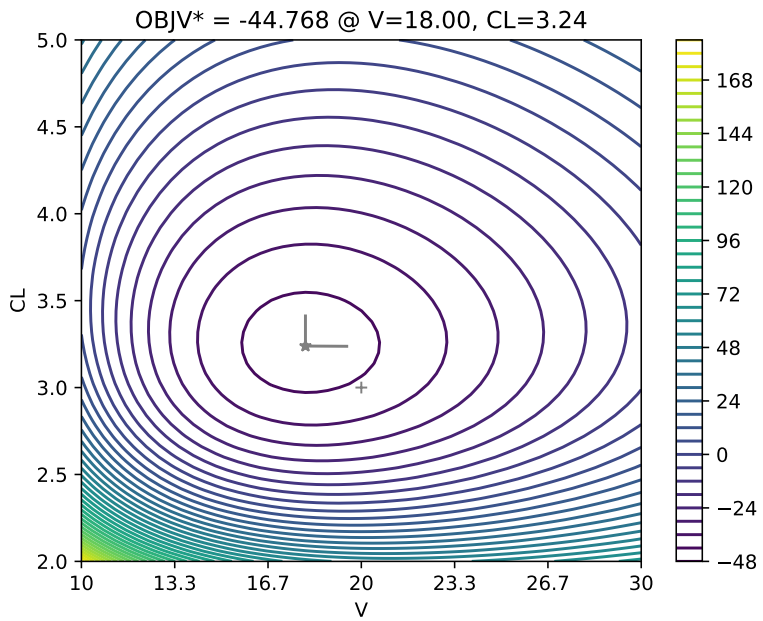


Fig. 2.24: *CL* vs *V* surface plot of the Objective Function Value (*ObjV*) for a bolus dose, administered to a one compartment model with absorption

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.

2.5.2 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.

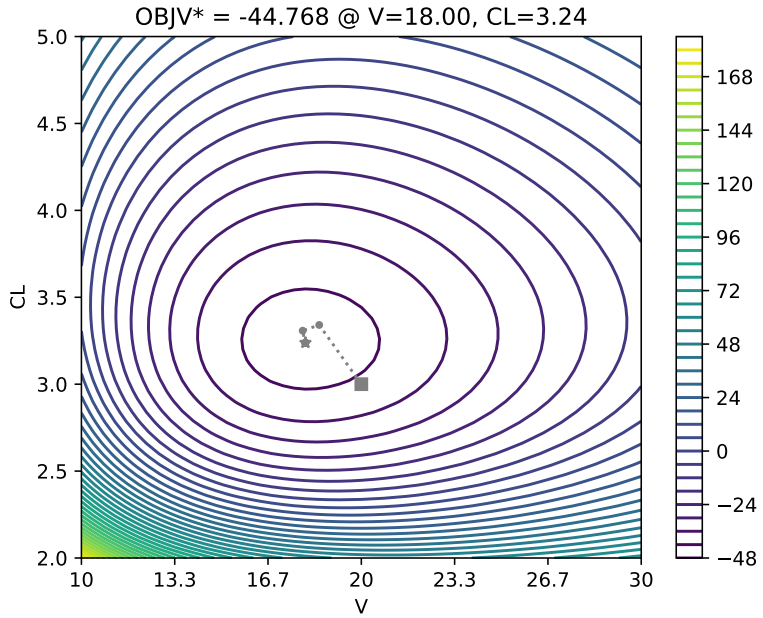


Fig. 2.25: CL vs V surface plot overlaid with the path taken by the optimization algorithm to the minimum. '+' shows starting values, square shows minimum values.

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. 2.25), 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.

2.5.3 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.

2.6 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).

2.6.1 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. 2.26).

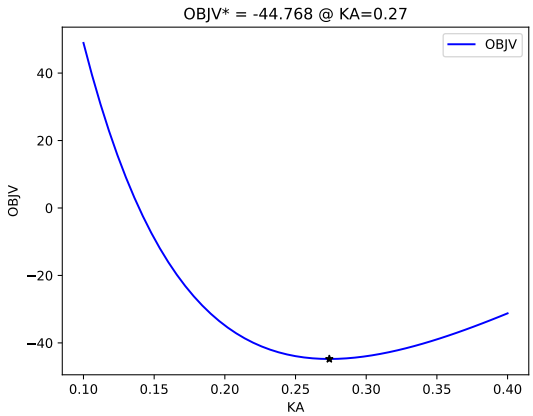


Fig. 2.26: *ObjV* vs *KA* with other parameters fixed at their true values

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. 2.27).

In effect, this approximates the likelihood surface with a Gaussian whose peak coincides with that of the true likelihood (Fig. 2.28).

Note: See *One Compartment Model with Absorption estimating KA and V*, *One Compartment Model with*

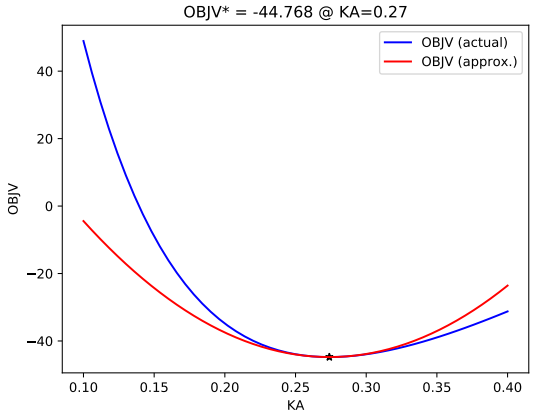


Fig. 2.27: *ObjV* vs *KA* with other parameters fixed at their true values, plus the quadratic approximation at the minimum

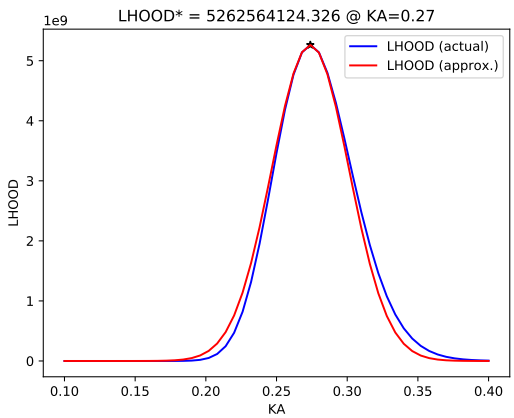
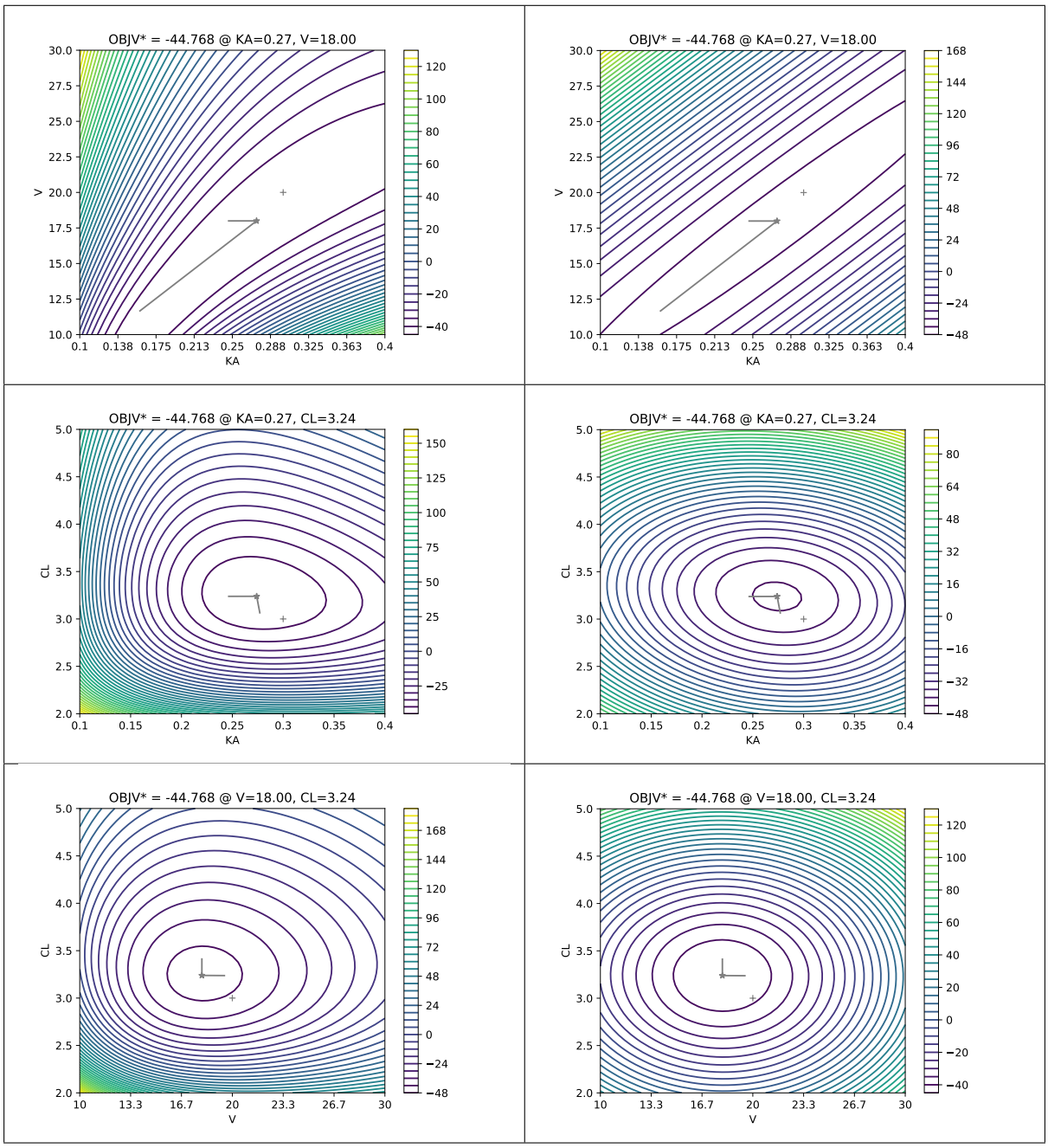


Fig. 2.28: Likelihood vs *KA* with other parameters fixed at their true values, plus the gaussian approximation at the maximum

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 2.2).

Table 2.2: Comparison of ObjV surface (left) with quadratic approximation (right)



2.6.2 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. 2.29).

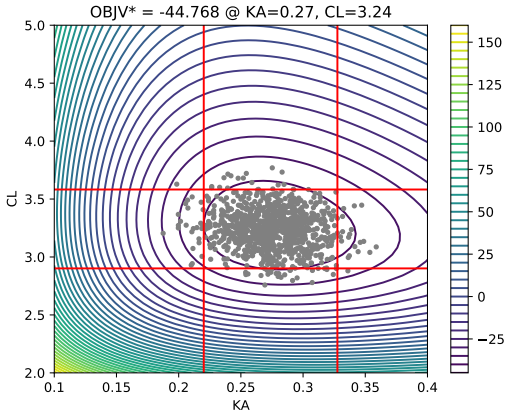


Fig. 2.29: *ObjV* surface with population parameters sampled from the approximating gaussian and 90% confidence intervals in red for *KA* and *CL*

```
CI(KA) = 0.274 + [-0.0538, 0.0537]
CI(CL) = 3.24 + [-0.337, 0.343]
```

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. 2.30).

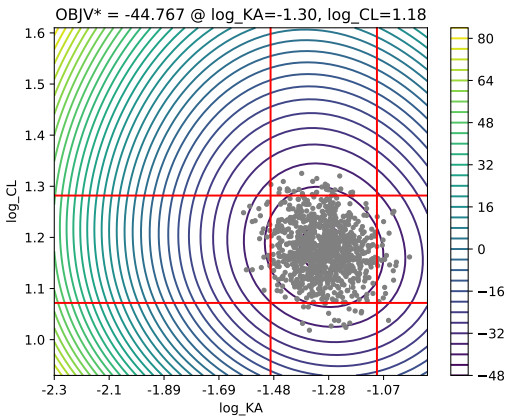


Fig. 2.30: *ObjV* surface with population parameters sampled from the approximating gaussian and 90% confidence intervals in red for log(*KA*) and log(*CL*)

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 + [-0.0489, 0.0601]
CI(CL) = 3.25 + [-0.325, 0.358]
```

2.6.3 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

THREE

POPULATION MODELS IN POPY

Estimating model parameters for a single subject requires many observations to differentiate between one model and another. However, because the observation process is usually intrusive (*e.g.* taking a blood sample to measure drug concentration) it is often practical to take only a few samples from one individual.

To overcome this limitation we can take a few samples from many individuals and pool the data, known as *Population PK*. Early attempts at doing this 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 predict a time course that is personalized to every individual so that the residual error model described earlier remains sensible. This personalization is done by introducing new parameters to capture variability:

1. A stochastic statistical model that uses *random effects* to capture *unpredictable*, random variability between subjects from the same population, and possibly between occasions for the same subject
2. A deterministic covariate model that uses additional *fixed effects* to capture *predictable* variability as a result of relationships between subject characteristics and structural model parameters (*e.g.* between weight and the volume of distribution)

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.

3.1 Inter-Subject Variation (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])
```

3.1.1 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.

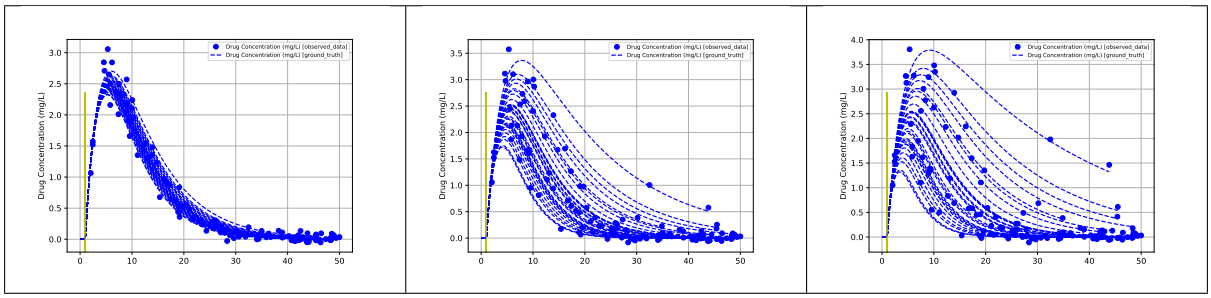
During data synthesis, PoPy adds random variation between individuals by sampling realizations of `r[CL_isv]` from the probability distribution. These realizations of `r[CL_isv]` are then used in the *MODEL_PARAMS* section to add inter-subject variability to the model parameter, `m[CL]`.

```
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 3.1).

Table 3.1: Population graphs with increasing ISV: (left) CL_isv=0.01; (centre) CL_isv=0.2; (right) CL_isv=0.5



3.1.2 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] = 0.2081
f[CL] = 3.0966
f[V] = 14.7707
f[PNOISE_STD] = 0.3766
f[ANOISE_STD] = 0.1619
f[CL_isv] = 0.0000
```

As a reference point, the final objective function value is

```
-200.93968711173534
```

3.1.3 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] = 0.3020
f[CL] = 3.2966
f[V] = 19.6506
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:

```
-394.58997461170543
```

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 Variation (IOV)*.

3.2 Inter-Occasion Variation (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]*.

3.2.1 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
        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*.)

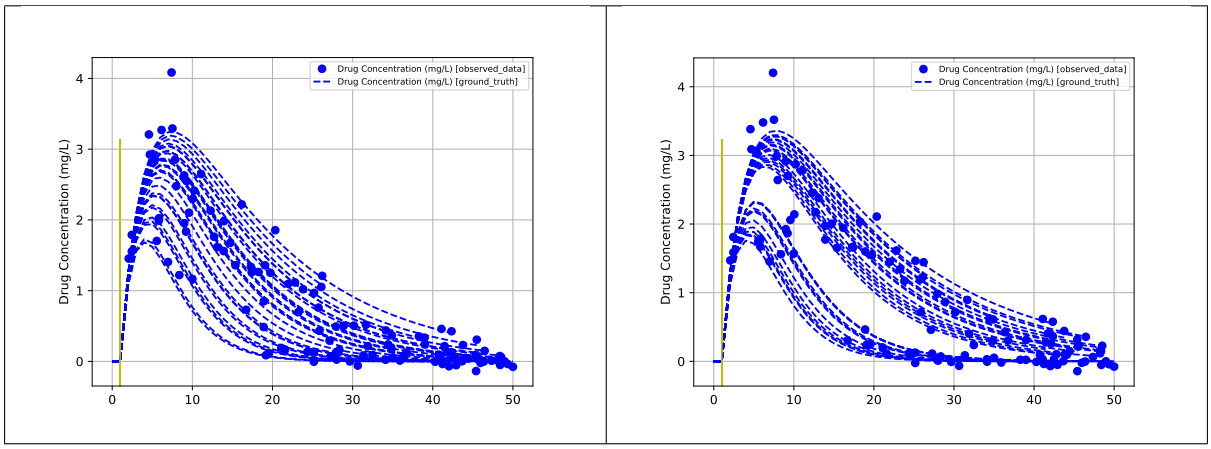
The IOV random effect is then added to its ISV sibling in the *MODEL_PARAMS* section of the script:

```
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 3.2). 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 3.2: Population graphs with increasing ISV: (left) CL_isv=0.2, CL_iov=0.1; (right) CL_isv=0.5, CL_iov=0.01



3.2.2 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] = 2.5350
f[V] = 20.9756
f[PNOISE_STD] = 0.2297
f[ANOISE_STD] = 0.0976
f[CL_isv] = 0.1395
f[CL_iov] = 0.0000
```

and we use the resulting objective function value,

```
-253.73371865528824
```

as a reference.

3.2.3 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] = 2.5856
f[V] = 20.2051
f[PNOISE_STD] = 0.1000
f[ANOISE_STD] = 0.0479
```

```
f[CL_isv] = 0.1199
f[CL_iov] = 0.0783
```

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

```
-363.8117557237518
```

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

3.3 Modelling Correlation in Random Effects

As shown in *Inter-Subject Variation (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.

3.3.1 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] )
        # 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. 3.1) 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).

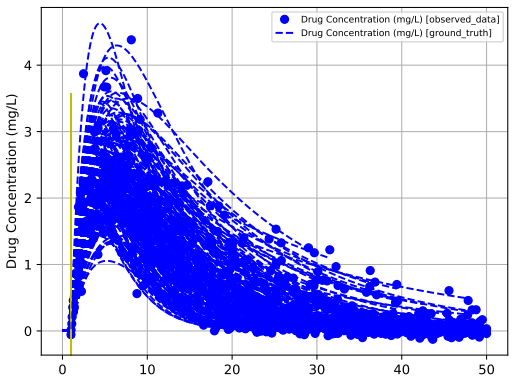


Fig. 3.1: Simulated prediction curves for a population with independent inter-subject variability on parameters CL and V

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}
```

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

```
-2215.9658412906333
```

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,

```
-2217.243790984624
```

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.

3.3.2 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] )
```

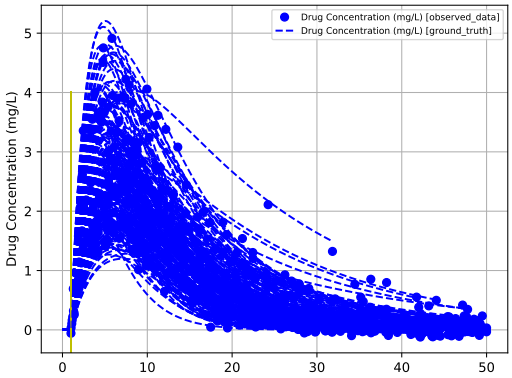


Fig. 3.2: Simulated prediction curves for a population with correlated inter-subject variability on parameters CL and V.

The generated predictions (Fig. 3.2) 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.)

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,

```
-2090.7014893307264
```

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.

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),

```
-2108.2792074137697
```

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.

3.4 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.

3.4.1 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).

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. 3.3).

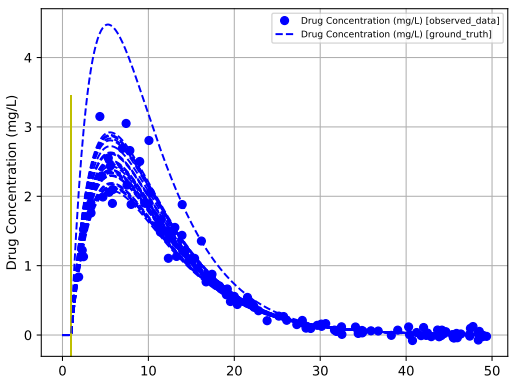


Fig. 3.3: Simulated prediction curves for a population of 40 individuals with weight-dependent clearance, CL, and volume of distribution, V

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

closely match those used to generate the observations.

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

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

3.4.2 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.

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.

CHAPTER

FOUR

POPY EXAMPLE MODELS

The easiest way to learn PoPy is by using working examples, that you can then adapt to your own requirements. See *HTML Summary Links* for links to **all** examples (both data and scripts) used in this documentation. For detailed walk through examples, see below:-

4.1 Creating Example Scripts

PoPy contains built in example scripts that can be generated on the command line. The general method is:-

```
$ popy_create [script_type] [script_name]
```

These built in scripts are designed to get you started with using a particular type of *script file*. Once generated you can edit the default text for your own PK/PD modelling requirements.

See *popy_create* documentation, or the *Creating an example Fit Script* and *Creating an example Tut Script* examples below.

4.1.1 Creating an example Fit Script

First, *Open a PoPy Command Prompt*. Then do:-

```
$ popy_create fit my_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 'my_fit_script.pyml' file is the name of the *Fit Script* written to disk.

You can generate a more verbose example script with comments using:-

```
$ popy_create -acsl fit my_fit_script.pyml
```

See *popy_create* for the meaning of the '-acsl' command line switch options. Or type 'popy_create -h'.

You will notice that *popy_create* also creates a file called *my_fit_script.pyml.create.main.log*. This is a record of the messages output by the *popy_create* tool. In this case the log file is very short. These log files, created by all of the *Command Line Tools*, act as part of the audit trail for your experiments. You can open the 'my_fit_script.pyml' in your system *editor* as follows:-

```
$ popy_edit my_fit_script.pyml
```

You can try running the fit_script:-

```
$ popy_run my_fit_script.pyml
```

However, you will likely see a message, similar to the following:-

```
CAST_ERROR= ERROR in value_record: ROOT->FILE_PATHS->input_data_file
ERROR when casting input file element
input file path = C:\Users\david\my_popy_egs\examples\fit_example4\input.csv
NOT present on file system.
```

Essentially this is telling you that this entry in 'builtin_fit_example.pyml':-

```
FILE_PATHS: {input_data_file: input.csv}
```

Is causing a problem because 'input.csv' does **not** exist. The popy_create tool outputs a script file only, so the 'input.csv' entry is just a place holder. You need to set the 'input_data_file' field to point at an existing data file to run the *Fit Script*.

Note one way of creating a data file is to create and then run a *Gen Script* or *Tut Script*, as described below.

The 'my_fit_script.pyml' is just meant to provide an easy template for you to use for your own analyses. The 'tut_script' example below is more complete, as a *Tut Script* works with just a single .pyml file.

4.1.2 Creating an example Tut Script

As always the first step is to *Open a PoPy Command Prompt*. Then do:-

```
$ popy_create tut my_tut_script.pyml
```

You can generate a more verbose example script with comments using:-

```
$ popy_create -acsl tut my_tut_script.pyml
```

See *popy_create* for the command line switch options. Or type 'popy_create -h'.

You can use this mechanism to create an example file for any of the *Script File Formats* in PoPy.

You can open the 'my_tut_script.pyml' in your system *editor* as follows:-

```
$ popy_edit my_tut_script.pyml
```

You should now have a run-able *Tut Script*. Run as follows:-

```
$ popy_run my_tut_script.pyml
```

You can use this mechanism to create an example file for any of the *Script File Formats* in PoPy.

4.2 Fitting a Two Compartment PopPK Model

The *Fitting a Simple PopPK Model using PoPy* shows fitting a one compartment PK model, to pre-existing data. In this example, we again utilise a pre-existing data set. We will demonstrate how to fit a two compartment model with absorption and bolus dosing, see Fig. 4.1:-

This model is called a **two compartment** model, because the **Depot** is not included, as is conventional in PK models.

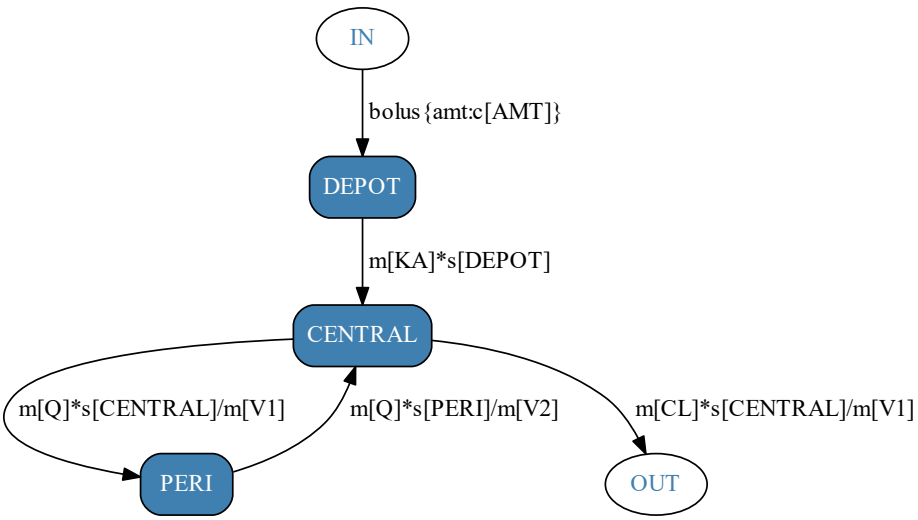


Fig. 4.1: Two compartment model with depot dosing for *Fit Script*. Here `c[AMT]` is the size of the bolus dose specified in the *data file*. `m[KA]`, `m[CL]`, `m[Q]`, `m[V1]`, `m[V2]` are all *MODEL_PARAMS* to be estimated for each individual.

This PoPy model is also created by default when *Creating an example Fit Script*, using *popy_create*.

Note: See the *First order absorption model with peripheral compartment* obtained by the PoPy developers for this example, including input script and input data file.

4.2.1 Run the Fit Script

This fitting example uses these two files:-

```
c:\PoPy\examples\builtin_fit_example.pyml
                 builitin_fit_example_data.csv
```

Open a PoPy Command Prompt to setup the PoPy environment in this folder:-

```
c:\PoPy\examples\
```

With the PoPy environment enabled, do:-

```
$ popy_edit builtin_fit_example.pyml
```

To view the script in an editor and then run the *Fit Script* using *popy_run* from the command line:-

```
$ popy_run builtin_fit_example.pyml
```

When the fit script has completed, you can view the output of the fit using *popy_view*, by typing the following command:-

```
$ popy_view builtin_fit_example.pyml.html
```

Note the extra '.html' extension in the above command. This command opens a local .html file in your web browser to summarise the result of the fitting.

You can compare your local html output with the pre-computed documentation output, see *First order absorption model with peripheral compartment*. You should expect some minor numerical differences when comparing results with the documentation.

4.2.2 Summary of Fit Results

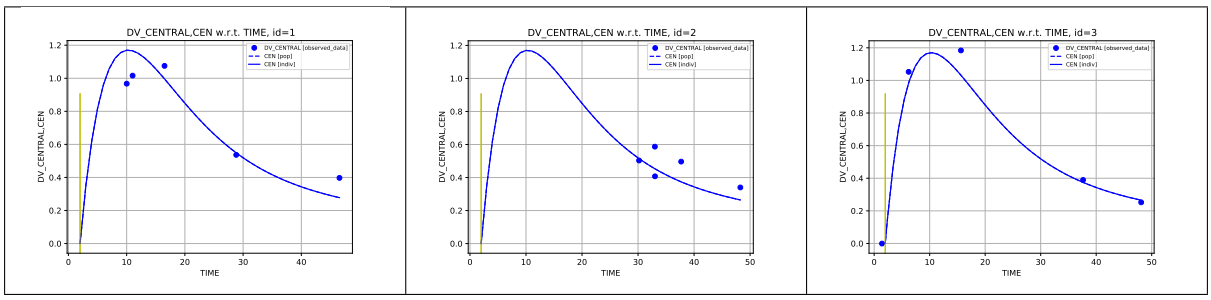
The results of running the fitting script are PoPy's best estimate for the presumed unknown *fixed effects* variables:-

```
f[KA] = 0.1089
f[CL] = 2.2693
f[V1] = 27.1342
f[Q] = 1.8655
f[V2] = 52.0726
f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv] = [
    [ 0.0807, 0.0287, 0.0288, 0.0131, -0.0713 ],
    [ 0.0287, 0.0228, 0.0284, 0.0062, -0.0334 ],
    [ 0.0288, 0.0284, 0.1865, 0.0233, -0.1768 ],
    [ 0.0131, 0.0062, 0.0233, 0.0109, -0.0022 ],
    [ -0.0713, -0.0334, -0.1768, -0.0022, 0.5211 ],
]
f[PNOISE] = 0.1395
```

The aim of a *Fit Script* is to optimise the *fixed effects* and *random effects* maximizing the likelihood of observing the input data given the model structure defined in 'builtin_fit_example.pyml'. The input data in this case, is the `c[DV_CENTRAL]` column in 'builtin_fit_example_data.csv', which contains 50 individuals each with 5 observations at random time points following a bolus dose event.

You can visually compare the fitted `f[X]` outputs with the input data, see Table 4.1.

Table 4.1: Model predictions vs original data points for first three individuals



In the graphs above the blue dots represent the original data points. The solid blue line represents the model individual predictions given the final `f[X]` parameters and fitted `r[X]` values for each individual. The dashed blue lines represent the model population predictions given final `f[X]` parameters and `r[X]` values set to zero.

Note in this model a bolus dose is received by all individuals at time 2.0. Then the amount of drug in the **Central** compartment follows a complex PK curve as it is first absorbed form the **Depot** compartment and then eliminated over time, whilst also interacting with the **Peripheral** compartment.

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

The data file included in this example is synthesized from the PK model of the same form described in 'builtin_fit_example.pyml' (see *Generate a Two Compartment PopPK Data Set*). So in this case, the model structure is known to be correct, so we should expect a good model fit.

4.2.3 Syntax of Fit Script

The mixed effect population structure is defined in the *EFFECTS* section 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[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])
```

There are 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. There are 50 individuals in the data set, therefore this model is attempting to estimate 6 main `f[X]` parameters, 15 variance `f[X]` parameters (the covariance matrix is symmetric) and 5 `r[X]` per individual. There are 271 parameters in total (i.e 15+6 `f[X]` + 50*5 `r[X]`).

The allowable ranges and starting values for the main `f[X]` are defined using the following syntax:-

```
f[X] ~ P start_x
```

Here the 'P' is short for 'positive'. This expression is actually a shortcut for:-

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

Where a *~unif()* distribution is used to define a range of allowed values [0.0, +inf]. Note, it's quite common to require PK/PD model parameters be non-negative, in order to make physical sense. The 'start_x' value is the initial value for `f[X]` used in the optimisation, which is usually an initial guess by the modeller.

Each individual has a unique `r[KA, CL, V1, Q, V2]` vector, because the random effects are defined at the ID level. For more info on the syntax above see *EFFECTS*. The `r[X]` are here defined as a zero-mean, multi-variate normal distribution:-

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

Note the second parameter of *~mnorm()* distribution, the square covariance matrix `f[KA_isv, CL_isv, V1_isv, Q_isv, V2_isv]` is a population parameter shared by all individuals.

When fitting a model, the `f[KA_isv, CL_isv, V1_isv, Q_isv, V2_isv]` matrix is defined using a *~spd_matrix()* distribution:-

```
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],
]
```

Where **spd** is short for *symmetric positive definite*. This distribution will always return a matrix with positive eigenvalues, starting with an initial matrix:-

$$\begin{pmatrix} 0.05 & 0.01 & 0.01 & 0.01 & 0.01 \\ 0.01 & 0.05 & 0.01 & 0.01 & 0.01 \\ 0.01 & 0.01 & 0.05 & 0.01 & 0.01 \\ 0.01 & 0.01 & 0.01 & 0.05 & 0.01 \\ 0.01 & 0.01 & 0.01 & 0.01 & 0.05 \end{pmatrix}$$

Note as the initial matrix is symmetric it is only necessary to specify the lower triangle elements. PoPy will update the 15 free elements of this matrix to increase the likelihood fit.

Given the `f[X]` and `r[X]` the mapping to the `m[X]` for each individual is defined in the *MODEL_PARAMS* section:-

```
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]
```

This shows that the `m[KA], m[CL], m[V1], m[Q], m[V2]` parameters for each individual are modelled as log normal distributions with median values of `f[KA], f[CL], f[V1], f[Q], f[V2]`. There is a shared proportional noise parameter `f[PNOISE]` for all individuals. And small fixed additive noise constant `m[ANOISE]`. For more info on the syntax above see *MODEL_PARAMS*.

A two compartment model with first order elimination and bolus dosing via a depot compartment is defined 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]
```

The bolus arrives in the **Depot** compartment, due to the `@bolus` term appearing on the right hand side of the `d[DEPOT]` equation:-

```
d[DEPOT] = @bolus{amt:c[AMT]} - m[KA]*s[DEPOT]
```

The amount of the bolus dose is `c[AMT]`, which is defined in the data file. In this case it is always 100 units and occurs at time point 2.0 for all individuals. The elimination rate from the **Depot** compartment is `m[KA]`, which is first order with respect to `s[DEPOT]`.

The **Central** compartment, which represents the blood plasma and where drug concentration *observations* are made is defined as follows:-

```
d[CENTRAL] = m[KA]*s[DEPOT]␣
↪- s[CENTRAL]*m[CL]/m[V1] - s[CENTRAL]*m[Q]/m[V1]  + s[PERI]*m[Q]/m[V2]
```

This consists of the input from the **Depot** `m[KA]*s[DEPOT]` and a first order elimination expression `s[CENTRAL]*m[CL]/m[V1]` that represents the removal of the drug from the blood plasma. Here the

elimination rate is expressed as a ratio of:-

- `m[CL]`: *clearance* from **Central** compartment
- `m[V1]`: *volume of distribution* of **Central** compartment

The last two terms `- s[CENTRAL]*m[Q]/m[V1] + s[PERI]*m[Q]/m[V2]` are simply the negative values of the rates for the **Peripheral** compartment:-

```
d[PERI] = s[CENTRAL]*m[Q]/m[V1] - s[PERI]*m[Q]/m[V2]
```

To compute the objective function for each row of the data set, the `s[X]` values are used to compute `p[X]` prediction variables which are then compared with the target `c[X]` values from the data file, as defined below:-

```
PREDICTIONS: |
    p[CEN] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[CEN]**2
    c[DV_CENTRAL] ~ norm(p[CEN], var)
```

The predicted variable `p[CEN]` is defined as follows:-

```
p[CEN] = s[CENTRAL]/m[V1]
```

Hence this is a concentration, because we are dividing the amount `s[CENTRAL]` by the *volume of distribution* of the **Central** compartment. The other two lines:-

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

show that we are comparing the model prediction `p[CEN]` with the data `c[DV_CENTRAL]` and using *~norm()* distribution likelihood error model. The variance is a proportional noise model, where the standard deviation of the proportional noise is `m[PNOISE]`. Here `m[ANOISE]` is fixed to a small positive constant, this is to avoid zero variances when `p[CEN]` is close to zero. For more info on the syntax above see *PREDICTIONS*.

PoPy is essentially trying to find the best combination of fixed parameters as follows:-

- `f[KA]` - the median elimination rate from the **Depot** -> **Central** compartment
- `f[CL]` - the median *clearance* of the **Central** compartment
- `f[V1]` - the median *volume of distribution* of the **Central** compartment
- `f[Q]` - the median *clearance* between the **Central** <-> **Peripheral** compartments
- `f[V2]` - the median *volume of distribution* of the **Peripheral** compartment
- `f[KA_isv, CL_isv, V1_isv, Q_isv, V2_isv]` - The covariance structure of the `f[X]` parameters above over the population of individuals.
- `f[PNOISE]` - the proportional noise not explained by the model in the `c[DV_CENTRAL]` data.

The unexplained noise `f[PNOISE]` and between subject variance `f[KA_isv, CL_isv, V1_isv, Q_isv, V2_isv]` obfuscate each other. However the population as a whole contains enough data to solve this problem using maximum likelihood *[Sheiner1980]*.

In PoPy the likelihood is optimised iteratively, with the `f[X]` and `r[X]` being updated at each iteration. In this case, the likelihood (or objective function) progressed as follows (Table 4.2):

Table 4.2: Objective values vs iteration number and time

Iteration	Time	OBJV

Continued on next page

Table 4.2 – continued from previous page

0.	0.00	4694.960923770783
0.	0.46	925.5746942755109
1.	2.38	925.5746942755109
1.1	2.53	-647.9716888729727
1.2	22.11	-709.3982861305856
1.3	33.06	-795.3954364398459
1.4	46.32	-835.8472850616511
1.5	56.24	-844.5443326611603
1.6	65.79	-847.5798357472844
1.7	77.21	-850.9366829942037
1.8	86.85	-852.523825175209
1.9	98.72	-853.8255559623917
1.10	110.44	-855.3704179834231
1.11	121.76	-860.0439762888727
1.12	132.15	-866.292551929844
1.13	142.71	-869.6903314278618
1.14	152.86	-872.0115456718268
1.15	169.94	-874.9343769366639
1.16	184.48	-876.591980985562
1.17	196.55	-880.8790734955087
1.18	207.65	-884.4774855727169
1.19	222.28	-886.9825001824557
1.20	236.16	-892.9493567116618
1.21	248.92	-894.278193027259
1.22	260.55	-897.4079567196504
1.23	273.04	-898.6664194606287
1.24	282.78	-900.2255171014091
1.25	294.78	-901.7780186834344
1.26	305.53	-903.0634884085612
1.27	317.28	-904.2825835322805
1.28	330.17	-906.5825115834821
1.29	348.63	-907.9108849096219
1.30	359.38	-908.9330529447419

Note that the objective function is defined as -2 * the log likelihood. Therefore the lower the objective function the more likely the input data will be observed given the current `f[X]` values. By default PoPy stops the fitting algorithm once the objective function has stopped decreasing.

4.2.4 Visual Predictive Check for Two Compartment PopPK Model

The *Fitting a Two Compartment PopPK Model* section showed fitting a PK/PD model to a data set.

As shown previous in *Visual Predictive Check for Simple PopPK Model*. It is possible to use the fitted *fixed effects* values, i.e the optimised `f[X]` variables, to generate a *visual predictive check*, often abbreviated to 'VPC'.

Running the MSim Script

It is presumed that you have already run the 'builtin_fit_example.pyml' script from *Fitting a Two Compartment PopPK Model*. If you have then you should have access to the following output folder:-

```
builtin_fit_example.pyml_output/
    msim/
        builtin_fit_example_msim.pyml
```

You need to *Open a PoPy Command Prompt* in the 'msim' sub folder then do:-

```
$ popy_edit builtin_fit_example_msim.pyml
```

To open the *MSim Script* in an editor. You can then run the script using:-

```
$ popy_run builtin_fit_example_msim.pyml
```

If you run this script the following .svg file is output:-

```
builtin_fit_example_msim.pyml_output/
    DV_CENTRAL_sim,DV_CENTRAL_wrt_TIME_SINCE_LAST_DOSE_comb_quant_sim_vpc/
        000000.svg
```

This graphic should look something like Fig. 4.2:-

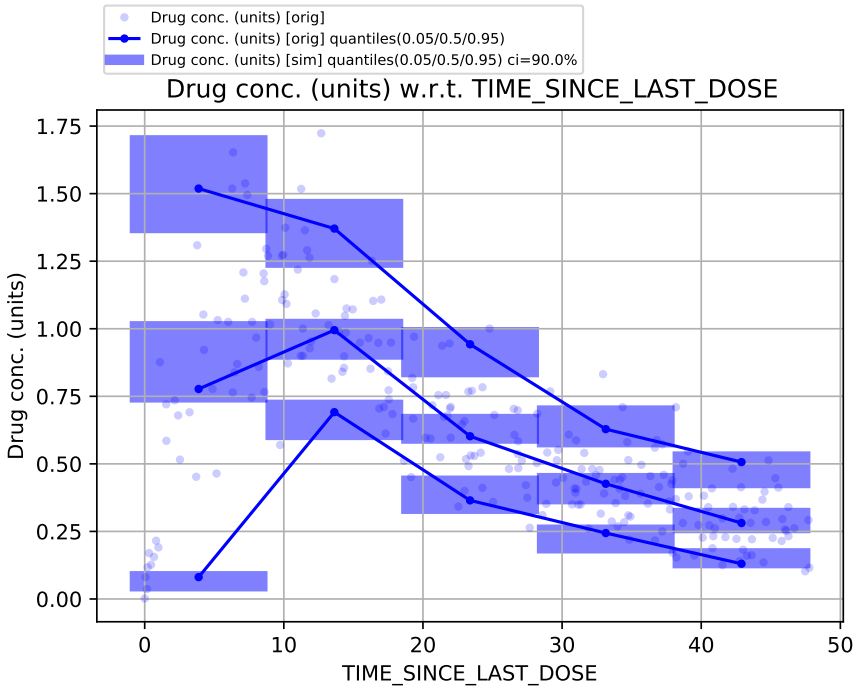


Fig. 4.2: Visual Predictive Check for Complex PopPK model.

Here the y axis is the concentration in the **Central** compartment and the x axis is the time since the last dose (*TSLD*). See *Visual Predictive Check for Simple PopPK Model* for a more general description of the *VPC* plot shown in Fig. 4.2.

In this case, the *TSLD* values are grouped into 12 equally spaced bins along the x axis. Note you need a minimum number of data points in each bin and there are only 250 data points in this toy example. Hence the small number of bins.

The blue dots (original data) are mainly shown to give some visual corroboration of the quantiles (solid blue line). In this graph there are only 12 bins and therefore each bin is quite wide, therefore some data points are grouped together inappropriately. This grouping issue is most obvious at the smaller values of TSLD, during the drug uptake period, when the drug is being mainly absorbed into **Central** compartment and has not been cleared from the blood plasma yet (see Fig. 4.2). Only more data and more bins can really fix the issue.

Syntax in the MSim Script

For each individual in the original data set, new synthetic data sets are created by sampling new *random effects* `r[X]` variables and new measurement noise for all data rows. *i.e.* The synthetic populations vary due to sampling the `r[X]` for each individual here:-

```
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])
```

And adding measurement noise here:-

```
PREDICTIONS: |
    p[CEN_sim] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[CEN_sim]**2
    c[DV_CENTRAL_sim] ~ norm(p[CEN_sim], var)
```

This procedure creates a set of N new data sets, which can be compared with the original data set. Where N is defined here:-

```
OUTPUT_OPTIONS:
    n_pop_samples: 100
```

You can increase the number of samples, to make the *VPC* more representative of your model. The more complex the PK/PD model, the more synthetic data samples you will need.

As this model has more parameters compared to the *Fitting a Simple PopPK Model using PoPy*, it may be worth increasing the 'n_pop_samples' and re-running the *MSim Script*. This is left as an exercise for the reader.

4.3 Generate a Two Compartment PopPK Data Set

The *Fitting a Two Compartment PopPK Model* section showed fitting a PK/PD model to a pre-existing data set. However in PoPy it is also possible to use a *Gen Script*, to generate a data set from a model file instead. *i.e.* The opposite of a *Fit Script*.

In this example we will demonstrate how to generate new data from a two compartment model with absorption and bolus dosing, see Fig. 4.3:-

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.

Note: See the *First order absorption model with peripheral compartment* obtained by the PoPy developers for this example, including input script and output data file.

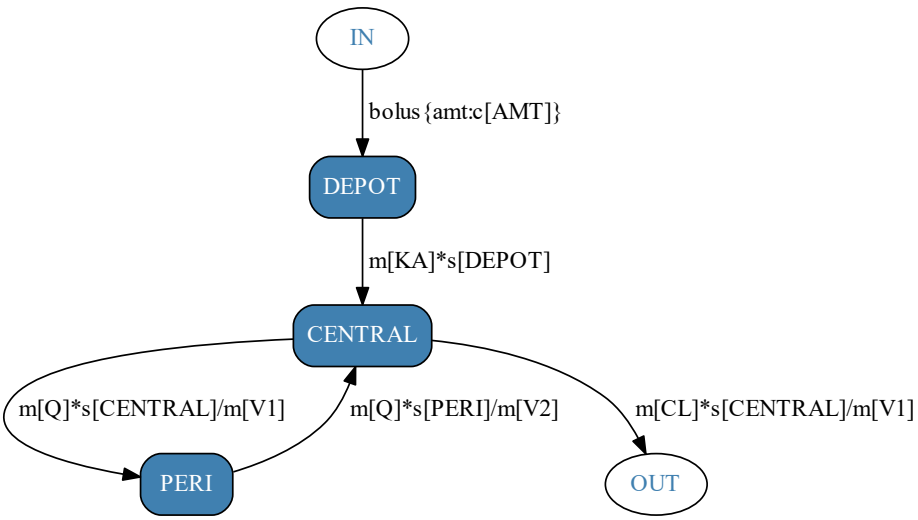


Fig. 4.3: Two compartment model with depot dosing for *Gen Script*. This is the same model as Fig. 4.1.

4.3.1 Running the Gen Script

This generating example make use of this single file:-

```
c:\PoPy\examples\builtin_gen_example.pyml
```

Open a PoPy Command Prompt to setup the PoPy environment in this folder:-

```
c:\PoPy\examples\
```

With the PoPy environment enabled, open the *Gen Script* in an editor as follows:-

```
$ popy_edit builtin_gen_example.pyml
```

then execute the script using *popy_run* from the command line:-

```
$ popy_run builtin_gen_example.pyml
```

When the gen script has completed, you can view the output of the generating process using *popy_view*, by typing the following command:-

```
$ popy_view builtin_gen_example.pyml.html
```

Note the extra '.html' extension in the above command. This command opens a local .html file in your web browser to summarise the result of the generating process.

You can compare your local html output with the pre-computed documentation output, see *First order absorption model with peripheral compartment*. You should expect some minor numerical differences when comparing results with the documentation.

4.3.2 Summary of Gen Results

The main inputs of the generating script are the *fixed effects* `f[X]` variables as defined in the *EFFECTS* section of the *Gen Script*. In this case the `f[X]` are all constant and summarised here:-

```
f[KA] = 0.2000
f[CL] = 2.0000
f[V1] = 50.0000
f[Q] = 1.0000
f[V2] = 80.0000
f[KA_isv,CL_isv,V1_isv,Q_isv,V2_isv] = [
    [ 0.1000, 0.0100, 0.0100, 0.0100, 0.0100 ],
    [ 0.0100, 0.0300, -0.0100, 0.0200, 0.0200 ],
    [ 0.0100, -0.0100, 0.0900, 0.0100, 0.0100 ],
    [ 0.0100, 0.0200, 0.0100, 0.0700, 0.0100 ],
    [ 0.0100, 0.0200, 0.0100, 0.0100, 0.0500 ],
]
f[PNOISE] = 0.1500
```

If the `f[X]` are random variables, which in PoPy are defined using a ~, then the *Gen Script* will sample each `f[X]` variable once. Sampling the `f[X]` however makes more sense if you are creating multiple synthetic data sets, see *MGen Script*.

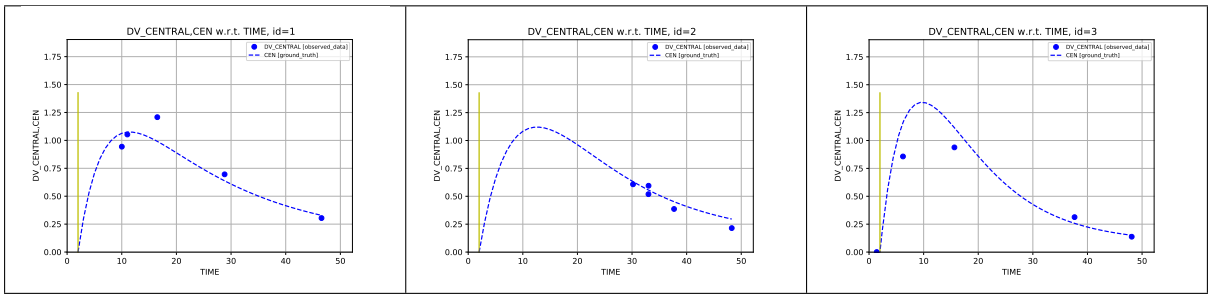
Given the population `f[X]` variables, the *Gen Script* then creates the requested number of individuals (in this case 50) and samples a set of time points (in this case 5) and dosing times (in this case a single bolus dose) for each individual. This step defines the number of rows in the synthetic data set.

The next stage is to sample any `c[X]` variables specified for each individual. In this example the only `c[X]` variables defined in the gen_script are the `c[ID]` field and `c[AMT]` value (which in this case is constant for all individuals). The `c[TIME]` and `c[TYPE]` fields are created by PoPy automatically. We now have most of a valid PoPy data set, but no observation values are defined yet.

To generate observations the `r[X]` variables for each individual are sampled. This along with the dose times and observation time period is enough to simulate smooth PK/PD curves from the *MODEL_PARAMS*, *DERIVATIVES* and *PREDICTIONS* defined in the script.

You can visualise the model predictions outputs (`p[X]` variables) by examining the plots for the first three individuals in the data set.

Table 4.3: Synthetic data plots for first three individuals



In Table 4.3 above, the dotted blue line represents the model predictions given the `f[X]` parameters and sampled `r[X]` values for each individual. No noise is added to this curve and it is plotted at regular unit time steps, therefore it is smooth.

The solid blue dots represent the observations with noise added at randomly sampled time points for each individual. The solid blue dots are the values that end up in the synthetic data file under the `c[DV_CENTRAL]` field.

Note in this model a bolus dose is received by all individuals at time 2.0. After the dose, the concentration of the drug in the **Central** compartment increases as drug is absorbed from the **Depot** compartment. Then the drug concentration falls as the drug is metabolised. The decay curve is first order with an inflection point due to the **Peripheral** compartment.

The doses are the same for all individuals, but the smooth curves generated by the model vary due to each individual having a different `r[X]` vector.

4.3.3 Syntax in the Gen Script

The *EFFECTS* section defines the population structure that the *Gen Script* will create 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[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])
```

This *EFFECTS* structure is similar to the *Syntax of Fit Script* with some additional lines to define new individuals, doses and observation times.

The number of individuals is defined by the following line:-

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

This specifies a sequence where the first individual is '1', the 2nd is '2' *etc.* up to '50'.

This line specifies a single dose record for each individual at time 2.0:-

```
t[DOSE] = 2.0
```

This line request a sample of 5 time points uniformly distributed in the period [1.0, 50.0]:-

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

The random effects are here sampled from a zero-mean, multi-variate normal distribution, as follows:-

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

Note the second parameter of mnorm, the square covariance matrix `f[KA_isv, CL_isv, V1_isv, Q_isv, V2_isv]` is a population parameter shared by all individuals. Each individual has a unique `r[KA, CL, V1, Q, V2]` vector, because the random effects are defined at the ID level. For more info on the syntax above see *EFFECTS*.

The *MODEL_PARAMS* and *DERIVATIVES* sections of this *Gen Script* are the same as the *Syntax of Fit Script*, so are not discussed here.

The *PREDICTIONS* section in the *Gen Script* defines how the dependent `c[X]` variables are sampled given the `p[X]` model predictions:-

```
PREDICTIONS: |
    p[CEN] = s[CENTRAL]/m[V1]
    var = m[ANOISE]**2 + m[PNOISE]**2 * p[CEN]**2
    c[DV_CENTRAL] ~ norm(p[CEN], var)
```

PoPy samples `c[DV_CENTRAL]` for each row of the data set, to create a synthetic noisy measurement at each time point for each individual.

4.3.4 Structure of output synthetic data file

The `c[X]` variables are saved to disk. For an example data file see *First order absorption model with peripheral compartment*. The first few lines of the 'synthetic_data.csv' are shown in (Table 4.4) below:-

Table 4.4: First 10 rows of 'synthetic_data.csv' file

TIME	ID	TYPE	AMT	DV_CENTRAL	DV_CENTRAL_FLAG	orig_data_row
2	1	dose	100	-0.000152985803347	0	0
10.0120217722	1	obs	100	0.943643872407	1	1
11.0234536491	1	obs	100	1.05284021173	1	2
16.5024021745	1	obs	100	1.20785360788	1	3
28.818526425	1	obs	100	0.695942623338	1	4
46.551188548	1	obs	100	0.304235321489	1	5
2	2	dose	100	0.00126357225648	0	0
30.181690446	2	obs	100	0.607437669642	1	1
33.0056777467	2	obs	100	0.518873750462	1	2
...	...	...	...	...	...	...

This shows some of the typical properties of PoPy's *PoPy Data Format*, where the main fields are:-

- TYPE - Specifies either a dose or an observation row.
- ID - The identifier for a given subject.
- TIME - The time stamp of the row event.
- AMT - The size of the dose at a given time.
- DV_CENTRAL - The synthetic observed values.
- DV_CENTRAL_FLAG - Indicates valid measurement rows, 1=valid 0=ignore.
- orig_data_row - The data row number within an individual subject.

4.3.5 Controlling Random Seed in PoPy scripts

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.

4.4 Generate data and Fit using a Two Compartment Model

In this example we will demonstrate a *Tut Script* using the same two compartment model with absorption and bolus dosing, as used in the *Fitting a Two Compartment PopPK Model* and *Generate a Two Compartment PopPK Data Set*, see Fig. 4.4:-

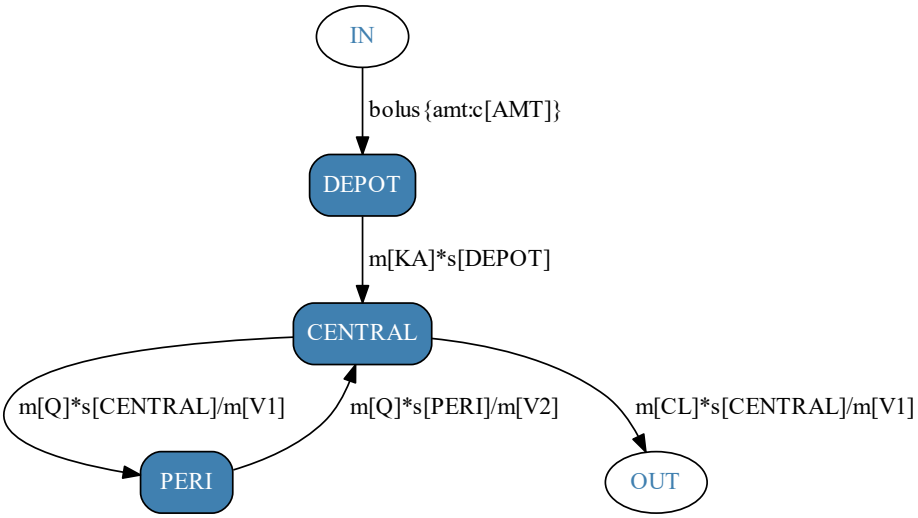


Fig. 4.4: Two compartment model with depot dosing for *Tut Script*.

This PoPy model is also used when *Creating an example Tut Script*.

Note: See the *First order absorption model with peripheral compartment* obtained by the PoPy developers for this example, including input script and output data file.

A *Tut Script* can be used as a theoretical tool to quickly investigate identifiability of PK/PD models, because the true `f[X]` parameters and the structure of the data are known.

The PoPy Manual makes extensive use of *tut_scripts* to create examples to illustrate different *Principles of Pharmacokinetics*.

4.4.1 Running the Tutorial Script

This tutorial example requires a single input file:-

```
c:\PoPy\examples\builtin_tut_example.pyml
```

Open a PoPy Command Prompt to setup the PoPy environment in this folder:-

```
c:\PoPy\examples\
```

With the PoPy environment enabled, you can open the script using:-

```
$ popy_edit builtin_tut_example.pyml
```

Again, with the PoPy environment enabled, call *popy_run* on the *Tut Script* from the command line:-

```
$ popy_run builtin_tut_example.pyml
```

When the tut script has terminated, you can view the output of the fit using *popy_view*, by typing the following command:-

```
$ popy_view builtin_tut_example.pyml.html
```

Note the extra '.html' extension in the above command. This command opens a local .html file in your web browser to summarise the result of the generating process.

You can compare your local html output with the pre-computed documentation output, see *First order absorption model with peripheral compartment*. You should expect some minor numerical differences when comparing results with the documentation.

4.4.2 Syntax of Tut Script

The major structural difference between a *Fit Script* or *Gen Script* and a *Tut Script* is that the generating *EFFECTS* are encoded in *GEN_EFFECTS* and the fitting *EFFECTS* are encoded in *FIT_EFFECTS*. For example the *GEN_EFFECTS* section for this tutorial example is 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[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])
```

And the *FIT_EFFECTS* section is as follows:-

```
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 *GEN_EFFECTS* get passed to the *Gen Script* and the *FIT_EFFECTS* get passed to the *Fit Script*. From the examples above you can see that the GEN_EFFECTS->POP section has:-

```
f[X] = true_value
```

Whereas the FIT_EFFECTS->POP section has:-

```
f[X] ~ P starting_value
```

Reflecting the fact that the `f[X]` are known constants for a *Gen Script*, but are unknown values to be estimated in a *Fit Script*.

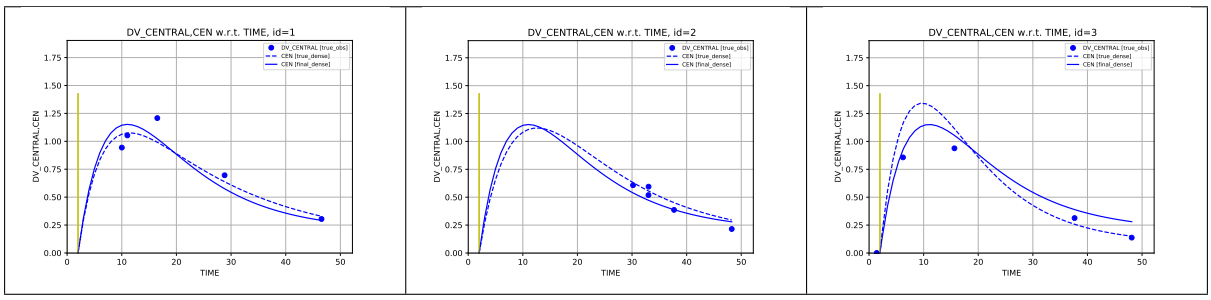
4.4.3 Summary of Tut Results

The *Tut Script* generates an output folder containing four new scripts:-

```
builtin_tut_example.pyml_output/
    builtin_tut_example_gen.pyml
    builtin_tut_example_fit.pyml
    builtin_tut_example_comp.pyml
    builtin_tut_example_tutsum.pyml
```

See *Files Generated by Tut Script* for more info. These four scripts are run in order. The *Gen Script* is described in *Generate a Two Compartment PopPK Data Set* and the *Fit Script* is described in *Fitting a Two Compartment PopPK Model*. Therefore here we will focus on the *Comp Script* outputs, which are fitted `f[X]` and generated `f[X]` plots and their objective values. The simplest comparison output is a visual comparison of the true and fitted `f[X]` PK curves and the synthetic measurement data, see Table 4.5.

Table 4.5: Fitted model PK curves vs true model PK curves for first three individuals



The solid blue lines in Table 4.5 show the predicted PK curves for the fitted model `f[X]` values. The dotted blue lines show the PK curves for the true `f[X]` values that were used to generated the data set (in the *Gen Script*). The blue dots are the target `c[DV_CENTRAL]` values from the data file.

The target `c[DV_CENTRAL]` values have measurement noise added, so blue dot data points do **not** lie on the true `f[X]` curves. The graphs show that the PK curves for the fitted `f[X]` are very similar to the true `f[X]` curves. Most divergence occurs away from the data points and most agreement close to the data points, for example for Individual 2 the fitted and true curves differ over time period [0,30], but are very similar in the period [30,50]. In the period [0,30], the model tends to impute the curve for the median individual, in the absence of actual data.

Fit vs True f[X] values

If the *Comp Script* has been run, the *TutSum Script* output also contains a convenient table to compare the initial, fitted and true `f[X]` values, see Table 4.6.

Table 4.6: Comparison of main initial, fitted and true `f[X]` values

Name	Initial	Fitted	True	Prop. Error	Abs. Error
f[KA]	1	0.121	0.2	39.56%	7.91e-02
f[CL]	1	1.56	2	22.22%	4.44e-01
f[V1]	20	33.9	50	32.12%	1.61e+01
f[Q]	0.5	2.22	1	122.23%	1.22e+00
f[V2]	100	120	80	49.45%	3.96e+01

Table 4.6 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 4.7: Comparison of fitted and true proportional noise `f[X]` values

Name	Initial	Fitted	True	Prop. Error	Abs. Error
f[PNOISE]	0.1	0.148	0.15	1.37%	2.05e-03

Here the `f[PNOISE]` starts at 0.1, 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. This is 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 5*5 `f[X]` comparisons. However the diagonal elements (only) are shown here:-

Table 4.8: Comparison of fitted and true isv variance diagonal `f[X]` values

Name	Initial	Fitted	True	Prop. Error	Abs. Error
f[KA_isv]	0.05	0.21	0.1	110.44%	0.11
f[CL_isv]	0.05	0.029956	0.03	0.15%	4.37E-05
f[V1_isv]	0.05	0.0473	0.09	47.47%	0.0427
f[Q_isv]	0.05	0.0729	0.07	4.14%	0.0029
f[V2_isv]	0.05	0.0567	0.05	13.45%	0.00673

This shows that `f[CL_isv]` and `f[Q_isv]`, the inter-subject variances of the clearances are estimated reasonably well. The `f[KA_isv]` estimate is far too high. The `f[V1_isv]` and `f[V2_isv]` estimates are worse than the starting value estimates.

The fact that `f[V1_isv]` and `f[V2_isv]` are badly estimated compared to the true values is not that surprising, since *volumes of distribution* are harder to estimate in PK models, compared to *clearances* which are rates. The difficulty in estimating `f[KA_isv]` may be due to the relative lack of data before the **cmax** peak of the PK curve, there being only 5 data points sampled per individual.

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

- 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, for example if the model is over-parametrised relative to the data set or even unidentifiable.

Given the relatively small amount of data generated (250 data points), the results in Table 4.6 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 here, see below for discussion.

Fit vs True Objective value

It is difficult to know by just comparing the fitted `f[X]` and true `f[X]` in section *Fit vs True f[X] values*, if 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 rational 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 sub optimal local minima, because the known true values are a better minima.

In this example, the true model objective function is:-

```
-881.0027
```

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]`. However it does **not** say if the global optimum solution was found, *i.e.* whether or not `f[X]` is a true global minima 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.

4.4.4 Re-run the tutorial

You can familiarise yourself with PoPy's various features by tweaking the tutorial example and re-running. A simple way of avoiding overwriting previous results is to do:-

```
$ copy builtin_tut_example.pyml builtin_tut_example_v2.pyml
$ popy_edit builtin_tut_example_v2.pyml
```

Then when you are happy with the edited file do:-

```
$ popy_run builtin_tut_example_v2.pyml
```

For example, you can adjust the amount of data generated, in this section:-

```
GEN_EFFECTS:
    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)
```

Note the 'range' function can sample time points evenly (instead of randomly). This usually makes the model fitting easier as the data points span the time range better. You can experiment with the number of doses or make a random sample of dose times for each individual.

You can also edit the underlying model or compartment structure, see the *MODEL_PARAMS* or *DERIVATIVES* sections.

Another possibility is changing the fitted model only, i.e take a copy of this file:-

```
builtin_tut_example.pyml_output/
    builtin_tut_example_fit.pyml
```

Name it 'builtin_tut_example_fit_v2.pyml', change the model structure or compartment. Then do:-

```
$ popy_run builtin_tut_example_fit_v2.pyml
```

Using a different model from the underlying generative model should result in a worse fit (using say an *Akaike information criterion* to compare fits).

4.5 Generate multiple data sets and Fit using a Two Compartment Model

The previous example showed how to *Generate data and Fit using a Two Compartment Model*, which generates data from a model then fits the model to the synthetic data.

In PoPy it is also possible to iterate over a *Tut Script*, using what we call a *MTut Script* or multi-tutorial script and apply the generate/fit process multiple times. In this example we will demonstrate a *MTut Script* using the same two compartment model with absorption and bolus dosing, as used in the Complex PopPK *Tut Script* example, see Fig. 4.5:-

An *MTut Script* is a tool for investigating identifiability of PK/PD models, but gives you more information than a *Tut Script* because the synthetic population data is sampled multiple times, which gives a spread of fitted `f[X]` results for each model parameter.

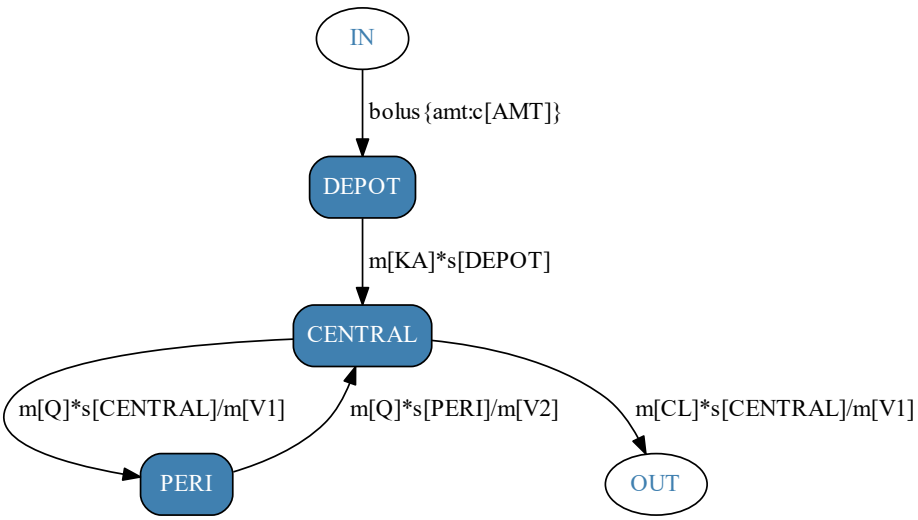


Fig. 4.5: Two compartment model with depot dosing for *MTut Script*.

4.5.1 Running the MTut Script

This multi tutorial example make use of this single file:-

```
c:\PoPy\examples\builtin_mtut_example.pyml
```

Open a PoPy Command Prompt to setup the PoPy environment in this folder:-

```
c:\PoPy\examples\
```

With the PoPy environment enabled, you can open the script using:-

```
$ popy_edit builtin_mtut_example.pyml
```

Again, with the PoPy environment enabled, call *popy_run* on the *MTut Script* from the command line:-

```
$ popy_run builtin_mtut_example.pyml
```

Running an 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 limit the `f[X]` parameters that are estimated.

4.5.2 Syntax of MTut Script

The *MTut Script* is very similar to the *Tut Script* in *Syntax of Tut Script*, the primary difference is that the *MTut Script* specifies the number of populations to sample as follows:-

```
OUTPUT_OPTIONS: {n_pop_samples: 30}
```

Like a *Tut Script*, the *MTut Script* encodes **both** the generation and fitting effects in the *GEN_EFFECTS* and *FIT_EFFECTS* sections. In this example the *GEN_EFFECTS* and *FIT_EFFECTS* sections are slightly different from *Syntax of Tut Script*, as follows:-

```
GEN_EFFECTS:
    POP: |
        c[AMT] = 100.0
```

```
        f[KA] ~ unif(0.1,0.3)
        f[CL] ~ unif(7.0, 13.0)
        f[V1] ~ unif(30.0,70.0)
        f[Q] ~ unif(0.5,1.5)
        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] ~ unif(0.1, 0.2)
    ID: |
        c[ID] = sequential(10)
        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] ~ P0.2
        f[CL] ~ P10.0
        f[V1] ~ P50.0
        f[Q] ~ P1.0
        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] ~ P0.15
    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 GEN_EFFECTS->POP level defines the following `f[X]`:-

```
f[KA] ~ unif(0.1,0.3)
f[CL] ~ unif(7.0, 13.0)
f[V1] ~ unif(30.0,70.0)
f[Q] ~ unif(0.5,1.5)
f[V2] = 80
f[PNOISE] ~ unif(0.1, 0.2)
```

This means that when running the *MGen Script* the `f[KA]`, `f[CL]`, `f[V1]`, `f[Q]` and `f[PNOISE]` are sampled 30 times to create 30 different synthetic data sets with different true `f[X]` values. The `f[V2]` parameters in contrast always has the constant value 80. Note the `f[KA_isv, CL_isv, V1_isv, Q_isv, V2_isv]` matrix is also constant here.

Whereas the FIT_EFFECTS->POP section tries to estimate the following `f[X]`:-

```
f[KA] ~ P0.2
f[CL] ~ P10.0
f[V1] ~ P50.0
```

```
f[Q] ~ P1.0
f[V2] = 80
f[PNOISE] ~ P0.15
```

These entries mean that when running the *MFit Script* the `f[KA]`, `f[CL]`, `f[V1]`, `f[Q]` and `f[PNOISE]` are estimated for each of the 30 synthetic data sets created by the *MGen Script*. The starting values for the parameter fitting are here set to the middle of the sampling region used in the **gen_params**. For example `f[KA]` has initial value 0.2, whereas the true value lies in the region [0.1,0.3] for each synthetic data set. `f[V2]` is not estimated at all, it is merely set to the true value *i.e.* 80. Similarly `f[KA_isv, CL_isv, V1_isv, Q_isv, V2_isv]` is set to the true constant value here. The 'P' in the **fit_params** forces the fitting process to estimate positive values.

Restricting the estimation to 5 `f[X]` parameters greatly speeds up the run time of *MFit Script*, which makes this toy example much faster to run. The *MTut Script* is only sampling 30 synthetic data sets, which is a relatively small number. You could increase this if you wish to obtain more detailed scatter plots (see below). Most of the run time is taken up by *MFit Script*, the *MGen Script* is relatively quick to run, as it only has to evaluate the *ordinary differential equations* in the *DERIVATIVES* block once per synthetic data set, where as the *MFit Script* must evaluate the *ordinary differential equations* multiple times to estimate `f[X]` for each synthetic data set.

4.5.3 Summary of MTut Results

The *MTut Script* should generate an output folder containing three new scripts:-

```
builtin_mtut_example.pyml_output/
    builtin_mtut_example_mgen.pyml
    builtin_mtut_example_mfit.pyml
    builtin_mtut_example_mcomp.pyml
```

The purpose of each of theses scripts is as follows:-

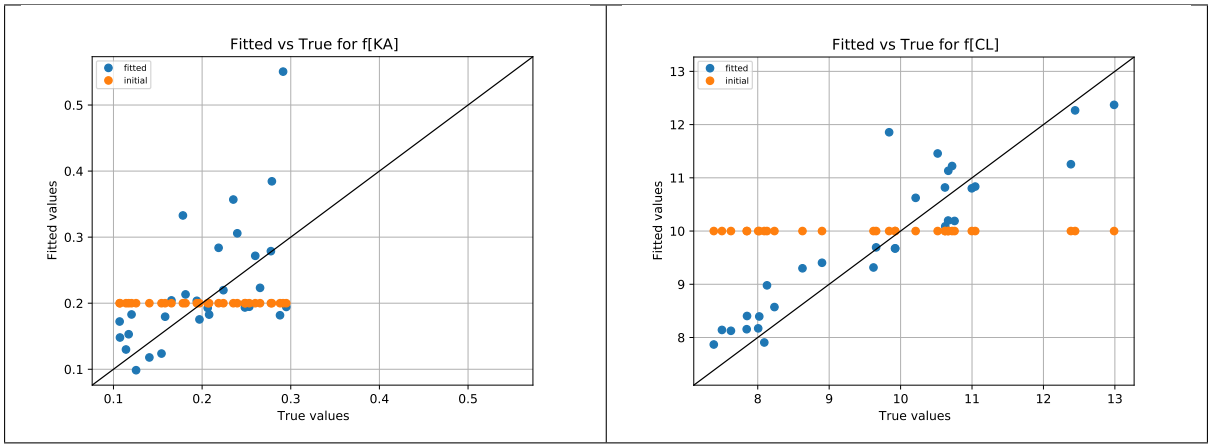
Table 4.9: Scripts output by a multi tutorial script

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 *Generate a Two Compartment PopPK Data Set* and the *MFit Script* is very similar to the *Fit Script* described in *Fitting a Two Compartment PopPK Model*. See *Files Generated by MTut Script* for more info.

Therefore 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 4.10 and Table 4.11.

Table 4.10: Fitted model vs true scatter plots for `f[KA]` and `f[CL]`



In Table 4.10 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[CL]` the true values are sampled as follows:-

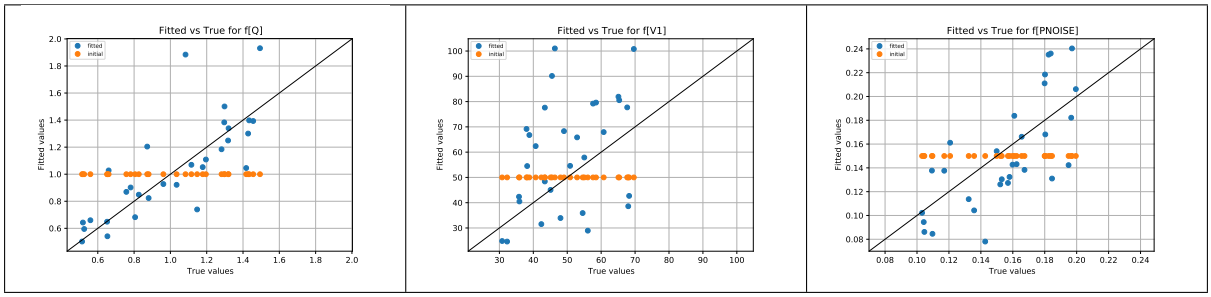
```
f[CL] ~ unif(7.0, 13.0)
```

i.e. the true values are uniformly sampled in the range [7.0,13.0]. The fitted `f[CL]` parameters are specified as follows:-

```
f[CL] ~ P10.0
```

The initial values for `f[CL]` are always 10.0. see green dots in a horizontal line on the right graph in Table 4.10. The 'P' specifies that the fitting value of `f[CL]` is restricted to positive numbers. The final fitting values are the blue dots on the right graph in Table 4.10. For `f[CL]` the blue dots are clustered along the black 45 degree line. Hence fitting for `f[CL]` works well, this agrees with the initial findings in *Fitting a Two Compartment PopPK Model*. The `f[KA]` fitted values (blue dots) on the left graph in Table 4.10 are less obviously grouped around the 45 degree line. Especially the outlier when the fitted value is estimated at 0.48 compared to a true value of 0.21.

Table 4.11: Fitted model vs true scatter plots for `f[Q]` and `f[V1]` and `f[PNOISE]`



In Table 4.11 (left graph) the `f[Q]` rate results for the **Peripheral** compartment are reasonably well estimated, apart from one outlier. The `f[V1]` parameter (middle graph) is really badly estimated, there does **not** seem to any correlation between true `f[V1]` values and fitted `f[V1]` values at all. The `f[PNOISE]` parameter (right graph) is reasonably well estimated, with the results mostly lying along the 45 degree line.

In this case, the `f[V1]` parameter (the *volume of distribution* of the **Central** compartment) is shown to be hard to estimate, compared to `f[CL]` and `f[Q]` *clearances*. The `f[KA]` estimate is a bit unstable, possibly due to this parameter requiring time point data shortly after the dose, which does not always exist. `f[PNOISE]` is reasonably well estimated, but it's likely that higher values of `f[PNOISE]` make the other parameters harder to identify.

Given the relatively small amount of data generated (50 data points spread across 10 individuals), the results in Table 4.10 and Table 4.11 are reasonably good.

4.5.4 Re-run the MTut Script

You can familiarise yourself with PoPy's various features by tweaking the multi-tutorial example and re-running. A simple way of avoiding overwriting previous results is to do:-

```
$ copy builtin_mtut_example.pyml builtin_mtut_example_v2.pyml
$ popy_edit builtin_mtut_example_v2.pyml
```

Then when you are happy with the edited file do:-

```
$ popy_run builtin_mtut_example_v2.pyml
```

For example, you can adjust the amount of data generated, in this section:-

```
EFFECTS:
    ID: |
        c[ID] = sequential(10)
        t[DOSE] = 2.0
        t[OBS] ~ unif(1.0, 50.0; 5)
        # t[OBS] = range(1.0, 50.0; 5)
```

Note the 'range' function can sample time points evenly (instead of randomly). This usually makes the model fitting easier as the data points span the time range better. You can experiment with the number of doses or make a random sample of dose times for each individual.

You could also change the initial fitting `f[X]` values, to make the `f[X]` estimation process start further away from the true `f[X]` values. You could also attempt to estimate more (or fewer) parameters, for example try estimating the `f[KA_isv, CL_isv, V1_isv, Q_isv, V2_isv]` covariance matrix.

You can also edit the underlying model or compartment structure, see the *MODEL_PARAMS* or *DERIVATIVES* sections.

4.5.5 Convert Tut to MTut

A *MTut Script* is essentially a *Tut Script* with an extra loop. Therefore it's fairly simple to convert an existing tutorial script to a multi tutorial script. First change the script type from tut->mtut:-

```
METHOD_OPTIONS: {py_module: mtut}
```

Add the mtut *OUTPUT_OPTIONS* section:-

```
OUTPUT_OPTIONS: {n_pop_samples: 30}
```

Change the *OUTPUT_SCRIPTS* section to this:-

```
OUTPUT_SCRIPTS:
    MGEN: {output_mode: run}
    MFIT: {output_mode: run}
    MCOMP: {output_mode: run, dot_size: 12}
```

You should then be able to run your new mtut script.

4.6 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:-

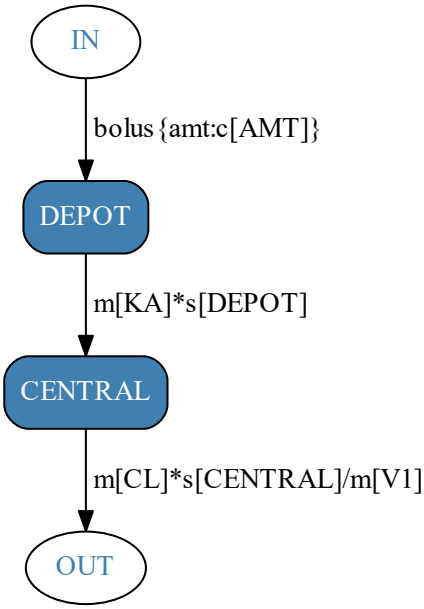


Fig. 4.6: One compartment model with depot dosing used to generate and fit **BLQ** PK data.

Note: See the *Depot + One compartment PK with BLQ* obtained by the PoPy developers for this example, including input script and output data file.

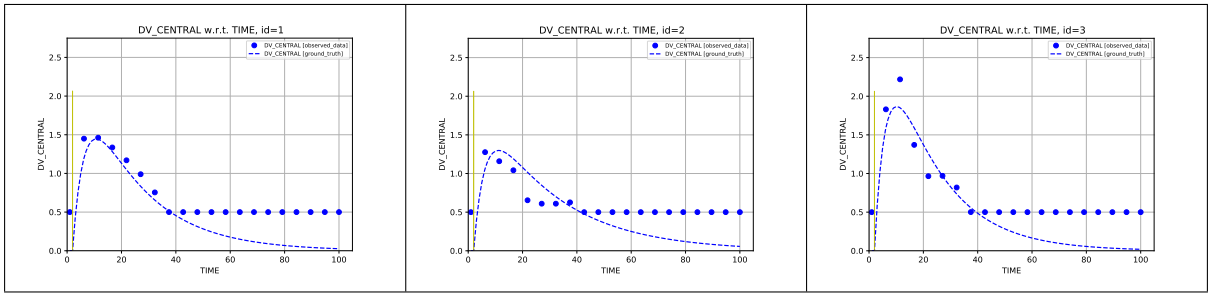
4.6.1 Generating BLQ observations

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 4.12.

Table 4.12: 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].

4.6.2 Fitting BLQ observations using rectnorm (correct error model)

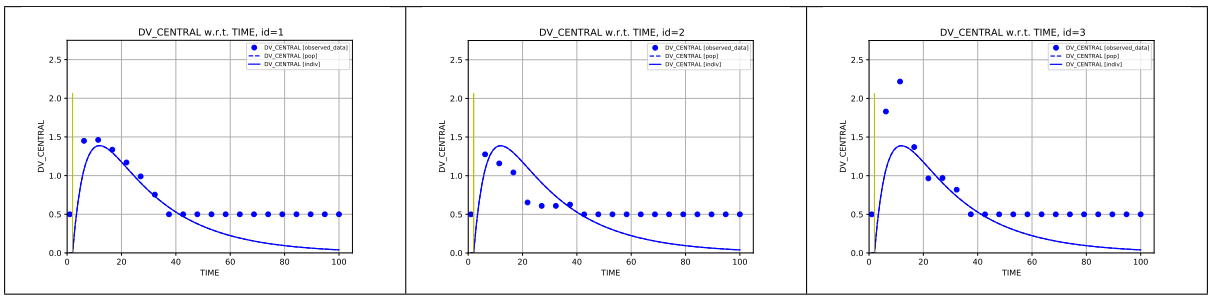
We fit the same (correct) error model, using the same *PREDICTIONS* section as follows:-

```
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 4.13 for PK curves after fitting.

Table 4.13: 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 4.14 and Table 4.15.

Table 4.14: Comparison of initial, fitted and true `f[X]` main values

Name	Initial	Fitted	True	Prop. Error	Abs. Error
f[KA]	1	0.195	0.2	2.28%	4.55e-03
f[CL]	1	2.03	2	1.27%	2.54e-02
f[V1]	20	47	50	6.06%	3.03e+00

Table 4.15: Comparison of initial, fitted and true `f[X]` noise values

Name	Initial	Fitted	True	Prop. Error	Abs. Error
f[PNOISE]	0.1	0.139	0.15	7.41%	1.11e-02

Table 4.14 and Table 4.15 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] = 0.1954
f[CL] = 2.0254
f[V1] = 46.9680
```

```
f[KA_isv,CL_isv,V1_isv] = [
    [ 0.1374, 0.0157, 0.0621 ],
    [ 0.0157, 0.0392, 0.0200 ],
    [ 0.0621, 0.0200, 0.1254 ],
]
f[PNOISE] = 0.1389
f[ANOISE] = 0.0100
```

These fitted values can be compared with fitting alternative error models below.

4.6.3 Fitting BLQ observations using norm (incorrect error model)

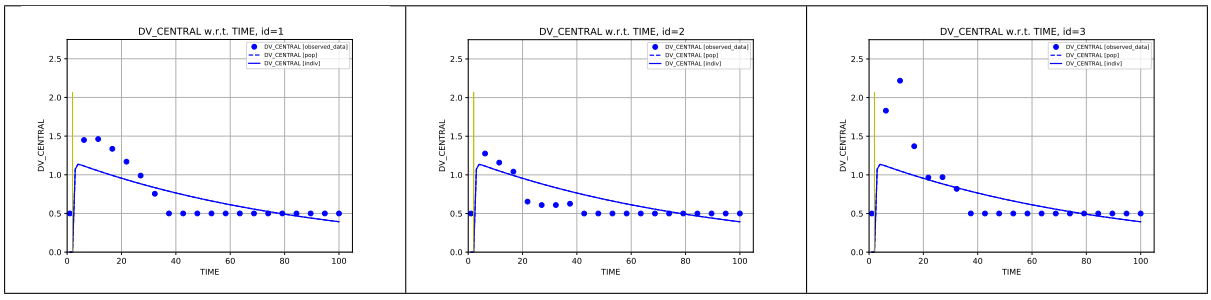
Note: See the *Depot One Comp PK with BLQ observations set to LLQ* script and results used by the PoPy developers for this example.

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 4.16.

Table 4.16: 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] = 2.6741
f[CL] = 0.9560
f[V1] = 86.0580
f[KA_isv,CL_isv,V1_isv] = [
    [ 1.1328, -0.0132, 0.2360 ],
    [ -0.0132, 0.0002, -0.0028 ],
    [ 0.2360, -0.0028, 0.0504 ],
]
f[PNOISE] = 0.2288
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.

4.6.4 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:-

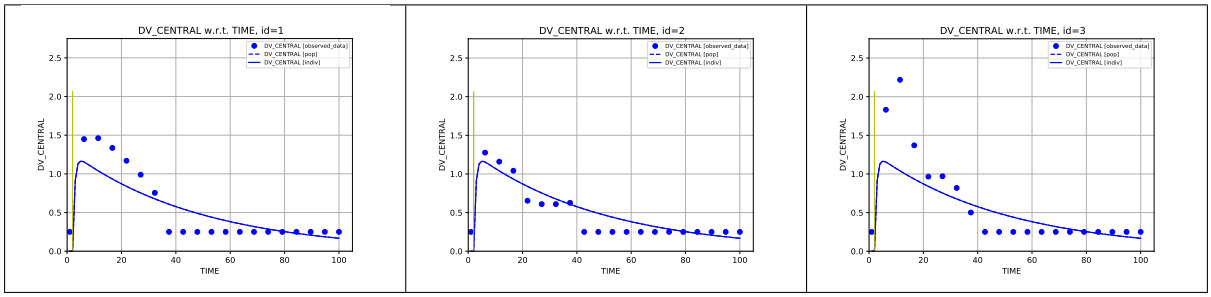
```
PREPROCESS: |
    # use halve value blq data
    if c[DV_CENTRAL] <= 0.5 and c[TYPE] == 'obs':
        c[DV_CENTRAL] = 0.25
```

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 4.17.

Table 4.17: 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] = 1.3656
f[CL] = 1.6576
f[V1] = 80.4195
f[KA_isv,CL_isv,V1_isv] = [
    [ 0.7858, 0.0888, 0.2165 ],
    [ 0.0888, 0.0103, 0.0250 ],
    [ 0.2165, 0.0250, 0.0610 ],
]
f[PNOISE] = 0.3307
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.

4.6.5 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:-

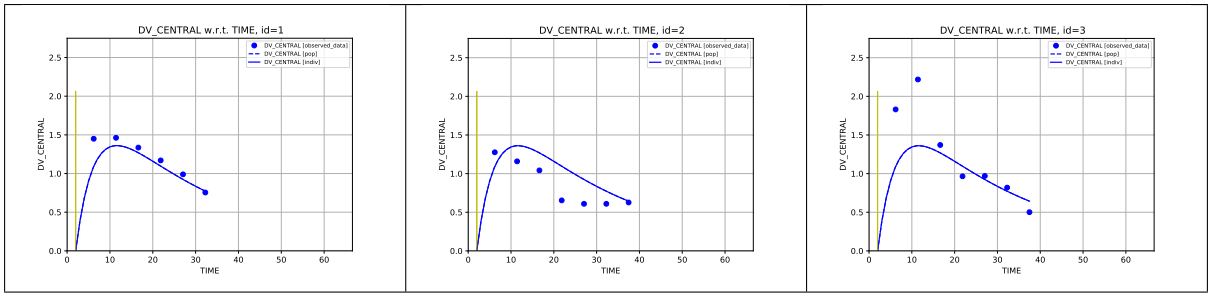
```
PREPROCESS: |
    # ignore blq data
    if c[DV_CENTRAL] <= 0.5 and c[TYPE] == 'obs':
        return
```

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 4.18.

Table 4.18: 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] = 0.2310
f[CL] = 1.8489
f[V1] = 52.3484
f[KA_isv,CL_isv,V1_isv] = [
    [ 0.0000, 0.0002, 0.0004 ],
    [ 0.0002, 0.0400, 0.0068 ],
    [ 0.0004, 0.0068, 0.0991 ],
]
f[PNOISE] = 0.1494
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)*.

CHAPTER

FIVE

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.

5.1 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 5.1 lists the the PoPy data fields for each *Nonmem* data field. Each subsection gives a one to one example conversion to PoPy format.

Table 5.1: 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*.

5.1.1 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 5.2.

Table 5.2: 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.

5.1.2 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 5.3.

Table 5.3: 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.

5.1.3 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 5.4.

Table 5.4: 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.

5.1.4 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 5.5.

Table 5.5: 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.

5.1.5 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.

5.1.6 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 5.6.

Table 5.6: 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 5.6 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.

5.1.7 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 5.7.

Table 5.7: 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.

5.2 Nonmem to PoPy Data conversions using P2NDAT and N2PDAT Scripts

See *Nonmem Data to PoPy Data File* for an overview of how the *Nonmem* data format maps to PoPy format. It is very possible to use this format information to write your own data conversion script in a general purpose programming language, *e.g.* *R* or *Python*.

However, we provide a convenient *N2PDat Script*, to automatically convert from *Nonmem* to PoPy without doing any programming. We actually provide two scripts that are mirror images of each other as follows:-

- *P2NDat Script* - converts from PoPy to *Nonmem* data
- *N2PDat Script* - converts from *Nonmem* to PoPy data

The two types of conversion scripts are illustrated in this section using the following files from the PoPy examples folder:-

```
c:\PoPy\examples\p2ndat_script.pyml
                 n2pdat_script.pyml
                 fit_example1_data.csv
```

Here 'fit_example1_data.csv' is in *PoPy Data Format* and is the simple PK data file discussed in *Fitting a Simple PopPK Model using PoPy*. 'p2ndat_script.pyml' is a PoPy script that will convert the original PoPy 'fit_example1_data.csv' to *Nonmem* format, see *P2NDAT Example*. The 'n2pdat_script.pyml' will convert the newly created *Nonmem* data file back to PoPy format, see *N2PDAT Example*.

The data files in this section form a loop:-

```
fit_example1_data.csv -p2ndat-> fit_example1_nm_data.csv -n2pdat-> fit_
↪example1_data_v2.csv
```

Where 'fit_example1_data.csv' and 'fit_example1_data_v2.csv' are both compatible PoPy data files and 'fit_example1_nm_data.csv' is in *Nonmem* format.

5.2.1 P2NDAT Example

The first few rows of the original 'fit_example1_data.csv' file are shown in Table 5.8.

Table 5.8: Original data in PoPy format (first ten rows)

TYPE	ID	TIME	AMT	DV_CENTRAL	DV_CENTRAL_FLAG
reset	1	0	100	0	0
dose	1	1	100	0	0
obs	1	7.22152181887	100	55.3986503177	1
obs	1	13.7633242874	100	43.5423043551	1
obs	1	19.4607360933	100	24.3960137842	1
obs	1	44.9645896939	100	3.06161955063	1
obs	1	48.3691740856	100	2.84311907493	1
reset	2	0	100	0	0
dose	2	1	100	0	0
obs	2	7.03200507014	100	48.1712193857	1

You can view the example *P2NDat Script*, *Open a PoPy Command Prompt* in this folder:-

```
c:\PoPy\examples\
```

And type:-

```
$ popy_edit p2ndat_script.pyml
```

Then run using:-

```
$ popy_run p2ndat_script.pyml
```

This will create a new *Nonmem* data file 'fit_example1_nm_data.csv'. The first ten rows are shown in Table 5.9.

Table 5.9: Output data in Nonmem format (first ten rows)

TIME	ID	AMT	DV	MDV	EVID	CMT
0	1	0	0	1	3	1
1	1	100	0	1	1	1
7.22152181887	1	0	55.3986503177	0	0	1
13.7633242874	1	0	43.5423043551	0	0	1
19.4607360933	1	0	24.3960137842	0	0	1
44.9645896939	1	0	3.06161955063	0	0	1
48.3691740856	1	0	2.84311907493	0	0	1
0	2	0	0	1	3	1
1	2	100	0	1	1	1
7.03200507014	2	0	48.1712193857	0	0	1

The differences between the input PoPy data Table 5.8 and the output *Nonmem* data Table 5.9. Are summarised in the Table 5.10

Table 5.10: Comparing PoPy 'fit_example1_data.csv' and Nonmem 'fit_example1_nm_data.csv'

Input PoPy column	Output *Nonmem* column	Comments
TYPE	*EVID*	reset->3, dose->1, obs->0
ID	*ID*	no change
TIME	*TIME*	no change
AMT	*AMT*	dose rows no change, obs/reset rows -> 0
DV_CENTRAL	*DV*	no change
DV_CENTRAL_FLAG	*MDV*	1-DV_CENTRAL_FLAG

Note the corresponding columns are not in the same order between 'fit_example1_data.csv' and 'fit_example1_nm_data.csv'. The *P2NDat Script* has removed the 'TYPE', 'DV_CENTRAL' and 'DV_CENTRAL_FLAG' PoPy fields, to leave 'TIME', 'ID' and 'AMT', then added the newly created *Nonmem* specific 'DV', 'MDV', 'EVID' and 'CMT' columns.

The 'fit_example1_nm_data.csv' contains a 'CMT' field to specify that the *Nonmem* dosing occurs in compartment 1. PoPy specifies the dosing compartment entirely in the script file, see *Dosing Fields*, so the output 'CMT' column has no corresponding column in the PoPy data file. You have to specify the 'CMT' value in your *P2NDat Script* manually, see *OUTPUT_NONMEM_FIELDS*.

P2NDAT Script Syntax

You can view the example *P2NDat Script* here:-

```
c:\PoPy\examples\p2ndat_script.pyml
```

Each section is discussed below.

METHOD_OPTIONS

Just specifies the script type:-

```
METHOD_OPTIONS: {py_module: p2ndat}
```

See *METHOD_OPTIONS* for more info.

FILE_PATHS

Just specifies the input PoPy data file and output *Nonmem* data file:-

```
FILE_PATHS:
    input_popy_file: fit_example1_data.csv
    output_nonmem_file: fit_example1_nm_data.csv
```

INPUT_POPY_FIELDS

Describes the columns of the input PoPy file:-

```
INPUT_POPY_FIELDS:
    time_field: TIME
    id_field: ID
    type_field: TYPE
    dv_fields: ['DV_CENTRAL']
    amt_fields: ['AMT']
    rate_fields: []
    dur_fields: []
    dose_labels: ['']
```

Here 'time_field', 'id_field' and 'type_field' are the PoPy data file *Required Fields*. They default to the above values.

The 'dv_fields' is a list of PoPy *Observation Fields* that will be moved into the *Nonmem DV* field. Note you can specify multiple observed columns, each observed field will result in extra rows in the *Nonmem* data output, as *Nonmem* only ever has **one** *DV* observation column.

The 'amt_fields' is a list of PoPy *Dosing Fields*, *i.e.* columns that contain dose amounts. Similar to the 'dv_fields', if you specify multiple dosing amount columns, then the *Nonmem* data output will contain extra rows, as *Nonmem* only has one *AMT* field.

The 'rate_fields' and 'dur_fields' are blank because we only have bolus dosing here. If you have infusion dosing then add the *@inf_rate* and *@inf_dur* rate and duration parameters here.

The 'dose_labels' field contains the dosing names used in the PoPy data file. In this case dose_labels= [''] means PoPy dose names are **not** used. *i.e.* the *TYPE* column just uses 'dose' values. If you use 'dose:my_dose_name', 'dose:my_other_dose_name' in your PoPy data file, to describe *Multiple Dose Types*, then you need to list the names here, *e.g.* ['my_dose_name', 'my_other_dose_name'].

OUTPUT_NONMEM_FIELDS

Describes the columns of the output *Nonmem* file:-

```
OUTPUT_NONMEM_FIELDS:
    comment_prefix: '#'
    column_names: auto
    time_field: TIME
    id_field: ID
    evid_field: EVID
    dv_field: DV
    mdv_field: MDV
    amt_field: AMT
    rate_field: none
    dur_field: none
```

```
    cmt_field: CMT
    obs_cmt_numbers: [1]
    dose_cmt_numbers: [1]
```

Here 'comment_prefix' allows loading of *Nonmem* data files with comment lines. Lines starting with the 'comment_prefix' symbol are ignored.

'column_names: auto', uses the columns names in the '.csv' data file. You could rename them using a list here, a bit like the *Nonmem $INPUT* section.

The 'time_field', 'id_field', 'evid_field', 'dv_field', 'mdv_field', 'rate_field', 'dur_field' and 'cmt_field' allows you to specify the *Nonmem* key fields 'ID', 'EVID', 'DV', 'MDV', 'AMT', 'RATE', 'DUR' and 'CMT'. These fields default to the *Nonmem* key names.

Note that if you do not require some of the *Nonmem* fields, *e.g.* in this case 'rate_field' and 'dur_field', because these only relate to infusion dosing and there is only bolus dosing in this example. Then you can assign null values using 'none'.

The 'obs_cmt_numbers' is a list of compartment indices to appear in the *CMT* column to be created by the *P2NDat Script*. The 'OUTPUT_NONMEM_FIELDS->obs_cmt_numbers' list must be the same length as the 'INPUT_POPY_FIELDS->dv_fields' list. The elements of both lists must correspond to the same type of observation. *e.g.* in this case all PoPy observations 'DV_CENTRAL' occur in *Nonmem* compartment one. The *P2NDat Script* will copy the PoPy 'DV_CENTRAL' value into the *Nonmem DV* column for all rows with *TYPE* ='obs' and set *MDV* =0 for these rows.

The 'dose_cmt_numbers' is a list of compartment indices to appear in the *CMT* column to be created by the *P2NDat Script*. The 'OUTPUT_POPY_FIELDS->dose_cmt_numbers' list must be the same length as the 'INPUT_POPY_FIELDS->amt_fields' list. The elements of both lists must correspond to the same type of dose. *e.g.* in this case all PoPy dose amounts 'AMT' occur in *Nonmem* compartment one. The *P2NDat Script* will copy the PoPy 'AMT' value into the *Nonmem AMT* column for all rows with *TYPE* ='dose' and set *AMT* =0.0 for other rows.

If you have multiple doses and multiple observation fields in your input PoPy data, then you have to specify the dv_fields/obs_cmt_numbers and amt_fields/dose_cmt_numbers list pairs carefully.

OUTPUT_OPTIONS

Describes the output options. Currently, the only option is to remove fields from the final data file:-

```
OUTPUT_OPTIONS:
    drop_fields: ['TYPE', 'DV_CENTRAL', 'DV_CENTRAL_FLAG']
```

Here we are removing the old PoPy fields from the *Nonmem* data output. This is useful in this case, as we wish to demonstrate regenerating the orig PoPy fields, when we use a *N2PDat Script* in the next section.

5.2.2 N2PDAT Example

The *P2NDat Script* converts data from PoPy to *Nonmem* format. Here we discuss the *N2PDat Script* that computes the inverse conversion from *Nonmem* to PoPy format.

Assuming you have run the *N2PDAT Example*, view the example *N2PDat Script* in your text editor, by typing:-

```
$ popy_edit n2pdat_script.pyml
```

Then run the *N2PDat Script* using:-

```
$ popy_run n2pdat_script.pyml
```

This will create a new PoPy data file 'fit_example1_data_v2.csv'. The first ten rows are shown in Table 5.11.

Table 5.11: Output data in Nonmem format (first ten rows)

TIME	ID	AMT	DV_CENTRAL	DV_CENTRAL_FLAG	TYPE
0	1	0	0	0	reset
1	1	100	0	0	dose:_bolus
7.22152181887	1	0	55.3986503177	1	obs
13.7633242874	1	0	43.5423043551	1	obs
19.4607360933	1	0	24.3960137842	1	obs
44.9645896939	1	0	3.06161955063	1	obs
48.3691740856	1	0	2.84311907493	1	obs
0	2	0	0	0	reset
1	2	100	0	0	dose:_bolus
7.03200507014	2	0	48.1712193857	1	obs

The differences between the input *Nonmem* data Table 5.9 and the output PoPy data Table 5.11. Are summarised in the Table 5.12

Table 5.12: Comparing Nonmem 'fit_example1_nm_data.csv' and PoPy 'fit_example1_data_v2.csv'

Input *Nonmem* column	Output PoPy column	Comments
TIME	*TIME*	no change
ID	*ID*	no change
AMT	AMT	no change
DV	DV_CENTRAL	no change
MDV	DV_CENTRAL_FLAG	1-MDV
EVID	*TYPE*	3->reset,1->dose:_bolus,0->obs
CMT	N/A	PoPy has no 'CMT' equivalent

The *N2PDat Script* has removed 'DV', 'MDV', 'EVID' and 'CMT' *Nonmem* fields from 'fit_example1_nm_data.csv' and replaced them with 'TYPE', 'DV_CENTRAL' and 'DV_CENTRAL_FLAG' PoPy fields in 'fit_example1_data_v2.csv'.

The 'fit_example1_data_v2.csv' contains **no** 'CMT' field because PoPy specifies the dosing compartment entirely in the script file, see *Dosing Fields*.

N2PDAT Script Syntax

You can view the example *N2PDat Script* here:-

```
c:\PoPy\examples\n2pdat_script.pyml
```

Each section is discussed below.

METHOD_OPTIONS

Specifies the script type:-

```
METHOD_OPTIONS: {py_module: n2pdat}
```

See *METHOD_OPTIONS* for more information.

FILE_PATHS

Specifies the input *Nonmem* data file and output PoPy data file:-

```
FILE_PATHS:
    input_nonmem_file: fit_example1_nm_data.csv
    output_popy_file: fit_example1_data_v2.csv
```

INPUT_NONMEM_FIELDS

Describes the columns of the input *Nonmem* file:-

```
INPUT_NONMEM_FIELDS:
    comment_prefix: '#'
    column_names: auto
    date_field: none
    date_format: none
    time_field: TIME
    id_field: ID
    evid_field: EVID
    dv_field: DV
    mdv_field: MDV
    amt_field: AMT
    rate_field: none
    dur_field: none
    cmt_field: CMT
    obs_cmt_numbers: [1]
    dose_cmt_numbers: [1]
```

This is the same as the *OUTPUT_NONMEM_FIELDS* section. The only difference is that this section is now describing an **input** *Nonmem* data file instead of an **output** *Nonmem* data file.

The 'obs_cmt_numbers' and 'dose_cmt_numbers' list have to correspond to the 'dv_fields' and 'amt_fields' in the *OUTPUT_POPY_FIELDS* section to get sensible PoPy data output. See below for more explanation.

OUTPUT_POPY_FIELDS

Describes the columns of the output PoPy file:-

```
OUTPUT_POPY_FIELDS:
    time_field: TIME
    id_field: ID
    type_field: TYPE
    dv_fields: ['DV_CENTRAL']
    amt_fields: ['AMT']
    rate_fields: []
    dur_fields: []
    dose_labels: ['']
```

This is the same as the *INPUT_POPY_FIELDS* section. The only difference is that this section is now describing an **output** PoPy data file instead of an **input** PoPy data file.

Here the 'dv_fields' is a list of PoPy observation columns to be created by the *N2PDat Script*, based on the input *Nonmem DV* field. The 'OUTPUT_POPY_FIELDS->dv_fields' list must be the same length as the 'INPUT_NONMEM_FIELDS->obs_cmt_numbers' list. The elements of both lists must correspond to the same type of observation. *e.g.* in this case all *Nonmem* observations occur in compartment one, so for *Nonmem* data rows with *EVID* =0 and *CMT* =1 the *Nonmem DV* value is copied into the PoPy DV_CENTRAL column with DV_CENTRAL_FLAG=1.

The 'amt_fields' is a list of PoPy dose amount columns to be created by the *N2PDat Script*, based on the input *Nonmem AMT* field. The 'OUTPUT_POPY_FIELDS->amt_fields' list must be the same length as the 'INPUT_NONMEM_FIELDS->dose_cmt_numbers' list. The elements of both list must correspond to the same type of dose. *e.g.* in this case all *Nonmem* doses occur in compartment one, so for *Nonmem* data rows with *EVID* =1 and *CMT* =1 the *Nonmem AMT* value is copied into the PoPy AMT column, with all other rows set to zero.

If you have multiple doses and multiple observation fields in your input *Nonmem* data, then you have to specify the obs_cmt_numbers/dv_fields and dose_cmt_numbers/amt_fields list pairs carefully.

OUTPUT_OPTIONS

Describes the output options, currently, just which fields to remove:-

```
OUTPUT_OPTIONS:
    drop_fields: ['DV', 'MDV', 'EVID', 'CMT']
```

Here we are removing the *Nonmem* specific fields. In a real life conversion it may be sensible to keep the *Nonmem* fields, so that you can perform a side by side sanity check from within the PoPy output file. Note the fields above will be of little use to a PoPy *Fit Script*, compared to the 'DV_CENTRAL', 'DV_CENTRAL_FLAG' and 'TYPE' fields, created by the *N2PDat Script*.

5.2.3 Compare original PoPy data with P2NDAT/N2PDAT version

In this walk through we have taken a PoPy data file 'fit_example1_data.csv', run *P2NDat Script* to create a *Nonmem* data version. Then we ran *N2PDat Script* to re-create the original PoPy data file 'fit_example1_data_v2.csv' from the *Nonmem* data.

You can compare the first 10 rows of both the input PoPy data set in Table 5.8 and the output PoPy data in Table 5.11.

Both files contain the same column headers *i.e.* 'TYPE', 'ID', 'TIME', 'AMT', 'DV_CENTRAL', 'DV_CENTRAL_FLAG'. The values in each column are the same apart from 'AMT' column has zero values in non-dose rows. Also the 'dose' value in the *TYPE* field is now 'dose:_bolus'. Both the input and output .csv files are valid PoPy data formats for the PK/PD problem described in *Fitting a Simple PopPK Model using PoPy*.

5.3 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 5.13 lists the equivalent script file sections.

Table 5.13: 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

5.3.1 $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.

5.3.2 $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*.

5.3.3 $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.

5.3.4 $SUBROUTINES

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.

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.

5.3.5 $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!

5.3.6 $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.

5.3.7 $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])
```

5.3.8 $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.

5.3.9 $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.

5.3.10 $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.

5.3.11 $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).

5.3.12 $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.

5.4 Nonmem Advan1,2,3,4,11,12 to PoPy analytic compartment models

Table 5.14 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 5.14: 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

5.4.1 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*.

5.4.2 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*.

5.4.3 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*.

5.4.4 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*.

5.4.5 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*.

5.4.6 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*.

5.4.7 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*.

5.4.8 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*.

5.4.9 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*.

5.4.10 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*.

5.4.11 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*.

5.4.12 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*.

5.5 DDMoRe0061 Conversion Example

5.5.1 Overview

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

5.5.2 Data Conversion

We describe the conversion of this data set in *Nonmem* format:-

```
c:\PoPy\validation\ddmore0061\Simulated_gentamicin_pk.csv
```

To this data set in PoPy format:-

```
c:\PoPy\validation\ddmore0061\popy\popy_data.csv
```

Using the *N2PDat Script* located here:-

```
c:\PoPy\validation\ddmore0061\popy\n2p_script.pyml
```

The original *Nonmem* data set looks something like Table 5.15.

Table 5.15: 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

The critical entries for the *N2PDat Script* are as follows:-

```
INPUT_NONMEM_FIELDS:
     obs_cmt_numbers: [1]
     dose_cmt_numbers: [1]
```

These two entries specify that the observations and doses all occur in compartment one. Technically the *Nonmem* data set above has no CMT field. However CMT defaults to one in *Nonmem* if it is NOT provided.

The dataset output by *N2PDat Script* is the same as the input, but with the 3 extra columns shown in Table 5.16.

Table 5.16: 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 the *Nonmem* 'MDV' flag. The new 'TYPE' field is formatted to process either *bolus* or *infusion* doses in PoPy.

5.5.3 Script Conversion

We describe the manual conversion of the *Nonmem* script file:-

```
c:\PoPy\validation\ddmore0061\Executable_gentamicin_pk.mod
```

To create the PoPy *Fit Script* here:-

```
c:\PoPy\validation\ddmore0061\popy\fit_script.pyml
```

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: popy_data.csv}
```

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 to be clumsily 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 sigmas are *fixed effects* and also defined in *EFFECTS*.

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 the PoPy *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
    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.

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.

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. It is not clear (to the PoPy developers) 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.

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.

5.6 DDMoRe0093 Conversion Example

5.6.1 Overview

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

5.6.2 Data Conversion

We describe the conversion of this data set in *Nonmem* format:-

```
c:\PoPy\validation\ddmore0093\Simulated_dataset.csv
```

To this data set in PoPy format:-

```
c:\PoPy\validation\ddmore0093\popy\popy_data.csv
```

Using the *N2PDat Script* located here:-

```
c:\PoPy\validation\ddmore0093\popy\n2p_script.pyml
```

The final data set looks something like Table 5.17.

Table 5.17: Main data fields for Nonmem and PoPy

ID	TIME	AMT	EVID	QTCF	MDV	CPP	TYPE
1001	0	0	2	0	1	0	pred
1001	31.2575	0	0	351.7	0	0	obs
1001	31.2655556	0	0	343.7	0	0	obs
1001	31.2736111	0	0	354.1	0	0	obs
1001	31.4072222	0	0	351.5	0	0	obs
1001	31.4152778	0	0	359	0	0	obs
1001	33.6672222	600	1	0	1	0	dose:dose_bolus
1001	34.1683333	0	2	0	1	1083	pred
1001	34.5841667	0	0	353.6	0	1447	obs

Note the main output of the data conversion is the 'TYPE' field which has the correct entries for PoPy corresponding to the *Nonmem* 'EVID' field. as follows:-

```
2 -> pred
0 -> obs
1 -> dose:dose_bolus
```

Note however that in this model the dosing is not actually used in the *Nonmem* control file.

5.6.3 Script Conversion

We describe the manual conversion of the *Nonmem* script file:-

```
c:\PoPy\validation\ddmore0093\Executable_run7b.ctl
```

To create the PoPy *Fit Script* here:-

```
c:\PoPy\validation\ddmore0093\popy\fit_script.pyml
```

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.

Whereas the PoPy script uses this syntax:-

```
FILE_PATHS: {input_data_file: output_popy_data.csv}
PREPROCESS: |
    if c[TYPE] == 'dose:dose_bolus': return
```

Which loads the PoPy data and also excludes the dosing rows.

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: |
        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
        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 ]
        ]
        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]
        )
```

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:-

```
$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)

    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*.

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.

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}]
```

5.7 DDMoRe0238 Conversion Example

5.7.1 Overview

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

5.7.2 Data Conversion

We describe the conversion of this data set in *Nonmem* format:-

```
c:\PoPy\validation\ddmore0238\Simulated_simdataDDM.csv
```

To this data set in PoPy format:-

```
c:\PoPy\validation\ddmore0238\popy\output_popy_data.csv
```

Using the *N2PDat Script* located here:-

```
c:\PoPy\validation\ddmore0238\popy\n2p_script.pyml
```

The original *Nonmem* data set looks something like Table 5.18.

Table 5.18: Main data fields for Nonmem (first 8 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

The final data set looks something like Table 5.19.

Table 5.19: Extra data fields for PoPy (first 8 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

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'.

5.7.3 Script Conversion

We describe the manual conversion of the *Nonmem* script file:-

```
c:\PoPy\validation\ddmore0238\Executable_run35b_ddm2.ctl
```

To create the PoPy *Fit Script* here:-

```
c:\PoPy\validation\ddmore0238\popy\fit_script.pyml
```

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: output_popy_data.csv
```

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
$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])
        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
```

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
    ;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
    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
    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.

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*.

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)
    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
```

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)
```

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.

CHAPTER

SIX

POPY REFERENCE GUIDE

6.1 Open a PoPy Command Prompt

To gain access to the *Command Line Tools* you need to invoke the PoPy command prompt environment.

There are three main ways of starting a PoPy command prompt:-

- *Desktop Shortcut Method*
- *Terminal popy_env Method*
- *Copy popy_cmd.exe Method*

To check that you are currently within a PoPy environment, see *Verify PoPy Environment*.

6.1.1 Desktop Shortcut Method

Left mouse click on the 'PoPy Command Prompt' shortcut on your Desktop or alternatively within the Start Menu (in the PoPy folder). This will open a PoPy command prompt in the directory where you installed PoPy.

For example, if you click the PoPy shortcut and installed PoPy to the directory:-

```
c:\PoPy
```

You should see something like Fig. 6.1:-

Fig. 6.1: PoPy dos prompt

6.1.2 Terminal popy_env Method

Open a command prompt in **any** folder on your computer. It is highly recommended that you use the traditional 'cmd' dos prompt, in preference to the Windows Powershell.

A simple way of opening a command prompt in a specific folder is to:-

- navigate to the folder in Windows Explorer
- left mouse click on the path dialog box
- type 'cmd'
- press return

For example if you navigate to the directory:-

```
c:\Users\david\
```

Then type 'cmd' over the Windows Explorer path, a plain dos command prompt should appear like this Fig. 6.2:-

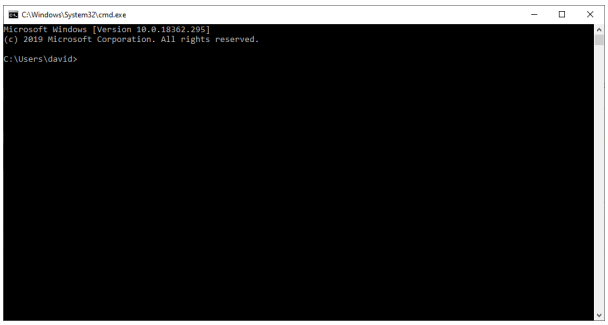


Fig. 6.2: Plain command prompt

Then type:-

```
$ popy_env
```

You should see the text:-

```
Welcome to PoPy Binary
```

After running *popy_env* the terminal should look something like Fig. 6.3:-

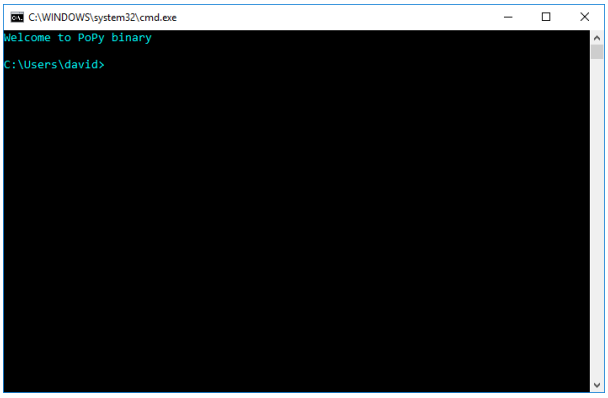


Fig. 6.3: PoPy command prompt

Note the colour of the dos prompt text will change from white to pale blue.

6.1.3 Copy popy_cmd.exe Method

An alternative method for starting PoPy in any folder is to simply copy the file 'popy_cmd.exe' from your PoPy install directory to a new location.

Then simply clicking on 'popy_cmd.exe' will start a PoPy environment in the same directory.

For example copy this file:-

```
c:\PoPy\popy_cmd.exe
```

to the directory:-

```
c:\Users\david\my_work\
```

Click on 'popy_cmd.exe' within the 'my_work' directory and you should see Fig. 6.4:-

Fig. 6.4: PoPy command prompt in the 'c:\Users\david\my_work' folder.

6.1.4 Verify PoPy Environment

The light blue text signifies that you are within a PoPy environment. You can verify this further by running *popy_info*:-

```
$ popy_info
```

You can also verify that the system path has changed. e.g in a command prompt:-

```
$ echo %PATH%
```

The system path should start with:-

```
c:\PoPy\bin
```

Or wherever PoPy is installed.

You should be able to run all the *Command Line Tools* now.

6.2 PoPy Activation

PoPy comes with a 60 day trial period. To continue using PoPy after this period has expired, you will need a *licence* to activate the product.

You should have been supplied with a free activation code if you signed up for the PoPy academic version.

To obtain a valid PoPy licence see *Licensing*.

6.2.1 Activation Status

To check the current activation status and licencing of your PoPy installation, *Open a PoPy Command Prompt* and type:-

```
popy_info
```

If you see output like this:-

```
INFO - product_key=None
```

Then PoPy is not yet activated. The trial period may or may not be active, which determines if PoPy will run without a *product key*. If you see the following:-

```
INFO - should_run=True
```

then PoPy is within the trial period.

If you have a *product key*, but this line is displayed:-

```
INFO - should_run=False
```

Then the *product key* is invalid and you need to obtain a new *product key* to successfully *Activate PoPy*.

6.2.2 Activate PoPy

To activate PoPy *Open a PoPy Command Prompt* and type:-

```
popy_activate XXXX-XXXX-XXXX-XXXX-XXXX-XXXX-XXXX
```

Where XXXX *etc.*, is the *product key* supplied by Wright Dose Ltd. Just cut and paste the *product key* from the email.

Verify Activation

Run:-

```
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
INFO - licence_end_date=2024-02-16 00:00:00
INFO - should_run=True
```

showing the paths used by PoPy, a few internal variables, and details about the licence (if you have run *popy_activate* successfully).

6.2.3 Deactivate PoPy

To remove a PoPy *product key* from your machine *Open a PoPy Command Prompt* and type:-

```
popy_deactivate
```

Verify Deactivation

Run:-

```
popy_info
```

and you should see something like this:-

```
INFO - product_key=None
```

6.2.4 Transfer Product Key Between Machines

If you wish to move a PoPy licence from one computer to another. Then you simply need to *Deactivate PoPy* on the current machine, noting the current *product key*. And then *Activate PoPy* on the new machine, using the same *product key*.

Note if you activate the same *product key* on multiple machines and you exceed the number of activations associated with the *product key*, this will invalidate your licence and may cause problems.

6.2.5 Licensing

There are currently two licences for PoPy:-

PoPy *Academic* Licence A free licence for PoPy distributed to academics, which may **not** be used for commercial analyses.

PoPy *Commercial* Licence A paid licence granting the end user one year of PoPy usage on a single machine. A commercial license is required to use PoPy in regulator submissions or publications regarding work that is sponsored or funded by drug companies *i.e.* research that leads to profit.

Both licences comes with a 60 day trial period, so you can try out PoPy before obtaining a *product key*.

All versions of PoPy have the same functionality. There are no *crippled* versions. Any version of PoPy can be upgraded to the latest version of PoPy at any time. The terms of the licence (*i.e.* duration *etc.*) are unaffected by upgrading PoPy.

Please email the following address for further information:-

info@popypkpd.com

We will endeavour to respond promptly to questions regarding our licencing.

6.3 Configure PoPy

6.3.1 PoPy Config File

The local settings of your PoPy installation can be altered by editing this file:-

```
C:\popy\popy_config.pyml
```

We recommend that you do not alter this file unless you know exactly what you are doing.

6.3.2 Configuration Options

You should probably restrict yourself to editing the following two paths:-

```
# path to text editor invoked using popy_edit
text_editor_path: "C:/Program Files/Notepad++/notepad++.exe"

# path to Inkscape binary invoked using popy_imconv
inkscape_exe_path: "C:/Program Files/Inkscape/inkscape.exe"
```

You should change these paths if you have installed *Notepad++* or *Inkscape* to non-standard locations.

6.3.3 Factory Reset Options

If you edit your 'popy_config.pyml' and PoPy no longer functions correctly, please restore to the following default settings:-

```
# path to favicon
favicon_path: ${POPY_PYTHON_PATH}/conf/4p_white_border.ico

# path to graphviz dot.exe for creating diagrams
graphviz_dot_path: ${POPY_PYTHON_PATH}/thirdparty/graphviz/bin/dot.exe

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

# path to memcached executable
memcached_path: ${POPY_PYTHON_PATH}/thirdparty/memcached-amd64/memcached.exe

# path rabbitmq sbin folder
rabbitmq_sbin_folder:␣
↪${POPY_PYTHON_PATH}/thirdparty/RabbitMQ Server/rabbitmq_server-3.5.1/sbin

# path to text editor invoked using popy_edit
text_editor_path: "C:/Program Files/Notepad++/notepad++.exe"

# path to Inkscape binary invoked using popy_imconv
inkscape_exe_path: "C:/Program Files/Inkscape/inkscape.exe"
```

6.3.4 Inkscape

We recommend Inkscape, an open source graphics tool, for converting between image file formats.

Install Inkscape

Download it from:-

https://inkscape.org/en/release/

If you install the 64bit binary installer the default install directory is:-

```
C:\Program Files\Inkscape
```

Configure Inkscape Path

Note that to use Inkscape with *popy_imconv* you need to make sure *PoPy Config File* contains this entry:-

```
inkscape_exe_path: "C:/Program Files/Inkscape/inkscape.exe"
```

With this path set correctly you can now *Open a PoPy Command Prompt* and do:-

```
$ popy_imconv *.svg png
```

To convert .svg images to .png format. See *popy_imconv*.

6.3.5 Uninstall PoPy

Note that if you are moving your *product key* to another machine, it is wise to *Deactivate PoPy* before uninstalling. If you are upgrading PoPy then it is best to keep the current product key.

Run the uninstaller

You can click on the shortcut at:-

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

Or you could, instead, run the .exe at:-

```
C:\PoPy\popy_uninstall.exe
```

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

6.4 PoPy Website

6.4.1 PoPy introductory site

We have an introductory site here:-

https://popypkpd.com/

This website describes the main features of PoPy and why you should use it.

6.4.2 PoPy demo site

We have a site where you can try Popy out online here:-

https://demo.popypkpd.com/

This site provides an easy-to-use interface to create and run Popy scripts for popular model types.

6.4.3 PoPy product website

The main website to support PoPy is here:-

https://product.popypkpd.com/

This site contains the binary downloads and up-to-date documentation for PoPy.

6.4.4 Obtain a website account

If you want to try out PoPy please contact:-

`info@popypkpd.com`

Your website account consists of an email address and password, and will permit access to the latest binary installer and up-to-date PoPy documentation.

6.4.5 Website Structure

Overview

Introduction to PoPy PK/PD system.

News

Recent news about PoPy. E.g. up-coming conferences + new releases/features.

Downloads

Access to all PoPy binaries. Note we will keep all our release binaries online indefinitely to support reproducibility of your results.

However you are encouraged to update to the latest version to get the most out of PoPy.

Documentation

Documentation for all releases of PoPy. Note the online documentation (for the latest release) will be updated continuously. So the online documentation will be more up-to-date than the static html supplied with the PoPy binary.

To access the online documentation for your version of PoPy simply type:-

```
popy_doc -o
```

Where the "o" stands for online. If you type:-

```
popy_doc
```

You will see the local documentation under:-

```
c:\PoPy\html
```

About

Some information about the developers and contact email.

6.5 Validate PoPy

Before using PoPy for an important analysis it is best to run the validation process. This verifies that the outputs of your PoPy installation agree with the outputs obtained by the developers of PoPy.

6.5.1 Check the PoPy Configuration

To see the current configuration, *Open a PoPy Command Prompt* 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
INFO - licence_end_date=2024-02-16 00:00:00
INFO - should_run=True
```

showing the paths used by PoPy, a few internal variables, and details about the licence (if you have run *popy_activate*).

6.5.2 Run the Validator

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 you compute locally to some reference results generated at Wright Dose Ltd.

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

To run this suite of examples *Open a PoPy Command Prompt* and type:-

```
$ popy_validate
```

Some of the examples take a few minutes to run, the full validation takes about 20 minutes, 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
```

which will also be written to the log file 'validation_script.pyml.run.main.log' in the 'validation' subdirectory of your PoPy installation, which is normally located here:-

```
c:\PoPy\validation\validation_script.pyml.run.main.log
```

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.

6.5.3 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 here:-

```
c:\PoPy\validation
```

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 6.1 lists each of the models in this folder:-

Table 6.1: 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]*

6.5.4 Troubleshoot

If the validation fails for any reason, please contact the PoPy developers at info@popypkpd.com.

6.6 Command Line Tools

The PoPy tools and their functions are summarised in Table 6.2:-

Table 6.2: PoPy Command Line Tools

Name	Example	Function
popy_env	popy_env	invoke the PoPy environment
popy_run	popy_run my_script.pyml	run a PoPy script
popy_check	popy_check my_script.pyml	check a PoPy script
popy_create	popy_create fit my_script.pyml	create an example PoPy script
popy_format	popy_format my_script.pyml	update PoPy script format
popy_edit	popy_edit my_script.pyml	open PoPy script in editor
popy_doc	popy_doc	open PoPy documentation in browser
popy_view	popy_view my_script.pyml.html	open PoPy html output in browser
popy_info	popy_info	display PoPy install details
popy_validate	popy_validate	run *validation*
popy_activate	popy_activate XXXX	*Activate PoPy* using a *product key*
popy_deactivate	popy_deactivate	*Deactivate PoPy* a *product key*
popy_imconv	popy_imconv *.svg png	convert images using *Inkscape*

6.6.1 popy_env

For all the *Command Line Tools* to work, we must first update the internal paths on your computer and set some environment variables. To do this, we provide a **popy_env** tool that you run from a command prompt (or Powershell):-

```
$ popy_env
```

If the command is successful, the text in the command prompt should change from white to light blue (if in a *dos prompt*).

You can achieve the same thing by opening the *Open a PoPy Command Prompt* from your desktop or the Start Menu - the shortcut starts a new DOS prompt then automatically calls 'popy_env'.

The settings initialised by **popy_env** are summarised by the *popy_info* tool.

6.6.2 popy_run

The most important command in the suite of *Command Line Tools* is **popy_run**. It processes PoPy scripts.

Running a script

To run a PoPy script *Open a PoPy Command Prompt* 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
- *n2pdat* - converts *Nonmem* to PoPy data
- *p2ndat* - converts PoPy to *Nonmem* data

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*.

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.

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.

Command line options

```
usage: popy_run [-h] [-a] [-c] [-f] [-i] [-l] [-m] [-o] [-r] [-s] [-t]
                [-v {noset,info,debug,warning,error,critical}]
                input_file

Runs a PoPy script

positional arguments:
  input_file            Required path to input configuration file.

optional arguments:
  -h, --help            show this help message and exit
  -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.
  -f, --format_on_fly   Optionally attempt to format the input config file
                        during run. And switch to the new formatted verison of
                        the file. Note this re-formatting is only useful if
                        the input config is a valid version of an older PoPy
                        format.
  -i, --i_am_feeling_lucky
                        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
                        your script regardless.
  -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.
  -m, --manual_mode     Optionally do NOT run output scripts automatically,
                        even if 'output_mode: run' set in config file.
                        Effectively uses 'output_mode: create' instead. Then
                        user has to run output scripts manually.
  -o, --overwrite       Optionally overwrite existing output files without
                        asking.
  -r, --replicate_scripts
                        Optionally replicate input config files in log files.
  -s, --spaces          Optionally add more spaces to the output config file
                        for greater clarity, but longer config files. Off by
                        default.
  -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 file
```

6.6.3 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.

Checking a script

Check a script by opening a *Open a PoPy Command Prompt* 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*.

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.

Log files created by popy_check

When **popy_check** processes a script file, it creates a log file:-

```
my_script.pyml.check.log
```

Hopefully the various error and warning messages will enable you to more easily edit and fix a broken script file.

Command line options

```
usage: popy_check [-h] [-o] input_file

Checks PoPy script for consistency and returns errors + warnings. Can use
optionally prior to running script.

positional arguments:
  input_file       Required path to input configuration file.

optional arguments:
  -h, --help       show this help message and exit
  -o, --overwrite  Optionally overwrite existing output files without asking.
```

6.6.4 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 *Open a PoPy Command Prompt* 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 *popy_edit* to suit current requirements or immediately run using *popy_run*. See *Creating Example Scripts* for more information.

Command line options

```
usage: popy_create [-h] [-a] [-c] [-l] [-o] [-s]
                   [-v {noset,info,debug,warning,error,critical}]
                   {fit,fitsum,gen,gensum,grph,
↪n2pdat,p2ndat,msim,rst,sim,sumdoc,tut,tutsum,val,vpc,mfit,mgen,mtut,mcomp}
                   output_file

Creates an example PoPy script

positional arguments:
  {fit,fitsum,gen,gensum,grph,
↪n2pdat,p2ndat,msim,rst,sim,sumdoc,tut,tutsum,val,vpc,mfit,mgen,mtut,mcomp}
                        type of script to create
  output_file           path to output configuration file

optional arguments:
  -h, --help            show this help message and exit
  -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.
  -o, --overwrite       Optionally overwrite existing output files without
                        asking.
  -s, --spaces          Optionally add more spaces to the output config file
                        for greater clarity, but longer config files. Off by
                        default.
  -v {noset,info,debug,warning,
↪error,critical}, --verbosity {noset,info,debug,warning,error,critical}
                        verbosity of output in log file
```

6.6.5 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.

Format a script

Do this by opening a *Open a PoPy Command Prompt* 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.

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.

Command line options

```
usage: popy_format [-h]
                   [-f {no_change,fit,fitsum,gen,gensum,grph,
↪n2pdat,p2ndat,msim,rst,sim,sumdoc,tut,tutsum,val,vpc,mfit,mgen,mtut,mcomp}]
                   [-i] [-d] [-a] [-c] [-l] [-o] [-s]
                   [-v {noset,info,debug,warning,error,critical}]
                   input_file

Tries to reformat an old version of a PoPy script

positional arguments:
  input_file            Required path to input configuration file.

optional arguments:
  -h, --help            show this help message and exit
  -f {no_change,fit,
↪fitsum,gen,gensum,grph,n2pdat,p2ndat,msim,rst,sim,sumdoc,tut,tutsum,val,vpc,
↪mfit,mgen,mtut,mcomp}, --force_type {no_change,fit,fitsum,gen,gensum,grph,
↪n2pdat,p2ndat,msim,rst,sim,sumdoc,tut,tutsum,val,vpc,mfit,mgen,mtut,mcomp}
                        Optionally force the input config file to be a certain
                        type of script, by default no change.
  -i, --inplace         Optionally overwrite original file, i.e fix in place.
  -d, --delete          Optionally delete extra files, e.g. log + backup file
                        etc.
  -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.
  -o, --overwrite       Optionally overwrite existing output files without
                        asking.
  -s, --spaces          Optionally add more spaces to the output config file
                        for greater clarity, but longer config files. Off by
                        default.
  -v {noset,info,debug,warning,
↪error,critical}, --verbosity {noset,info,debug,warning,error,critical}
                        verbosity of output in log file
```

6.6.6 popy_edit

Opens a text file in an editor from the command line.

Edit a script

To edit a script in a text editor, *Open a PoPy Command Prompt* and call:-

```
$ popy_edit my_script.pyml
```

The editor invoked by **popy_edit** is defined in the *PoPy Config File* file.

Edit multiple scripts

Note you can also open all scripts in a directory using:-

```
$ popy_edit *.pyml
```

This opens all files with the '.pyml' extension in your default editor.

Specify text editor

By default this uses the entry **text_editor_path** in this file:-

```
c:\PoPy\popy_config.pyml
```

If the default editor does not exist then the chosen editor is the first binary found in this list:-

- C:\Program Files\Notepad++\notepad++.exe
- C:\Program Files (x86)\Notepad++\notepad++.exe
- C:\Windows\notepad.exe

Also see *Configure Editor*.

Command line options

```
usage: popy_edit [-h] input_file

Opens a PoPy .pyml script file with an appropriate editor. By default uses
editor specified in $POPY_PYTHON_PATH\popy_config.pyml. Alternatively will
look in the following paths:- C:\Program
Files\Notepad++\notepad++.exe,C:\Program Files
(x86)\Notepad++\notepad++.exe,C:\Windows\notepad.exe

positional arguments:
  input_file  Required path to input configuration file.

optional arguments:
  -h, --help  show this help message and exit
```

6.6.7 popy_doc

For convenient access to PoPy's documentation, *Open a PoPy Command Prompt* and type:-

```
$ popy_doc
```

This opens the *The PoPy Manual HTML* documentation page.

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
```

Command line options

```
usage: popy_doc [-h] [-o]
                [-s {none,fit,fitsum,gen,gensum,grph,
↪n2pdat,p2ndat,msim,rst,sim,sumdoc,tut,tutsum,val,vpc,mfit,mgen,mtut,mcomp}]
↪       [-t {none,activate,create,check,deactivate,doc,edit,env,fix,info,run}]

Opens PoPy html documentation in web browser.

optional arguments:
  -h, --help            show this help message and exit
  -o, --online          Optionally access online documentation instead of
                        local.
  -s {none,
↪fit,fitsum,gen,gensum,grph,n2pdat,p2ndat,msim,rst,sim,sumdoc,tut,tutsum,
↪val,vpc,mfit,mgen,mtut,mcomp}, --script {none,fit,fitsum,gen,gensum,grph,
↪n2pdat,p2ndat,msim,rst,sim,sumdoc,tut,tutsum,val,vpc,mfit,mgen,mtut,mcomp}
```

```
                        Optionally open documentation for a specific script
                        type.
  -t {none,activate,create,check,deactivate,doc,edit,env,fix,info,run}
↪, --tool {none,activate,create,check,deactivate,doc,edit,env,fix,info,run}
                        Optionally open documentation for a specific PoPy tool
```

6.6.8 popy_view

Opens a html file in a browser from the command line.

Open a html file

For convenient access to PoPy's html output, *Open a PoPy Command Prompt* 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.

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.

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.

optional arguments:
  -h, --help  show this help message and exit
```

6.6.9 popy_info

A handy tool, **`popy_info`**, displays information about the installation so that you can check at a glance that the installation worked.

Open a PoPy Command Prompt 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
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 *PoPy Activation* status of PoPy on your machine.

6.6.10 popy_validate

A tool to *Validate PoPy*. To run **popy_validate**, *Open a PoPy Command Prompt* and type:-

```
$ popy_validate
```

See *Validate PoPy* for more information.

6.6.11 popy_activate

A tool to *Activate PoPy*. To run **popy_activate**, *Open a PoPy Command Prompt* and type:-

```
$ popy_activate [product key]
```

where **[product key]** is the *product key* supplied by Wright Dose Ltd.

See *Activate PoPy* for more information.

Command line options

```
usage: popy_activate [-h] product_key

Activates a PoPy install using a product key.

positional arguments:
  product_key  A bare product key, cut and paste to the command line.

optional arguments:
  -h, --help   show this help message and exit
```

6.6.12 popy_deactivate

A tool to *Deactivate PoPy*. To run **popy_deactivate**, *Open a PoPy Command Prompt* and type:-

```
$ popy_deactivate
```

See *Deactivate PoPy* for more information.

6.6.13 popy_imconv

A convenient multi-image conversion tool. *Open a PoPy Command Prompt* and type:-

```
$ popy_imconv *.svg png
```

This relies on *Inkscape* to convert all images on disk matching the pattern `*.svg` to .png format.

A .png file is created for each .svg file in the folder.

Note to get this working correctly you need *Inkscape* installed and configured on your machine.

Examples

You can convert to many output formats using *Inkscape*:-

```
$ popy_imconv *.svg jpg
```

You can also just convert a single image:-

```
$ popy_imconv graph.svg tiff
```

Command line options

```
usage: popy_imconv [-h] input_images {png,jpg,emf,svg,pdf,wmf,tiff}

Converts images using inkscape.

positional arguments:
  input_images          input images path, you can use *.svg.
  {png,jpg,emf,svg,pdf,wmf,tiff}
                        image output format

optional arguments:
  -h, --help            show this help message and exit
```

6.7 Script File Formats

The PoPy script formats are shown in Table 6.3:-

Table 6.3: 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
n2pdat	converts *Nonmem* to PoPy data	none
p2ndat	converts PoPy to *Nonmem* data	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, *Open a PoPy Command Prompt* 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}
```

6.7.1 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 *Fitting a Simple PopPK Model using PoPy* or *Fitting a Two Compartment PopPK Model* for walk through examples.

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*

6.7.2 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.

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*

6.7.3 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.

Main Sections of a Sim Script

- *METHOD_OPTIONS*
- *PARALLEL*
- *DESCRIPTION*
- *FILE_PATHS*
- *DATA_FIELDS*
- *PREPROCESS*
- *STATES*
- *DERIVATIVES*
- *PREDICTIONS*
- *POSTPROCESS*
- *ODE_SOLVER*
- *OUTPUT_OPTIONS*
- *OUTPUT_SCRIPTS*

6.7.4 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 *Generate data and Fit using Simple PopPK Model* and *Generate data and Fit using a Two Compartment Model* for walk through examples.

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*

6.7.5 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*.

6.7.6 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*.

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*

6.7.7 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*.

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*

6.7.8 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 *Visual Predictive Check for Two Compartment PopPK Model* for a working example.

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*

6.7.9 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 *Generate multiple data sets and Fit using a Two Compartment Model* for walk through examples.

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*

6.7.10 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*.

6.7.11 Grph Script

Outputs basic plots given input data sets and fitting results.

6.7.12 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 *Visual Predictive Check for Simple PopPK Model* and *Visual Predictive Check for Two Compartment PopPK Model* for working examples.

6.7.13 FitSum Script

A *FitSum Script* generates a *HTML* report on the results of a *Fit Script*.

6.7.14 GenSum Script

A *GenSum Script* generates a *HTML* report on the results of a *Gen Script*.

6.7.15 TutSum Script

A *TutSum Script* generates a *HTML* report on the results of a *Tut Script*.

6.7.16 N2PDat Script

Converts a data set in *Nonmem* format to PoPy format. See *Nonmem to PoPy Data conversions using P2NDAT and N2PDAT Scripts* for a walk through example.

6.7.17 P2NDat Script

Converts a data set in PoPy format to *Nonmem* format. See *Nonmem to PoPy Data conversions using P2NDAT and N2PDAT Scripts* for a walk through example.

6.8 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 6.4 we list the main sections that are common to many *Script File Formats*.

Table 6.4: 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

6.8.1 METHOD_OPTIONS

This details the methods to be used in the script and is a required section.

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
```

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 6.5 shows both named float formats.

Table 6.5: 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.

6.8.2 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.

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.

6.8.3 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.

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']
```

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 'l' 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.

6.8.4 FILE_PATHS

This required field, defines where any input files can be found and where the results of the computation are to be stored.

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

    # 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
```

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/fit/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.

6.8.5 DATA_FIELDS

An optional section where the names of the *type_field*, *id_field* and *time_field* can be redefined.

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
```

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.

6.8.6 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.

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
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.

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.

6.8.7 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 Variation (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*.

EFFECTS with two levels from a fit_script

The example below is used in *Fitting a Two Compartment PopPK Model*.

```
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. 6.5.

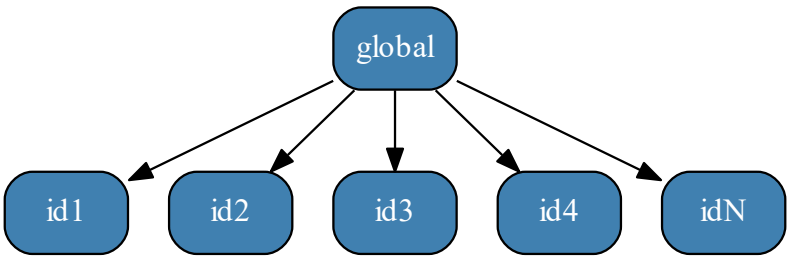


Fig. 6.5: *EFFECTS* structure with **two** levels

Here the number of `r[X]` values is dependent on the number of individuals in the *data file*.

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. 6.6.

For more information on this topic see *Inter-Occasion Variation (IOV)*.

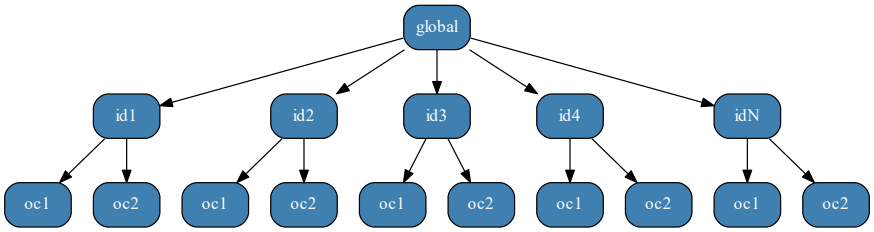


Fig. 6.6: *EFFECTS* structure with **three** levels

EFFECTS with two levels from a gen_script

The example below is used in *Generate a Two Compartment PopPK Data Set*.

```
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].

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)
        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.

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.

EFFECTS with two levels from a tut_script

The examples below are used in *Generate data and Fit using a Two Compartment Model*.

```
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.

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
```

```
    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.

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.

6.8.8 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.

Example MODEL_PARAMS section

The example below is used in *Fitting a Two Compartment PopPK Model*.

```
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.

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.

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.

6.8.9 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.

Example STATES for PK Model

In *Fitting a Two Compartment PopPK Model*, 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.

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.

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.

6.8.10 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.

Example DERIVATIVES for PK Model

The example below is used in *Fitting a Two Compartment PopPK Model*.

```
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],
        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. 6.7.

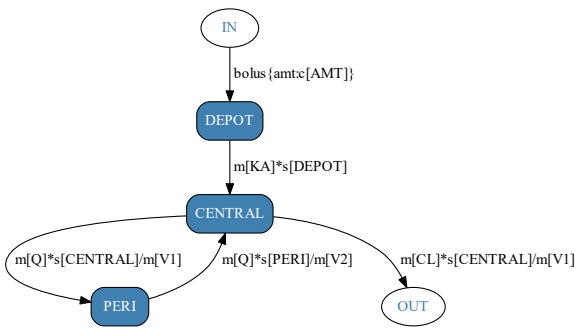


Fig. 6.7: Two compartment model with depot dosing, computed automatically from *DERIVATIVES* section.

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.

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. 6.8.

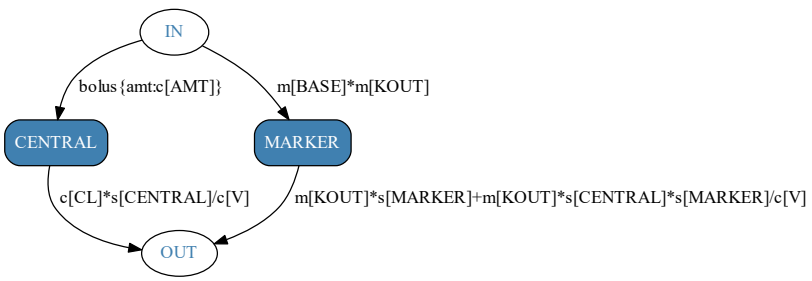


Fig. 6.8: direct PD model, computed automatically from *DERIVATIVES* section.

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. 6.9.

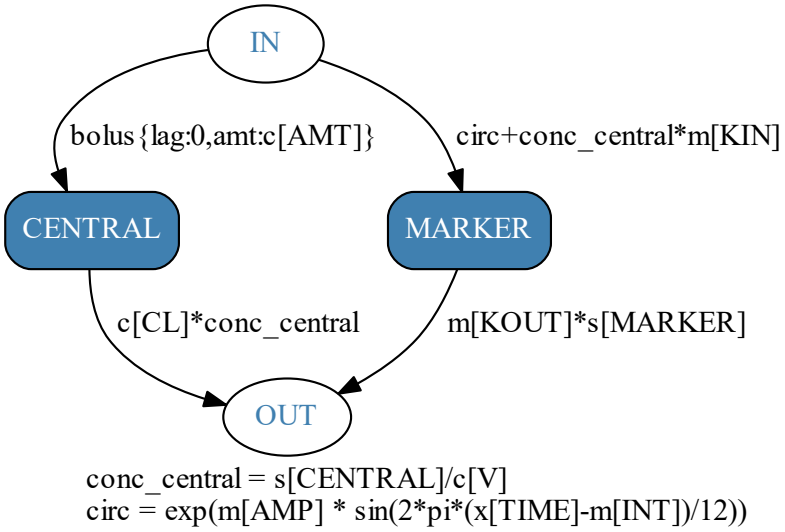


Fig. 6.9: circadian PD model, computed automatically from *DERIVATIVES* section.

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.

6.8.11 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.

Example PREDICTIONS for PK Model

The example below is used in *Fitting a Two Compartment PopPK Model*.

```
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 *Generate a Two Compartment PopPK Data Set*, 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.

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.

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]
    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.

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.

6.8.12 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.

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.

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.

6.8.13 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.

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.

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.

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'.

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
```

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:-

http://lagrange.univ-lyon1.fr/docs/scipy/0.17.1/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.

6.8.14 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*.

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.

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.

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]*.

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*.

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*.

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.

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.

6.8.15 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.

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.

6.8.16 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.

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.

Example OUTPUT_SCRIPTS for tut_script

The example below is used in *Generate data and Fit using a Two Compartment Model*.

```
OUTPUT_SCRIPTS:
    GEN: {output_mode: run, sim_time_step: 1.0, share_axes: True}
    FIT: {output_mode: run, sim_time_step: 1.0, share_axes: True}
    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, share_axes: True}
    FIT: {output_mode: run, sim_time_step: -1.0, share_axes: True}
    COMP: {output_mode: none}
    TUTSUM: {output_mode: run}
```

This results in more limited tutorial outputs compared to the previous example.

Example OUTPUT_SCRIPTS for gen_script

The example below is used in *Generate a Two Compartment PopPK Data Set*.

```
OUTPUT_SCRIPTS:
    SIM: {output_mode: run, sim_time_step: 1.0, share_axes: True}
    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}
    SIM: {output_mode: none}
    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.

Example OUTPUT_SCRIPTS for fit_script

The example below is used in *Fitting a Two Compartment PopPK Model*.

```
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}
    SIM: {output_mode: none}
    MSIM: {output_mode: none}
    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.

Example OUTPUT_SCRIPTS for sim_script

The example below is from a *Sim Script* generated by a *Fit Script* in *Fitting a Two Compartment PopPK Model*.

```
OUTPUT_SCRIPTS:
    GRPH:
        output_mode: run
        grph_list: ['SPAG_GRPH']
        x_var: TIME
        y_var_list: ['DV_CENTRAL', 'CEN', 'CEN']
        y_var_src_list: ['observed_data', 'pop', 'indiv']
        y_var_label_list: ['DV_CENTRAL', 'CEN', 'CEN']
```

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.

Example OUTPUT_SCRIPTS for msim_script

The example below is from a *MSim Script* generated by a *Fit Script* in *Fitting a Two Compartment PopPK Model*.

```
OUTPUT_SCRIPTS:
    VPC:
        output_mode: run
        vpc_list: ['COMB_QUANT_SIM_VPC']
        y_var_src_list: ['sim', 'orig']
        y_var_list: ['DV_CENTRAL_sim', 'DV_CENTRAL']
        x_var: TIME_SINCE_LAST_DOSE
```

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.

Example OUTPUT_SCRIPTS for mtut_script

The example below is used in *Generate multiple data sets and Fit using a Two Compartment Model*.

```
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.

6.8.17 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.

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'.

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.

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.

6.9 Script File Elements

See Table 6.6 for more detail on various script file elements.

Table 6.6: 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
Script Nodes	Elements of a *YAML* file

6.9.1 Variable Types

Table 6.7 shows the different type of script variables that are available in the *verbatim* sections of PoPy scripts:-

Table 6.7: 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 6.7, 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 6.7 above means the following sections - *MODEL_PARAMS*/*STATES*/*DERIVATIVES*/*PREDICTIONS*.

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}])
    ...
```

6.9.2 Probability Distributions

The distributions available for use in PoPy models are shown in Table 6.8:-

Table 6.8: 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)

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
```

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'.

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:
        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.

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.

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
    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**.

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.

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*.

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.

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.

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]`.

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.

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.

6.9.3 Matrices

The matrices available for use in PoPy models are shown in Table 6.9:-

Table 6.9: 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)]]

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.

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.

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*.

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.

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.

6.9.4 Dosing Functions

Table 6.10 shows the dosing functions that are available in the *DERIVATIVES* section and their parameters:-

Table 6.10: 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

@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*.

@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*.

@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*.

@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*.

@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*.

6.9.5 Analytic Compartment Functions

Table 6.11 shows the inbuilt compartment functions that are available in the *DERIVATIVES* section using the '_cl' suffix:-

Table 6.11: 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*.

@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.

@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.

@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.

@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.

@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.

@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 6.12:-

Table 6.12: 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

@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]
```

@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]
```

@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]
```

@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]
```

@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]
```

@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]
```

6.9.6 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.

6.10 Script File Outputs

The files generated by PoPy scripts are described in Table 6.13:-

Table 6.13: 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*

6.10.1 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: create}
    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.

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*,|focel 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/
```

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*.

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 *Visual Predictive Check for Simple PopPK Model*.

For more information on files output once the *MSim Script* is run see *Files Generated by MSim Script*.

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.

6.10.2 Files Generated by Gen Script

This section discusses how the *Gen Script* saves files to disk, you can see an example in *Generate a Two Compartment PopPK Data Set*. 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.

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. 4.3.

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*.

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.

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.

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.

6.10.3 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
```

6.10.4 Files Generated by Tut Script

This section describes the output files from the *Tut Script*, you can see examples in *Generate data and Fit using Simple PopPK Model* and *Generate data and Fit using a Two Compartment Model*.

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 6.10.4 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/
```

6.10.5 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 *Generate data and Fit using a Two Compartment Model*.

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.

6.10.6 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 *Generate multiple data sets and Fit using a Two Compartment Model*.

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*.

6.10.7 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 *Generate multiple data sets and Fit using a Two Compartment Model*.

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 mgen-output_options_spec 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*.

6.10.8 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 *Fitting a Two Compartment PopPK Model*.

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.

6.10.9 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 6.10.9 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/
```

6.10.10 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 *Generate multiple data sets and Fit using a Two Compartment Model*.

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 6.14: .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.

6.10.11 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 *Fitting a Two Compartment PopPK Model*.

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 *First order absorption model with peripheral compartment*.

6.10.12 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 *Generate a Two Compartment PopPK Data Set*.

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
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 *First order absorption model with peripheral compartment*.

6.10.13 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 *Generate data and Fit using Simple PopPK Model*.

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*.

6.11 PoPy Quick Reference Guide

See Table 6.15:-

Table 6.15: PoPy Quick Reference

Commands	*Scripts*	*Sections*
• *popy_env* • *popy_run* • *popy_check* • *popy_create* • *popy_format* • *popy_edit* • *popy_doc* • *popy_view* • *popy_info* • *popy_validate* • *popy_activate* • *popy_deactivate* • *popy_imconv*	• *fit* • *gen* • *sim* • *tut* • *comp* • *mfit* • *mgen* • *msim* • *mtut* • *mcomp* • *grph* • *vpc* • *fitsum* • *gensum* • *tutsum* • *n2pdat* • *p2ndat*	• *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

SEVEN

APPENDICES

7.1 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 online repository of PK/PD models 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 *PoPy Activation*.

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

7.2 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.

7.2.1 Example Summaries

Simple Fit Example

Used in *Fitting a Simple PopPK Model using PoPy* to demonstrate running a *Fit Script*.

Note: Online Example (v1.1.2): quick_start/fit_example1

Simple Tut Example

Used in *Generate data and Fit using Simple PopPK Model* to demonstrate running a *Tut Script*.

Note: Online Example (v1.1.2): quick_start/tut_example1

First order absorption model with peripheral compartment

Used in *Fitting a Two Compartment PopPK Model* to demonstrate running a *Fit Script*.

Note: Online Example (v1.1.2): quick_start/builtin_fit_example

First order absorption model with peripheral compartment

Used in *Generate a Two Compartment PopPK Data Set* to demonstrate running a *Gen Script*.

Note: Online Example (v1.1.2): quick_start/builtin_gen_example

Note: Example Data: https://product.popypkpd.com/docs/en/1.1.2/_autogen/quick_start/builtin_gen_example/builtin_gen_example.pyml_output/gen/synthetic_data.csv

First order absorption model with peripheral compartment

Used in *Generate data and Fit using a Two Compartment Model* to demonstrate running a *Tut Script*.

Note: Online Example (v1.1.2): quick_start/builtin_tut_example

7.2.2 Individual Model Summaries

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 (v1.1.2): indiv_examples/elimination/elim_ke_example

Elimination Example with Volume of Distribution

Used in *Volume of Distribution* to demonstrate elimination with the apparent volume of distribution, *V*.

Note: Online Example (v1.1.2): indiv_examples/elimination/elim_v_example

Elimination Example with Clearance

Used in *Clearance* to demonstrate elimination with clearance, *CL*.

Note: Online Example (v1.1.2): indiv_examples/elimination/elim_cl_example

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 (v1.1.2): indiv_examples/compartment_models/odes/iv_one_cmp_cl

One Compartment Model with Absorption

Used in *One Compartment Model with Absorption* to demonstrate a one compartment model with absorption.

Note: Online Example (v1.1.2): indiv_examples/compartment_models/odes/dep_one_cmp_cl

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 (v1.1.2): indiv_examples/compartment_models/odes/iv_two_cmp_cl

Two Compartment Model with Absorption

Used in *Two Compartment Model with Absorption* to demonstrate a two compartment model with absorption.

Note: Online Example (v1.1.2): indiv_examples/compartment_models/odes/dep_two_cmp_cl

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 (v1.1.2): indiv_examples/compartment_models/odes/iv_three_cmp_cl

Three Compartment Model with Absorption

Used in *Three Compartment Model with Absorption* to demonstrate a three compartment model with absorption.

Note: Online Example (v1.1.2): indiv_examples/compartment_models/odes/dep_three_cmp_cl

Bolus Dose with no elimination.

Used in *Bolus Dose* to demonstrate a single bolus dose with no elimination.

Note: Online Example (v1.1.2): indiv_examples/dosing/bolus_tut

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 (v1.1.2): indiv_examples/dosing/inf_dur_tut

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 (v1.1.2): indiv_examples/dosing/inf_rate_tut

Gamma Dose with no elimination.

Used in *Gamma Dose* to demonstrate a single gamma dose with no elimination.

Note: Online Example (v1.1.2): indiv_examples/dosing/gamma_tut

Weibull Dose with no elimination.

Used in *Weibull Dose* to demonstrate a single weibull dose with no elimination.

Note: Online Example (v1.1.2): indiv_examples/dosing/weibull_tut

Repeated Bolus Dose with first order elimination.

Used in *Repeated Dosing* to demonstrate a repeated bolus dose with first order elimination.

Note: Online Example (v1.1.2): indiv_examples/dosing/bolus_repeated_tut

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 (v1.1.2): indiv_examples/dosing/inf_dur_repeated_tut

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 (v1.1.2): indiv_examples/dosing/inf_rate_repeated_tut

Repeated Gamma Dose with first order elimination.

Used in *Repeated Dosing* to demonstrate a repeated gamma dose with first order elimination.

Note: Online Example (v1.1.2): indiv_examples/dosing/gamma_repeated_tut

Repeated Weibull Dose with first order elimination.

Used in *Repeated Dosing* to demonstrate a repeated weibull dose with first order elimination.

Note: Online Example (v1.1.2): indiv_examples/dosing/weibull_repeated_tut

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 (v1.1.2): indiv_examples/error_models/ao_tut

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 (v1.1.2): indiv_examples/error_models/po_tut

Model containing both proportional and additive error

Tut script used in *Residual Error Model* to demonstrate proportional and additive noise model.

Note: Online Example (v1.1.2): indiv_examples/error_models/pa_tut

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 (v1.1.2): indiv_examples/error_models/pa_gen_pa_fit_badvar

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 (v1.1.2): indiv_examples/pd_models/circ_sin

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 (v1.1.2): indiv_examples/pd_models/direct_pd

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 (v1.1.2): indiv_examples/pd_models/direct_pd_simul

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 (v1.1.2): indiv_examples/uncertainty/d1cmp_ka_stderr

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 (v1.1.2): indiv_examples/uncertainty/d1cmp_ka_v_stderr

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 (v1.1.2): indiv_examples/uncertainty/d1cmp_ka_cl_stderr

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 (v1.1.2): indiv_examples/uncertainty/d1cmp_v_cl_stderr

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 (v1.1.2): indiv_examples/uncertainty/d1cmp_ka_v_cl_stderr

7.2.3 Population Model Summaries

One Compartment Model with Absorption and Inter-subject Variance f[CL_isv]=0.2

Used in *Inter-Subject Variation (ISV)* to demonstrate inter-subject (or between-subject) variability.

Note: Online Example (v1.1.2): pop_examples/compartment_models/d1cmp_cl_isv

One Compartment Model with Absorption and Inter-subject Variance f[CL_isv]=0.01

Used in *Inter-Subject Variation (ISV)* to demonstrate inter-subject (or between-subject) variability.

Note: Online Example (v1.1.2): pop_examples/compartment_models/d1cmp_cl_isv_01

One Compartment Model with Absorption and Inter-subject Variance f[CL_isv]=0.5

Used in *Inter-Subject Variation (ISV)* to demonstrate inter-subject (or between-subject) variability.

Note: Online Example (v1.1.2): pop_examples/compartment_models/d1cmp_cl_isv_05

One Compartment Model with Absorption and no inter-subject Variance f[CL_isv]=0

Used in *Inter-Subject Variation (ISV)* to demonstrate inter-subject (or between-subject) variability.

Note: Online Example (v1.1.2): pop_examples/compartment_models/d1cmp_cl_isv_naive

One Compartment Model with Absorption and Inter-occasion Variance f[CL_isv]=0.2

Used in *Inter-Occasion Variation (IOV)* to demonstrate inter-occasion (or between-occasion) variability.

Note: Online Example (v1.1.2): pop_examples/compartment_models/d1cmp_cl_iov

One Compartment Model with Absorption and Inter-occasion Variance f[CL_isv]=0.5

Used in *Inter-Occasion Variation (IOV)* to demonstrate inter-occasion (or between-occasion) variability.

Note: Online Example (v1.1.2): pop_examples/compartment_models/d1cmp_cl_iov_05

One Compartment Model with Absorption and no inter-occasion Variance f[CL_iov]=0

Used in *Inter-Occasion Variation (IOV)* to demonstrate inter-occasion (or between-occasion) variability.

Note: Online Example (v1.1.2): pop_examples/compartment_models/d1cmp_cl_iov_naive

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 (v1.1.2): pop_examples/re_covariance/gen_indep_fit_indep

Diagonal matrix generation diagonal matrix fit

Used in *Modelling Correlation in Random Effects* to demonstrate correlation between random effects.

Note: Online Example (v1.1.2): pop_examples/re_covariance/gen_diag_fit_diag

Diagonal matrix generation full matrix fit

Used in *Modelling Correlation in Random Effects* to demonstrate correlation between random effects.

Note: Online Example (v1.1.2): pop_examples/re_covariance/gen_diag_fit_full

Full matrix generation diagonal matrix fit

Used in *Modelling Correlation in Random Effects* to demonstrate correlation between random effects.

Note: Online Example (v1.1.2): pop_examples/re_covariance/gen_full_fit_diag

Full matrix generation full matrix fit

Used in *Modelling Correlation in Random Effects* to demonstrate correlation between random effects.

Note: Online Example (v1.1.2): pop_examples/re_covariance/gen_full_fit_full

Body Weight Covariate

Used in *Covariates* to demonstrate using weight as a covariate.

Note: Online Example (v1.1.2): pop_examples/covariates/weight_covariate

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 (v1.1.2): pop_examples/blq/blq_pk_tut

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 (v1.1.2): pop_examples/blq/blq_pk_norm_fit

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 (v1.1.2): pop_examples/blq/blq_pk_norm_fit_half

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 (v1.1.2): pop_examples/blq/blq_pk_norm_fit_ignore

7.3 Troubleshooting

Links to common errors when running PoPy models are shown in Table 7.1:-

Table 7.1: Solutions to common PoPy issues

Problem	Solution
'popy_run' is not recognized	run *popy_env* before *popy_run*
The term 'popy_run' is not recognized as the name of a cmdlet	run *popy_env* before *popy_run*
'popy_env' is not recognized	put 'popy_env.bat' on system path
The term 'popy_env' is not recognized as the name of a cmdlet	put 'popy_env.bat' on system path
'popy_env' permission error	use *dos prompt* to call *popy_env*

7.3.1 'popy_run' is not recognized

Scenario

You get this error if you open a *dos prompt* and attempt to run the *popy_run* program as follows:-

```
$ popy_run
```

Then get the response:-

```
'popy_run' is not recognized as an internal or external command,
operable program or batch file.
```

Cause

The problem is that the PoPy environment has not been enabled, so *Microsoft Windows* does **not** know where to look for the command 'popy_run'.

Solution

The solution is to run *popy_env* before you do *popy_run*, as follows:-

```
$ popy_env
$ popy_run
```

popy_run should now work as expected.

7.3.2 The term 'popy_run' is not recognized as the name of a cmdlet

Scenario

You get this error if you open a *powershell prompt* and attempt to run the *popy_run* program as follows:-

```
$ popy_run
```

Then get the response:-

```
popy_run : The term 'popy_run' is not recognized␣
↪as the name of a cmdlet, function, script file, or operable program.
Check the spelling of the name,
↪ or if a path was included, verify that the path is correct and try again.
At line:1 char:1
+ popy_run
+ ~~~~~~~~
    + CategoryInfo␣
↪          : ObjectNotFound: (popy_run:String) [], CommandNotFoundException
    + FullyQualifiedErrorId : CommandNotFoundException
```

Usually in difficult to read red on blue writing.

Cause

The problem is that the PoPy environment has not been enabled, so *Microsoft Windows* does **not** know where to look for the command 'popy_run'.

Solution

The solution is to run *popy_env* before you do *popy_run*, as follows:-

```
$ popy_env
$ popy_run
```

popy_run should now work as expected.

7.3.3 'popy_env' is not recognized

Scenario

You start a *dos prompt* and type the following command:-

```
$ popy_env
```

Then you see the message:-

```
'popy_env' is not recognized as an internal or external command,
operable program or batch file.
```

Cause

PoPy is either not installed or your system path is not configured correctly.

Solution

In a *dos prompt* type:-

```
$ echo %PATH%
```

In order for PoPy to work the directory containing 'popy_env.bat' on your machine, needs to be listed in the *Microsoft Windows* system path. If the 'popy_env.bat' directory is **not** present, then you need to manually add it. In a dos prompt you can do this:-

```
$ set PATH=%PATH%;c:\PoPy
```

Where 'c:\PoPy' is the recommended install directory for PoPy.

To save the path permanently do:-

```
$ setx PATH %PATH%
```

Note you probably need **admin rights** to use the 'setx' command. Once your *Microsoft Windows* path is sorted out you should be able to run and test the PoPy environment as follows:-

```
$ popy_env
$ popy_info
```

7.3.4 The term 'popy_env' is not recognized as the name of a cmdlet

Scenario

You start a *powershell prompt* and type the following command:-

```
$ popy_env
```

Then you see the message:-

```
popy_env : The term 'popy_env' is not recognized␣
↪as the name of a cmdlet, function, script file, or operable program.
Check the spelling of the name,
↪ or if a path was included, verify that the path is correct and try again.
At line:1 char:1
+ popy_run
+ ~~~~~~~~
```

```
    + CategoryInfo␣
↪          : ObjectNotFound: (popy_run:String) [], CommandNotFoundException
    + FullyQualifiedErrorId : CommandNotFoundException
```

Cause

PoPy is either not installed or your system path is not configured correctly.

Solution

In a *powershell prompt* type:-

```
$ $env:Path
```

In order for PoPy to work the directory containing 'popy_env.bat' on your machine, needs to be listed in the *Microsoft Windows* system path. If the 'popy_env.bat' directory is **not** present, then you need to manually add it. In a *powershell prompt* prompt you can do this:-

```
$ $env:Path = "$env:Path;c:\PoPy"
```

Where 'c:\PoPy' is the recommended install directory for PoPy.

To save the new path permanently do:-

```
$ setx PATH $env:Path
```

Note you probably need **admin rights** to use the 'setx' command. Once your *Microsoft Windows* path is sorted out you should be able to run and test the PoPy environment as follows:-

```
$ popy_env
$ popy_info
```

7.3.5 'popy_env' permission error

Scenario

You start a *powershell prompt* and type the following command:-

```
$ popy_env
```

Then you see the message:-

```
popy_env : File␣
↪C:\PoPy\popy_env.ps1 cannot be loaded because running scripts is disabled on
this system. For more information, see␣
↪about_Execution_Policies at https:/go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:1
+ popy_env
+ ~~~~~~~~
    + CategoryInfo          : SecurityError: (:) [], PSSecurityException
    + FullyQualifiedErrorId : UnauthorizedAccess
```

Cause

You do not have sufficient Powershell script execution rights to run the *popy_env* setup script.

Solution1

If you are an Administrator on your *Microsoft Windows* system, you may grant execution rights to all powershell scripts, by opening a **Administrator** *powershell prompt* and running this command:-

```
$ set-ExecutionPolicy unrestricted
```

Or possibly just set the execution policy for the *popy_env* script alone:-

```
$ set-ExecutionPolicy ByPass -File c:\PoPy\popy_env.ps1
```

This should then allow you to run:-

```
$ popy_env
```

From a normal **non-admin** *powershell prompt*.

For more information on the Powershell script permission system see this page:-

https://docs.microsoft.com/en-gb/powershell/module/microsoft.powershell.core/about/about_execution_policies

Solution2

An alternative solution maybe to *Uninstall PoPy* and reinstall PoPy in a location where your *Microsoft Windows* user has more permissions. It's a good idea to **avoid** folders like:-

```
c:\Program Files\PoPy
```

We recommend this directory, if you want non-admin users to run PoPy:-

```
c:\PoPy
```

For more information see *Install PoPy*

Solution3

You could invoke *popy_env* from a *dos prompt* instead and just **not** use Powershell. For example start PoPy using *Desktop Shortcut Method*.

7.4 Bug Reporting

7.4.1 Check you have a bug!

Are you using the latest version of PoPy? It is possible that the bug you are encountering has already been fixed. Examine the *Release Notes*.

It might be worth upgrading PoPy anyway, to see if the bug gets fixed. All PoPy end user *Licensing* allow free upgrades to the latest version.

Have you managed to reproduce the bug reliably? Preferably on more than one machine. Sometimes the process of reproduction reveals an easy solution.

Note that often PoPy throws an error due to a user error in the input script file. This is **not** classed as a bug. However if the error message is unclear and confusing, that is a bug!

7.4.2 Bug Report Advice

Three elements that help produce a good bug report, are as follows:-

Steps to reproduce

It is important that the developers can reproduce the bug, otherwise the bug **cannot** be fixed! It is useful to include **all** details, *e.g.* Operating system and current version of PoPy. The output of *popy_info* is very helpful. Also any input script files and data sets. If you can reproduce the problem on a small example that fails quickly, that is also very helpful.

Note, confidentiality may prevent you from sharing your original data with us. In this case you may consider:-

- Renaming the data header file to remove context
- Add random noise to your data (check the bug still occurs)
- Removing unused columns and rows (check the bug still occurs)
- Creating a new data set using a *Gen Script*, that exhibits the same bug.

It is helpful to use a zip utility, so we can easily set up your files and reproduce the bug from an email.

Expected result

State what you expected to happen!

Sometimes this is obvious. For example if PoPy crashes mid run. However sometimes it is more subtle, for example, you expected PoPy to give an answer much closer to a ground truth value for a particular model and data set.

Actual result

Clearly state what actually happened. Reporting any error messages is particularly vital. Note, that a PoPy program typically produces a log file like this:-

```
my_script.pyml.run.main.log
```

However if a runtime error occurs an additional log file is created as follows:-

```
my_script.pyml.run.error.log
```

7.4.3 Reporting the bug

Currently PoPy does not have a formal bug tracking system (it may do in the near future), so for now email your bug to:-

info@popypkpd.com

We will endeavour to respond quickly with a time estimate of how long it will take to fix the bug.

If the bug is serious enough, we will create a new PoPy binary as quickly as possible, otherwise the bug will be fixed in the next planned release for PoPy.

7.5 Credits

PoPy could not have been developed without the efforts of many people.

7.5.1 Authors

Main contributors are:-

- Phil Tresadern
- David Cristinacce
- Andrew Cristinacce
- James Wright

7.5.2 Sponsor

Wright Dose Ltd.

7.5.3 Python Libraries

Many thanks to the people responsible for the following *Python* gems:-

- NumPy - fast c style arrays + matrices
- SciPy - ODE solver + stats functions
- matplotlib - Plotting graphs
- SymPy - Symbolic Python - thanks for all those Jacobians
- sphinx - for generating this documentation
- pylint - for diligently pointing out all our errors
- jenkins - continuous integration server - for helping pylint shout louder

7.6 Release Notes

This page lists the various PoPy releases to date:-

7.6.1 PoPy v1.1.2

Release date = 11 Mar 2022

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.

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.

7.6.2 PoPy v1.1.1

Release date = 22 Oct 2021

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.

Bug Fixes

- Fixed various bugs with fitting in parallel

7.6.3 PoPy v1.1.0

Release date = 22 Sep 2021

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.

Bug Fixes

- Fixed bug when passing string data into *Cython* functions.

7.6.4 PoPy v1.0.5

Release date = 13 May 2021

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.

Bug Fixes

- Improvements in error catching and reporting when running PoPy in parallel using *MPI*.
- Fixed bug when using backslashes in *DERIVATIVES* section.

7.6.5 PoPy v1.0.4

Release date = 23 Dec 2020

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.

Bug Fixes

- Bugs in gradient computation due to variable `C[X]` within individual data fixed.
- Fixed bugs in *standard error* computation, now more reliable.

7.6.6 PoPy v1.0.3

Release date = 05 Sep 2020

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.

Format Changes

- 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.

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*.

7.6.7 PoPy v1.0.2

Release date = 18 Sep 2019

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, see *Running multiple scripts*, *Checking multiple scripts*, *Format multiple scripts*, *Edit multiple scripts*, *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.

Format Changes

- 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.

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 *Objective function at each iteration for simple PopPK example*

7.6.8 PoPy v1.0.1

Release date = 28 Jun 2019

This is mainly a bug fix release of PoPy

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.

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.

7.6.9 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!

BIBLIOGRAPHY

[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.