An Advanced User’s Guide to Stat-JR (Beta) version 0.2

Programming and Documentation by

William J. Browne
Bruce Cameron
Christopher M.J. Charlton
Danius T. Michaelides*
Richard M.A. Parker
Camille Szmaragd
Huanjia Yang*
and
Zhengzheng Zhang

Centre for Multilevel Modelling,
University of Bristol.

*Electronics and Computer Science,
University of Southampton.

December 2012
## Contents

1. About Stat-JR (e-STAT) ................................................................. 1
   1.1 Stat-JR: software for scaling statistical heights .......................... 1
   1.2 About the Advanced User’s Guide ........................................... 2
2. Installation instructions ............................................................... 2
3. A simple regression template example ............................................. 4
   3.1 Running a first template ....................................................... 4
   3.2 Opening the bonnet and looking at the code .............................. 9
      3.2.1 Invars ........................................................................... 10
      3.2.2 Outbug ......................................................................... 11
      3.2.3 Outlatex ....................................................................... 12
      3.2.4 Some points to note ....................................................... 14
   3.3 Writing your own first template ................................................. 14
4. Running templates with the e-STAT engine ....................................... 15
   4.1 Algebra and Code Generation ................................................... 15
   4.2 The Demo algebraic software system ......................................... 21
5. Including Interoperability ............................................................... 25
   5.1 eSTAT.py ........................................................................... 25
   5.2 Regression2.py ................................................................. 26
   5.3 WinBUGS and Winbugsscript.py .............................................. 26
   5.4 MLwiN ................................................................................ 30
   5.5 R ...................................................................................... 36
   5.6 Other packages .................................................................... 41
6. Input, Data manipulation and output templates .................................. 42
   6.1 Generate template (generate.py) ............................................. 42
   6.2 Recode template (recode.py) .................................................. 46
   6.3 AverageandCorrelation template ............................................. 49
Acknowledgements

The Stat-JR software is very much a team effort and is the result of work funded under three ESRC grants. The LEMMA 2 and LEMMA 3 programme nodes (Grant: RES-576-25-0003 & Grant:RES-576-25-0032) as part of the National Centre for Research Methods programme and the e-STAT node (Grant: RES-149-25-1084) as part of the Digital Social Research programme. We are therefore grateful to the ESRC for financial support to allow us to produce this software.

All nodes have many staff that, for brevity, we have not included in the list on the cover. We acknowledge therefore the contributions of:

Fiona Steele, Harvey Goldstein, George Leckie, Rebecca Pillinger, Kelvyn Jones, Paul Clarke, Mark Lyons-Amos, Liz Washbrook, Sophie Pollard, Robert French, Nikki Hicks, Mary Takahama and Hilary Browne from the LEMMA nodes at the Centre for Multilevel Modelling for their input into the software.

David De Roure, Luc Moreau, Tao Guan, Alex Fraser, Toni Price, Mac McDonald, Ian Plewis, Mark Tranmer, Pierre Walthery, Paul Lambert, Emma Housley, Kristina Lupton and Antonina Timofejeva from the E-STAT node for their input into the software.

A final acknowledgement to Jon Rasbash who was instrumental in the concept and initial work of this project. We miss you and hope that the finished product is worthy of your initials.

WJB December 2012.
1. About Stat-JR (e-STAT)

1.1 Stat-JR: software for scaling statistical heights.

The use of statistical modelling by researchers in all disciplines is growing in prominence. There is an increase in the availability and complexity of data sources and this has been followed by a corresponding increase in the sophistication of statistical methods that can be used. For the novice practitioner of statistical modelling it can seem like you are stuck at the bottom of a mountain, and current statistical software allows you to progress slowly up certain specific paths depending on the software used. Our aim in the Stat-JR package is to assist practitioners in making their initial steps up the mountain, but also to cater for more advanced practitioners who have already journeyed high up the path, but want to assist their novice colleagues in making their ascent as well.

One issue with complex statistical modelling is that to use the latest techniques can involve having to learn new pieces of software. This is a little like taking a particular path up a mountain with one piece of software, spotting a nearby area of interest on the mountainside (e.g. a different type of statistical model), and then having to descend again and take another path, with another piece of software, all the way up again to eventually get there, when ideally you’d just jump across! In Stat-JR we aim to circumvent this problem via our interoperability features so that the same user interface can sit on top of several software packages thus removing the need to learn multiple packages. To aid understanding, the interface will allow the curious user to look at the syntax files for each package to learn directly how each package fits their specific problem.

To complete the picture, the final group of users to be targeted by Stat-JR is the statistical algorithm writers. These individuals are experts at creating new algorithms for fitting new models, or better algorithms for existing models, and can be viewed as sitting high on the peaks with limited links to the applied researchers who might benefit from their expertise. Stat-JR will build links by incorporating tools to allow this group to connect their algorithm code to the interface through template-writing, and hence allow it to be exposed to practitioners. They can also share their code with other algorithm developers, and compare their algorithms with other algorithms for the same problem.

Many of the ideas within the Stat-JR system were the brainchild of Jon Rasbash (JR). Sadly, Jon died suddenly just as we began developing the system, and so we dedicate this software to his memory. We hope that you enjoy using Stat-JR and are inspired to become part of the Stat-JR community: either through the creation of your own templates that can be shared with others, or simply by providing feedback on existing templates.

Happy Modelling,

The Stat-JR team.
1.2 About the Advanced User’s Guide

The Advanced Guide is meant to complement the Beginners guide and we recommend that users read that guide first to get an idea of how the Stat-JR software works. A major component of the Stat-JR package is the use of (often user written) templates. Templates are pieces of computer code (written in the Python language) that perform a specific task. Many of the templates are used to fit a family of statistical models although there are other templates that perform data input, data manipulation and data and graphical output.

In this document it is our aim to give users who intend to write their own templates, or more generally are interested in how the Stat-JR system works, more details about how to write templates and to some degree how the system fits together. We will do this by showing the code for several of the templates we have written and giving a detailed explanation of what each function and even in places each line of code does.

An initial question posed by potential template writers has been what language are templates written in and when told ‘Python’ then ask whether we are providing an introductory chapter on this language. We are not specifically writing an introductory chapter on Python (good books include Hetland (2005) and Lutz and Ascher (2004)) as it has a vast language and we will be interested in specific aspects of the language, some of which are non-standard and specific to Stat-JR. In fact many of the functions that make up a template in Stat-JR are designed to create text blocks in other languages, for example C++, WinBUGS or any of the other macro languages associated with the software packages supported via inter-operability. This is not to say that reading up on Python is without merit and certainly Python experts will find writing templates initially easier than others (though more because of their programming skills than their Python skills per se).

Our advice is therefore to work through this guide first and try the exercises and have a Python book as a backstop for when you are stuck writing your own templates. We will now give instructions into how to install all the software needed to run Stat-JR before moving on to our first example template.

2 Installation instructions

Stat-JR has a dedicated website for requests for a copy of the software and which contains instructions for installation. This is currently located at

http://www.bristol.ac.uk/cmm/software/statjr/index.html

To run the software:

Stat-JR runs in a web browser; whilst it will work in most web browsers we suggest not using Internet Explorer, although it is hoped support for more browsers will be added in future. To start Stat-JR, open Windows Explorer / My Computer and go to the directory that contains the Stat-JR software, typically C:\Stat-JR. Here double-click on the file webtest.cmd; this should bring up a (Firefox) web browser.
When you double-click on `webtest.cmd`, this action starts a Command prompt window in the background to which commands are printed out. This window is useful for viewing what the system is doing: for example, on the machine on which I have run `webtest.cmd` I can see the following commands:

```
F:\JRdecrel>cd src\apps\webtest
F:\JRdecrel>cd src\apps\webtest\webtest

Failed to load package GenStatModel (GenStat not defined)
Failed to load package MatlabScript (Matlab not found)
Failed to load package MinitabModel (Minitab not found)
Failed to load package MintabScript (Minitab not found)
Failed to load package OctaveScript (Octave not found)
Failed to load package Sabre (Sabre not found)
Failed to load package SASModel (SAS not found)
Failed to load package SAScript (SAS not found)
Failed to load package SPSSModel (SPSS not found)
Failed to load package SPSScript (SPSS not found)
Trying to locate and open Firefox
could not locate runnable browser
Trying to locate and open Chrome
could not locate runnable browser
Trying to locate and open default web browser
http://0.0.0.0:1878/
127.0.0.1:1880 - - [17/Dec/2012 10:02:52] "HTTP/1.1 GET /" - 200 OK
```

The most important command when starting up is the line:

```
http://0.0.0.0:1878/
```

Note that the number 1878 is specific to this run of the program and will be different on your machine. This line only appears when the program has performed all its initial set-up routines. This may take a while, particularly the first time you use the program. You should then be able to view
the main page of webtest in your browser (note: not all browser packages are supported: see note above); if you can’t, then try refreshing the browser window, or typing localhost:1878 into the address bar (substituting the number you see for 1878).

3 A simple regression template example

3.1 Running a first template

We will firstly consider a very simple template that is included in the core model templates distributed with Stat-JR which has the title Regression1. This template is used in the Beginners guide and perhaps before looking at the code it would be good to run the template again in the web interface to see what it does. To do this fire up the Stat-JR package as directed in section 2. If you refresh the screen you should be greeted by the following display:

Here you will see that the template Regression1 is the default template on start up and the default dataset is called tutorial [see the beginner’s guide for more information on the dataset]. We will here click on the Run button to run the template and be greeted by the following display:
Here you will see a Configuration box which is looking for inputs for a response (a single select list) and explanatory variables (a multiple select list). We will here select normexam as the response and cons and standlrt as the explanatory variables as shown below:

Next we click on the Next button and fill in the input boxes that appear as follows:
Note that when all boxes on the screen are filled in, clicking the **Next** button will show further inputs if there are any. Here we have given a name of the object where the results are to be stored. The other inputs are for the MCMC estimation methods we are using. When you click **Next** again the software will perform some procedures in the background and after a short while the screen will expand to include two panes with accompanying pull down lists in which various input (to the left) and output (to the right) objects can be displayed thus:
The left pane in the main window has some code that is written in a variant of the model specification language associated with the WinBUGS package. Other objects, for example, a nicely formatted mathematical description of the model (in LaTeX code) can be shown by picking `equation.tex` in the right hand box and these can be displayed in their own box by right clicking in the pane and choosing This Frame -> Open Frame in New Tab:

There is a green Run button above the two panes. If we click this button then after a short while the model will run and we will have more objects to choose from in the right hand output list including the results (ModelResults):
This screen contains summary statistics for four parameters (in fact sigma and tau are functions of each other). We can also look at diagnostic plots for the parameters e.g. beta0.png:
The purpose of this document is not to go into details about what these figures mean – interested readers can look at the accompanying Beginner’s guide for such information. Instead we want to teach you here how to write a similar template yourself.

3.2 Opening the bonnet and looking at the code

The operations that we have here performed in fitting our first model are shared between the user written template Regression1 and other code that is generic to all templates and which we will discuss in more detail later.

So our next stage is to look at the source python file for Regression1. All templates are stored in the templates subdirectory under the base directory and have the extension .py and so if we open regression1.py (in Wordpad/Notepad and not Python) we will see the following:

```python
from EStat.Templating import *
from mako.template import Template as MakoTemplate
import re

class Regression1(Template):
    'A model template for fitting 1 level Normal multiple regression model in E-STAT only. To be used in documentation.'

    tags = ['Model', '1-Level', 'E-Stat', 'Normal']
    engines = ['eSTAT']

    invars = '''
y = DataVector('response: ')
tau = ParamScalar()
sigma = ParamScalar()
sigma2 = ParamScalar()
x = DataMatrix('explanatory variables: ')
beta = ParamVector(parents=[x], as_scalar=True)
'''

    outbug = '''
model{
    for (i in 1:length(${y})) {
        ${y}[i] ~ dnorm(mu[i], tau)
        mu[i] <- ${mmult(x, 'beta', 'i')}
    }

    # Priors
    % for i in range(0, x.ncols()):
    beta$i$ ~ dflat()
    % endfor
    tau ~ dgamma(0.001000, 0.001000)
    sigma2 <- 1 / tau
    sigma <- 1 / sqrt(tau)
}
'''

    outlatex = r'''
\begin{aligned}
\mbox{${y}}_i & \sim \mbox{N}({\mu_i}, \sigma^2) \\
\mu_i & = ${mmulttex(x, r'\beta', 'i')} \\
%for i in range(0, len(x)):'''
```
We will now describe in some detail what this code does. The first 3 lines here are simply importing information needed by the template and are generic to many templates. We then have a class statement which defines a class \texttt{Regression1} which is a subclass of a generic \texttt{Template} class. There is then a sentence known as a descriptor that describes what the template does. For those unfamiliar with the terminology we are using think of a class as being a definition of a type of object, for example we might have a class of rectangles where each rectangle might be described by two attributes, length and width. Then an instance of the class which we might call Dave will have these values instantiated e.g. Dave’s length is 3 and width is 1. We might think of the subclass of rectangles the squares which again have the two attributes length and width. We could state that class \texttt{Square} (Rectangle): in which case we know that as squares are a subclass of rectangles they have a height and width but we would now redefine the attribute width within the squares definition to equal height. This terminology is what is used in what is called \textit{object orientated programming}. In the definition here five attributes (tags, engines, invars, outbug, and outlatex) are then defined as being parts of a \texttt{Regression1} class although there will be other attributes that are generic to the template class and are defined elsewhere.

Briefly:

The \texttt{tags} attribute identifies the template as belonging to the tag groups ‘Model’, ‘1-Level’, ‘e-STAT’ and ‘Normal’ and this is used in the web interface to decide which templates to show in specific template lists.

The \texttt{engines} attribute identifies which estimation and or graphical engines can be used with this template, in this case just the built-in ‘e-STAT” estimation engine, which is used by Stat-JR to decide which options to offer. This attribute is how, along with additional attributes, we allow Stat-JR to interoperate with other software.

The \texttt{invars} attribute is a text string (hence the starting and ending “\”) which consists of a list of the inputs in this template.

The \texttt{outbug} attribute is a text string that will produce the model code we saw in the web interface for this template.

The \texttt{outlatex} attribute is a text string that will produce a piece of LaTeX code which is converted into the nice maths we saw in the web interface. We will next look at the last three attributes in more detail.

\subsection*{3.2.1 Invars}

When this template has been selected in the web interface it will firstly have its inputs interrogated and start creating an instance of a model object. Stat-JR has a list of object types that can be thought of as the building blocks of a model object. Statements like

\begin{verbatim}
  y = DataVector('response: ')
\end{verbatim}

...can be thought of as defining the components that make up a model object, so here we are building a model object that contains a data vector called y. The text in the brackets is used by the web
interface as a piece of text to place on the screen alongside the appropriate input device (in the case of a data vector a single select list).

Stat-JR distinguishes between Data objects which require user inputs and Parameters (Param) which just need to be declared. This template therefore has 6 components (2 pieces of data and 4 parameters) that make up the model. The DataMatrix declaration for x will correspond to the multiple-select list that we saw when running the template.

3.2.2 Outbug

The outbug attribute gives a definition (as a text string) of an instance of a model set up using this template. The definition is in a language that very much resembles the language used by the WinBUGS package (with some minor differences) and will be used in Stat-JR to create code to run the model. The definition is also shown on the screen in the left hand pane under the label model.txt so you can for example see the definition for the model we fitted to the tutorial dataset earlier by turning back a few pages. As the text is specific to the inputs given, the definition is a text string containing some quantities that depend on inputs. These are integrated into the text string via the $ symbol for substitutions, through conditional and looping computation achieved via % commands and through the calling of external functions. The outbug code for this template uses all three devices and so we will here go through stage by stage the instance of outbug shown in the earlier screen shots.

We start with the raw code:

```r
outbug = ''
model{
  for (i in 1:length(${y})) {
    ${y}[i] ~ dnorm(mu[i], tau)
    mu[i] <- ${mmult(x, 'beta', 'i')}
  }
  # Priors
  % for i in range(0, x.ncols()):
  beta${i} ~ dflat()
  % endfor
  tau ~ dgamma(0.001000, 0.001000)
  sigma2 <- 1 / tau
  sigma <- 1 / sqrt(tau)
}
```

Now we can substitute normexam for ${y} as this is the column we chose for y thus:

```r
outbug = ''
model{
  for (i in 1:length(normexam)) {
    normexam[i] ~ dnorm(mu[i], tau)
    mu[i] <- ${mmult(x, 'beta', 'i')}
  }
  # Priors
  % for i in range(0, x.ncols()):
  beta${i} ~ dflat()
  % endfor
  tau ~ dgamma(0.001000, 0.001000)
  sigma2 <- 1 / tau
  sigma <- 1 / sqrt(tau)
}
```

We start with the raw code:
Next we can evaluate the for loop with, in our example \( x \) having 2 columns:

```r
outbug = '''
model{
    for (i in 1:length(normexam)) {
        normexam[i] ~ dnorm(mu[i], tau)
        mu[i] <- $\text{mmult}(x, \text{\textquoteleft beta\textquoteright, \textquoteleft i\textquoteright})$
    }

    # Priors
    beta0 ~ dflat()
    beta1 ~ dflat()
    tau ~ dgamma(0.001000, 0.001000)
    sigma2 <- 1 / tau
    sigma <- 1 / sqrt(tau)
}
'''
```

Finally the function \textit{mmult} is a function written separately and is used to create the products of the \( x \) variables and their associated \textit{betas} with appropriate indexing. When run we get:

```r
outbug = '''
model{
    for (i in 1:length(normexam)) {
        normexam[i] ~ dnorm(mu[i], tau)
        mu[i] <- \text{cons}[i]\times\beta_0 + \text{standlrt}[i]\times\beta_1
    }

    # Priors
    beta0 ~ dflat()
    beta1 ~ dflat()
    tau ~ dgamma(0.001000, 0.001000)
    sigma2 <- 1 / tau
    sigma <- 1 / sqrt(tau)
}
'''
```

This is identical to the code we see under \textbf{Model} in the webtest interface and is one way of displaying the model we wish to fit. Another way is to write the model in mathematical form using the Latex language and this is also shown in the web output in the right hand pull down list under \texttt{equation.tex}. Basically we are using a program called MathJax which will display LaTeX code in a nice format embedded within a webpage. The attribute that is used for creating this code is outlatex.

### 3.2.3 Outlatex

If you click on the right button on the part of the screen showing the Equations and click on the \textit{Show Maths as TeX commands} option you will get a window popping up that shows the source:
This code is created via the \textit{outlatex} function and we will now look at how we get from \textit{outlatex} to this source for our example. The generic code is as follows:

\begin{aligned}
\text{${y}_i \sim \text{N}({\mu}_i, \sigma^2)} \setcounter{equation}{0} \\
\mu_i = \\
\beta_0 (\text{cons})_{(i)} + \beta_1 (\text{standlt})_{(i)} \\
\beta_0 \propto 1 \\
\beta_1 \propto 1 \\
\tau \sim \text{Gamma} (0.001,0.001) \\
\sigma^2 = 1 / \tau
\end{aligned}

We have three steps as with the \textit{outbug} function, firstly we will substitute \textit{normexam} for ${y}$

\begin{aligned}
\text{normexam}_i \sim \text{N}({\mu}_i, \sigma^2) \setcounter{equation}{0} \\
\mu_i = \\
\beta_0 \propto 1 \\
\beta_1 \propto 1 \\
\tau \sim \text{Gamma} (0.001,0.001) \\
\sigma^2 = 1 / \tau
\end{aligned}

Next we can evaluate the for loop with, in our example x having 2 columns:

\begin{aligned}
\text{normexam}_i \sim \text{N}({\mu}_i, \sigma^2) \setcounter{equation}{0} \\
\mu_i = \\
\beta_0 \propto 1 \\
\beta_1 \propto 1 \\
\tau \sim \text{Gamma} (0.001,0.001) \\
\sigma^2 = 1 / \tau
\end{aligned}
and finally we have the step to expand a function – this time called mmulttex:

```latex
\begin{aligned}
\text{normexam}_i & \sim \text{N} (\mu_i, \sigma^2) \\
\mu_i & = \beta_0 \text{cons}_{i} + \beta_1 \text{standlrt}_{i} \\
\beta_0 & \propto 1 \\
\beta_1 & \propto 1 \\
\tau & \sim \Gamma (0.001,0.001) \\
\sigma^2 & = 1 / \tau
\end{aligned}
```

3.2.4 Some points to note

You will notice that the string object created in `outlatex` has an `r` before the `'` and that similarly there is an `r` inside the `mmulttex` function call before the `. Basically the triple quotes are used in place of quotes to allow the use of single quotes within the expression. The `r` is used to let the computer know that the expression in the quotes is a raw string and so for example although the \ character is often used as a control character, in a raw string it will be treated simply as a \ and passed through to the LaTeX reading software. This avoids the use of lots of double \ for each \. One debugging tip is that lines often finish with a double slash to denote a new line in LaTeX. It is important to add a space after the double slash in the text file as otherwise it will be concatenated onto the next line.

Some of you will know LaTeX and so the code in the source window will be familiar. It is however not essential to write an `outlatex` function for your own templates as the code is purely decorative. We will not give a crash course on LaTeX here but essentially the `aligned` environment is used to write a set of mathematical equations with the & sign denoting the place where to line up horizontally the lines and the double slashes denoting new lines. LaTeX uses the \ preceding terms to denote special characters e.g. \beta gives a Greek lowercase beta. The aligned environment is for mathematics and so if we wish to write words in normal font we enclose them in a `mbox`. With this basic knowledge you should be able to compare the source code and the maths it produces and thus see what each of the special characters is.

3.3 Writing your own first template

We haven’t at this stage explained how the `outbug` function is used to create code to fit the model. This is all generic code that is common to all templates and which we will discuss a bit more later. It is enough for now to realise that to write some basic templates simply requires writing code similar to that seen here and the Stat-JR system will do the rest of the hard work for you. We will now test your understanding by getting you to construct your own first template:
Exercise 1
It is best when starting writing templates to start from a template that works and modify it to confirm you understand what is going on. You will therefore now take the Regression1 template and construct a template for an even simpler model – a simple linear regression. To do this in the template directory copy the file Regression1.py to LinReg.py. It is also sensible to change the classname in the template.

For a linear regression we want a template with two inputs y and x – only this time x is a vector rather than a matrix i.e. there is only one predictor plus a constant. Try changing the text to ask specifically for a Y variable and an X variable for the inputs. You will need to change invars a little. Try also then simplifying the outbug and outlatex functions – you should be able to get away without needing the mmult/mmulttex functions.

In fact mu[i] should be something like alpha + beta*x[i], though if you use alpha and beta they will both need declaring as ParamScalar’s in the invars function.

When you think you have the template correctly written save it and rerun Stat-JR and test it out. If it is saved in the templates directory it will be automatically picked up. It should give similar results to Regression1 for the example shown earlier.

4 Running templates with the e-STAT engine

4.1 Algebra and Code Generation

In section 3 we have seen the code required to create a template that fits a simple template using the built-in e-STAT estimation engine. We have however hidden away many of the details. In this section we will expose a few more details, including a little section on the algebra system. Let us start by returning to the same example and show a few more screens that we have not yet exposed.

We will begin however by switching a few of the settings so that we can easier see what is going on. To do this return to the main menu by clicking on the Change buttons next to either the Template or the Dataset information at the top of the screen. Then click on the Settings button upon which you will be greeted by a settings screen where you will need to change the buttons to look as follows:
Here we have switched on standalone and also switched off optimisation. Click on the Set button when you have made the changes. Now choose **Regression1** as the Template and **tutorial** as the dataset as before and set up the inputs as follows:
Now clicking on **Next** we can choose other options from the right hand pane so firstly choose **algorithm.txt** from the right hand list and display it in a new tab.

Basically this window shows a nicely presented result of what is returned from the algebra system when it is given the model description constructed by the **outbug** method. We will look at the algebra system in a little more detail. You will see that three of the parameters (beta0, beta1 and tau) have posterior distributions that require sampling from a conditional distribution using a method called Gibbs sampling whilst two (sigma and sigma2) are simply calculated as deterministic functions of the other parameters. Finally a formula for the deviance function is also returned. In fact the algebra system returns a series of files (in xml format), one for each parameter and we can also view these (in nicely presented form) for example **tau.xml**.
Here we get the same line repeated twice as the second line shows the posterior after optimisation (which here we have switched off). Stat-JR takes these files and converts each of them into the C++ programming language so if we look at the file `modelcode.cpp` we will see the actual C++ code constructed below (note we will not go into detail as to how this is achieved):

```cpp
double ysum0 = pow(((normexam[i] - (beta0*cons[i])) - (beta1*standlrt[i])), 2) - csum0; double tsum0 = sum0 + ysum0; csum0 = (tsum0 - sum0) - ysum0; sum0 = tsum0;
```

Here after some code that is required for passing the variables back and fore from Python to C++ we see the step for tau. This is similar to that given in the algebra. One difference is that the length of normexam has it’s value (4059) substituted in. The code also uses a technique called Kahan summation and so what would have been the line

\[
dsum0 = \text{pow}\left(\left(\text{normexam}[i] - (\beta_0 \times \text{cons}[i]) - (\beta_1 \times \text{standlrt}[i])\right), 2\right);
\]

is expanded to the following

```cpp
double ysum0 = pow(((normexam[i] - (beta0*cons[i])) - (beta1*standlrt[i])), 2) - csum0;
```

to deal with potential rounding issues.

If you scroll down you will see similar code to perform the steps for the other parameters and the deviance. There are further C++ files which contain supporting routines (`supportcode.cpp` – note this
used to contain random number generators but they are now included via a library instead), perform the DIC calculation (dic.cpp) and set up proposal distributions via adaptation when using Metropolis Hastings sampling but not in this example (adapt.cpp).

When run in the usual way i.e. without switching settings to run as standalone each of these pieces of C code is compiled separately and Python code within Stat-JR pieces everything together. If as we have done we choose run as standalone and now click on Run then the software does as it suggests and creates standalone C++ files. In the current version of Stat-JR we have included parallel processing and so only one standalone file is constructed, standalone0.cpp which contains the starting values for all three chains. If we look at standalone0.cpp in a new tab we see the following:

```c++
#include <cstdlib>
#include <math>
#include <time>
#include <iostream>
#include <string>
#include <fstream>
#include <vector>
#include <random>
#include <sstream>
#include <iostream>
#include <fstream>
#include <string>
#include <limits>

if defined(_EMCC_)
#include <util/random>

class RunningStat
{
    public:
        RunningStat() : n(0) {}

    void Clear()
    {
        n = 0;
    }

    void Push(double x)
    {
        n += 1;
        // See Numerical Recipes vol 1, 3rd edition, page 292
        if (n == 1)
        {
            m = x;
            m_min = -m;
            m_max = 0;
        }
        else
```
If you view the rest of this C++ code in detail you will see that there is a chunk at the top that is common to all models but the rest of the code is mostly model specific. If you return to the Settings screen and switch back on optimisation and switch off standalone C code then repeating the model setup you can fit the model and view the code in modelcode.cpp:
Here the code is much harder to link to the algebra system as the data has been included into the model steps and any constants have thus been evaluated. You might like to compare the code for the tau step and see if you can spot the links, for example 2029.501 is 4059/2 + 0.001. Our advice is that if you are interested in understanding the C++ code and the algorithm generally then it is probably easier to switch off optimisation whereas if you want the code to run faster then switch it on. We will revisit the C++ code in later sections when we introduce the use of the precode method.

We will next look at how the algebra system converts the model statements into a set of steps in more detail.

4.2 The Demo algebraic software system

The algebra system that we have developed for the Stat-JR system (with main developer Bruce Cameron) will take a WinBUGS like input file and produce output files for each parameter giving their full conditional posterior distribution either as a known distribution with formula for each parameter or as an unknown distribution function.

We can run this algebraic processing system in isolation as it also has some nice output screens that are useful for teaching purposes. Start up the Demo.exe executable on your machine (you can find this in the Demo\bin subdirectory underneath the base directory where you extracted StatJR). This will bring up the program and loads of windows (Note that you may need to install font files on your machine if some of the symbols look incorrect):
You will need to copy the model (model.txt) for the Regression1 template onto the clipboard and then paste this into the window entitled **BUGS Input** as follows:

\[
\text{Complementary log-log} \\
\text{cloglog}(x) = y \\
\Leftrightarrow x = \exp(-\exp(y))
\]

Be careful if you are copying from the web browser as it may lose the endlines and this will then not work properly but things are OK if you copy somewhere else first. You will notice when the model code has been copied that the program takes a little while computing things and the other windows
will then change. If we select the **Graph Nodes and Deviance** window and make this full screen and choose *beta0* we get the following:

Here we see some algebraic processing and if we scroll to the bottom of the window we get:

The program decides which lines in the model specification involve *beta0*. It then finds the prior and likelihood parts before merging together to find the posterior and log posterior. It then uses its features for matching conjugate distributions to spot that the posterior for *beta0* is a normal distribution. Finally it gives the conditional posterior distribution in terms of other objects in the model. We can view *beta1* and see similarly a Normal posterior:
Finally \( \tau \) the precision has a Gamma posterior distribution:

When run from Stat-JR, the algebraic processing software then saves these three distributions in XML file format so that they can be read in later when we create code to fit the model and the distribution you see here correspond to those displayed in Stat-JR.
5 Including Interoperability

The Stat-JR package has its own new algebra system and estimation engine as illustrated in the last section. Another aspect of the package is its ability to interface with other software packages and in particular (but not exclusively) their estimation engines. This feature doesn't however come for free and translator methods that are often template specific need writing to achieve interoperability. The work here can be broken down into generic work that is built into the software and includes interfacing with the external software and managing the output received, and other work such as construction of data and script files for the external package that may be template specific and thus written by the template writer or generic as well.

In this section we will describe the (generic) Python code that is written to support interoperability and found in the packages subdirectory of Stat-JR. We will return to the regression modelling template and take a look at how we can include interoperability via an adapted template (Regression2.py). We will here describe work on three of the software packages that have been considered for interoperability, namely WinBUGS, MLwiN, and R but first we will delve a little further into the workings of the e-STAT engine and look at the file eSTAT.py.

5.1 eSTAT.py

When running a model in Stat-JR with a specific estimation engine an object is constructed of a unique class related to that engine. These objects are what pull together inputs and data, perform the estimation and store the results. The files for the various engines are found in the packages subdirectory which also contains equivalent files for use with templates not related to model estimation. The object of type eSTAT is defined in the file eSTAT.py and you will see if you access this code that it is rather long and complicated. We will not here try and go through everything here as this would only be useful for the most expert Python coders. There are however some commonalities across engines and so we will give very brief indications of what certain methods do in the template:

- The MethodInput method is present in each engine and contains the user's estimation method inputs that we saw when running the Regression1 template earlier.
- The init method is what is called after the estimation method inputs have been added and the Next button is pressed. It calls lots of other methods to perform the various tasks here including getting the algorithm from the algebra system and constructing the code for running the model.
- The applydata method is used with eSTAT to construct starting values for parameters in the model.
- The compilemodel method is used to call the (Demo) algebra system and get back algebraic steps for each parameter.
- The calcconsts method is what is run with eSTAT when optimisation is switched on to pull out terms in the algebra that are purely data and evaluate them.
- The run method is used to run the current model with the prescribed estimation settings and is called when the Run button is pressed.
- The runmore method is used when the More button has been pressed for further iterations.
- The genCPP method is used to generate the C++ code for the standalone engine.
• The `runCPP` method is used to run the estimations when standalone C++ code is selected.
• The `saveresults` method brings together the (potentially multi-chain) output and constructs the `ModelResults` and `output` chain objects.
• The `dic` method constructs code if required to calculate the DIC diagnostic for the model.

With regard engine classes in general we would expect to find a `MethodInput` method, an `init` method, a `run` method and often a `saveresults` method but also some engine specific methods. The `MethodInput` method always contains any additional engine specific inputs that are displayed on the screen. The `init` method contains the Python code to be run upon pressing the Next button prior to running and the `run` method contains the Python code to be run after pressing the Run button. The `saveresults` method, where present, is usually called from the `run` method. If the estimation method allows more iterations then there will be a `runmore` method that is called after pressing the More button. We will now look at a second template that contains further interoperability.

5.2 Regression2.py
In this section we will consider a second template – Regression2 that extends the first template by including the option to fit the same model in a variety of packages. If you look at the code in the Python file you will see that this template has identical code for the attributes defined in Regression1 but in addition has methods to allow the user to call other programs. We will begin however by looking at the engines attribute:

```python
```

Here we see that this template offers very many software packages to be used. For several packages there is simply one engine whereas for MLwiN and R which we will see later there are two engines as these packages have both classical and MCMC engines built-in. If you scroll down the file you will see additional methods: WriteMLwiNOut, REstimationInput, STATAInput, SPSSInterop, SASScript, MTBScript, SabreScript MatlabScript, and GenStatScript. Each of these methods is used to produce scripts or parts of scripts for the specific package and the main point to take here is that although some template specific coding is required, the code required is generally short functions (and in the case of WinBUGS, OpenBUGS and JAGS non-existent) and so the bulk of the work, at least for this template is done by the generic code within the package files for the engines.

5.3 WinBUGS and Winbugsscript.py
We will begin by looking at the WinBUGS package (Lunn et al., 2000) as the model code we have been creating for the Stat-JR engine has many similarities with WinBUGS code. We will begin by running the template and viewing the output. It should be noted that in order to run the WinBUGS engine Stat-JR needs to be able to find it. In the webtest directory you will find a file called settings.cfg which contains directory names for each package. For example on my machine I have:

```python
[WinBUGS]
executable=../../../WinBUGS14/WinBUGS14.exe
```

If you wish to use this option you need to either install WinBUGS in this directory or change these paths to point to WinBUGS on your machine. When you have done this restart the webtest program
so that it uses these settings and select Regression2.py from the template choices and tutorial for the dataset. Next select the following inputs:

Clicking on Next will show the Model code and mathematical representation as we saw for Regression1 with the e-STAT engine apart from that this model code has been modified slightly to be in line with standard WinBUGS code, in this case length(normexam) has been replaced with 4059 in code (Note that this template also supports the eSTAT estimation engine). We can look at the other input files required by WinBUGS for example here is the file containing initial values for chain 1:
If we next click on **Run** you will see a WinBUGS window appear on your toolbar and in the background whilst WinBUGS is fitting the model. When it finishes it will disappear and we will get the following output in the browser:

As we chose 2 chains you will also observe a green and blue output for both the chains and kernel density plots. If you look at **ModelResults** you will notice that we get results for each parameter (including the addition of the deviance and some reordering of the output). We now need to see how the connection to WinBUGS was achieved. Interestingly for the **Regression2** template you will not find any additional code to run WinBUGS within the template itself. This means that all the code is generic and will be found in the **WinBUGS.py** file within the packages directory.

As mentioned in the last section these engine files give class definitions for classes that will perform the interoperability work for specific packages and the file WinBUGS.py gives the class definition for the **WinBUGSScript** class. This class has as expected **MethodInput**, **init** and **run** methods and as WinBUGS supports running further MCMC iterations there is also a **runmore** method. The **MethodInput** method is fairly self explanatory and contains the various additional estimation method inputs required by WinBUGS. The **init** method is split into two parts written in two methods: **PrepareWBugsInputs** which is used to create in turn the three files that are needed to fit a model in WinBUGS, namely, the data, model and initial value files; **WriteScript** which creates the script file that WinBUGS uses to perform the model fitting and extraction of results etc.
It is worth looking at the first few lines of the PrepareWBugsInputs as they show the mechanism for creating the data file from within a template:

```python
# Write the data file
data = OrderedDict()
if hasattr(self.template, 'WinBUGSData'):
data = self.template.WinBUGSData()
else:
```

Here we see the first line checks whether the template has it’s own method called WinBUGSData and if not it will go through the data objects in turn to construct the data file required by WinBUGS. Basically the next bit of code interrogates all the templates objects and adds to a dictionary labelled data any objects that are data. Then the next chunk of code forms the text string datastr which will become the data file.

In many simple model scenarios, the WinBUGS model file will aside from simple substitutions be identical to the input file for the Demo algebra system and so the chunk of code dealing with model construction is fairly simple:

```python
# Model spec file
modspec='
# This file contains the model specification based on the
eSTAT system or newly defined
# check if there is a self.template.WinBUGSMod
if hasattr(self.template, 'WinBUGSMod'):
    modspec+=str(self.template.WinBUGSMod())
else :
    modspec+=str(self.template.output())

for d in data.keys():
    if isinstance(data[d], list) or isinstance(data[d],
numpy.ndarray):
        modspec = modspec.replace('length(' + d + ')',
str(len(data[d])))
    for p in self.params.keys():
        modspec = modspec.replace('length(' + p + ')',
str(self.params[p]))

    modspec=modspec.replace('ln(','log('

self.eng.inputs['model.txt'] = TextOutput(modspec, description =
'Model Specification')
```

Here again we check for template specific code, in this case a method called WinBUGSMod within the template. There are then two substitutions – strings of the form length(object) are substituted by the actual length and the log function ‘ln’ is substituted by the WinBUGS form ‘log’. The final chunk which is for initial value file formation can again be overwritten by having a method called WinBUGSInits in the template. At present it very much relies on naming of variables in terms of forming the file and we may change this in the future. For an example of a template with it’s own WinBUGS functions you might look at 1LevelMVNormal which is described later in this manual.

The WriteScript function is totally generic as it creates the script file to be run in WinBUGS and this at least at present is consistent across templates.

The run function which is run when the Run button is pressed is fairly short:
def run(self):
    self.eng.outputs['resultsIndex.txt'] = TextOutput()
    for i in range(0, self.template.EstObjects['nchains']):
        self.eng.outputs['results' + str(i + 1) + '.txt'] = TextOutput()
    self.eng.outputs['log.odc'] = TextOutput()
    self.eng.outputs['log.txt'] = TextOutput()

    self.eng.run('script.txt')

    try:
        self.saveresults()
    except:
        print 'There was a problem running the model'

The first series of lines simply define the outputs and types that will come back from WinBUGS and then the command `self.eng.run('script.txt')` actually runs WinBUGS. The last command `self.saveresults()` both extracts the numbers from the text files returned from WinBUGS and constructs the `ModelResults` object that can be viewed. We have limited this section to a broad description of the purposes of specific functions used in the interoperability and how an advanced user if required, might write their own methods for their template. Stat-JR also supports OpenBUGS (Lunn et al., 2009) and JAGS (Plummer, 2003) which are in terms of input files similar to WinBUGS. There are differences in their script files and so the files `OpenBUGS.py` and `JAGS.py` have similar but slightly differing code to account for this. JAGS also has a slightly different format for data files which `JAGS.py` takes care of.

We next look at MLwiN.

### 5.4 MLwiN

MLwiN (Rasbash et al. 2009) is another package with MCMC functionality but which can also fit multilevel models using classical statistical methods. In Stat-JR for this template we offer the option of fitting in MLwiN using either approach. Having seen how WinBUGS links into Stat-JR we will now show the similarities and differences in how MLwiN links in. The first observation is that MLwiN doesn’t use a model description language like Stat-JR or WinBUGS. It is also more restrictive in terms of which models it can fit which means that it will not be available for all templates but many of the templates we have written thus far fit models that MLwiN can also fit. Although MLwiN has a GUI user interface which is typically how users will use it, it also has a macro language and it is this language that we have to make use of when writing interoperability code for Stat-JR. So as with WinBUGS we need to tell Stat-JR where to find MLwiN and this is found in the `settings.cfg` file, for example:

```
[MLwiN]
executable=../../../MLwiN v2.26/i386/mlnscript.exe
```

Let us demonstrate using MLwiN and MCMC for the `tutorial` dataset and `Regression2` template. Here select the `template` and `dataset` and next choose inputs as follows:
The first thing to note is that the two approaches for MLwiN have their own engine name, and we will see later their own python files in the package directory. A further thing to note is that MLwiN only offers single chains for MCMC. However if you run it from Stat-JR you can get the illusion of multiple chains as Stat-JR will run it several times, once for each chain. Currently each chain has the same initial values but different random number seed but in the future we hope to allow different starting values as well. Clicking on the Next button we see the following:
In the left pane is the script file for one of the chains and this is a series of commands in the MLwiN macro language. In order to minimise the amount of template specific code this macro is rather long containing many conditional statements. There are 3 almost identical macros that only vary in a single line where the MCMC seed is set and a datafile, `datafile.dta` which is called in the second line of each macro. Clicking on Run will fire off the three instances of MLwiN and bring back the output as follows (after changing the right hand list to show ModelResults):

\[\begin{align*}
\text{Note input data file} \\
\text{DATA} 'datafile.dta' \\
\text{Note set up the model} \\
\text{EXEC 'newname'} \\
\text{DATA 1 'LD'} \\
\text{ADD 'l scanning'} \\
\text{DATA 1 'l scanning'} \\
\text{PARAM 6 'l scanning'} \\
\text{ADD 'c om' } \\
\text{ADD 'w scanning'} \\
\text{Note join the model} \\
\text{EXEC 1} \\
\text{READ 1} \\
\text{START} \\
\text{Note find model type for MCMC command} \\
\text{COORD 1 b100} \\
\text{SWITCH b100} \\
\text{CASE 4:} \\
\text{Note unordered multinomial} \\
\text{SET b100 6} \\
\text{CASE 5:} \\
\text{CASE 6:} \\
\text{Note ordered multinomial} \\
\text{SET b100 7} \\
\text{CASE 7:}
\end{align*}\]
Apart from the speed of estimation (which is much quicker than Stat-JR and WinBUGS) the results are very similar. Note that the fixed effect names here include the variable they are associated with.

We could as an alternative run the model not using MCMC by clicking on Change Estimation Settings and choosing the following and clicking Next:
Note that as the non-MCMC engines do not create a datafile of the output that question is not asked. Clicking on Run you get almost instantaneous answers:

You will notice here that the results produced are simply point estimates and standard errors as the method doesn’t construct chains. We also do not see the plots that we get with MCMC methods. As we mentioned earlier there are two engines and hence two files in the packages directory, MLwiNMCMC.py and MLwiNIGLS.py. We will look briefly at MLwiNMCMC.py which is currently, apart from eSTAT.py and a couple of associated eStat engines (eSTATPred and CustonC) the biggest file in the directory.

As usual the file defines a class, this time for an MLwiNMCMC object. The class has the usual MethodInput, init and run (and runmore) methods. The init method will construct the dataset and script files for running in MLwiN. It begins

```python
def init(self):
    self.itnum = int(self.template.EstObjects['iterations'])
    self.input()
    mlnscript = '''
    NOTE input data file
    RSTA 'datafile.dta'
    NOTE Set up the model
    ```
```
Here we see a call to the `input` method which calculates which variables in the dataset are needed for the model fitting and then constructs the data file to send to MLwiN. Next we see the beginning of the script file being constructed in the object `mlnscript`. The call to the template specific code is done via the line `mlnscript += self.template.WriteMLwiNOut(self)` and this forms the top part of the macro file. The next five or six lines simply substitute objects into the code, so for example the `Regression2` code is as follows:

```python
def WriteMLwiNOut(self, MLwiNOut):
    mlnscript = '''
RESP "$y"
IDEN 1 "id"
ADDT "levres"
SETV 1 "levres"
FPAR 0 "levres"
% for i in range(len(x)):
  ADDT "$x[i]"
% endfor
'''
    return mlnscript
```

and obviously we wish to substitute for example `$y` with `normexam` in this example etc. If you look back at the screen shot with the macro file produced you will see these lines clearly. The final for loop in `init` will loop over the requested number of chains producing the full macro code including the bulk of the macro that is constructed by the `script` method (which takes as an argument the chain number in order to set the random seed appropriately). This method is very long but is really just constructing generic macro code, in an analogous way to the chunk of generic C code used in eSTAT.

The `run` method simply runs MLwiN using the constructed datasets and macro files and then calls the `output` method that creates the output objects that are available in Stat-JR and the `saveresults` method which creates the `ModelResults` object. Again we omit details of precisely how these work as they are generic code and not template specific.

The `MLwiNIGLS.py` file has a similar form to `MLwiNMCMC.py` except the macros are slightly shorter and the objects produced in the `output` and `saveresults` methods are different. We will leave MLwiN here and move onto another package with some functionality for the use of both MCMC and classical estimation methods, R.
5.5 R

R (Development Core Team, 2011) is a general purpose statistics programme that consists of a framework of interlinking statistical commands that are known as packages. The R installation consists of a base package containing many of the standard statistical operations and to this can be added user written packages. For our Regression2 template we will utilise the glm function which requires the MASS package when performing classical inference. For MCMC methods we use the MCMCglmm package (Hadfield 2009), a user-defined function that can fit many models using MCMC. As with the earlier programs we need to include details of the location of R in settings.cfg on our machine prior to running webtest. On my machine this is as follows:

[R]
executable=../../R-2.14.1/bin/i386/R.exe

We can again first run the Regression2 template to see what it gives for R so choose Regression2 as the template and tutorial as the dataset and then input the following:

Clicking on Run will fire off R in the background window and then give the following output:
Here by default we see the dataset sent to R in the left pane and a quantile-quantile plot, which is one of the outputs produced, in the right hand pane. We can select `script.R` in the left pane and `ModelResults` in the right pane to get the following:

It is also possible to view the full log file from R and a plot of residuals against fitted values in the right hand pane as this is also created by the R script. If we wish to instead use MCMC estimation we
Clicking on **Next** and then **Run** will give the following outputs upon running and changing the left hand pane to *script.R* and the right hand pane to *ModelResults*:

It should be noted that *MCMCglmm* is a single chain package and so *Stat-JR* does not give the option for multiple chains here. Let us now look at how interoperability is performed in the code. At present R interoperability is possibility the opposite extreme to MLwiN interoperability. In the packages directory there are files for each R package, here we are interested in *RMASS.py* and *RMCMMCglmm.py* which are quite short files defining classes for each object and then in the *Regression2* template file there is a longer function *RestimationInput* which is used to construct the R script for model fitting.

Looking firstly at *RMASS.py*, it has a **MethodInput** method but this doesn’t have any additional inputs as they are not required for this package. The **inits** method code begins as follows:
def init(self):
    self.WriteData()

    self.RScript = 'library(foreign)\n'  
    self.RScript += 'mydata<-read.dta("datafile.dta")\n'
    # print summary of the data
    self.RScript += 'summary(mydata)\n'
    self.RScript += 'PACKages<-as.character(data.frame(installed.packages())$Package)\n'
    self.RScript += 'test<-("MASS" %in% PACKages)\n'
    self.RScript += 'if (!test){\n'
    self.RScript += 'install.packages("MASS",repos="http://cran.r-project.org")\n'
    self.RScript += '}\n'
    self.RScript += 'library(MASS)\n'

    Here we see a call to the WriteData method that constructs the data file that needs to be sent to R and is similar to the WriteData method used with MLwiN. Then we begin to construct the script file for R which here initially involves loading up the data and installing the R packages required if they are not already present. Next is the call to the template specific code via the RestimationInput method as shown below:

    script = self.template.REstimationInput()
    #args = {'data': self.template.data}
    args = {}
    for k in self.template.objects.keys():
        args[k] = self.template.objects[k]
    for k in self.template.EstObjects.keys():
        args[k] = self.template.EstObjects[k]

    try:
        self.RScript += MakoTemplate(script).render(**args)
    except:
        self.RScript += exceptions.text_error_template().render() 

    Here the substitutions into the template specific code are made and it is added to the script file. Finally some more generic code to interrogate the output produced by R and create datafiles to be used as output objects in Stat-JR is written.

    # Assume that there is a glm object called MyModel
    self.RScript += 'stats <- list()\n'
    self.RScript += 'stats$deviance <- myModel$deviance\n'
    self.RScript += 'stats$nulldeviance <- myModel$null.deviance\n'
    self.RScript += 'stats$aic <- myModel$aic\n'
    self.RScript += 'stats$converged <- myModel$converged\n'
    self.RScript += 'stats$iter <- myModel$iter\n'
    self.RScript += 'write.dta(data.frame(stats), file="stats.dta")\n'
    self.RScript += 'write.dta(data.frame(rbind(myModel$coefficients, sqrt(diag(vcov(myModel))))), file="estimates.dta")\n'
    self.RScript += 'write.dta(data.frame(myModel$residuals), file="residuals.dta")\n'

    self.eng.inputs['script.R'] = TextOutput(self.RScript, description = "Script to run model")
Looking at the template specific code in the template Regression2 we see the following code at the beginning of the method:

```python
def REstimInput(self):
    script = ''
    import re
    from EStat.Templating import Rmmult

    RMod = y + ' ~ ' + Rmmult(x)
    RMod = re.sub('s*','',RMod)
    ## always remove the intercept and let the user add the constant or not
    RMod += '-1'
    ## ensure no two consecutive + are present
    RMod=re.sub('++','+',RMod)
    data = {}
    for var in datafile.variables.keys():
        data[var] = datafile.variables[var]['data']
%
```

This first chunk simply defines the model statement into a variable called RMod and then the code goes on to specific code for the MASS package.

```python
% if Rpackage == 'MASS':
    % Rfamily='gaussian(identity)'
    myModel< glm(${RMod},data=mydata,family=${Rfamily})
    summary(myModel)
    png("ResivsFitted.png",width=733,height=550)
    plot(myModel,1)
    dev.off()
    png("qqNorm.png",width=733,height=550)
    plot(myModel,2)
    dev.off()
% endif
```

This second chunk is the code specific to MASS and the glm function and consists of the call to that function followed by calls to two plotting functions. There is then a large chunk of code specific for the MCMCglmm function which essentially runs that code and then the script is returned. So we should now see how the full script file for R is formed and what needs to be placed in the template.

The `run` method that is executed when the `Run` button is pressed is in this case very short and basically consists of a command to run the script followed by a call to the `saveresults` method.

```python
def run(self):
    self.eng.run('script.R')
    try:
        self.saveresults()
    except:
        print 'There was a problem running the model'
```

The `saveresults` method itself is quite short and creates the `ModelResults` object. Here it involves interrogation of the two output files from R, `estimates.dta` and `stats.dta` to extract the appropriate numbers for the `ModelResults` object.
def saveresults(self):
    results = ModelOutput()

    dta = self.eng.outputs['estimates.dta']
    for var in dta.variables.keys():
        results.add(var, 'est', dta.variables[var]['data'][0])
        results.add(var, 'se', dta.variables[var]['data'][1])

    statsdta = self.eng.outputs['stats.dta']
    for stat in statsdta.variables.keys():
        results.add('model', stat, statsdta.variables[stat]['data'][0])

    self.eng.outputs['ModelResults'] = results

The RMCMCglmm.py file performs the equivalent operations when this engine is called from Stat-JR and the same methods are present. The code is slightly more involved and in this case the MethodInputs method has inputs to tell R how long to run MCMC for. There are also other differences in the functions to account for the method being MCMC and hence returning chains to be summarised. These methods are however similar to those for MLwiN and WinBUGS and we will not detail them here.

Finally here we should point out that there are R specific templates that perform other functions. For example PlotsViaR.py which allows the user to use the R lattice graphical functions from within Stat-JR. These templates generally have a very simple structure: the engine attribute is set to Rscript, the invars attribute gives all the inputs required by the template and the REstimationInput method is used to construct a script file to be used in R to perform the required operations. There is therefore a file in the packages directory named RScript.py which handles such templates. It has a very simple structure and contains the usual methods we have become familiar with when looking at the files in this directory. It basically contains code to construct the data file for R, call the template code to construct the script and then run the script and store the output objects. We will give no further details here but finish by mentioning the other packages supported by Stat-JR.

5.6 Other packages

For our Regression2 template you will see that we also offer interoperability with other packages: Stata, SPSS, SAS, Minitab, SABRE, Matlab, Octave and GenStat. Each of these packages will have a python file in the packages directory which deals with getting the data in the correct format for the package, calling the template specific code for the package and interrogating the output files received back by Stat-JR from the package. Some packages will have two python files in the packages directory, for example for STATA we have files StataModel.py and StataScript.py, and here the distinction is between calls from templates that fit models and thus need to create a ModelResults object and templates that use other functionality e.g. graphs from within the package.

For our Regression2 template you will see that for most packages the code is quite short, for example for STATA, which is perhaps the longest we have:

def STATAInput(self):
    script = ''
    <\% family='gaussian' \%>
    <\% link='identity' \%>
    glm ${y} \$
    % for p in x:
    ${p} \$

41
and here the code not only fits a model but also produces two plots. This ends our whirlwind description of the interoperability features in the Stat-JR program. The interoperability features are still a work in progress and although they are present in many of the templates that we will describe in later sections we will not be going into details on this aspect of these templates. The interested reader can look at these templates and see how they perform interoperability and try writing their own interoperability code for their own templates.

6 Input, Data manipulation and output templates

The Stat-JR system does not simply consist of templates for fitting models to datasets. There are in addition templates that allow the user to input their own datasets, manipulate datasets and plot features of datasets. In many ways these templates are much simpler to write and understand. We will here look at a few examples of the templates along with their code and explain how they fit into the webtest interface.

6.1 Generate template (generate.py)

Our first template to look at is used for generating columns to add to a dataset. These columns can be constants, sequences, repeated sequences or random numbers. As this template doesn’t have any exciting outputs we will not see much happen after execution. Let’s look at an example of adding a vector of uniform random numbers to the tutorial dataset.

We firstly choose the Generate template from the template list on the main window and press the Use button. Next we press the Run button to run the template. The template will look as follows:
Now we select *random* for the output column name and stick with *Uniform Random* for the type. After clicking **Next** we are asked for a name of output results, and here if we enter *tutorial* the new column will be appended to the dataset and the tutorial dataset (in memory) will have an additional column. If we choose a new name then a new dataset containing all the columns from *tutorial* along with this new dataset will be formed (in memory) and *tutorial* will persist without the new column. Pressing **Next** will finish the inputs and Pressing **Run** will run the template and the **Run** button will then disappear.
To see what the template has done if you open the tutorial object in a new tab you will see a new column labelled random to the right of the dataset:

<table>
<thead>
<tr>
<th>row</th>
<th>school</th>
<th>student</th>
<th>normexam</th>
<th>cons</th>
<th>standart</th>
<th>grade</th>
<th>shcol</th>
<th>year</th>
<th>schurband</th>
<th>random</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.261524</td>
<td>1</td>
<td>0.05909</td>
<td>1</td>
<td>1</td>
<td>0.16075</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0.134067</td>
<td>1</td>
<td>0.30302</td>
<td>1</td>
<td>1</td>
<td>0.16075</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>-1.72328</td>
<td>1</td>
<td>-1.36458</td>
<td>0</td>
<td>1</td>
<td>0.16075</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>0.96786</td>
<td>1</td>
<td>0.30302</td>
<td>1</td>
<td>1</td>
<td>0.16075</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>5</td>
<td>0.34544</td>
<td>1</td>
<td>0.371105</td>
<td>1</td>
<td>1</td>
<td>0.16075</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>6</td>
<td>1.7136</td>
<td>1</td>
<td>2.18544</td>
<td>0</td>
<td>1</td>
<td>0.16075</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>7</td>
<td>1.03961</td>
<td>1</td>
<td>1.11162</td>
<td>0</td>
<td>1</td>
<td>0.16075</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>8</td>
<td>-0.129083</td>
<td>1</td>
<td>-1.0397</td>
<td>0</td>
<td>1</td>
<td>0.16075</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>9</td>
<td>0.239378</td>
<td>1</td>
<td>0.329261</td>
<td>1</td>
<td>1</td>
<td>0.16075</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>10</td>
<td>-1.21919</td>
<td>1</td>
<td>-1.44723</td>
<td>0</td>
<td>1</td>
<td>0.16075</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>11</td>
<td>2.40669</td>
<td>1</td>
<td>2.43739</td>
<td>0</td>
<td>1</td>
<td>0.16075</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>12</td>
<td>0.61729</td>
<td>1</td>
<td>2.06797</td>
<td>0</td>
<td>1</td>
<td>0.16075</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>13</td>
<td>-1.83669</td>
<td>1</td>
<td>0.40409</td>
<td>0</td>
<td>1</td>
<td>0.16075</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>14</td>
<td>-0.129083</td>
<td>1</td>
<td>-1.0397</td>
<td>0</td>
<td>1</td>
<td>0.16075</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>15</td>
<td>2.03012</td>
<td>1</td>
<td>2.52004</td>
<td>0</td>
<td>1</td>
<td>0.16075</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>16</td>
<td>1.24033</td>
<td>1</td>
<td>1.11147</td>
<td>1</td>
<td>1</td>
<td>0.16075</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Examining the code it is first worth noting that the template has

engines = ['PyScript']

This tells Stat-JR that this template is not a model template and therefore needs to be treated differently. The template has an `invars` function as shown below:

```python
invars = '''
outcol = Text('Output column name: ')
#carry = DataMatrix('Carried Data: ')

if type == 'Binomial Random':
    prob = Text('Probability')
    numtrials = Integer('Number of Trials')

if type == 'Chi Squared Random':
    degreefree = Integer('Degrees of Freedom')

if type == 'Gamma Random':
    shape = Text('Shape')

if type == 'Poisson Random':
    exp = Text('Expectation')

if type == 'Constant':
```

44
Here we see that there are two main attributes, a name for the column to add (outcol) and a type of column to generate. Depending on the type there may be additional attributes and these are catered for through a set of if statements in Python. So for example if we want a constant column we will have an additional attribute, value which gives the value of the constant. Note that the length of the vector is controlled by the lengths of the columns already in the dataset, as a dataset is currently restricted to be a set of columns of equal length.

As this template is not a model template there is no outbug or outlatex attributes instead the computations are performed within a method called pyscript which basically performs the required calculation in Python and adds the column to the output. The method code is as follows:

```python
import numpy
import EStat
from EStat.Templating import *
from EStat.DTAFile import DTAFile

def PyScript(self):
    script = ''
    retval = DTAFile()
    retval.nobs = datafile.nobs
    for k in datafile.variables.keys():
        retval.addvariable(k, data = datafile.variables[k]['data'])

    datalen = datafile.nobs

    if type == 'Uniform Random':
        outvar = numpy.random.uniform(size = datalen)
    elif type == 'Binomial Random':
        outvar = numpy.random.binomial(float(numtrials), float(prob), size = datalen)
    elif type == 'Chi Squared Random':
        outvar = numpy.random.chisquare(float(degreefree), size = datalen)
    elif type == 'Exponential Random':
        outvar = numpy.random.exponential(size = datalen)
    elif type == 'Gamma Random':
        outvar = numpy.random.gamma(float(shape), size = datalen)
    elif type == 'Normal Random':
        outvar = numpy.random.normal(size = datalen)
    elif type == 'Poisson Random':
        outvar = numpy.random.poisson(float(exp), size = datalen)
    elif type == 'Constant':
        outvar = numpy.ones(datalen) * float(value)
    elif type == 'Sequence':
```
outvar = numpy.arange(int(start), int(start) + (datalen * int(step)) - 1, int(step))
if type == 'Repeated sequence':
    outvar = numpy.array(list(numpy.repeat(numpy.arange(1, int(max) + 1),
        int(repeats))) * (datalen / int(max) * int(repeats)))

retval.addvariable(str(outcol), data = outvar)
outputs[str(outdata)] = retval
'''

Here although the function is long this is in the main due to the many if statements to cope with each type of vector to be generated. So for example if we wanted a vector of Uniform random numbers to be stored in random then the only lines to be executed are:

retval = DTAFile()
retval.nobs = datafile.nobs
for k in datafile.variables.keys():
    retval.addvariable(k, data = datafile.variables[k]['data'])
datalen = datafile.nobs
outvar = numpy.random.uniform(size = datalen)
retval.addvariable(str(outcol), data = outvar)
outputs[str(outdata)] = retval

Here the code calculates the length of vector in the fifth line, uses the numpy random generator in the next line to create the column of numbers in outvar. In the remaining lines we link the column into the dataset and finally return the dataset to the output name we gave as an input.

**Exercise 2**

Try modifying this template so that it only offers the random number generators. Try expanding the inputs so for example the Normal random generator will allow a mean and a variance, the Gamma has a scale parameter and the exponential has a rate parameter.

### 6.2 Recode template (recode.py)

The Generate template allows the user to add new columns to their existing dataset. There are many templates that expand or manipulate a dataset and we will here look at a second template, the Recode template. The Recode template as the name suggests recodes values – in this case recoding values within a contiguous range to a specific new value. This can be useful for creating categorical values – although this might involve several repeated uses of the Recode template!

We will demonstrate this with the tutorial dataset and look at recoding the school gender (schgend) column. In the original dataset schgend takes values 1 for a mixed school, 2 for a boys school and 3 for a girls school. We might want to recode this to take values 1 for mixed and 2 for single sex i.e. convert the 3s for girls schools to 2s.

To do this first we select Recode from the template list and hit Use followed by the Run button. Next we select schgend from the list of columns and select the other options as below:
Clicking on Run will run the template and then returning to the main menu (by clicking on Change) and clicking on the View Summary link shows a data summary:
Here we see that schgend now goes from 1 to 2 as expected. Let us now look at the code for this template. As with generate this template has an invars attribute and a Pyscript method. These are both quite short:

```python
invars = '''
incol = DataVector('Input column name: ')
rangestart = Text('Start of range: ')
rangend = Text('End of range: ')
newval = Text('New value: ')
#carry = DataMatrix('Carried Data: ')
outdata=Text('Name of output dataset: ')
'''
```

Here the invars attribute gets the five inputs that we saw when running the template. Next the Pyscript method:

```python
def PyScript(self):
    script = ''
    import numpy
    import numexpr

    import EStat
    from EStat.Templating import *
    from EStat.DTAFile import DTAFile

    retval = DTAFile()
    retval.nobs = datafile.nobs
    for k in datafile.variables.keys():
        retval.addvariable(k, data = datafile.variables[k]['data'])
    #for i in carry:
    #    retval.addvariable(i, data = datafile.variables[i]['data'])

    # Copy data into numpy array for processing
    var = numpy.array(datafile.variables[incol]['data'])
    var[(var >= float(rangestart)) & (var <= float(rangeend))] = float(newval)
    retval.addvariable(incol, data = var)
    outputs[str(outdata)] = retval

    return script
```

After some importing lines, the Pyscript method firstly copies the original column to the object var and then performs the recoding by finding the values in the original column within the correct range and then replacing them with the newval. Note the >= and <= operators mean that the range is inclusive of its end points. Finally when var is modified it is then linked back to the input column and the dataset is returned.

**Exercise 3**

This template applies the recoding by copying the recoded column over itself. As an exercise, try modifying the template so that it will place the recoded column into a new location i.e. have another name that is where to output the column to. Note the code for Generate should help here.
6.3 AverageandCorrelation template

Another template that one might consider using prior to fitting a model is the AverageandCorrelation template. This template will give either some summary statistics (including the averages) for a series of columns or the correlation matrix for a set of columns.

The template has a very short invars attribute:

```python
invars = ''
op = Text('Operation: ', ['averages', 'correlation'])
vars = DataMatrix('Variables: ')
''
```

Here op allows the user to choose between averages and correlations whilst vars stores which columns to perform the operation on. This template again uses the PyScript method but this time creates an output called table which will give the averages or correlations in tabular form.

The code for PyScript is as follows:

```python
def PyScript(self):
    script = '''
import numpy
import EStat
from EStat.Templating import *

tabout = TabularOutput()
if op == 'averages':
    tabout.column_headings = ['name', 'count', 'mean', 'sd']
    for i in range(0, len(vars)):
        var = numpy.array(datafile.variables[vars[i]]['data'])
        tabout.add_row(vars[i], [len(var), var.mean(), var.std()])

if op == 'correlation':
    invars = numpy.empty((len(vars), datafile.nobs))
    for i in range(0, len(vars)):
        invars[i, :] = datafile.variables[vars[i]]['data'][:]
    corrs = numpy.corrcoef(invars)
    tabout.column_headings = ['name']
    for j in range(0, len(vars)):
        tabout.column_headings.append(vars[j])
        for i in range(0, len(vars)):
            row = []
            for j in range(0, len(vars)):
                row.append(corrs[i, j])
            tabout.add_row(vars[i], row)

outputs['table'] = tabout
'''
    return script

You will see here separate chunks of code for averages and correlations. The average code basically initializes a table output with column heading and then loops through the columns in vars setting each in turn as a numpy array stored in var. An array of text strings are then constructed and added to the tabular output, tabout, and here we are utilising the len function to get the number of data items and the built in numpy functions mean and std to get the mean and standard deviation respectively.

49
The correlation code is slightly longer, we here firstly need to construct the data as a matrix `invars` from which we can construct the correlations (`corrs`) by a call to the `numpy.corrcoef` function. Then we again format the output nicely into `tabout`.

The line `outputs['table'] = tabout` creates the table object which is then included in the output object list. If we consider using this template with the `tutorial` dataset we first need to `Use` and `Run` it to get the default screen:

![StatJR Demonstrator - Mozilla Firefox](image)

We can now try this with some of the variables and in turn averages and correlation. It is worth noting here that you need not fill in the output results box as this isn’t used by this template. Here is an example of averages:
and the correlations for the same four variables:
Exercise 4
Why not try and add the option to this template to give the standard error of the mean and also to allow the template to output both averages and correlations together for the same variables. Remember to rename the template first!

6.4 XYPlot template
Our final template in our whistle-stop tour of non-model templates is a graphing template. Python has excellent graphing facilities and so we have created a few very basic graphing templates that demonstrate some of these facilities. The xyplot template basically allows the user to plot one or more Y variables against an X variable on the same plot.

The template has an invars attribute as shown below:

```python
invars = ''
yaxis = DataMatrix('Y values: ')
xaxis = DataVector('X values: ')
''
```

Here we have two inputs, the various Y variables and the corresponding X variable to plot against. For a graph template we once again use the PyScript method but this time the method constructs an object called graphxy which is in fact a .png image file and is constructed by a function called ImageOutput.

The PyScript code is as follows:

```python
def PyScript(self):
    script = ''
    import subprocess
    import os

    import numpy
    import numexpr

    from matplotlib.figure import Figure
    import matplotlib.pyplot as plt
    from matplotlib.backends.backend_agg import FigureCanvasAgg

    import EStat
    from EStat.Templating import *

    fig = Figure(figsize=(8,8))
    ax = fig.add_subplot(100 + 10 + 1, xlabel = str(xaxis))
    for n in yaxis:
        ax.plot(datafile.variables[xaxis]['data'],
                datafile.variables[n]['data'], 'x', label = n)
    ax.legend()
    canvas = FigureCanvasAgg(fig)
    canvas.print_figure(os.path.join(directory, 'graphxy.png'), dpi=80)
    outputs['graphxy'] = ImageOutput(filename='graphxy.png', directory =
    directory)
    '
    return script
```

Here we have to firstly import lots of Python libraries in order to call the graphics functions. The function we are using is the Figure function from the matplotlib package. We then make a blank plot sticking on the axes labels before looping over the y variables and plotting their points. The 'x' is the
symbol to be plotted for each plot. The last four lines are used to store the plotted figure as a .png file and the full pathname for this file is returned by the function.

To see this template in action we will pick it from the webtest main screen (along with the tutorial dataset) and we will be greeted by the following in the browser:

![Screenshot of Stat-JR Demonstrator interface](image)

Perhaps the simplest plot here would be to plot normexam against standlrt which you can try yourself. Here we illustrate instead the use of more than one y variable by making the following selections:

![Screenshot of Stat-JR Demonstrator interface](image)

Clicking on the Run button and choosing to display in a separate tab gives the following graph:
Here we see plotted the actual intake scores for each pupil against school number in green and the school average in blue.

**Exercise 5**

Simplify this template to only allow a single y variable. Try adding a main title to the graph and varying the symbol and colours – maybe make this an option for the user to choose. Remember to rename the template before you start!

---

**7 Single level models of all flavours – A logistic regression example**

We have so far met two model templates, *Regression1* which could be used to fit normal response multiple regression models in the Stat-JR built in MCMC engine and *Regression2* which allowed the same models to be fitted in other statistics packages. We will now look at a generalisation of these templates, *1LevelMod* that allows other response types including Binomial and Poisson responses. This template will illustrate the use of conditional statements within the *invars* and *outbug* functions.

We will begin by looking at the template in action in Stat-JR. The template should be able to fit all the models that *Regression1* fits and so you could test the earlier regressions but here we will look at a logistic regression. So from the main browser screen we need to set the *template* to be *1LevelMod* and the *dataset* to be *bang*, our example binary response dataset taken from the 1988 Bangladeshi Fertility Survey. Clicking on **Run** gives the following output in the browser:
We will now set up the various inputs and the screen will look as follows:

We are here fitting a logistic regression to the response variable which is whether the women in Bangladesh in the dataset use contraception or not. We are regressing this against age and using the Stat-JR built-in MCMC engine with some default settings for estimation.
Clicking on **Next** and selecting **equation.tex** in the right hand list will display the model:

\[
\text{\texttt{\textbackslash \texttt{makefile}}}\\
\text{\texttt{\textbackslash \texttt{\texttt{\\n}}}}\\
\text{\texttt{\texttt{\texttt{\\n}}}}\\
\text{\texttt{\texttt{\\n}}}}
\]

We will next click on **Run** to run the model and then get the following results by selecting **ModelResults** from the right hand list:
Here we see that the age coefficient is positive and significant meaning that older women are more likely to use contraceptives. So we now want to look at the template to see what the code looks like. We will only concern ourselves with the Stat-JR built in engine here and so will not look at how the template works with interoperability as this will be an extension of the code for Regression2 in section 6.

7.1 Invars
The code for invars is as follows:

```python
invars = '''
y = DataVector('response: ')
D = Text('specify distribution: ', ['Normal', 'Binomial', 'Poisson'])
if D == 'Binomial':
    n = DataVector('denominator: ')
    link = Text('specify link function: ', ['logit', 'probit', 'cloglog'])
if D == 'Poisson':
    link = Text(value = 'ln')
    offset = Boolean('Is there an offset: ')
    if offset:
        n = DataVector('offset: ')
if D == 'Normal':
    tau = ParamScalar()
    sigma = ParamScalar()
    sigma2 = ParamScalar()
    x = DataMatrix('explanatory variables: ')
    beta = ParamVector(parents = [x], as_scalar = True)
'''
```

Compared to Regression1 you will see that we have introduced an input $D$ for distribution and that we introduce conditional statements (if statements). The distribution $D$ is defined as a Text input and you will see that there are a limited number of choices given as a second argument to the statement. The webtest program will treat this as a pull-down list input with the limited number of choices populating the list.

As we saw in our example when fitting a Binomial model we introduce additional inputs $n$ – the denominator column and $link$ a Text based input to indicate the link function. We also see that for non-normal models there is no level 1 variance and so the quantities $tau$, $sigma$ and $sigma2$ are not included.

7.2 Engines
This template allows many estimation engines as shown below:

```python
engines = ['eSTAT', 'WinBUGS', 'OpenBUGS', 'JAGS', 'MLwiNMCMC',
'MLwinNIGLS', 'RMASS', 'RMCMCglm', 'RMCMCpack', 'StataModel', 'SPSSModel',
'SASModel', 'RINLA', 'rstan']
```

when we originally wrote Stat-JR each template had it’s own inputs for these engines defined in a Methodinput function but now these are generic inputs and so simply by including an engine here, Stat-JR knows which inputs to use.

7.3 Outbug
The outbug attribute now also contains conditional statements as shown below:
outbug = '''
model{
    for (i in 1:length(${y})) {
        ${y}[i] ~ \\
        % if D == 'Normal':
        dnorm(mu[i], tau)
        mu[i] <- \\
        % endif
        % if D == 'Binomial':
        dbin(p[i], ${n}[i])
        ${link}(p[i]) <- \\
        % endif
        % if D == 'Poisson':
        dpois(p[i])
        ${link}(p[i]) <- \\
        % endif
        % if offset:
        ${n}[i] + \\
        % endif
        % endif
        ${mmult(x, 'beta', 'i')}
    }

    # Priors
    % for i in range(0, x.ncols()):
    beta${i} ~ dflat()
    % endfor
    % if D == 'Normal':
    tau ~ dgamma(0.001000, 0.001000)
    sigma <- 1 / sqrt(tau)
    sigma2 <- 1 / tau
    % endif
}
}

Basically in the outbug method, conditional statements are started by a %if and the code to be conditionally executed is ended by a %endif. The conditional statements can be hierarchical for example the line
% if offset:
is within another %if statement and now the %endif will correspond to the latest %if. In our example we have D == 'Binomial' and so the code simplifies to:
outbug = '''
model{
    for (i in 1:length(${y.name})) {
        ${y}[i] ~ \\
        dbin(p[i], ${n}[i])
        ${link}(p[i]) <- \\
        ${mmult(x, 'beta', 'i')}
    }

    # Priors
    % for i in range(0, x.ncols()):
    beta${i} ~ dflat()
    % endfor
}
'''

and as we demonstrated for Regression1 we can fill in the $ calls and unwind the %for loop and the $mmult function to get the code we saw in the example output window.
7.4 Outlatex

Finally the outlatex method now also contains conditional statements.

```latex
outlatex = r'''
\begin{aligned}
%if D == 'Normal':
\mbox{$(y)_i \sim N(\mu_i, \sigma^2)$} \\
\mu_i &= 
%endif
%if D == 'Binomial':
\mbox{$(y)_i \sim \text{Binomial}(n_i, \pi_i)$} \\
\mbox{$(\text{link})$} &= 
%endif
%if D == 'Poisson':
\mbox{$(y)_i \sim \text{Poisson}(\pi_i)$} \\
\mbox{$(\text{link})$} &= 
%endif
%if offset:
\mbox{$(n)_i +$} \\
%endif
%endif
$\text{mmulttex}(x, r'\beta', i)$) \\
%for i in range(0, len(x)):
\beta_(i) \propto 1 \\
%endfor
%if D == 'Normal':
\tau \sim \text{Gamma}(0.001,0.001) \\
\sigma^2 = 1 / \tau 
%endif
\end{aligned}
'''
```

and as with the outbug function we achieve conditional operations via the %if and %endif pairs.

Again for our example we can strip out the conditionals to get

```latex
outlatex = r'''
\begin{aligned}
\mbox{$(y)_i \sim \text{Binomial}(n_i, \pi_i)$} \\
\mbox{$(\text{link})$} &= \\
%for i in range(0, len(x)):
\beta_(i) \propto 1 \\
%endfor
\end{aligned}
'''
```

If you look at the code you will see other functions for the various other software packages but we will not discuss these here.

**Exercise 6**

Convert the more general 1LevelMod template into a specific logistic regression template. To do this copy 1LevelMod.py to 1LevelLogit.py and simply remove the conditional statements and additional options so that the template only allows the user to fit logistic regression models. You can check the template works by attempting the example given in the section with your new template.
8 Including categorical predictors

The template 1LevelMod can fit many response types but treats all predictor variables as if they are continuous. This means that if we have a categorical predictor, as in our example we have with school gender, we will need to perform some data manipulation to construct the dummy variables that are used for a categorical predictor in order to use this template. We will here look at an alternative template that has built in functionality for constructing these categorical variables within the template. This template is called 1LevelCat and we will first look at its invars function to see how it gets the user to input the model structure before demonstrating its use on the tutorial dataset.

The invars function is as follows:

```python
invars = '''
y = DataVector('response: ')
D = Text('specify distribution: ', ['Normal', 'Binomial', 'Poisson'])
if D == 'Binomial':
    n = DataVector('denominator: ')
    link = Text('specify link function: ', ['logit', 'probit', 'cloglog'])
if D == 'Poisson':
    link = Text(value = 'ln')
    offset = Boolean('Is there an offset: ')
    if offset:
        n = DataVector('offset: ')
if D == 'Normal':
tau = ParamScalar()
sigma = ParamScalar()
sigma2 = ParamScalar()
x = DataMatrix('explanatory variables: ')
for var in x:
    context[var + '_cat'] = Boolean('Is ' + var + ' categorical? ')
origx = Text(value = [])
beta = ParamVector()
'''
```

This code section is the same as that in 1LevelMod upto the point that x is input. We next see a for loop that includes the use of the context function which is used to construct attribute names that are a combination of text and variable names. If for example x contains the three variable list ['cons','standlrt','schgend'] then the context statements will create 3 variables 'cons_cat', 'standlrt_cat' and 'schgend_cat' which will store the text strings ‘yes’ or ‘no’ depending on whether the variables are categorical or not. The line

```python
origx = Text(value = [])
```

will be used to store the original x variables prior to manipulating the categorical variables. By setting its value in the assignment we will not get an input widget appearing in the browser.

Let us demonstrate fitting this model so choose 1LevelCat from the template list and tutorial as the dataset. Note if you have previously used the recode template on this dataset, on the main menu click on Debug/Reload Datasets to get back the original tutorial dataset. Firstly we will choose the inputs as follows:
Next we click on the Next button and we will be able to look at the model code and equation (by selecting `equation.tex` in the right hand list):

Here we see that in both the maths and the model code the expression for the linear predictor has two terms to represent two of the possible categories for school gender (schgend_2 and schgend_3). The important method here is the `preparedata` method. The `preparedata` method basically links the data with the template but we can write template specific versions of the method that replace this generic linkage and allow preprocessing of the data. In this case the code is as below:
def preparedata(self, data):
    mydata = data['datafile']
    for var in self.objects['x']:
        self.objects['origx'].name.append(var)  # Save user's original selection
        #self.objects['x'] = []
        del self.objects['x'][:]
    for var in self.objects['origx'].name:
        if self.objects[var + '_cat']:
            uniqvals = list(set(mydata.variables[var]['data'].compressed()))
            uniqvals.sort()
            uniqvals.remove(uniqvals[0])
            for i in uniqvals:
                mydata.addvariable(var + '_' + str(int(i)), data = (mydata.variables[var]['data'][:] == i).astype(float))
                self.objects['x'].name.append(var + '_' + str(int(i)))
        else:
            # TODO: fix this
            self.objects['x'].name.append(var)

    self.objects['beta'].ncols = len(self.objects['x'])

This code firstly retains the named predictor variables in origx by copying the contents of x to origx and then deleting them from x. Then the code loops over the variables via the second for statement and conditionally (the if statement) on a particular variable being categorical does some processing. The lines

    uniqvals = list(set(mydata.variables[var]['data'].compressed()))
    uniqvals.sort()
    uniqvals.remove(uniqvals[0])
    for i in uniqvals:
        mydata.addvariable(var + '_' + str(int(i)), data = (mydata.variables[var]['data'][:] == i).astype(float))
        self.objects['x'].name.append(var + '_' + str(int(i)))
    else:
        # TODO: fix this
        self.objects['x'].name.append(var)

    self.objects['beta'].ncols = len(self.objects['x'])

firstly find all unique values in the categorical predictor which are then stored in uniqvals. We then sort these into ascending order before removing the first as it will play the role as the base category in the model. We then have a second loop over this list of uniqvals where we create the dummy variables. The lines

    uniqvals = list(set(mydata.variables[var]['data'].compressed()))
    uniqvals.sort()
    uniqvals.remove(uniqvals[0])

firstly construct an array which takes value 1 if the original variable has value i or 0 otherwise. This newly constructed predictor variable is then appended to the new variable list. If the variable is not categorical it is simply added to this new variable list itself. We finally adjust the length of beta to account for the expansion of the categorical variables and return the new dataset.

This preparedata method is run before the outbug and outlatex methods and so these are identical to those we saw in 1LevelMod. To continue running the example we can press the Run button and then select the ModelResults object from the right hand list to get the following results:
This completes this section and is the last single level model we will meet for a while. Another extension would be to allow the inclusion of interactions into the model. This has been done in the template 1LevelInteractions (which is available of the template repository but not part of the core release). Here the modifications are done in the invars and outbug/outlatex methods as no new predictor variables are created. Instead the model code includes multiplications between the variables. We will leave you to try out this template as an exercise.

9 Multilevel models

Our next step is to move onto templates for models for more complex data structures. In this section we look at multilevel modelling templates — templates that allow random effects to account for clustering in the data. We will look at two templates of increasing complexity, firstly a template for fitting models that have 2 levels i.e. 1 higher level of clustering and then secondly a more general template that will fit models with any number of levels clustering whether nested or crossed. Note here that these templates allow only random intercepts in the models we are fitting.

9.1 2LevelMod template

We will begin our investigation of 2LevelMod by looking at its invar function:

```python
invars = '''
y = DataVector('response: ')
L2ID = IDVector('Level 2 ID: ')
D = Text('specify distribution: ', ['Normal', 'Poisson', 'Binomial', 't'])
if D == 'Binomial':
    n = DataVector('denominator: ')
    link = Text('specify link function: ', ['logit', 'probit', 'cloglog'])
if D == 'Poisson':
    link = Text(value = 'ln')
    offset = Boolean('Is there an offset: ')
    if offset:
        n = DataVector('offset: ')
if D == 'Normal':
```
\[ \tau = \text{ParamScalar()} \]
\[ \sigma^2 = \text{ParamScalar()} \]

if D == 't':
    \[ \tau = \text{ParamScalar()} \]
    \[ \sigma^2 = \text{ParamScalar()} \]
    \[ \#d = \text{ParamScalar()} \]
    \[ u = \text{ParamVector(parents=[L2ID], as_scalar=False)} \]
    \[ \tau_u = \text{ParamScalar()} \]
    \[ \sigma^2_u = \text{ParamScalar()} \]

\[ x = \text{DataMatrix('explanatory variables: ')} \]
beta = ParamVector(parents=[x], as_scalar=True)
storeresid = Boolean('Store level 2 residuals?')

If you compare this with the `invars` function for `1LevelMod` you will see we have added 2 additional inputs: `L2ID` to allow the user to input the column containing the level 2 identifiers and `storeresid`, a Boolean indicator of whether to store the level 2 residuals or not. We also have three additional parameters \( u, \tau_u \) and \( \sigma^2_u \) (to represent the level 2 residuals, their precision and variance respectively) that have been included. We can try out an example of these inputs by selecting the template `2LevelMod` and the dataset `tutorial` and applying the following inputs:

Clicking on **Next** and choosing **equation.tex** in the right hand pane will bring up the maths and model code:
Here we see a mathematical representation of the model created in outlatex along with the model code in outbug. We saw in the last section the use of preparedata and another new method, monitor_list is used here:

```python
def monitor_list(self):
    mon = []
    old_mon = Template.monitor_list(self)
    for m in old_mon:
        if m != "u" or self.objects['storeresid']:
            mon.append(m)
    return mon
```

Basically all we use monitor_list for is to construct a list of those parameters which we want to store chains for. If no monitor_list method is present then Stat-JR assumes all parameters are to have their chains stored as has been the case in the other model templates covered so far. Here the method loops through all the parameters and if one is called “u” and residuals are NOT to be stored then this parameter is not included in the list.

Let’s look next at outbug:

```
outbug = '''
model {
    for (i in 1:length(${y})){
        ${y}[i] ~ % if D == 'Normal':
        dnorm(mu[i], tau)
        mu[i] <- % endif
        % if D == 'Binomial':
        dbin(p[i], ${n}[i])
        ${link}(p[i]) <- % endif
        % if D == 'Poisson':
        dpois(p[i])
    }
}
```
The code has become quite long mainly due to the conditional statements for the different distribution types and the fact that we also allow the use of the t distribution for random effects (although we will not demonstrate or discuss this here).

We see that the term \( u[L2ID[i]] \) has been appended to the linear predictor where L2ID is inserted for a particular model. The chunk of code

```c
  for (j in 1:length(u)) {
    % if D == 't':
      u[j] ~ dt(0, tau_u, df)
    % else:
      u[j] ~ dnorm(0, tau_u)
    % endif
  }
```

then gives the random effect distribution and finally the chunk

```c
  tau_u ~ dgamma(0.001000, 0.001000)
  sigma2_u <- 1 / tau_u
```

...
gives a prior distribution for the variance of the random effects. The `outlatex` function is adapted in very similar ways and so for brevity we omit this code here. We will finish off this template by running it and looking at ModelResults (in a new tab). If we do this we only get results for the variables that have had their chains stored:

```
<table>
<thead>
<tr>
<th>parameter</th>
<th>mean</th>
<th>sd</th>
<th>ESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>sigma2_u</td>
<td>0.097111</td>
<td>0.003012</td>
<td>2400</td>
</tr>
<tr>
<td>mu</td>
<td>1.7639011</td>
<td>0.009133</td>
<td>5771</td>
</tr>
<tr>
<td>Decl</td>
<td>0.0293602</td>
<td>0.121719</td>
<td>5502</td>
</tr>
<tr>
<td>beta_1</td>
<td>0.032039</td>
<td>0.04039</td>
<td>231</td>
</tr>
<tr>
<td>beta_0</td>
<td>0.054402</td>
<td>0.012556</td>
<td>466</td>
</tr>
<tr>
<td>tau_u</td>
<td>10.73609</td>
<td>2.39654</td>
<td>3407</td>
</tr>
<tr>
<td>sigma2</td>
<td>0.56655</td>
<td>0.012550</td>
<td>5771</td>
</tr>
</tbody>
</table>
```

Basically although we didn’t store chains for each of the 65 random effects $u$ we can store summary statistics for them but by default we do not display them unless we change the output options on the settings screen. As usual we also can get the MCMC plots e.g. for $\beta_0$:
**Exercise 7**
There also exists a template that allows for categorical predictors (2LevelCat) adapt this template so that it allows the user to incorporate interactions.

### 9.2 NLevelMod template

The NLevelMod template as the name suggests extends the 2LevelMod template to an unlimited number (input by the user) of levels of clustering. Note that these clusters can be either nested or cross-classified. We will once again start by looking at the invars attribute to see how it differs from 2levelmod:

```plaintext
invars = ''
NumLevs = Integer('Number of Classifications: ')
for i in range(0, int(NumLevs)):
    selstr = 'Classification ' + str(i + 1) + ':
    context['C' + str(i + 1)] = IDVector(selstr)
    context['u' + str(i + 1)] = ParamVector(parents=[context['C' + str(i + 1)]], as_scalar=False)
    context['tau_u' + str(i + 1)] = ParamScalar()
    context['sigma2_u' + str(i + 1)] = ParamScalar()

y = DataVector('response: ')
D = Text('specify distribution: ', ['Normal', 'Binomial', 'Poisson'])
if D == 'Binomial':
    n = DataVector('denominator: ')
    link = Text('specify link function: ', ['logit', 'probit', 'cloglog'])
if D == 'Poisson':
    link = Text(value = 'ln')
    offset = Boolean('Is there an offset: ')
    if offset:
        n = DataVector('offset: ')
if D == 'Normal':
    tau = ParamScalar()
    sigma2 = ParamScalar()
x = DataMatrix('explanatory variables: ')
beta = ParamVector(parents=[x], as_scalar=True)
storeresid = Boolean('Store residuals?')
'''
```

Here we have needed to replace the code for inputting the level 2 identifier with code to input the number of classifications (levels of clustering) and then we have looped over the number of classifications constructing both the names of the columns that contain the classification vectors (which will labelled \(C1, C2\)…) and the new parameters associated with each classification (\(u1, tau_u1\) and \(sigma2_u1\) etc). To achieve these inputs we have used the `context` command to construct attribute names by concatenating strings and also a simple string concatenation to create `selstr` which contains the question associated with inputting each classification name. The rest of the code is similar to before. It should be noted that a 2 level model in this template has 1 classification as we are not considering level 1 here.

We can consider using this template on a cross-classified example with two higher classifications. This example is a dataset from Fife in Scotland where we are looking at the impact of both primary school and secondary school on the attainment of children at age 16. To do this select the template NLevelMod from the template list and the dataset xc from the dataset list. Select the inputs as shown:
Clicking on the **Next** button will display the mathematical formulation of the model and the model code if you choose `equation.tex` in the right hand list:
We can see that the model code is similar to that we had for the 2 level model and 2LevelMod. Again we have a `monitor_list` function that decides whether or not to monitor the residuals based on user input.

```python
def monitor_list(self):
    mon = []
    old_mon = Template.monitor_list(self)
    for m in old_mon:
        if not m.startswith("u") or self.objects['storeresid']:
            mon.append(m)
    return mon
```

The model code is created by the `outbug` attribute and here we see the code:

```plaintext
outbug = ''
<% numlevs = int(NumLevs) %>
model {
    for (i in 1:length(${y})) {
        ${y}[i] ~ \$
        if D == 'Normal':
            dnorm(mu[i], tau)
            mu[i] <- \$
            endif
        if D == 'Binomial':
            dbin(p[i], ${n}[i])
            $(link)(p[i]) <- \$
            endif
        if D == 'Poisson':
            dpois(p[i])
            $(link)(p[i]) <- \$
            if offset:
                ${n}[i] + \$
            endif
            endif
            ${mmult(x, 'beta', 'i')} \$
            if for i in range(0, numlevs):
                + u${i + 1}[${context['C' + str(i + 1)]}[i]]
            }% endfor
        }% endfor
        if i in range(0, numlevs):
            for (i${i + 1} in 1:length(u${i + 1})) {
                u${i + 1}[i${i + 1}] ~ dnorm(0, tau_u${i + 1})
            }% endfor
        % endfor
# Priors
% for i in range(0, x.ncols()):
    beta${i} ~ dflat()
% endfor
% if D == 'Normal':
    tau ~ dgamma(0.001000, 0.001000)
    sigma2 <- 1 / tau
% endif
% for i in range(0, numlevs):
    tau_u${i + 1} ~ dgamma(0.001000, 0.001000)
    sigma2_u${i + 1} <- 1 / tau_u${i + 1}
% endfor
}
...
Here we introduce the use of local variable `numlevs`. Basically the `outbug` attribute is a text string with substitutions. Then if we wish to include a Python statement, whilst inside the text string, we place it within a `<%` and a `%>` in this case we set a value to `numlevs` and then use it as a looping upper bound later in the code. You will see that the rest of the code contains many of the features we have discussed in earlier examples. You do however have to be careful as the code is such a mixture of WinBUGS like model code and Python code. For example if we consider the chunk:

```% for i in range(0, numlevs):
  for (i$+1 in 1:length(u$i+1)) {
    u$i+1[i$+1] ~ dnorm(0, tau_u$i+1)
  }
% endfor```

with our cross-classified model. Here we are using `i` in both the model code we are constructing and as a python variable. So `numlevels` in our example is 2 and so the outside `%for (Python)` can be expanded out and the `i` substitutions made and we get:

```for (i1 in 1:length(u1)) {
  u1[i1] ~ dnorm(0, tau_u1)
}
for (i2 in 1:length(u2)) {
  u2[i2] ~ dnorm(0, tau_u2)
}```

as we see in the browser. Once again the `outlatex` function which creates the LaTeX output will have similar substitutions via Python but we will not describe this in detail here.

Clicking on Run will run the model and the output contains information for all the residuals (but not ESS) and the usual MCMC plots are available. There are several other N level modelling templates included with the software that you can also look at. We will describe one further such template (`NLevelRS`) which allows random slopes in section 11. This template will need to utilise the `preccode` feature and so we will first explain this with a simpler 1 level example.

**Exercise 8**
Try adapting this templates to allow categorical predictors as in 1LevelCat calling your new templates `nlevelcatint` and also possibly adding the option for interactions.

### 10 Using the Preccode method

One of the aims of the Stat-JR system is to allow other estimation engines aside from our built-in MCMC engine to be used with templates. We saw in section 6 details of how the system can interact with third-party software. In this section (and in fact the following three sections) we will see how through the inclusion of additional C++ code the user can increase the set of models and methods that can be fitted using the built-in engine. At present the methods we describe are partly to advance the modelling but also partly to cover current limitations in the algebra system which will hopefully eventually be rectified. As the names suggest the `preccode` function will involve writing C++ code and so some knowledge of the C/C++ languages would be useful. The examples given here will however allow the user with some modification to use similar chunks of code for their examples. We begin in this chapter with a simple example of a 1 level probit regression model.
10.1 The 1LevelProbitRegression template

We have seen already that the 1LevelMod template can be used to fit binary response models and we have demonstrated a logistic regression model for the bang dataset. A probit regression is similar to a logistic regression but uses a different link function. One interesting feature of a probit regression is that the link function is the inverse normal distribution cdf. This means that we can interpret the model using latent variables in an interesting way.

Imagine that you had a variable which was a continuous measurement but that we can only observe a binary indicator as to whether the variable was above or below a threshold, for example in education we might have a mark in an exam but the student is only told whether they pass or fail. If we model the pass/fail indicators using a probit regression then this is equivalent to assuming the unobserved (latent) continuous measure follows a normal distribution (with threshold 0 and variance 1).

We can use this fact in our modelling when we use MCMC by generating the latent continuous variables as part of the algorithm. Then having generated the latent variables we have a normal response model for these variables which is easy to fit. The 1LevelProbitRegression template therefore fits a probit regression using this technique and we will add the step to update the continuous response variables via the preccode methods.

We will start as usual by looking at the invars attribute which is quite short:

```
invars = ''
v = DataVector('response: ')
y = ParamVector(parents=[v], as_scalar=False)
x = DataMatrix('explanatory variables: ')
beta = ParamVector(parents=[x], as_scalar=True)
''
```

Here you will see that the column containing the 0/1 response is actually stored as v in this template as we will use y to be the underlying continuous response. As y is latent it is defined as a Paramvector rather than data, and the parents term links the lengths of the two vectors together which basically ensures that the continuous response vector y is the same length as the observed binary response vector v. As always it helps to demonstrate the template with an example so we will fit a probit regression model (equivalent to the logit regression in section 7) to the Bangladeshi dataset. Select 1LevelProbitRegression from the template list and bang from the dataset list and then fill in the template as shown below:
Clicking on the **Next** button and choosing `equation.tex` in the right hand pane we see the model described mathematically:

Here you see how use is the latent continuous variable written as $y$ in the model code. We also see that the model code is really just fitting a normal model as if we already know the values of $y$ and if we look at `outbug` we can see that clearly.
Here the code is fairly straightforward so the interesting thing is how we actually include a step for y to make this the correct model. We have a `customsteps` function:

```python
def customsteps(self):
    ret = ['y']
    return ret
```

This function basically constructs a list of parameters for which we will ignore the step from the algebra system. In this case the list simply contains `y`. We will now look at two more methods in this template that we haven’t looked at before in great detail.

**10.2 monitor_list method**

We saw that in the previous couple of templates that we asked whether to store residuals. The problem of storing the residuals is this can mean that the program uses lots of memory if the number of parameters is large, and in the current template we have a parameter for each data point (the latent continuous responses). We would therefore like the ability to not store every value. This is done via the `monitor_list` method which has the following code:

```python
def monitor_list(self):
    mon = []
    old_mon = Template.monitor_list(self)
    for m in old_mon:
        if not m == 'y':
            mon.append(m)
    return mon
```

Here we have a method that goes through all the parameters in the model and as long as they are not equal to `y` they are appended to the monitor list. To see this in practice we will continue our example and press the Run button. If you select ModelResults then the results will look as follows:
One word of caution is that the DIC diagnostic here is using the normal model that the algebra system knows about so should be disregarded. We will fix this in a later version of the template.

10.3 \texttt{preccode} method

So we have seen that the code works but we need now to look and see how the step for updating the latent \( y \) variable is incorporated into the code. This is done via the \texttt{preccode} function which for this template looks as follows:

```python
def preccode(self):
    extraccode = ''
    <%!
    def mmult(names, var, index):
        out = ''
        count = 0
        for name in names:
            if count > 0:
                out += ' + ' + double(' + name + ' + index + ') * ' + var + str(count)
            count += 1
        return out
    %>
    double mean;
    for(int i=0;i<length(y);i+)
    {
        mean = mmult(x, 'beta', 'i');
        if(v[i] <= 0)
            y[i] = dtnormal(mean,1,2,0,0);
        else
            y[i] = dtnormal(mean,1,1,0,0);
    }
    return extraccode
```
This function could have been even shorter except we need to include in here a definition for the \texttt{mmult} function that is used to construct the linear predictor, this time as C++ code.

The actual C++ code is the chunk

```cpp
    double mean;
    for(int i=0;i<length(y);i++)
    {
        mean = \$\texttt{mmult}(x.name, 'beta', 'i'));
        if($v.name[i] <= 0)
            y[i] = d\texttt{tnormal}(mean,1,2,0,0);
        else
            y[i] = d\texttt{tnormal}(mean,1,1,0,0);
    }
```

Here we see that the code involves looping over all data points via a \texttt{for} loop and for each point evaluating the mean value which is the linear predictor calculated via the substitution. Then depending on the value of the binary response a call is made to the truncated normal random number generator via the \texttt{dtnormal} function. Here \texttt{dtnormal} takes 5 arguments, the mean, the sd, the type of truncation with 1 left truncation, 2 right truncation and 3 both, and finally the left and right truncation values.

To see this in action choose \texttt{modelcode.cpp} from the right hand list and choose to view in it’s own tab:
Here we see that after some initial setup lines, the \texttt{preccode} chunk, as the name suggests appears at before the steps for other parameters (in this case beta0). This is important as the \( y \) variable needs initialising before the other parameters are updated and updating it first ensures \( y \) is positive when \textit{use} is 1 and negative when \textit{use} is 0.

\section{Multilevel models with Random slopes and the inclusion of Wishart priors}

One limitation of the Demo algebra system in it’s current form is that it treats all parameters as scalars. This means for example that for the \textit{Regression1} template, the set of beta parameters are all updated individually through univariate normal steps. We will investigate the implications of this in section 12. In section 9 we introduced our first multilevel models all of which only had random intercepts. To extend such models to include random slopes requires (assuming slopes and intercepts are correlated) the use of a multivariate normal distribution for the random effects. Multivariate normal distributions by their nature have vector and not scalar parameters and so our model code diverges from standard WinBUGS model code here (and hence this is an example template where template specific methods are required for WinBUGS). Our improvised model code depends on the number of response variables i.e. we have bivariate, trivariate etc normal distributions. We will see how these work in practice via the template \textit{NLevelRS}. It should be noted that we now also have templates that completely circumvent the algebra system and simply write custom C code. These templates have the postfix ‘cc’ at their end.

\subsection{An example with random slopes}

Firstly select \textit{NLevelRS} from the \textit{template} list and \textit{tutorial} from the \textit{data} list. Then choose the inputs as follows:
You will see that there are lots of inputs here and correspondingly the \texttt{invars} function for this template is therefore quite long as we see below:

```python
invars = ''
NumLevs = Integer('Number of Classifications: ')
for i in range(0, int(NumLevs)):
    selstr = 'Classification ' + str(i + 1) + ': '
    context['C' + str(i + 1)] = IDVector(selstr)
y = DataVector('response: ')
D = Text('specify distribution: ', ['Normal', 'Binomial', 'Poisson'])
if D == 'Binomial':
    n = DataVector('denominator: ')
    link = Text('specify link function: ', ['logit', 'probit', 'cloglog'])
if D == 'Poisson':
    link = Text(value = 'ln')
    offset = Boolean('Is there an offset: ')
    if offset:
        n = DataVector('offset: ')
if D == 'Normal':
```

The `invars` function includes a loop to handle multiple classifications, setting up contexts, collecting responses, and handling different distributions and link functions.
tau = ParamScalar()
sigma = ParamScalar()
x = DataMatrix('explanatory variables: ')
beta = ParamVector(parents=[x], as_scalar=True)

for i in range(0, int(NumLevs)):
    context['x'+str(i+1)] = DataMatrix('explanatory variables random at ' +
    context['C' + str(i + 1)] + ' classification: ')
    for var in range(0, len(context['x'+str(i+1)])):
        context['u' + str(var) + '_' + str(i)] =
        ParamVector(parents=[context['C' + str(i + 1)]], as_scalar=False)
    num = len(context['x'+str(i+1)])
    if num == 1:
        context['tau_u0_'+str(i+1)] = ParamScalar()
        context['sigma_u0_'+str(i+1)] = ParamScalar()
    else:
        context['omega_u'+str(i+1)] = ParamMatrix()
        context['omega_u'+str(i+1)].size = num
        context['d_u'+str(i+1)] = ParamMatrix()
        context['d_u'+str(i+1)].size = num
        context['priors' + str(i)] = Text('Priors (NB Uniform not supported
by WinBUGS / OpenBUGS): ', ['Uniform', 'Wishart'])
        if context['priors' + str(i)] == 'Wishart':
            context['R' + str(i)] = List('R matrix: ')
            context['v' + str(i)] = Integer('Degrees of Freedom:')
    storeresid = Boolean('Store residuals?')

The template is initially like the NLevelMod template but then has an additional section that is used
to input the variables that have random effects associated with them (at each level), and then any
priors at those levels are input. You will see that we use the context function to construct the
variable names a lot and that there are different parameters for classifications where there is a
single random parameter and where there are more than one. In brief parameters beginning tau_u0
and sigma_u0 are the precision and variance of the random effects if there is a single set of random
effects; those beginning omega_u and d_u are the variance matrix and precision matrix if we have
multiple sets of random effects at a classification. Finally in this case there are two possible priors
and for the (informative) Wishart priors an estimate (beginning with R) and degrees of freedom
(beginning with v) parameter are required.

Having completed our inputs we now need to click on Next to see what the model code looks like
(also selecting equation.tex in the right hand window):
Here we see the LateX code including the multivariate normal distribution for the random intercepts and slopes. You will see how this is written in the model code:

```latex
for(i1 in 1:length(u0_0)) {
  dummy_0[i1] ~ dnormal2a(u0_0[i1], u1_0[i1], 0, 0, d_u1[0], d_u1[1], d_u1[2])
  u0_0[i1] ~ dflat()
  u1_0[i1] ~ dflat()
}
```

Basically the dnormal2a distribution has as its first two arguments the two responses. Next we get the 2 means and then the 3 parameters that make up the precision matrix. As the algebra system expects all parameters to appear on the left-hand side we complete our workaround for a multivariate Normal distribution by including the two dflat statements which do not change the posterior but mean that the `u0_0[i1]` and `u1_0[i1]` are regarded in the algebra system as parameters. Note that the `dummy_0` parameters are simply placeholders as each distribution needs a scalar left-hand side. The definition of dnormal2a does not depend on the left hand side term and as the `dummy_0` parameters do not appear anywhere on the right hand side the algebra system doesn’t think they are really parameters and so does not attempt to find their posteriors.

The code for creating the model code is in `outbug` but doesn’t contain anything very new that needs reporting here. The `outlatex` code might interest those trying to learn LateX as it contains a chunk to produce the multivariate Normal line as follows:

```latex
\left(\begin{array}{l}
% for i in range(0, len(context['x'+str(lev+1)])):
u^\langle$\{lev + 2\}\rangle_{S(i),S(context['C' + str(lev + 1)])(i)}
% if i != len(context['x'+str(lev+1)]) -1:
\right)
```
Here we use Python %ifs and %fors to allow conditional code and the array environment and \left and \right (for big brackets) in LaTeX to deal with vectors and matrices. The actual code that is produced can be looked at by right clicking on the LateX and selecting show source and selecting the appropriate lines. It looks as follows:

\begin{array}{l}
\left( \begin{array}{l}
u^{(2)}_{0,\text{school}(i)}
\\u^{(2)}_{1,\text{school}(i)}
\end{array} \right) \sim \mbox{N} \left( \begin{array}{l}
0
0
\end{array} \right), \Omega^{(2)}_{u} \right] \end{array}

Looking at the model code we have not included a prior for $d_{u1}$ and so here we again resort to writing our own preccode chunk.

11.2 Preccode for NLlevelRS
We will here look at the preccode in chunks. The preccode is being used to add a step for updating the precision matrix $d_{u1}$ and the corresponding variance matrix $omega_{u1}$. Looking at the start of the code:

```python
def preccode(self):
    extracode = ''
    
    <% numlevs = int(NumLevs) %>
    bool fail = false;
    % for i in range(0, numlevs):
    <%= n = len(context['x'+str(1 + i)]) %>
    % if n > 1 :
    // Note currently using a uniform prior for variance matrix
    SymMatrix sb$i+1$(n);
    for(int i = 0; i < length(u0_S{i}); i++) {
    % for j in range(0, n):
```
This first section stores the number of levels (numlevs) for looping purposes and also within the loop the number of random effects are constructed (as \( n \)) because for classifications with only 1 set of random effects nothing needs doing as the algebra system has evaluated the posterior required. We next construct a matrix variable \( sb1 \) which initially stores the crossproduct matrix of the residuals before moving to the next chunk of code:

In this chunk of code we have different blocks of code depending on prior distribution types. For the uniform prior we simply construct the degrees of freedom parameter (\( vw1 \)), which equals the number of higher level units minus the number of sets of random effects + 1. We also have some code for the first iteration (\( runstate = 0 \)) to avoid numerical problems as the residual starting values are all the same. For the Wishart prior we have to add the prior parameters to the \( sb1 \) and \( vw1 \) parameters. Next we have:
In this last chunk of code we invert the \texttt{sb1} parameter before drawing the new precision matrix which we store in \texttt{mat\_d\_u1} and the inverse matrix to the vector \texttt{mat\_omega\_u1}. To see the code that the \texttt{preccode} method generates for our example we can select \texttt{modelcode.cpp} and scroll down a few lines as shown below:

As the name \texttt{preccode} suggests the code appears before the other steps in the algorithm. We can finally run the template by clicking on the \texttt{Run} button and selecting \texttt{ModelResults} from the right hand list. The results appear as follows:
and as usual we also get MCMC output graphs for the fixed effect parameters, variances and precisions via the right hand list.

**Exercise 9**
Try adapting the NLevelRS template so that it only allows one higher classification and compare your results with the 2LevelRS template. This exercise will be in essence a merging of features of two templates, 2LevelMod and NLevelRS and will test your understanding of the various chunks of code.

**12 Improving mixing (1LevelBlock and 1LevelOrthogParam)**
In this section we will return once again to our first template, Regression1 but use it on a different dataset, rats. This dataset consists of the weights of 30 laboratory rats at weekly intervals from 8 days old and here we will consider a regression looking at the impact on their final weight at 36 days of their initial weight at 8 days old.

**12.1 Rats example**
We will set up a simple regression for this rather small dataset as follows:

We will then run the model by clicking the Next and Run buttons. If we look at the output and change the right hand list to beta_1.png so that we have the MCMC plot for beta1 visible we will see the following:
In *ModelResults* we see that both the regression coefficients have very small effective sample sizes (28 and 29 respectively) and the chains we observe in the graphs above are not mixing well. Aside from being a small dataset a difference between the *rats* and the *tutorial* dataset is that the data have not been centred. This means that the joint posterior distribution of $\beta_0$ and $\beta_1$ has a large correlation between the pair of parameters and so if we update them separately we will have problems. We will look at two templates that will rectify this problem.

### 13.2 The 1LevelBlock template

Most software packages will update the parameters $\beta_0$ and $\beta_1$ together in one multivariate normal block. As we have seen in chapter 12 the current algebra system in Stat-JR does not produce multivariate posterior distributions. We can however work out the correct posterior distribution by hand and plug this into the code via the `preccode` options we have seen earlier. This is performed by the 1LevelBlock template. If you look at the template code you will see it has an initial input in the `invars` attributes as to whether or not to block the fixed effects.

```python
mv = Boolean('Use MVNormal update for beta?: '

Otherwise the code is little changed from the template 1LevelMod until we come to a new `customsteps` method:

```python
def customsteps(self):
    ret = []
if not self.objects['mv']:
    return ret
nbeta = len(self.objects['x'])
for i in range(0, nbeta):
    ret.append('beta' + str(i))
return ret

This code is informing the code generator that custom steps are to be used for the beta parameters when the block updating option (mv) is selected and that it should ignore whatever has been returned from the algebra system for these steps. The precode method then contains the code to update the beta vector which has mean (in matrix form) \((X'X)^{-1}X'y\) and variance \((X'X)^{-1}\) times the residual variance. The code which uses matrix classes is as follows:

def preccode(self):
    if not self.objects['mv']:
        return ''
    extraccode = ''
    \<% nbeta = len(context['x']) %> \$
    bool fail = false;
    static RectMatrix xtxchol(${nbeta}, ${nbeta});
    static RectMatrix mean(${nbeta}, 1);
    // Setting up constant terms for beta step
    if (runstate == 0) {
        xtxchol = mat_x.T() * mat_x;
        chol_decomp(xtxchol, 0);
        RectMatrix xty = mat_x.T() * mat_y;
        mean = matrix_cholsolve(xtxchol, xty);
    }
    // Multivariate step for beta
    RectMatrix taudiag(${nbeta}, ${nbeta});
    for (int i = 0; i < ${nbeta}; i++) {
        taudiag(i, i) = 1/tau;
    }
    SymMatrix variance = matrix_cholsolve(xtxchol, taudiag);
    mat_beta = dmultnormal(mean, variance, fail);
    
    return extraccode

If we want to test this template we can choose it (along with rats) from the template list and set up the inputs as follows:
Running the template by pressing the **Next** and **Run** buttons results in the following output. Note here we have selected *beta_1.png* for comparison with the *Regression1* output.
We can see that the method has given much better mixing for $beta1$. Looking at the ModelResults the effective sample size values have increased from 28/29 to 6422 for both $beta0$ and $beta1$. We have in other templates (2LevelBlock and NLevelBlock) implemented similar block updating of fixed effects for multilevel models. We will next look at an alternative method that has the advantages of not needing to use precode and also of being useful for non-normal response models.

### 13.3 The 1LevelOrthogParam template

The alternative approach to blocking variables that are correlated is to reparameterise the parameters to a configuration that are less correlated. We will achieve this by using an orthogonal parameterisation for the fixed effects rather than the standard parameterisation. The template we will use is called 1LevelOrthogParam and the inputs are very similar to the 1LevelMod template (as this approach also works for non-normal responses). The template does have 2 additional inputs in $invars$ which are used to find out whether or not to use a transformed parameterisation and if so whether to use an orthogonal or orthonormal parameterisation.

This can be seen in the following lines:

```python
orthog = Boolean('Do you want to use orthogonal parameterisation?: ') if orthog:
    betaort = ParamVector(parents=[x], as_scalar=True)
    orthomat = Text(value = [])
```

Here we add an additional vector of responses, $betaort$ if the orthogonal parameterisation is to be used. Let us try out the template on the rats example so choose 1LevelOrthogParam from the template list and input the following:
Clicking on the Next button and choosing equation.tex from the right hand list will give the following output for the model:

\[
\begin{align*}
\beta_0 & = N(\mu_0, \sigma^2) \\
\mu_0 & = \frac{\text{orthcons} + \beta_0 \text{orthy8}}{\sigma^2} \\
\beta_0 & \approx 1 \\
\beta_1 & \approx 1 - 1.0 \beta_0 + 152.16666667 \beta_0^2 \\
\beta_2 & \approx 0.8 \beta_1 + 1.0 \beta_2 \\
\gamma & \approx 1(0.681, 0.061) \\
\sigma^2 & \approx 1/y
\end{align*}
\]

The method of using an orthogonal parameterisation is mentioned in Browne et al. (2009) for non-normal examples and has also been implemented in MLwiN. For details on how we construct orthogonal vectors we refer the reader to Browne et al. (2009) but note that a function to do the procedure named orthog can be viewed in 1LevelOrthogParam.py. Here you will see that we fit a model with the parameters betaort placed in the linear predictor along with data vectors orthcons and orthy8. These data vectors are constructed in the function preparedata that we detail here:

```python
def preparedata(self, data):
    mydata = data['datafile']

    if self.objects['orthog']:
        self.objects['orthogmat'] = []
        orth = numpy.zeros([len(mydata.variables[self.objects['x'][0]]['data']),
                            len(self.objects['x'])])

        for i in range(0, len(self.objects['x'])):
            orth[:, i] = mydata.variables[self.objects['x'][i]]['data']

        if self.objects['orthotype'] == 'Orthogonal':
            om = orthog(orth)
            for i in om.flat:
                self.objects['orthogmat'].append(str(i))

    for n in range(0, len(self.objects['x'])):
```

89
tmp =
numpy.zeros(len(mydata.variables['x'][n]['data']))
for n1 in range(0, len(self.objects['x'])):
    tmp +=
    mydata.variables[self.objects['x'][n1]['data']] * om[n1, n]
mydata.addvariable('orth' + self.objects['x'][n], data = tmp)

if self.objects['orthtype'] == 'Orthonormal':
    (tmp, om) = numpy.linalg.qr(numpy.mat(orth))
    for i in om.I.flat:
        self.objects['orthogmat'].append(str(i))
    for n in range(0, len(self.objects['x'])):
        mydata.addvariable('orth' + self.objects['x'][n], data =
        numpy.array(tmp[:, n]).flatten())
self.objects['x'][:] = map(lambda n: 'orth' + n,
self.objects['x'])

We begin by constructing a blank list 'orthogmat' and an empty matrix orth. We then implement the orthogonalising algorithm by filling orth with the original x variable vectors and then calling the orthog function. om is the matrix that performs the orthogonalisation and we store this as a vector in the object 'orthogmat'. A slightly different routine is given if the user chooses orthonormal instead here. The last few lines then multiply the original x variables by 'orthogmat' storing the results in a matrix tmp. The columns of this tmp matrix are then placed in objects that have the string 'orth' appended to the front of the original x variables names. Finally the map function replaces the original x variable names with these new orthogonal variable names before the data is returned. The function outbug then constructs the model code:

    outbug = '
model{
    for (i in 1:length(${y})) {
        ${y}[i] ~ \$
        % if D == 'Normal':
        dnorm(mu[i], tau)
        % endif
        % if D == 'Binomial':
        dbin(p[i], ${n}[i])
        % endif
        % if D == 'Poisson':
        dpois(p[i])
        % endif
        % if offset:
        ${n}[i] + \$
        % endif
        % endif
        %if orthog:
        ${mmult(x, 'betaort', 'i')}
        % else:
        ${mmult(x, 'beta', 'i')}
        % endif
    }
}
# Priors
% for i in range(0, beta.ncols):
  %if orthog:
    betaort${i} ~ dflat()
  % else:
    beta${i} ~ dflat()
  % endif
% endfor

% if orthog:
  <% count = 0%>
  % for i in range(0, beta.ncols):
    beta${i} <- \ \\n  % for j in range(0, beta.ncols):
    ${orthogmat[count]} * betaort{j}\\
  % if j == (beta.ncols - 1):
  % else:
    % if float(orthogmat[count+1])  >= float(0.0) :
      + \ \n    % endif
    % endif
  <\% count += 1 %>\
  % endfor
% endif

% if D == 'Normal':
  tau ~ dgamma(0.001000, 0.001000)
  sigma <- 1 / sqrt(tau)
  sigma2 <- 1 / tau
% endif

Here we see that a different `mmult` function is performed for the orthogonal parameterisation and priors are given for `betaort` rather than `beta` in this case. Finally code is given to allow us to recover `beta` from `betaort` deterministically. We construct the product of the `orthogmat` terms and the `betaorts` placing + signs between the terms unless the `orthogmat` term is negative.

We can run the model by clicking on the Run button and we will see the following results for `beta1` if we select `beta_1.png` in the right hand list:
We again see good mixing of the chains and very similar estimates to the blocking approach (Effective sample sizes for beta0 and beta1 are 5948 and 5923 respectively). The other advantage of this orthogonal approach is in its generalisability to non-normal response models. In these cases Metropolis Hastings algorithms are used and so a blocking approach is not so straightforward.

**Exercise 10**
Convert this template so that it is analogous to the `Regression1` template but uses the orthogonal parametrisation. Call this new template `orthogression`.

### 12.3 Multivariate Normal response models
Having established a method of including multivariate distributions for use with random slopes in the preccode we can reuse the same method to allow us to fit multivariate Normal response models We will here consider the template for fitting 1 level multivariate response models, `1LevelMVNormal.py`. This template can be used to fit models with missing data for some responses which is achieved by a method similar to that used for the probit regression and so the preccode will generate (at least) two steps, one for the variance matrix of the responses and an initial step to set up the missing responses. Looking at the `invars` function we see the following:

```python
invars = '''
D = Text(value = 'MVNormal')
y = DataMatrix('responses: ')
lenbeta = 0
for i in range(0, len(y)):
    context['x'+str(i+1)] = DataMatrix('explanatory variables for response ' + y[i] + ': ')
    lenbeta += len(context['x'+str(i+1)])
context['miss'+y[i]] = ParamVector()
```
Here you will notice that we construct parameter vectors that are a combination of the string 'miss' and the y variable names input and these will be used in the model. Note that in line with the 1LevelBlock template we have also given the option to update beta as a block but for now we will ignore this here. Let us run the template with the gcsemv1 dataset that contains two responses for secondary school pupils taking General Certificate of Secondary Education (GCSE) exams in 1989, a written and a coursework test score. We will set up the inputs as follows:
Here we allow the two responses to both depend on one predictor female. Note that both responses contain missing values as there are some pupils with only a written score and some with only a coursework score. The missing values are given the value $-9.999e29$ and this value will be looked for in the preccode function. You will also note the extra input for imputing datasets. Here we will return datasets with the current values of missing data at the prescribed iteration numbers for each chain. Clicking on the Next button and looking at the model output having chosen equation.tex in the right hand window we see:

```
\begin{verbatim}
\begin{verbatim}

Here we see again the use of the \texttt{dnormal2a} function and also that we have included \texttt{dflat} statements for both the \texttt{misswritten} and \texttt{misscsework} responses to let the algebra system know that these are parameters. We will not look in detail at the \texttt{outbug} method as we can see the output it produces on the screen.

There is a \texttt{monitor_list} method to stop us storing chains for the missing data:

```python
def monitor_list(self):
    mon = []
    old_mon = Template.monitor_list(self)
    for m in old_mon:
        if not "miss" in m:
            mon.append(m)
    return mon
```

There is also a \texttt{preparedata} method that is used to set the length of the missing data vector to equal the original response vector:

```python
def preparedata(self, data):
    mydata = data['datafile']
    for i in range(0, len(self.objects['y'])):
        self.objects['miss'+self.objects['y'][i]].ncols = -1*len(mydata.variables[self.objects['y'][i]]['data'])
```
We next turn our attention to the `preccode` function.

### 12.4 The `preccode` function for this template

We will deal with the code here in chunks. We begin with a definition of the `mmult2` function that we will use to work out the linear predictors for each response. The `mmult2` function is specifically useful for multivariate response models as it contains a count parameter which informs us which element of beta to start with in the linear predictor:

```python
extraccode = '''
<%!
def mmult2(names, var, index,count):
    out = ""
    first = True
    for name in names:
        if first == False:
            out += ' + ' +
        else:
            first = False
            out += 'double(' + name + ' + index + ') * ' + var + str(count)
        count += 1
    return out
%>

{% n = len(context['y']) %>

Next we have the code chunk for generating the step for the level 1 variance matrix. This is almost identical to the random slopes code except we need the crossproduct of the level 1 residuals e (instead of the higher level random effects u) and this needs constructing which is done in the initial code using the `mmult2` function:

```c
// Note currently using a uniform prior for variance matrix
    SymMatrix sb(${n});
    for(int i = 0; i < length(miss${y[0]}); i++) {
        % for i in range(0, n):
            double e${i} = double(miss${y[i]}[i]) -
            (${mmult2(context['x' + str(i+1)], 'beta', 'i', lenbeta)});
        % endfor
        % for i in range(0, n):
            % for j in range(i, n):
                sb(${i}, ${j}) += e${i} * e${j};
        % endfor
    % endfor
```

Once constructed the remainder of the code follows the same pattern as random slopes and so for brevity we omit this code here, it can be viewed in `1LevelMVNormal.py`.

The one thing we have not mentioned is how the missing data is updated and here this is currently done in a slightly undesirable way, and relies on the parameter name beginning with the character string `miss`. To see how this is done we once again have to delve deeper into the code. In the subdirectory of Stat-JR with path `src/lib/EStat/templates` you will find some of the files that are used in the code generation. The file `gibbsstep.cpp` contains the template that is used by Stat-JR to convert the step from the algebra system into C code and in here we can modify what precisely is
written in the C code. You will notice a few statements that involve the “miss” prefix at the start and end of the code:

```c
% if "miss" in theta:
<% temp = theta.replace('miss', '',1) %>
if (${temp} <= -9.999e29) {
% endif

and

% if "miss" in theta:
}
% endif

This code recognises the prefix “miss” in a variable name and places the condition statements around the the update step for that parameter. There are also some more complicated reliance on various prefixes involving “mis” but these are primarily for the mixed response modelling which we do not discuss here. Basically for the case “miss” we have:

```c
% if fn == "dnorm" and "mis" in theta:
% if
% elif "miss" in theta:
${theta} = ${expr};
% endif
% endif
```

which simply translates to equating the variable name of interest (theta) to the expression the algebra system gives for its posterior (expr). This reliance on the parameter name is undesirable and we will hopefully come up with a better method for making such algorithmic changes in later releases.

It would be good at this point to look at the code generated for this example. To do this choose `modelcode.cpp` from the right hand list and scroll down. Here we see the step for the variance matrix `omega_e` near the top of the code:

```c
{
  // Note currently using a uniform prior for variance matrix
  SymMatrix sb(2);
  for(int i = 0; i < 1905; i++) {
    double e0 = double(misswritten[i]) - (double(cons[i]) * beta0 + double(female[i]) * beta1);
    double e1 = double(misscsework[i]) - (double(cons[i]) * beta2 + double(female[i]) * beta3);
    sb(0, 0) += e0 * e0;
    sb(0, 1) += e0 * e1;
    sb(1, 1) += e1 * e1;
  }
  if (runstate == 0) {
    sb(0, 0) += 0.0001;
    sb(1, 1) += 0.0001;
  }
  matrix_sym_invinplace(sb);
  int vw = 1905 - 3;
  bool fail = false;
  mat_d_e = dwishart(vw, sb, fail);
}
mat_omega_e = matrix_sym_inverse(mat_d_e, fail);

and later on the steps for the missing data:

// Update misswritten
for(unsigned int i=0; i<1905; i++){
    if (written[i] <= -9.999e29) {
        misswritten[i] =
            dnorm(((beta0*cons[i]) + (beta1*female[i]) + ((d_e[1]*misscsework[i]*pow(d_e[0],(-1.0)))*(-1.0)) + (d_e[1]*pow(d_e[0],(-1.0))*beta2*cons[i]) + (d_e[1]*pow(d_e[0],(-1.0))*female[i]*beta3)), d_e[0]);
    }
}

// Update misscsework
for(unsigned int i=0; i<1905; i++){
    if (csework[i] <= -9.999e29) {
        misscsework[i] =
            dnorm(((d_e[1]*pow(d_e[2],(-1.0))*misswritten[i])*(-1.0)) + (d_e[1]*beta0*pow(d_e[2],(-1.0))*cons[i]) + (d_e[1]*beta1*female[i]*pow(d_e[2],(-1.0)))+ (beta2*cons[i]) + (female[i]*beta3)), d_e[2]);
    }
}

We can run the template by clicking on the Run button and choosing the ModelResults in the right hand list:
Note here we do not see the missing values as by default non-monitored nodes are not displayed in ModelResults. To view the missing values you need to return to the Settings screen (from the main screen) and click on the tick box that allows you to Include unmonitored values in results and click on Set before setting up the model again. If you do this the screen will look as follows:

Here you will see that for the missing data variables the ones that correspond to actual data have standard deviation zero in the output as they shouldn’t change from iteration to iteration so for example the first and third written scores were observed. We can also look at the values of these missing data at prescribed iterations/chains and so selecting impute_datafile_chain0_iter1000 gives the following:
Here the second written test score has value 47.19, and the ninth written score is 24.98 whilst their means are 44.05 and 33.13 respectively across all iterations and chains. We have extended these multivariate normal modelling templates to more levels and to include random slopes. They also form the basis for the mixed response templates which allow other response types via the use of latent variables and mimic and extend the functionality that exists in the REALCOM software program. You will see that these templates are pretty big and involve coding in several languages (Python, WinBUGS model code, LaTeX and C++). It is hoped that with advances in the algebra system that the reliance on the `preccode` functions will reduce but if you want to look at the other multivariate templates you will see many similarities in the code in these functions. This is one of the plus points of the ability to view the code in the templates within the Stat-JR system.

We will finish this documentation by considering one more example of getting more from the MCMC estimation engine.

### 13 Out of sample predictions

Most of the statistical modelling templates we have thus far created are primarily being used for statistical inference. We might however be interested in using the model to predict future responses. The advantage of a simulation-based approach is that we can easily get confidence intervals about these predictors at the same time as we estimate the model. We do however have to be careful that we do not feedback the results of our predicting into the estimation part of the model. WinBUGS has a method to do this with its cut function and we have developed a similar method which we will demonstrate here.

#### 13.1 The 1LevelOutSampPred template – using the zxfd trick

We will illustrate our approach on a 1 level model which we can fit using the 1LevelOutSampPred template. We will firstly choose this template along with the tutorial dataset and then select the following inputs:
To explain what is going on we are planning to fit a regression model to `normexam` with predictor `standlrt` as we have done previously using the `1LevelMod` template. We will then use the predictors given in ‘missing explanatory variables’ to predict the 10 individuals who in this case have the same scores as the first 10 in the model actually fit. Note if you want to predict other individuals you need to form new columns of the same length as the data although the values below the ‘Number of missing’ row will be ignored. Clicking on the Next button gives the following output:

Here you will notice that we have an additional `j` loop in the model loop for the out-of-sample predictions which will be stored in `mnormexam`. There are two interesting parts to this code: Firstly the line

```
```

has the strange string `zxfd` placed in the middle of the two parameter names. This is our way of stopping the predictions from feeding back to the model parameter estimation (equivalent to performing the cut function in WinBUGS). Basically as the predictors in this line are not `beta0` and `beta1` then this line will not influence the posteriors for the fixed effects. The posterior for `mnormexam` will be calculated but this is only because we include the line

```
dummy[j] ~ ddummy(mnormexam[j])
```

so that `mnormexam` appears on both the left and right hand side within the model code to differentiate it from data. The algebra system will formulate the posterior which will depend on `beta[1]` and `beta[2]`. Of course in practice we want these replaced by the correct `beta0` and `beta1` and this is done in the bowels of the code generator with the lines:

```
eelif type == 'variable':
    result=l['name'].replace('.', '_').replace('zxfd', '')
```

which is in code that is not currently available to the user. If we run the template we get the following output:
Here you can see that the out of sample predictions, *mnormexam* have been estimated with standard errors. We have recently incorporated the ability to use additional datasets in a template and we may in the future update this template to allow the data to be predicted to be in a different dataset. We hope that this and other templates give you a flavour of the possibilities that are available in the Stat-JR package. The package is still evolving and so we very much welcome feedback and suggestions for improvement. We also encourage you to send us your own templates for inclusion with the software.

**Exercise 11**

Try modifying the 1levcat template to allow for out of sample predictions. Call the new template 1levelcatpred.

## 14 Solutions to the exercises

Rather than fill many pages with Python code we will place potential solutions to each of the exercises in a solutions directory on the e-STAT website at a later date. Below we list the filenames for each of the exercises:

- **Exercise 1** LinReg.py
- **Exercise 2** Random.py
- **Exercise 3** RecodeNew.py
- **Exercise 4** AvandCorr.py
- **Exercise 5** XYPlotNew.py
References


