

# **An Advanced User's Guide to Stat-JR (alpha release)**

Programming and Documentation by

William J. Browne

Bruce Cameron

Chris M.J. Charlton

Danius T. Michaelides\*

and

Camille Szmaragd

Centre for Multilevel Modelling,

University of Bristol.

\*Electronics and Computer Science,

University of Southampton.

December 2010

## Acknowledgements

The Stat-JR software is very much a team effort and is the result of work funded under two ESRC grants. The LEMMA 2 programme node (Grant: RES-576-25-0003) 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.

Both 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 and Hilary Browne from the LEMMA 2 node at the Centre for Multilevel Modelling for their input into the software.

David De Roure, Luc Moreau, Tao Guan, Toni Price, Mac McDonald, Ian Plewis, Mark Tranmer, Paul Lambert, Emma Housley 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.

# Contents

1.	About Stat-JR (e-STAT)	6
1.1	Stat-JR: software for scaling statistical heights	6
1.2	About the Advanced User's Guide	7
2.	Installation instructions	8
3.	A simple regression template example	12
3.1	Running a first template	12
3.2	Opening the bonnet and looking at the code	17
3.2.1	Invars	18
3.2.2	Outbug	19
3.2.3	Outlatex	21
3.2.4	Some points to note	22
3.3	Writing your own first template	23
4.	Structure of templates	24
4.1	Templating.py	24
5.	The parts of the Stat-JR system	27
5.1	webtest.py and the web interface	27
5.2	Model Templates and model.py	29
5.2.1	The compilemodel method	31
5.2.2	The Demo algebraic software system	31
5.2.3	Continuing the go function / applydata method	35
5.2.4	Parameter.py	36
5.2.5	Running a model – the run method and XMLtoC	37
5.2.6	Summarising the results	40
5.3	C code generation (XMLtoPureC)	41
5.3.1	Gencpp	42

5.3.2 XMLtoPureC .....	43
5.3.3 Example code for our template .....	43
5.4 JavaScript generation (XMLtoJS).....	44
5.4.1 Running the Regression1 example.....	46
5.4.2 JavaScript source code.....	47
6. Including Interoperability.....	48
6.1 Regression2.py .....	48
6.2 WinBUGS and Winbugsscript.py .....	49
6.3 MLwiN .....	53
6.4 R .....	58
6.5 STATA .....	61
7. Input, Data manipulation and output templates.....	65
7.1 Generate template (generate.py).....	65
7.2 Recode template (recode.py) .....	69
7.3 AverageandCorrelation template .....	71
7.4 XYPlot template.....	75
8. Single level models of all flavours – A logistic regression example .....	79
8.1 Invars.....	81
8.2 Methodinput .....	82
8.3 Outbug .....	83
8.4 Outlatex.....	84
9. Including categorical predictors.....	86
10. Multilevel models .....	91
10.1 2levelmod template.....	91
10.2 Nlevelmod template .....	96
11. Using the Precode/PreJcode methods.....	100
11.1 The ProbitRegression template .....	100

11.2	monitor_list method .....	102
11.3	precode method.....	103
11.4	prejcode method.....	105
12.	Multilevel models with Random slopes and the inclusion of Wishart priors .....	107
12.1	An example with random slopes .....	107
12.2	Precode for NlevwithRS.....	110
12.3	Multivariate Normal response models .....	114
12.4	The precode function for this template .....	116
13.	Improving mixing (1levblock and 1levorthogparam).....	121
13.1	Rats example.....	121
13.2	The 1levelblock template.....	122
13.3	The 1levelorthogparam template.....	125
14.	Out of sample predictions.....	130
14.1	The 1levpred template – using the zxfd trick .....	130
15.	Solutions to the exercises .....	133

# 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 on their initial journey up the mountain but also to cater for more advanced practitioners who are potentially having difficulty progressing further up the mountain but also want to assist their novice colleagues in making their own ascent as easy as possible so that they too can ascend the mountain of statistical knowledge.

One issue with complex statistical modelling is that to use the latest techniques can involve having to learn new pieces of software. To continue the mountain analogy – having climbed up one path with one piece of software this is like having to descend again and choose another path and another piece of software. 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. For example a user familiar with Stata could look at equivalent R code for fitting their model.

To complete the picture, the final group of users to be targeted by Stat-JR are 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 on peaks (possibly isolated) with limited links to the applied researchers who might benefit from their expertise. Stat-JR will again build links by including 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. As described in the next section, the Stat-JR system is built around the construction and use of templates – self contained objects to perform specific statistical tasks. Templates will be created by both advanced practitioners and algorithm writers and used and shared by these two groups as well as novice practitioners.

Many of the ideas within the Stat-JR system were the brain child 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 through the creation of your own templates that can be shared with others or simply through feedback on the existing templates

Happy Modelling. The Stat-JR team.

## 1.2 About the Advanced User's Guide

A major component of the Stat-JR package is the use of (often user written) templates. Templates are pieces of computer code (written in Python) 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 outputs. Templates appear to the end user as a window with a series of widgets and drop-down boxes which allow the user to specify the model to be fit, the data to be manipulated etc.

In this document it is our aim to give advanced users who intend to write their own templates, or more generally are interested in how the Stat-JR system works, more details about 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 does, and even in places each line of code. An initial question posed by potential template writers is what computer language are templates written in and when told 'Python' then ask whether we provide an introductory chapter on this language. We have not written an introductory chapter on Python (good books include Hetland (2005) and Lutz and Ascher (2004)) as it has many features 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 (or look at the Python website at <http://www.python.org/doc/> for further documentation) 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

To use the Stat-JR software requires several pieces of software to be present on your PC. We will endeavour in later versions to construct a purpose built install program but for now we provide a list of instructions for the user.

### 2.1 Install Python

You will firstly need to download and install Python from <http://www.python.org/download/>. Python is the language we use to construct much of Stat-JR and it is also the language that the templates are written in. We recommend the most recent in the 2.7 series of installs (currently 2.7.1). For now do not use the Python 3 series. If you have a 32bit operating system, choose the "Windows installer". If you have a 64bit operating system you can, if you wish, choose "Windows X86-64 installer" instead. If you choose this version then setting up the C++ compiler will be slightly different.

### 2.2 Install accompanying packages for Python

Next we need to install some of Python's accompanying packages so open <http://www.lfd.uci.edu/~gohlke/pythonlibs/>. The package names here take the form <package name>-<version>-<windows API version>-<Python version>.

You must choose packages based on the version of Python you selected above, for example if you choose the 32bit version of 2.7 you need those packages with a windows API version of win32 and a Python version of py2.7.

Locate and install the following five packages (choose the mkl version of the packages if it exists) :

1. NumPy
2. SciPy
3. Numexpr
4. Matplotlib
5. Setuptools

Note, you may want to use the search facility in your internet browser to locate the packages. When installing the packages, accept all the default file directories.

### 2.3 Install two further packages

Next, open a command prompt window (for example click the *Start* button then select *run*. Type *cmd* in the run window and click OK) and run the following commands:

```
cd \Python27\Scripts (If you installed Python elsewhere substitute this directory)
easy_install mako
```

```
easy_install web.py
```

These will install two further packages that we use with Python. Close the command window by running the exit command.

## 2.4 Install MinGW

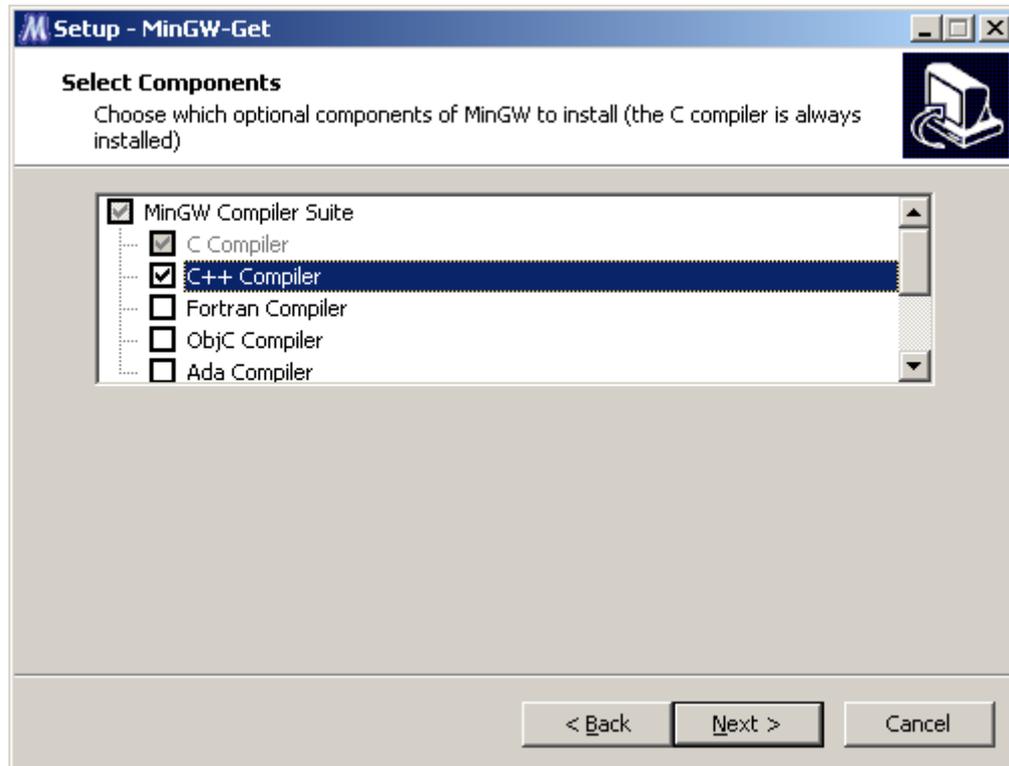
Next as we are using the C++ language within Stat-JR you will need to install a compiler. Again this will depend on your machine.

For a 32bit machine:

Select the following weblink:

<http://sourceforge.net/projects/mingw/files/Automated%20MinGW%20Installer/mingw-get-inst/mingw-get-inst-20101030/mingw-get-inst-20101030.exe/download>

Note, when you get to the following screen shot, make sure to click on the C++ Compiler check box before proceeding



For a 64bit machine:

Select the following web link:

<http://sourceforge.net/projects/mingw-w64/files/Toolchains%20targetting%20Win64/Personal%20Builds/>

Choose 4.5 x86\_64

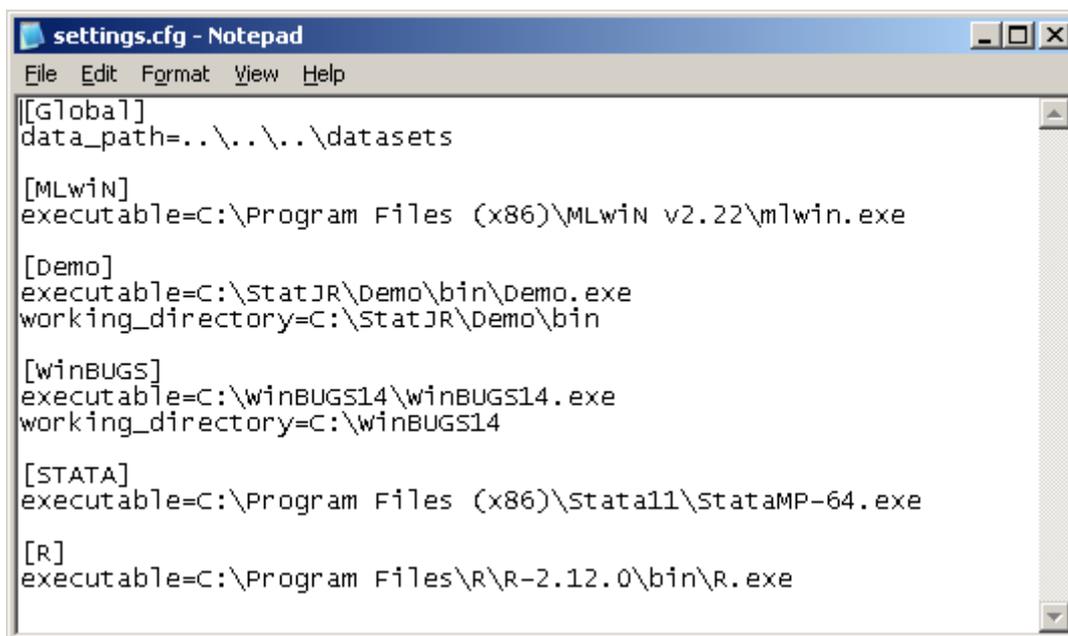
After unzipping the files rename the folder to MinGW

## 2.5 Download Stat-JR source files

We finally need to download the source files that are specific to Stat-JR. You will find these at (weblink to be determined but for the alpha release we will have E-mailed you a fluff link)

Unzip StatJR.zip to C:\StatJR (or a location of your choosing) which we will refer to as the Stat-JR base directory. When we refer to subdirectories they will be housed under this directory. When all the files are installed you will find a file called *webtest.cmd* in the base directory. This is the file we will use for running the software. If you have installed MinGW to a location other than C:\MinGW edit the *webtest.cmd* to reflect this.

The *webtest.cmd* file runs a Python script file called *webtest.py* which can be found in the subdirectory *src/apps/webtest*. In this directory you will also find a *settings.cfg* which contains paths for various components of the Stat-JR system. A screen shot of this file is given below:



```
settings.cfg - Notepad
File Edit Format View Help
[[Global]
data_path=..\..\..\datasets

[MLwin]
executable=C:\Program Files (x86)\MLwin v2.22\mlwin.exe

[Demo]
executable=C:\StatJR\Demo\bin\Demo.exe
working_directory=C:\StatJR\Demo\bin

[winBUGS]
executable=C:\winBUGS14\winBUGS14.exe
working_directory=C:\winBUGS14

[STATA]
executable=C:\Program Files (x86)\Stata11\StataMP-64.exe

[R]
executable=C:\Program Files\R\R-2.12.0\bin\R.exe
```

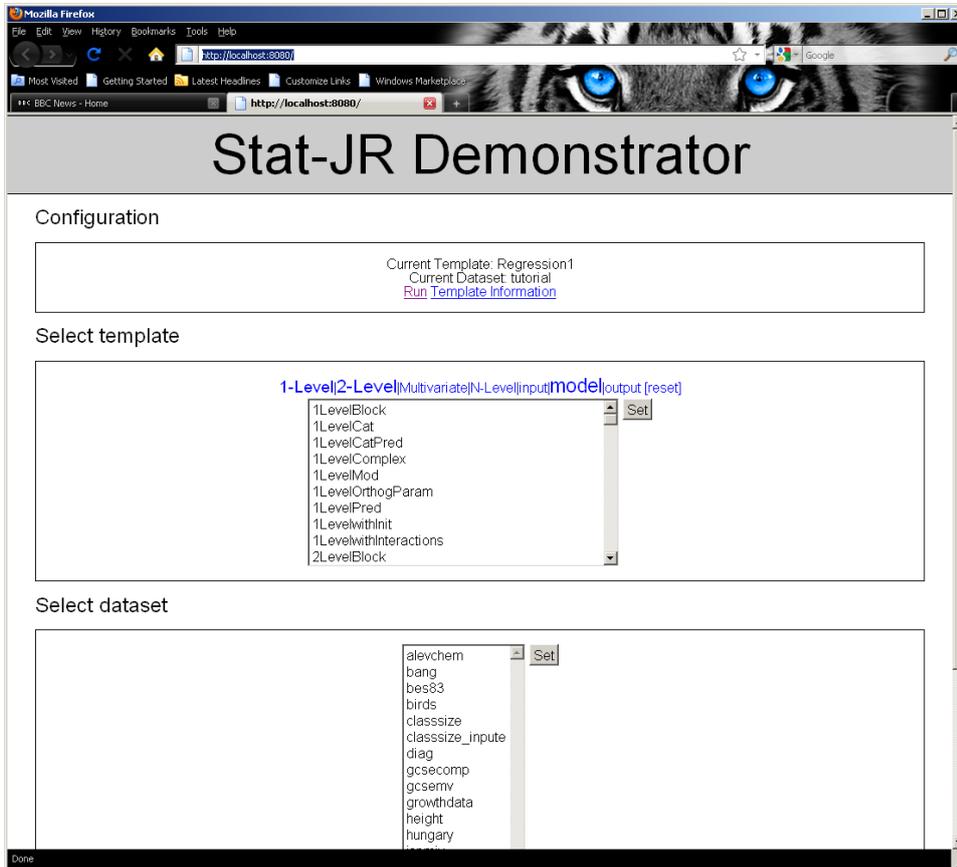
Note that if your base directory is something other than C:\StatJR then you will need to change the path name in this file for the Demo algebra system accordingly.

For interoperability you will need to change the paths for the various third-party software packages (WinBUGS, Stata and R).

## 2.6 Check that the installation has been successful

You can now try to run Stat-JR to check that the installation has been successful. Double click on the *webtest.cmd* file in the base directory. This will fire up a command prompt window and your default browser. At present you will normally get an error in your browser as it fires up before the command

prompt window is ready but this should be fixed by waiting and refreshing the window. As an alternative the Python file webtest.py (which is being run by webtest.cmd and is located at src\apps\webtest) can be run directly. Once this has started you can open your browser and type in the web link <http://localhost:8080/> to connect with webtest. If the installation has been successful, you will see the following webpage.



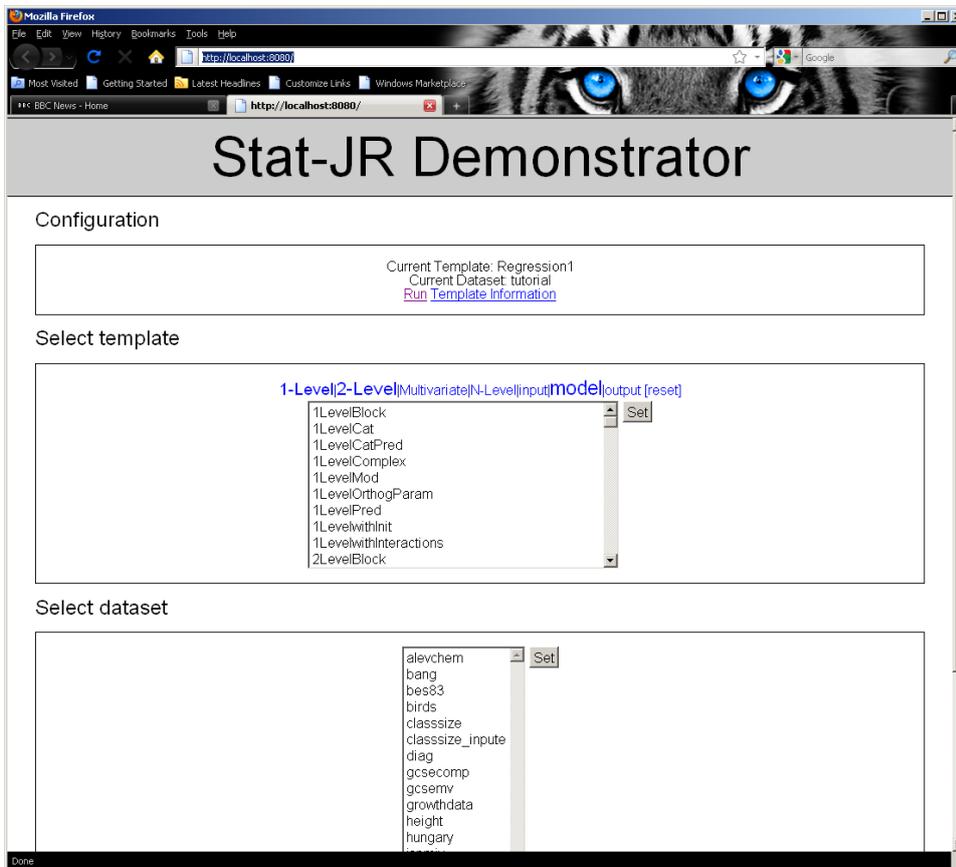
## 2.7 Add a shortcut to Stat-JR on your Start menu or on your desktop

Finally, add a short cut to Stat-JR on your Start menu or on your desktop to enable you to easily find and launch the software at the beginning of each session.

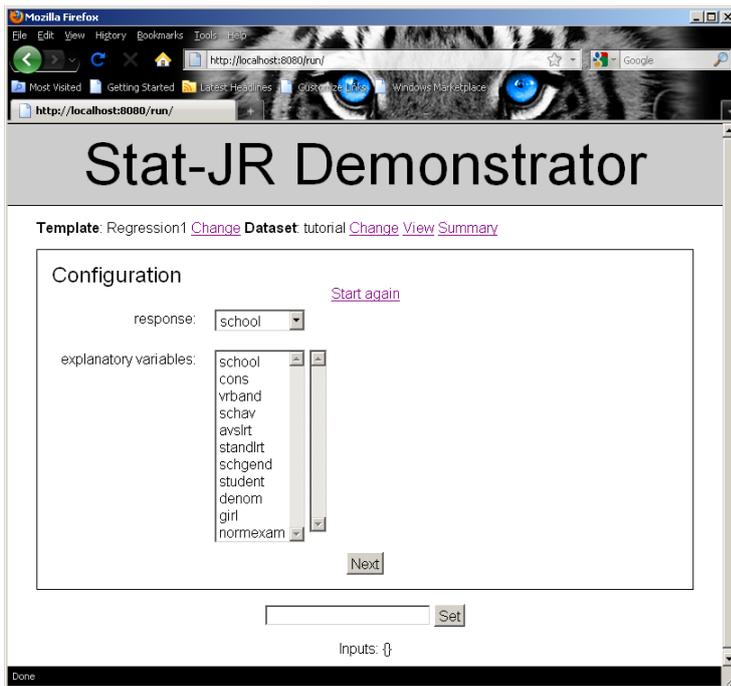
### 3. A simple regression template example

#### 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*. Perhaps before looking at the code it would be good to run the template in the web interface to see what it does. To do this, launch Stat-JR (full installation instructions are given in Chapter 2). If you refresh the screen you should be greeted by a display similar to the following:

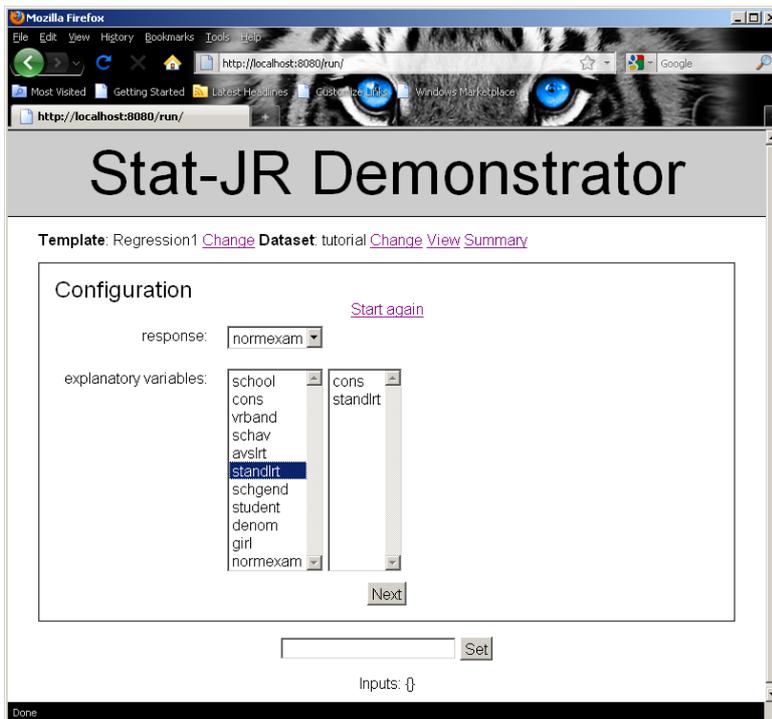


The Configuration box shows that the template *Regression1* is the default template and the default dataset is called tutorial. We will click on the **Run** button to run the template and obtain the following display:

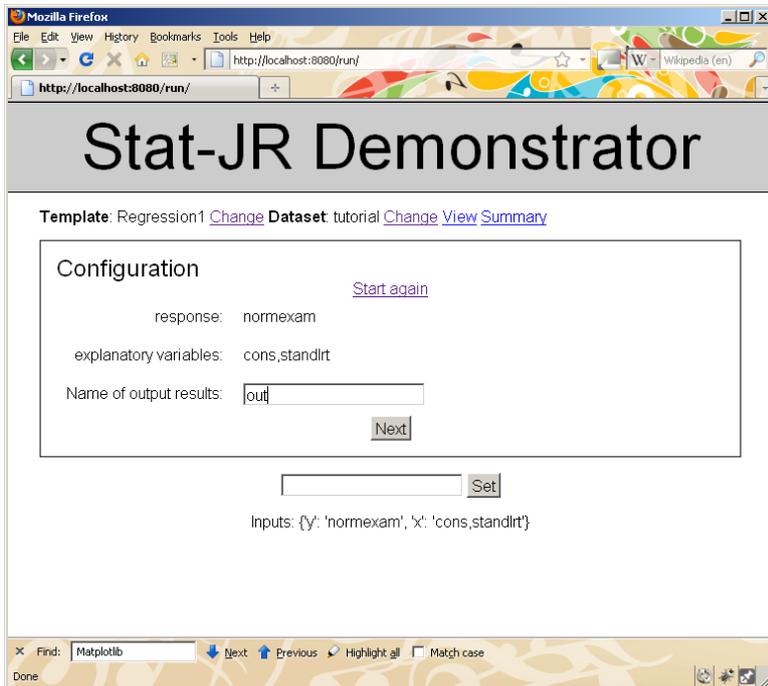


Here you will see a **Configuration** box which is looking for inputs for a response (a single select list) and explanatory variables (a multiple select list). We will start by fitting a very simple regression model, regressing *normexam* (exam scores at age 16) on *standlrt* (A London reading test taken by the same children at age 11).

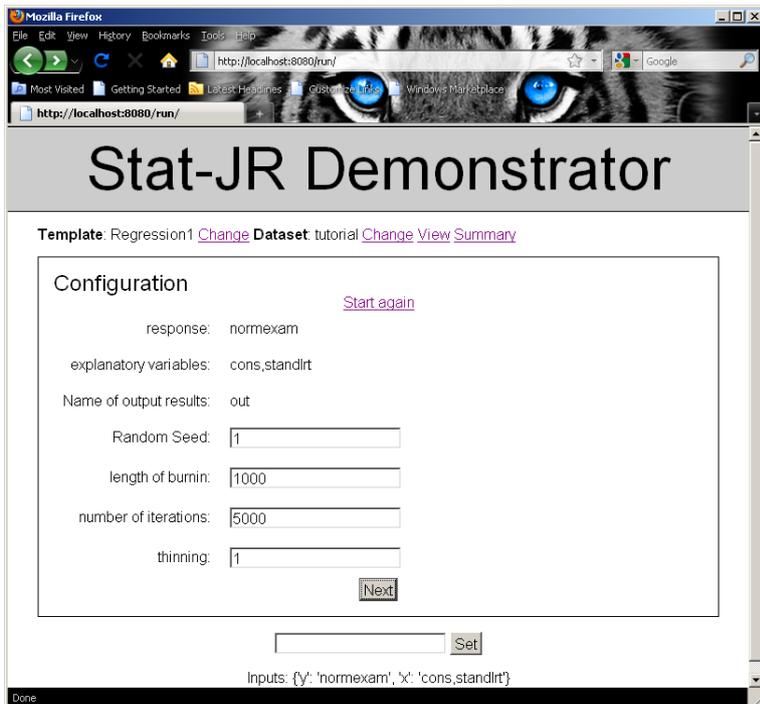
We will select *normexam* as the response and *cons* and *standlrt* as the explanatory variables as shown below:



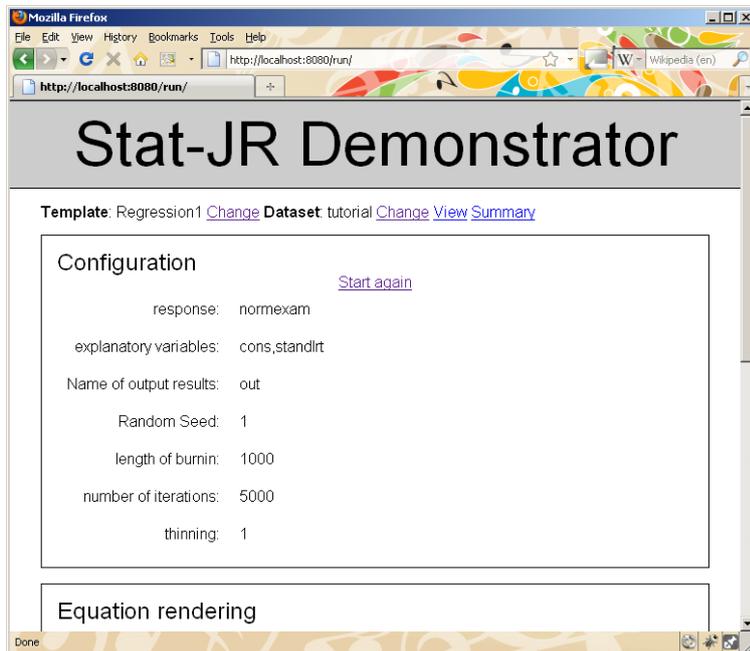
Click on the **Next** button and then specify the Name of output results (we have used 'out' here) as shown below.



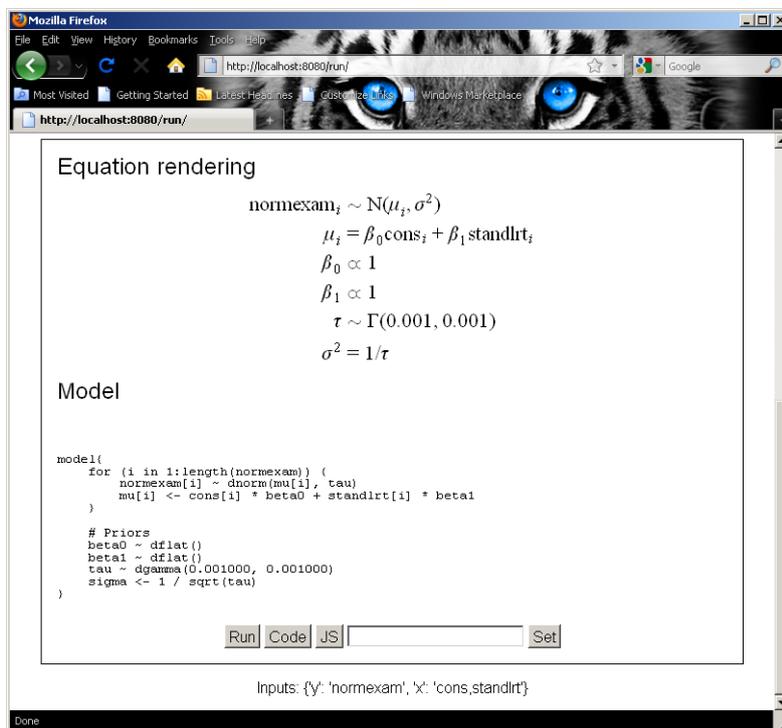
Click Next and we will next have to fill in some estimation method specific boxes. The built-in engine within STAT-JR is an MCMC engine and so we hence have to type in some boxes specific to this type of estimation as follows:



Click **Next**. The configuration box now summarises the specified model and the MCMC estimation options.

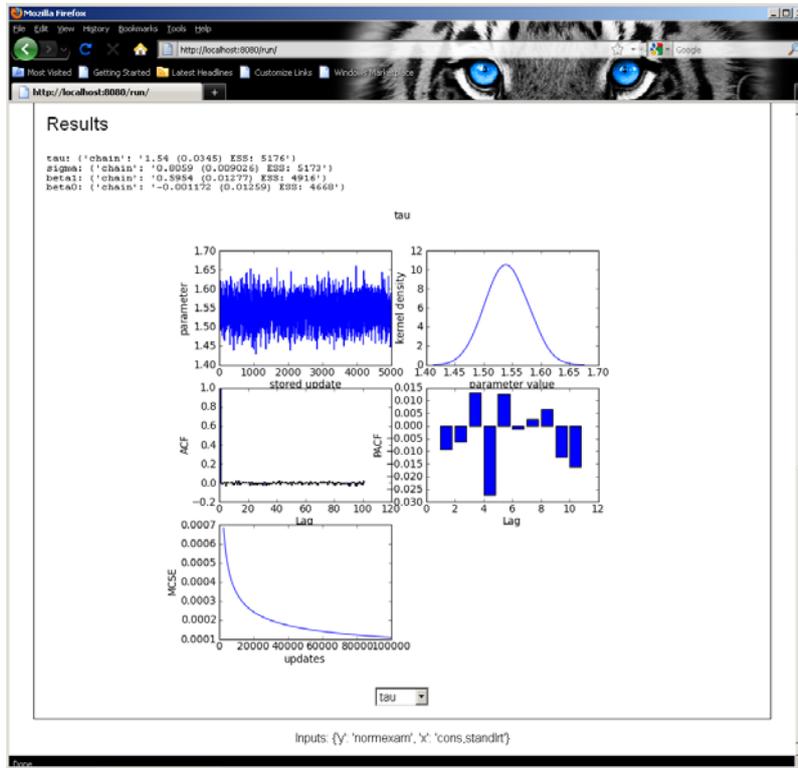


Scroll down the web page to the **Equation rendering** box:



The top half of the box contains a nicely formatted mathematical description of the model (in LaTeX code). The bottom half has some code that is written in a variant of the model specification language associated with the WinBUGS package. There are several buttons at the bottom of the window and if

we click on the **Run** button we will run the model and after a short while we will discover the page lengthens to incorporate a final box entitled **Results**:



This screen contains summary statistics for four parameters: beta0, beta1, tau and sigma. The last of these, sigma, is simply a function of the precision parameter, tau. Thus, the model intercept and slope coefficients are -0.001 and 0.595 respectively while the corresponding standard errors are 0.013 and 0.013. The residual variance is estimated as 0.65 (= 0.806<sup>2</sup>).

The screen also displays a range of figures for the MCMC output for the tau parameter. There is a drop-down box at the bottom of the screen from which we can choose to view the MCMC graphs for the other parameters. The five figures contain a trace plot of the 5,000 estimates, a kernel density plot, plots of the autocorrelation (ACF) and partial auto-correlation (PACF) functions for assessing mixing of the chains and a Monte Carlo standard error (MCSE) plot.

We will not go into detail about what these figures mean in this Advanced User's Guide; interested readers can look at the accompanying Novice Practitioner's Guide (to be written in 2011) for such information. The aim here is to teach you how to write a similar template yourself.

## 3.2 Opening the bonnet and looking at the code

The operations that we have 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*. If we open *regression1.py* (in a text editor such as Wordpad or Notepad rather than Python) we will see the following:

```
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' ]

    invars = '''
y = DataVector('response: ')
tau = ParamScalar()
sigma = ParamScalar()
x = DataMatrix('explanatory variables: ')
beta = ParamVector()
beta.ncols = len(x)
'''

    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)
  sigma <- 1 / sqrt(tau)
}
'''

    outlatex = r'''
\begin{aligned}
& \mbox{\${y}}_i \text{ \& \sim \mbox{N}(\mu_i, \sigma^2) \ \} \\
& \mu_i \text{ \& = } \\
& \text{\${mmulttex}(x, r'\beta', 'i')} \ \} \\
& \% \text{for i in range(0, len(x))} \\
& \beta_{\text{\$i}} \text{ \& \propto 1 \ \} \\
& \% \text{endfor} \\
& \tau \text{ \& \sim \Gamma (0.001,0.001) \ \} \\
& \sigma^2 \text{ \& = 1 / \tau}
\end{aligned}
'''
```

```
\end{aligned}
'''
```

We will now describe in some detail what this code does. The first three 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 *Regression1* which is a subclass of a generic *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 the word class to describe 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 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 (class, object, subclass, attribute) is used in what is called object orientated programming.

In the definition here four attributes (**tags**, **invars**, **outbug**, and **outlatex**) are then defined as being parts of a *Regression1* class although there will be other attributes that are generic to the template class and are defined elsewhere.

Briefly:

1. The **tags** attribute identifies the template as belonging to the tag groups 'model' and '1-Level' and this is used in the web interface to decide which templates to show in specific template lists. Thus, the regression1 class is a template for specifying 1-level (i.e. single-level) models.
2. The **invars** attribute is a text string (hence the starting and ending "'") which consists of a list of the inputs in this template.
3. The **outbug** attribute is a text string that will produce the model code we saw in the web interface for this template.
4. The **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 now look at the last three attributes in more detail.

### 3.2.1 Invars

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

```
y = DataVector('response: ')
```

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 drop-down box).

Stat-JR distinguishes between Data objects (these start with the prefix Data) which require user inputs and Parameters (these start with the prefix Param) which just need to be declared. This template therefore has 5 declarations for the 5 components in the template - 2 pieces of data ( $y$  and  $x$ ) and 3 parameters ( $\tau$ ,  $\sigma$  and  $\beta$ ) that make up the model. The DataMatrix declaration for  $x$  will correspond to the multiple select list that we saw when running the regression1.py template in Section 3.1. The final command

```
beta.ncols = len(x)
```

is used to define the size (`ncols`) of `beta` so that there is a separate `beta` associated with each explanatory variable.

### 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 under the **Model** banner 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:

```
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)
  sigma <- 1 / sqrt(tau)
}
'''
```

Now we can substitute *normexam* for  $\$y$  as this is the variable we chose for  $y$  thus:

```
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)
sigma <- 1 / sqrt(tau)
}
'''

```

Next we can evaluate the for loop with, in our example  $x$  having 2 columns (as input by the user), and the `.ncols` function looks up the number of columns in  $x$ . The Python range function does not include the upper bound in the range, so in this case  $i$  will take values 0 and 1:

```

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

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

```

Finally the function `mmult` is a function written separately and is used to create the products of the  $x$  variables (i.e. matrix multiplication) and their associated *betas* with appropriate indexing:

```

outbug = '''
model{
  for (i in 1:length(normexam)) {
    normexam[i] ~ dnorm(mu[i], tau)
    mu[i] <- cons[i]*beta0 + standlrt[i]*beta1
  }

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

```

This is identical to the code we see under **Model** in the webtest 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 under **Equation Rendering**. 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 Show Source option you will get a Window popping up that shows the source:

```

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

```

This code is created via the *outlatex* function and we will now look at how we get from *outlatex* to this source for our example. The generic code is as follows:

```

outlatex = r'''
\begin{aligned}
\mbox{\${y}}_i & \sim \mbox{N}(\mu_i, \sigma^2) \\
\mu_i & = \\
& \${mmultttx(x, r'\beta', 'i')} \\
%for i in range(0, len(x)):
\beta_{\${i}} & \propto 1 \\
%endfor
\tau & \sim \Gamma(0.001, 0.001) \\
\sigma^2 & = 1 / \tau
\end{aligned}
'''

```

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

```

outlatex = r'''
\begin{aligned}
\mbox{normexam}_i & \sim \mbox{N}(\mu_i, \sigma^2) \\
\mu_i & = \\
& \${mmultttx(x, r'\beta', 'i')} \\
%for i in range(0, len(x)):
\beta_{\${i}} & \propto 1 \\
%endfor
\tau & \sim \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:

```
outlatex = r'''
\begin{aligned}
\mbox{normexam}_i & \sim \mbox{N}(\mu_i, \sigma^2) \\
\mu_i & = \\
& \${mmulttex(x, r'\beta', 'i')} \\
\beta_0 & \propto 1 \\
\beta_1 & \propto 1 \\
\tau & \sim \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` :

```
outlatex = r'''
\begin{aligned}
\mbox{normexam}_i & \sim \mbox{N}(\mu_i, \sigma^2) \\
\mu_i & = \\
& \beta_0 \mbox{cons}_{i} + \beta_1 \mbox{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 i.e. a string that doesn't contain any special control characters. 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 introduce in chapter 5. It is enough for now to realise that to write some basic templates simply requires writing code similar to that seen here; 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 to write templates to begin with 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 (i.e. with one explanatory variable in addition to a constant). 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. (Note that you will have to close the command window before rerunning Stat-JR and then refreshing the browser) 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. Structure of templates

From Chapter 3 you should now have an idea of the tasks involved in writing a ‘simple’ template. In later chapters we will give further information on how one writes more complicated templates. We will use a similar structure of firstly introducing the template and using it in the software. Then opening up the bonnet and explaining aspects of the code, in particular what is different in this template from earlier templates.

We mentioned in chapter 3 that in the code for a template we are effectively giving a definition for a specific subclass of the *template* class and that there is a generic *template* class that these templates inherit attributes from. In this chapter we will give a little more detail of some of these additional attributes before explaining further how the model description generated by the *outbug* attribute links in to the rest of the software in Chapter 5.

You will find many of the inner workings of the Stat-JR system in python files within the subdirectory `src/lib/EStat`. The file *Templating.py* contains amongst other things the class definition for the generic *template*.

### 4.1 Templating.py

This python file contains the class definitions of many of the building blocks used in Stat-JR including the different data and parameter types we have seen used in the *invars* attribute of the *Regression1* class. We are here primarily interested in the *template* class which is the last class defined within the file (although the code is near the top of the file).

The *template* class contains definitions of lots of methods that can be used for a template. Again we are using object orientated programming terminology here and so a *method* is a function described within a *class* definition which performs operations on the *instance* of the *class*. Let us return to our rectangle *class* which had two *attributes* length and width. An example *method* for this *class* could be area which would be a function that multiplies the length *attribute* by the width *attribute* and hence returns the area. A second *method* might be perimeter which adds 2\*length to 2\*width. We will illustrate just two *methods* which might help explain a little about what is going on. Firstly all *classes* will have an *initialisation method*, often called a *constructor* in object oriented programming terminology, and in Python these are usually written as `__init__`

The code for initialisation is short:

```
def __init__(self):
    self.objects = MyDict()
    self.outputs = {}
    self.EstObjects=MyDict()
```

You will notice that all *methods* refer to ‘self’ which says that the *method* operates on this particular *instance* of the *class*. The initialisation code simply sets up 3 *attributes* within the *instance* of the

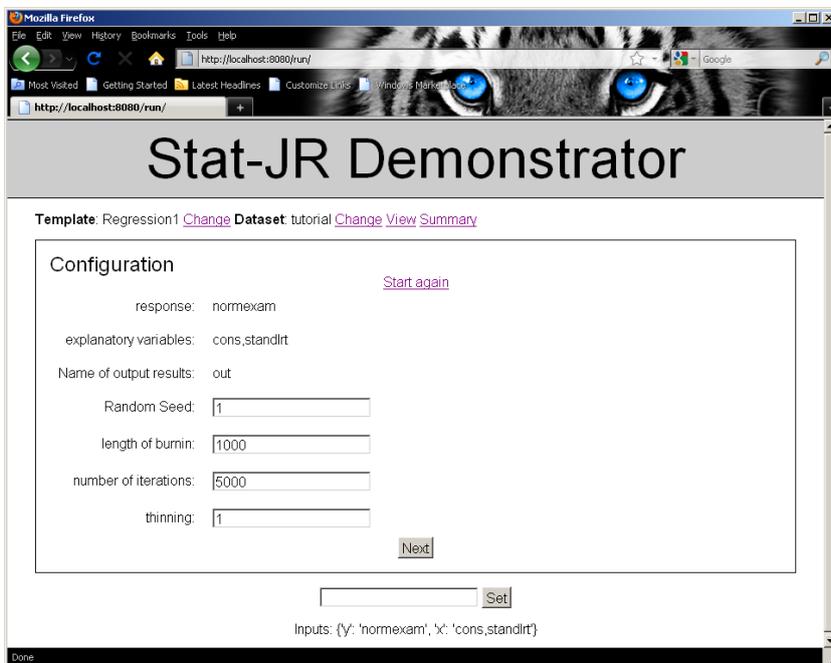
*class* – *objects*, *outputs* and *EstObjects*. Currently *objects* and *EstObjects* are empty MyDict *objects* and *outputs* is an empty dictionary. So we have here populated an *instance* of the *class* with some placeholders for *attributes* that will be created properly elsewhere.

To illustrate this further we will look at the *MethodInput* *method*. This *method* is used to ask for some of the other user inputs we saw being input when we ran the *Regression1* template.

The code for this *method* is as follows:

```
# Create another set of inputs related to estimation method and engine.
Will also define type of outputs specific to method/engine
def MethodInput(self):
    self.EstObjects['Engine']=Text(value = 'eSTAT')
    self.EstObjects['seed']=Integer('Random Seed: ')
    self.EstObjects['burnin']=Integer('length of burnin: ', min=0)
    self.EstObjects['iterations']=Integer('number of iterations: ',
min=0)
    self.EstObjects['thinning']=Integer('thinning: ')
    self.EstObjects['nchains']=1
```

This function is essentially populating the dictionary known as *EstObjects*. *EstObjects* is an instance of a MyDict *class* that has been defined elsewhere and is essentially a list of *objects* of varying types where each *object* has a label and a value. The set of lines within the *MethodInput* *method* then define the labels and types of all *objects* in the dictionary and either assign a value to the *object* - as for the *Engine* *object* or give a text string that appears on the screen when this function is called as for the *seed* *object*. We repeat below the inputs for the *Regression1* example:



You will see the four inputs correspond to the four inputs in the *MethodInput* *method*. The *template* *class* has a large number of *methods* and here we will simply just list them and give a very brief indication of what they do. We will then revisit some of them later:

- `__init__` - this is the initialisation *method* that we described earlier
- `get_tags` – a class *method* linked to the *tags*
- `render_text` – renders the text in the browser
- `applydata` – this function is more important as it actually connects the real data and initial parameter values to the model object that is being formed.
- `preparedata` – this function can do preprocessing of the data – although this is template specific and so the default function simply attaches the data into the instance of the class under the attribute `data`.
- `resultdata` – this function is used to construct the results of a model fit and place within the instance of the class as an attribute named `results`.
- `monitor_list` – this function sets which of the variables are to have their changes monitored
- `Run` – This may be used in the future to give templates more control over what happens when they are run
- `resultout` – A function that gets the only other input – the name of the dataset where the results will be stored.
- `MethodInput` – the function previously described to get hold of estimation method inputs
- `input` - a function that links with the *invars* attributes to get hold of model inputs
- `output` – a function that links with the *outbug* attribute to create the model output code for the algebra system.
- `outputlatex` – a function that links with the *outlatex* attribute to create the LaTeX output.
- `outputBUGS` – a function that will link with the *outWBug* attribute (where present) to create true WinBUGS model files.
- `preccode`, `postcode`, `prejcode`, `postjcode` – by default each returns a blank string but can be replaced by template specific code that allows the user to insert sections of C code or Java script into the code formed by Stat-JR. We will see how this works later.

At this stage it is enough to know that there is a lot of generic functionality that has been written and is common to all templates and that although the example template is quite small a lot is going on behind the scenes. In the next chapter we will look at how creating an output BUGS like model statement fits into the whole Stat-JR system.

## 5. The parts of the Stat-JR system

In this chapter we will look at how the model templates that we write link into the system as a whole. We will show the template classes link with a separate generic model class that is used within the bowels of the Stat-JR system to store the model and create an algorithm to fit the model. This chapter is fairly technical and some readers might find it easier to skip through to chapter 6. We will begin by looking briefly at the current **webtest** interface into the Stat-JR system.

### 5.1 webtest.py and the web interface

To start up the software for the earlier Regression1 example we used a piece of Python code called *webtest.py* found in the `src/apps/webtest` subdirectory. The **webtest** file as the name suggested allows a web interface to be used to run templates from.

The *webtest.py* file basically starts up a web based application on your machine and links together html files that can be found in the *html\_templates* subdirectory of the webtest directory with the user written templates.

There are several class definitions within *webtest.py*: `run`, `info`, `summary`, `data`, `image`, `index` which relate to which html file is currently being viewed. When *webtest* is started up the `index.html` file which contains the lists of templates and datasets is used. There are two ways in which user input is then dealt with.

Firstly the user might click on for example the **Set** buttons which will involve the code in the class `index` being executed. You will see in the `index` class definition the lines

```
i = web.input()
if 'dataset' in i:
    context['datasetname']=i['dataset']
if 'template' in i:
    context['templatename']=i['template']
```

and this is how the user input is captured from this initial screen i.e. here the `web.input` function will return which object on the index window has changed and what it's value is to allow the user to change the dataset and/or the template that is currently set. We will not go into detail on how the lists are populated with the available templates/datasets and how tagging is implemented but for the interested reader this is done directly in the html file.

The second sort of input is a click on a hyper-link, so for example a click on the word `Run` at the top of the screen. This works simply through HTML and the *run.html* page is called and replaces the *index.html* file in the browser. An object of type *run* is created and now the class code for the *run* object is important.

The *run* object is driving much of the Stat-JR package. When called firstly the *GET* method is called which initialises the screen and ends with a call to the *build\_form* method whose code is at the bottom of the class definition. Here there is some initial setting up code including the line:

```
t=context['template']
```

which sets up *t* to be an instance of the current template and then the following chunk of code:

```
try:
    t.input()
    print "finished"
    t.resultout()
    if GUIObject.interface.unanswered_questions():
        print "not finished yet"
        raise GUIExceptionInputNeeded()
    if hasattr(t, 'outbug'):
        t.MethodInput()
        if GUIObject.interface.unanswered_questions():
            print "not finished yet"
            raise GUIExceptionInputNeeded()
    elif hasattr(t, 'graphdata'):
        t.EstObjects['Engine'] = Text()
        t.EstObjects['Engine'].name = 'output'
    elif hasattr(t, 'tabledata'):
        t.EstObjects['Engine'] = Text()
        t.EstObjects['Engine'].name = 'taboutput'
    else:
        t.EstObjects['Engine'] = Text()
        t.EstObjects['Engine'].name = 'service'
```

This code shows the general order of things: we start by getting the template's inputs through the call to *t.input*, then we get hold of the output file name through *t.resultout* and then we check out which attributes the template has (a way of deciding on the type of template) in order to know which methods to run next. For our *Regression1* example we have an attribute *outbug* and so we call *MethodInput* which gets the remaining inputs e.g. random number seed etc. Non-model templates will not have an attribute *outbug*.

You will note in two places the lines:

```
if GUIObject.interface.unanswered_questions():
    print "not finished yet"
    raise GUIExceptionInputNeeded()
```

These lines interrogate the window to check whether all inputs have been entered and so stop the additional buttons being shown until all inputs are given. Having got all inputs we are ready to call other template functions using the following chunks of code:

```
state = "ready"
context['variables']=t.preparedata(context['data'])
if hasattr(t, 'WinBUGSMod'):
    statstemplate=t.WinBUGSMod()
elif hasattr(t, 'outbug'):
    statstemplate=t.output()
if hasattr(t, 'outlatex'):
    latextemplate=t.outputlatex()
```

The *preparedata* method is called to link in data into an attribute called *variables* and then, depending on the template type (and whether we are using WinBUGS – see chapter 6), the *output* and *outputlatex* methods are called and allocated to *statstemplate* and *latextemplate* respectively. These calls are to update the screen and display the model code and LaTeX.

At this stage the buttons to allow the model to be run or for C++ or Java code to be generated appear in the web-browser. Each of these buttons correspond to a different value of input within the *POST* method chunk of code. (Note in the html code we switch to the *POST* method once all inputs are generated i.e. at the same point as the buttons appear.) We will describe what happens when we fit a model or generate source code in later subsections but we next need to turn our attention to the concept of a model (as opposed to a template).

## 5.2 Model Templates and model.py

In Stat-JR we are anticipating advanced users writing their own templates and our definition of a template is an object that takes user input and produces sufficient output for the system to do a particular task. When we describe model templates their task is to get enough information to set up a model object that can then be used to fit a statistical model. We have a specific type of model object that is used to fit models using the estimation engine within Stat-JR. In chapter 6 we will look at how things work when we don't use this engine but instead use other packages.

The definition of the *model* class can be found in the file *model.py* found in the *src/lib/EStat* subdirectory and the methods within the *model* class are what are used to fit a model using the Stat-JR system. Continuing our exploration of the *webtest* code you will find that if the *run* button is pressed then the following code is executed:

```
if 'run' in i:
    #MLwiNEst = None
    m = None

    if t.EstObjects['Engine'].name=='eSTAT':
        if 'seed' in t.EstObjects != '':
            seed = int(t.EstObjects['seed'].name)
        else:
            seed = 1
        thinning = 1
        if 'thinning' in t.EstObjects:
            thinning = int(t.EstObjects['thinning'])
        m = go(t, context['variables'], int(t.EstObjects['burnin']),
            int(t.EstObjects['iterations']), thinning, seed, t.monitor_list, callback)
```

Here the first if statement establishes that we are intending to run a model and the next if statement checks we are running the in-house estimation engine (currently called eSTAT). Then we have a few initialising lines before the important *go* function is called and the instance of model object *m* is allocated the output of *go*.

The *go* function can be found near the top of *webtest.py* and has the following code:

```
def go(t, variables, burnin, iterations, thinning, seed, mon, c):
    m = Model()
    m.modelspec = t.output()
    m.preccode = t.preccode()
```

```

m.postccode = t.postccode()
m.compilemodel()
m.callback = c
m.burnin = burnin
m.iterations = iterations
m.thinning = thinning
m.seed = seed
t.applydata(m, variables)

allmissing = True
for p in m.params.values():
    if p.tree != '':
        allmissing = False
if allmissing == True:
    print "Error in algebra system, nothing generated"
    return None

if hasattr(t, 'customsteps'):
    for p in t.customsteps:
        print 'Custom step for parameter: ' + p
        m.params[p].tree = ''

for p in mon():
    m.params[p].monitor = True
m.run()
m.Summarise()
return m

```

Most of what this code is doing is setting up the model object *m*. The first line initialises the model object and this code can be found by looking at the `__init__` method for the *model* class (in *model.py*). Basically this is just giving names to all the attributes within the *model* class and setting default values for these attributes.

```

def __init__(self):
    self.directory = ''
    self.modelspec = ''
    self.params = {}
    self.data = {}
    self.burnin = 5000
    self.iterations = 100000
    self.thinning = 1
    self.seed = 1
    self.callback = lambda i, p: i
    self.identify = True
    self.code = ''
    self.preccode = ''
    self.postccode = ''
    self.prejcode = ''
    self.postjcode = ''
    self.__initialised = True

```

Having initialised the model we next pass on some of the information from the template to the model, the model specification and any pre or post C++ code that is needed (see chapter 11 for an example of such code). There is then a call to a method *compilemodel* which does a large number of tasks and which needs a chapter of its own.

### 5.2.1 The compilemodel method

The *compilemodel* method basically takes the modelspec and sends it to an algebra system that constructs the conditional posterior distributions for the parameters in the model. The code is as follows:

```
def compilemodel(self):
    if self.modelspec != '':
        print "Compiling Model..."
        self.directory = tempfile.mkdtemp()
        handle, name = tempfile.mkstemp(dir=self.directory)
        f = os.fdopen(handle, 'w+b')
        f.write(self.modelspec)
        f.flush()
        f.close()
        executable = EStat.configuration.get('Demo', 'executable')
        print "Running " + executable

        cmdline = "%s %s %s /Identify:%s" % (executable, name,
self.directory, self.identify)
        cwd = ''
        if EStat.configuration.get('Demo', 'working_directory'):
            print "cd-ing from: " + cwd
            cwd=os.getcwd()
            os.chdir(EStat.configuration.get('Demo',
'working_directory'))

        os.system(cmdline)

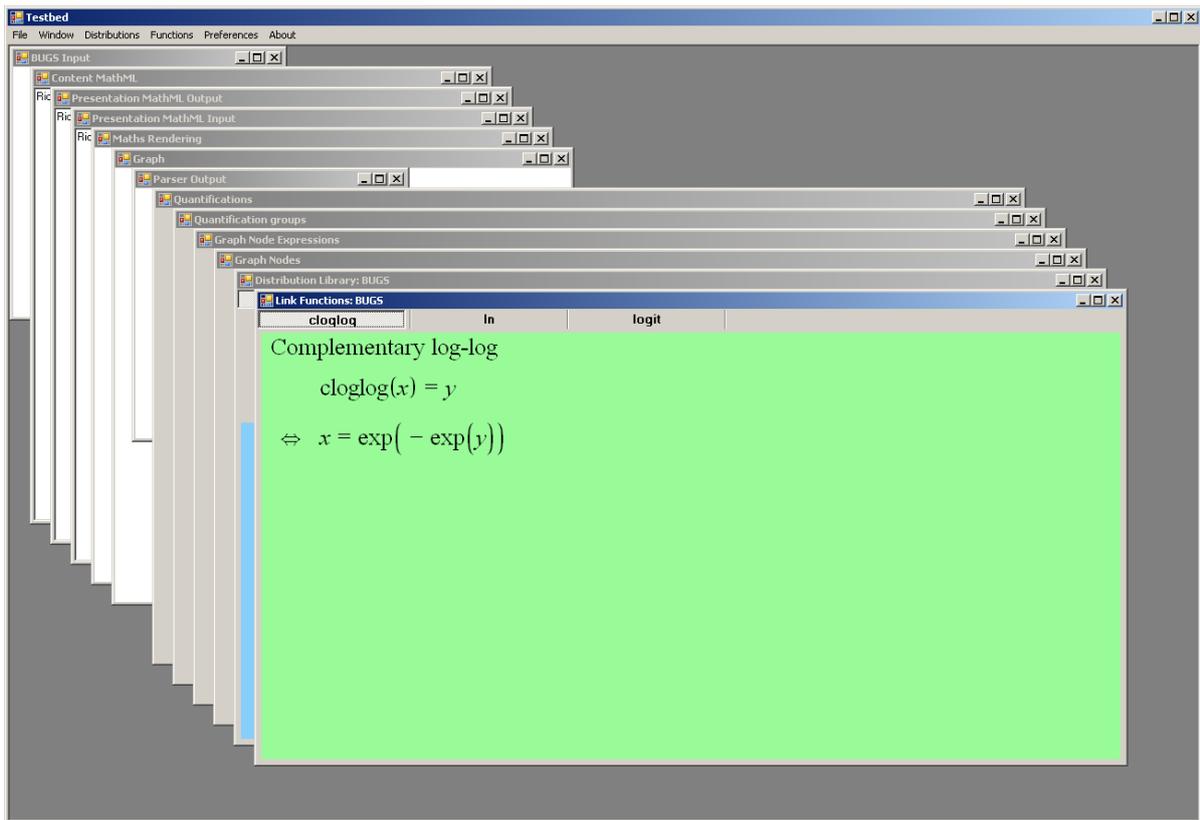
        if cwd:
            os.chdir(cwd)
```

This code basically writes the model specification to a file and then runs another piece of software using this file as input.

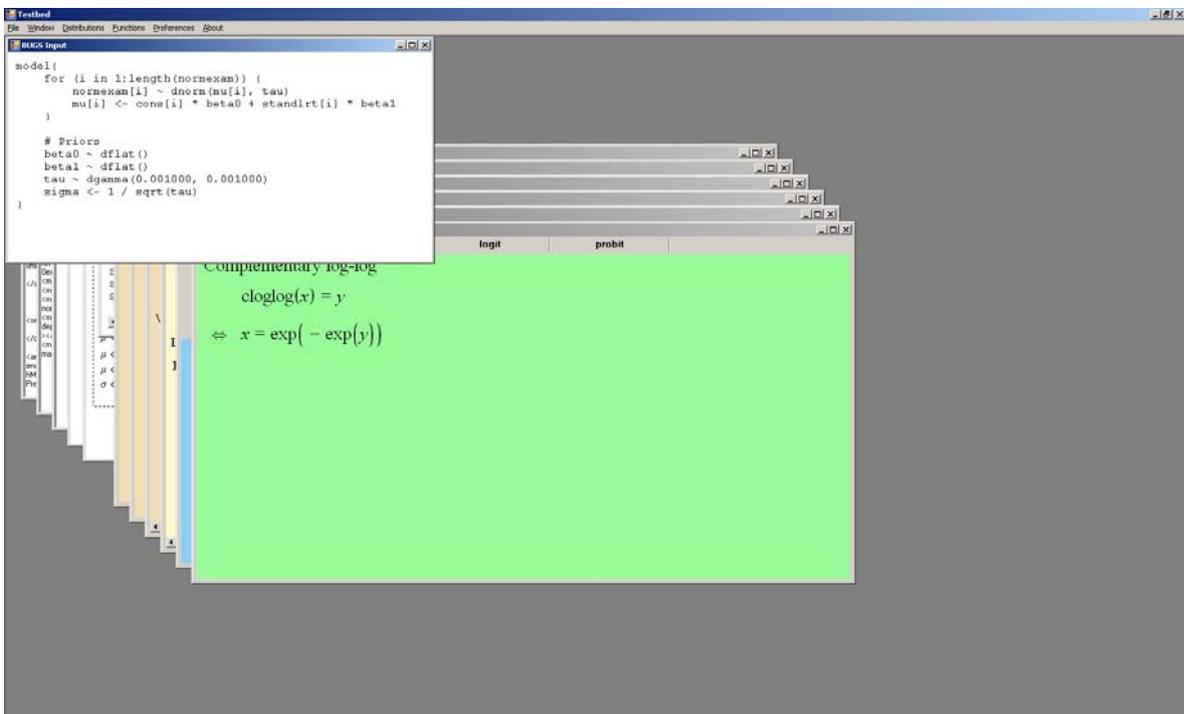
### 5.2.2 The Demo algebraic software system

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** 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 a large number of windows:

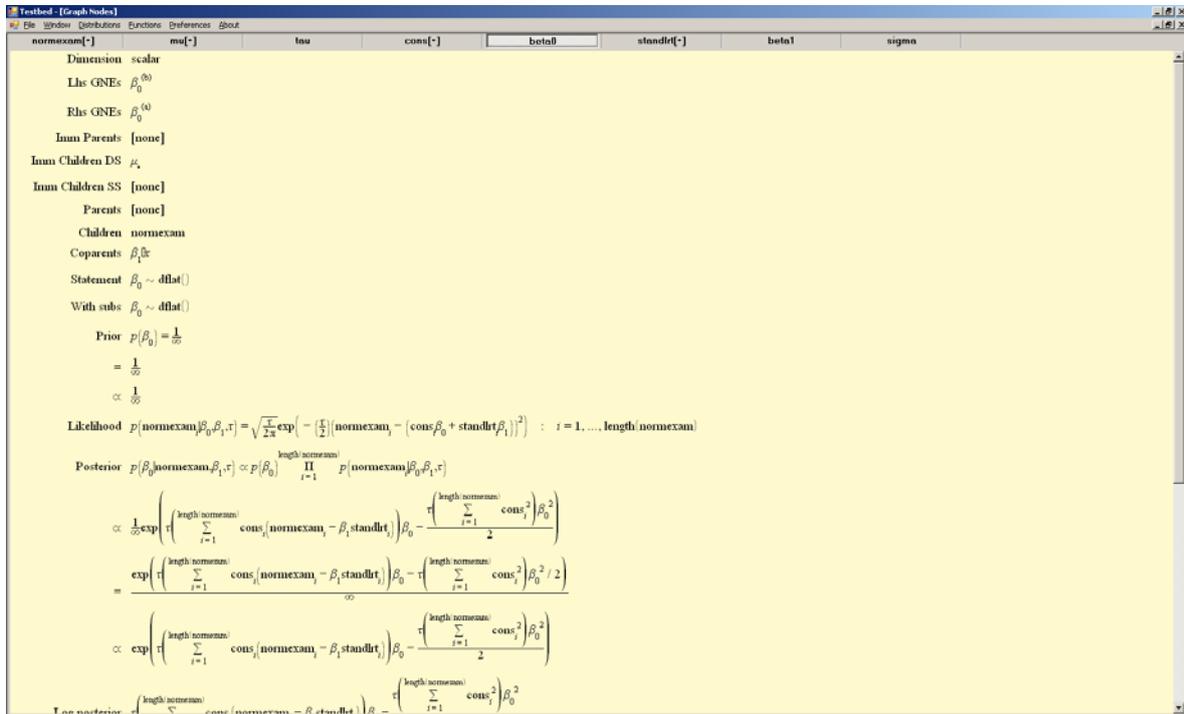


You will need to copy the model for the *Regression1* template onto the clipboard and then paste this into the window entitled **BUGS Input** as follows:

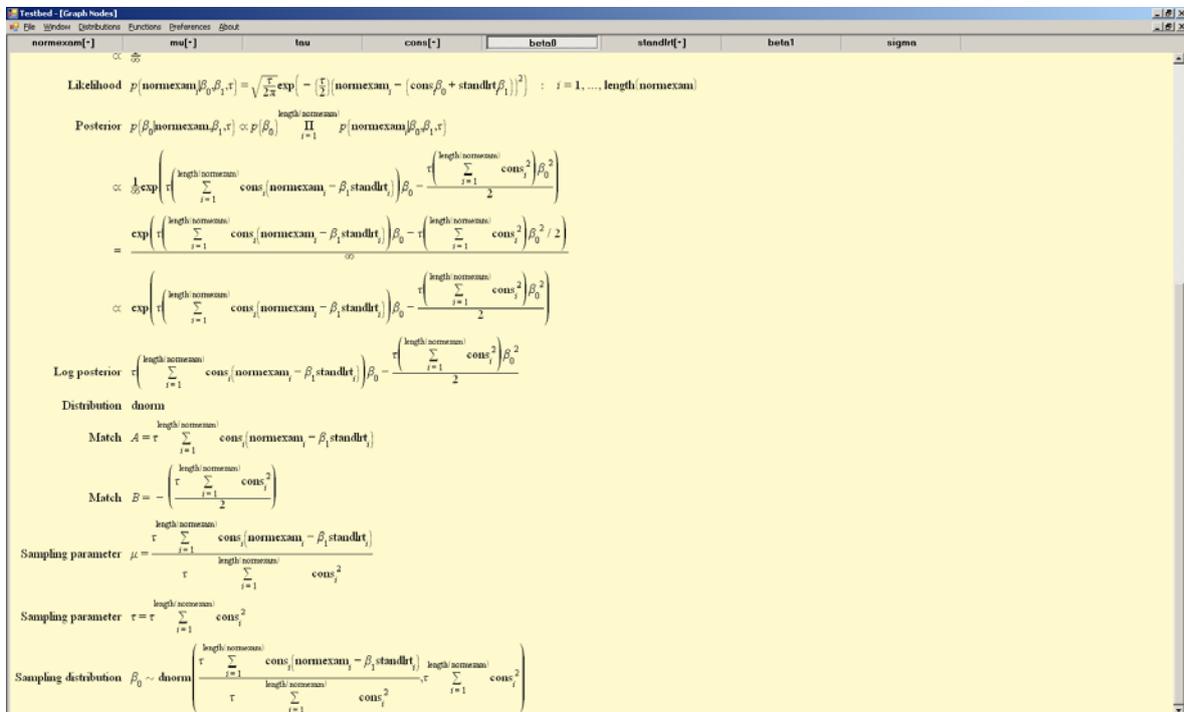


Be careful if you are copying from the web browser as it may lose the endlines and this will then not work properly but things are OK if you copy somewhere else first. You will notice when the model

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** 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  $\beta_0$  is a normal distribution. Finally it gives the conditional posterior distribution in terms of other objects in the model. We can view  $\beta_1$  and see similarly a Normal posterior:

**Likelihood**  $p(\text{normexam}_i | \beta_0, \beta_1, \tau) = \sqrt{\frac{\tau}{2\pi}} \exp\left(-\frac{\tau}{2} (\text{normexam}_i - (\text{cons}_i \beta_0 + \text{standit}_i \beta_1))^2\right) : i = 1, \dots, \text{length}(\text{normexam})$

**Posterior**  $p(\beta_1 | \text{normexam}, \beta_0, \tau) \propto p(\beta_1) \prod_{i=1}^{\text{length}(\text{normexam})} p(\text{normexam}_i | \beta_0, \beta_1, \tau)$

$$\propto \beta_1 \exp\left(\tau \left( \frac{\sum_{i=1}^{\text{length}(\text{normexam})} \text{standit}_i (\text{normexam}_i - \beta_0 \text{cons}_i)}{\sum_{i=1}^{\text{length}(\text{normexam})} \text{standit}_i^2} \right) \beta_1 - \frac{\tau \left( \sum_{i=1}^{\text{length}(\text{normexam})} \text{standit}_i^2 \right) \beta_1^2}{2}\right)$$

$$= \exp\left(\tau \left( \frac{\sum_{i=1}^{\text{length}(\text{normexam})} \text{standit}_i (\text{normexam}_i - \beta_0 \text{cons}_i)}{\sum_{i=1}^{\text{length}(\text{normexam})} \text{standit}_i^2} \right) \beta_1 - \frac{\tau \left( \sum_{i=1}^{\text{length}(\text{normexam})} \text{standit}_i^2 \right) \beta_1^2}{2}\right)$$

**Log posterior**  $\tau \left( \frac{\sum_{i=1}^{\text{length}(\text{normexam})} \text{standit}_i (\text{normexam}_i - \beta_0 \text{cons}_i)}{\sum_{i=1}^{\text{length}(\text{normexam})} \text{standit}_i^2} \right) \beta_1 - \frac{\tau \left( \sum_{i=1}^{\text{length}(\text{normexam})} \text{standit}_i^2 \right) \beta_1^2}{2}$

**Distribution** dnorm

**Match**  $A = \tau \frac{\sum_{i=1}^{\text{length}(\text{normexam})} \text{standit}_i (\text{normexam}_i - \beta_0 \text{cons}_i)}{\sum_{i=1}^{\text{length}(\text{normexam})} \text{standit}_i^2}$

**Match**  $B = -\frac{\tau \left( \sum_{i=1}^{\text{length}(\text{normexam})} \text{standit}_i^2 \right)}{2}$

**Sampling parameter**  $\mu = \frac{\tau \sum_{i=1}^{\text{length}(\text{normexam})} \text{standit}_i (\text{normexam}_i - \beta_0 \text{cons}_i)}{\sum_{i=1}^{\text{length}(\text{normexam})} \text{standit}_i^2}$

**Sampling parameter**  $\tau = \tau$

**Sampling distribution**  $\beta_1 \sim \text{dnorm}\left(\frac{\tau \sum_{i=1}^{\text{length}(\text{normexam})} \text{standit}_i (\text{normexam}_i - \beta_0 \text{cons}_i)}{\sum_{i=1}^{\text{length}(\text{normexam})} \text{standit}_i^2}, \frac{\tau \sum_{i=1}^{\text{length}(\text{normexam})} \text{standit}_i^2}{2}\right)$

Finally  $\tau$  the precision has a Gamma posterior distribution:

**Prior**  $p(\tau) = 0.001000 \dots \tau^{-0.999} \exp(-0.001000\tau) / \Gamma(0.001000)$

$$= \frac{0.993116048420934 \exp(-0.001000\tau)}{\Gamma(0.001000) \tau^{0.999}}$$

$$\propto \frac{\exp(-0.001000\tau)}{\tau^{0.999}}$$

**Likelihood**  $p(\text{normexam}_i | \beta_0, \beta_1, \tau) = \sqrt{\frac{\tau}{2\pi}} \exp\left(-\frac{\tau}{2} (\text{normexam}_i - (\text{cons}_i \beta_0 + \text{standit}_i \beta_1))^2\right) : i = 1, \dots, \text{length}(\text{normexam})$

**Posterior**  $p(\tau | \text{normexam}, \beta_0, \beta_1) \propto p(\tau) \prod_{i=1}^{\text{length}(\text{normexam})} p(\text{normexam}_i | \beta_0, \beta_1, \tau)$

$$\propto \frac{\exp(-0.001000\tau)}{\tau^{0.999}} \exp\left(-\frac{\tau}{2} \left( \sum_{i=1}^{\text{length}(\text{normexam})} (\text{normexam}_i - \beta_0 \text{cons}_i - \beta_1 \text{standit}_i)^2 \right)\right) \tau^{\text{length}(\text{normexam})}$$

$$= \frac{\exp\left(-0.001000 + \left( \sum_{i=1}^{\text{length}(\text{normexam})} (\text{normexam}_i - \beta_0 \text{cons}_i - \beta_1 \text{standit}_i)^2 / 2 \right) \tau\right)}{\tau^{0.999 - 0.5 \text{length}(\text{normexam})}}$$

$$\propto \frac{\exp\left(-\left(0.001000 + \left( \sum_{i=1}^{\text{length}(\text{normexam})} (\text{normexam}_i - \beta_0 \text{cons}_i - \beta_1 \text{standit}_i)^2 / 2 \right) \tau\right)}{\tau^{0.999 - 0.5 \text{length}(\text{normexam})}}$$

**Log posterior**  $-\left(0.001000 + \frac{\sum_{i=1}^{\text{length}(\text{normexam})} (\text{normexam}_i - \beta_0 \text{cons}_i - \beta_1 \text{standit}_i)^2}{2}\right) \tau - (0.999 - 0.5 \text{length}(\text{normexam})) \ln \tau$

**Distribution** dgamma

**Match**  $A = -0.001000 - \frac{\sum_{i=1}^{\text{length}(\text{normexam})} (\text{normexam}_i - \beta_0 \text{cons}_i - \beta_1 \text{standit}_i)^2}{2}$

**Match**  $B = -0.999 + 0.5 \text{length}(\text{normexam})$

**Sampling parameter**  $\mu = 0.001000 + \frac{\sum_{i=1}^{\text{length}(\text{normexam})} (\text{normexam}_i - \beta_0 \text{cons}_i - \beta_1 \text{standit}_i)^2}{2}$

**Sampling parameter**  $r = 0.001 + 0.5 \text{length}(\text{normexam})$

**Sampling distribution**  $\tau \sim \text{dgamma}\left(0.001 + 0.5 \text{length}(\text{normexam}), 0.001000 + \frac{\sum_{i=1}^{\text{length}(\text{normexam})} (\text{normexam}_i - \beta_0 \text{cons}_i - \beta_1 \text{standit}_i)^2}{2}\right)$

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.

### 5.2.3 Continuing the go function / applydata method

Having run the *compilemodel* function the next few lines of the go function are as follows:

```
m.callback = c
m.burnin = burnin
m.iterations = iterations
m.thinning = thinning
m.seed = seed
t.applydata(m, variables)
```

Here we are copying more information into our instance of the model before calling a method of our template *t* to link up the model with the data which have been stored in *variables*.

The generic *applydata* function can be found in *templating.py*:

```
def applydata(self, m, variables):
    for var in self.objects:
        print 'var :' + var
        if type(self.objects[var]) == DataScalar:
            print 'var name: ' + self.objects[var].name
            m.adddata(self.objects[var].name)
            m.data[self.objects[var].name] =
variables[self.objects[var].name]
        if type(self.objects[var]) == ParamScalar:
            print 'initial values: ' + str(self.objects[var]._name)
            m.addparam(var)
            if self.objects[var].answered == False:
                m.params[var].current_value = 1.0
            else:
                m.params[var].current_value =
float(self.objects[var].name)
        if type(self.objects[var]) == DataVector:
            print 'var name: ' + self.objects[var].name
            m.adddata(self.objects[var].name)
            if variables.has_key(self.objects[var].name):
                m.data[self.objects[var].name] =
variables[self.objects[var].name]
        if type(self.objects[var]) == ParamVector:
            print 'initial values: ' + str(self.objects[var]._name)

            if self.objects[var].ncols < 0:
                m.addparam(var)
                if self.objects[var].answered == False:
                    m.params[var].current_value = [0.1] * (-1 *
self.objects[var].ncols)
            else:
                m.params[var].current_value = map(float,
self.objects[var].name)
                #m.params[var].current_value = [0.1] *
len(variables[t.objects['y'].name])
            else:
                for i in range(0, self.objects[var].ncols):
                    m.addparam(var + str(i))
                    if self.objects[var].answered == False:
                        m.params[var + str(i)].current_value = 1.0
                    else:
                        m.params[var + str(i)].current_value =
map(float, self.objects[var].name)[i]
```

```

        if type(self.objects[var]) == DataMatrix:
            for i in range(0, self.objects[var].ncols()):
                print 'var name: ' + self.objects[var].name[i]
                m.adddata(self.objects[var].name[i])
                m.data[self.objects[var].name[i]] =
variables[self.objects[var].name[i]]
        if type(self.objects[var]) == ParamMatrix:
            pass # Still need to add this

```

This method passes through each of the objects that are in the object list of the template and performs different tasks depending on their type. Basically if the object is one of the various data types then we call the *model* object's method to attach the actual data to the object (*m.adddata*) with the code looping over the columns if the object is a matrix. For objects that are parameters the code both adds the parameter and also sets an initial value for it.

Adding a parameter does actually more than simply adding the parameter to the model; it also adds the algorithm used for that parameter as an attribute of the parameter. The method of adding a parameter to a model initialises the parameter and then loads up the algorithm:

```

def addparam(self, name, mode=0, algorithm=''):
    self.params[name] = Parameter(self, name, mode, algorithm)
    self.params[name].load(self.directory)

```

Here we need to look at the class definition for a parameter to see what exactly is going on and this appears in *parameter.py*.

#### 5.2.4 Parameter.py

The file *parameter.py* contains the definition of the *parameter* class and the *RunningStat* class (which is one of the attributes of a parameter). The *parameter* class has an initialising method but here we are more interested in the *load* method:

```

def load(self, directory):
    fname = directory + '/' + self.name + '.xml'
    if os.path.isfile(fname):
        f = open(fname, 'r')
        self.xml = f.read()
        f.close()
        self.tree = XMLtoTree(self.xml).tree
        # Temporarily detect and set algorithm
        # NOTE: default algorithm should be based on existence of
sampling distribution, as ideally logpost should always be present
        if type(self.tree) == dict:
            self.algorithm = "gibbs"
        else:
            self.algorithm = "mh"
        print 'Algorithm for ' + self.name + ': ' + self.algorithm
    else:
        print "Missing formula for " + self.name
        self.tree = ''

```

The parameter has three interesting attributes that are set here. Firstly the particular XML file for this parameter output by the algebra system is read and stored in the attribute *xml*. This is converted into a *tree* attribute which is an object of type *XMLtoTree* (the code for this is in *XMLtoTree.py* in the *src/lib/Estat* directory) but we will not go into details here. The tree will have an attribute type that allows the software to know whether it will use Gibbs sampling or MH sampling here. Note that the posterior distribution is stored as a tree at this stage as the model object is used both when fitting models using the engine and also when converting to C++ or Java code.

So we should now have data and parameters linked into our model object. Returning to the *go* function this continues as follows:

```

allmissing = True
    for p in m.params.values():
        if p.tree != '':
            allmissing = False
    if allmissing == True:
        print "Error in algebra system, nothing generated"
        return None

if hasattr(t, 'customsteps'):
    for p in t.customsteps:
        print 'Custom step for parameter: ' + p
        m.params[p].tree = ''

    for p in mon():
        m.params[p].monitor = True
    return m

```

Basically we check that something is generated by the algebra system else display an error. The last few lines allow custom estimation steps (steps written by the user that replace those output by the algebra system) by removing the tree code for particular parameters (such custom steps will be included in the *precode/postcode* functions as explained later). Finally the list of parameters that we wish to store MCMC chains for is set via the *monitor* attributes for each parameter and the *run* method for the *model* object is called.

### 5.2.5 Running a model – the run method and XMLtoC

The *run* method for the Model class contains a lot of Python code and does many things. We now need to piece together the parameter trees to form a set of algorithmic steps to fit the model to the data. The first chunk of code is as follows:

```

def run(self):
    run_time = time.clock()
    total = self.burnin + self.iterations

    vars = {}
    localvars = {}
    self.scode = stdtmpl.get_template('statlib.cpp').render(seed =
self.seed)
    self.code = ''
    self.code += self.precode
    for k in self.params.keys():
        vars[k] = self.params[k].current_value
        if self.params[k].monitor == True:

```

```

        if type(vars[k]) == list:
            self.params[k].chain =
numpy.array([[0.0]*(self.iterations/self.thinning)]*len(vars[k]]))
        else:
            self.params[k].chain =
numpy.array([0.0]*(self.iterations/self.thinning))
        else:
            if type(vars[k]) == list:
                self.params[k].rs = []
                for ind in range(0, len(vars[k])):
                    self.params[k].rs.append(RunningStat())
            else:
                self.params[k].rs = RunningStat()
                self.code += '\t// Update ' + k + '\n'
                self.code += str(XMLtoC(self.params[k].tree,
self.params[k].mode, self.params[k].algorithm).toC())
                self.code += '\n'
            self.code += self.postccode

```

Here we begin by setting a timer and calculating in total how many iterations we need to run for. We then set up attributes that contain the code that will be run to fit the model. The attribute *scode* is a string containing C++ code that contains all the random number generators and matrix algebra functions (written in C++) that are required. The file *statlib.cpp* contains these generators and the only thing that is variable here is the *seed* used.

Next we define the *code* attribute which will be the body of the code, that is run for each iteration of an MCMC algorithm. Here you will see that *code* is constructed by concatenating *preccode*, a chunk of code for each parameter (each preceded with a comment to identify what is updated) and *postccode*. We also set up the memory required for storing the chains of parameters, which are for now initialised with zeros.

The chunk of code for each parameter is created using the *XMLtoC* call and here we need to look at the *XMLtoC* class which is found in *XMLtoC.py* (in the *src/lib/Estat* directory). You will see that this function requires the tree that contains the parameter posterior, the algorithm for the parameter posterior and the mode (which is not important here). Briefly, this function converts the tree object into a string of C++ code that will update the parameter by randomly drawing either from its conditional posterior (when algorithm is Gibbs) or indirectly via a proposal distribution and comparison of the posterior probability of proposed and current values (when algorithm is MH).

At this stage in the *run* function the *code* attribute contains a fully formed piece of C++ code for performing one iteration of an MCMC algorithm for the current model. The next chunk of the *run* method of the *model* class is as follows:

```

for k in self.params.keys():
    if type(self.params[k].current_value) == list:
        vars[k + '_accept'] = map(lambda x: 0,
self.params[k].current_value)
        localvars[k + '_acceptinrow'] = map(lambda x: 0,
self.params[k].current_value)

```

```

        localvars[k + '_acceptdone'] = map(lambda x: 0,
self.params[k].current_value)
        vars[k + '_sd'] = map(lambda x: 0.1,
self.params[k].current_value)
    else:
        vars[k + '_accept'] = 0
        localvars[k + '_acceptinrow'] = 0
        localvars[k + '_acceptdone'] = 0
        vars[k + '_sd'] = 0.1

    adapt = True
    adapt_count = 0
    it = 0
    desacc = 44.1 # Desire 44.1% acceptance
    tol = 10 # with tolerance of 10%

    print 'Finished initialising:' + str(time.clock() - run_time)

```

Here we are simply setting starting values for all parameters and MH settings for each parameter. The next chunk of code:

```

algorithm = """
if ${accept} > (desacc - tol) and ${accept} < (desacc + tol):
    ${acceptinrow} += 1
    if ${acceptinrow} == 3:
        ${acceptdone} = 1
else:
    ${acceptinrow} = 0
if ${accept} > desacc:
    ${sd} *= (2 - ((100 - ${accept}) / (100 - desacc)))
else:
    ${sd} /= (2 - (${accept} / desacc))
${accept} = 0"""

```

gives the basic form for the adapting steps that are part of the MH algorithm. Then we have

```

while adapt:
    scipy.weave.inline(self.code,
                        vars.keys() + self.data.keys(),
                        local_dict=vars,
                        global_dict=self.data,
                        support_code=self.scode,
                        compiler='gcc',
                        type_converters=converters.blitz,
                        headers=['<vector>'],
                        extra_compile_args=['-O3'])

    it = it + 1

```

This basically compiles and runs one iteration of the code and adds 1 to an iteration counter. There is then a long section of code (not shown) that forms the steps to be used to adapt the proposal distributions for MH sampling. After a message to show adapting has finished the above code for compiling and running one iteration of the algorithm is repeated, this time within a loop for the desired number of iterations.

The *code* is completed by a chunk of code that stores the values of parameters for iterations after the burnin (if we are storing chains) and updates summary statistics (if we are not storing the chains):

```

        if iteration + 1 > self.burnin and iteration % self.thinning ==
0:
            for k in self.params.keys():
                if self.params[k].monitor == True:
                    if type(vars[k]) == list:
                        for p in range(len(vars[k])):
                            self.params[k].chain[p][(iteration-
self.burnin)/self.thinning] = vars[k][p]
                    else:
                        self.params[k].chain[(iteration-
self.burnin)/self.thinning] = vars[k]
                else:
                    if type(vars[k]) == list:
                        for p in range(len(vars[k])):
                            self.params[k].rs[p].Push(vars[k][p])
                    else:
                        self.params[k].rs.Push(vars[k])
            print 'Finished iterating:' + str(time.clock() - run_time)
            for k in self.params.keys():
                self.params[k].current_value = vars[k]

```

This *run* function therefore runs the whole model. You will note that this is done in Python but with calls to C++ code to update the parameters for each iteration. This can be contrasted from the code generation which we will look at in section 5.3 where the code is all written in C++.

### 5.2.6 Summarising the results

Having run the model the last call before returning the model is to the model *summarise* method. This method is used to calculate summary statistics for all parameters as shown in the following code (*run.html* then loops through the dictionary returned by the method and prints out the value stored for each key *parameter*):

```

def Summarise(self):
    prec = 4
    results={}
    for p in self.params.keys():
        results[p]={}
        if self.params[p].monitor == True:
            print '\n' + p + ': '
            if len(self.params[p].chain.shape) > 1:
                out = ''
                for i in range(0, len(self.params[p].chain)):
                    out += '<br/>%.*g (%.*g) ESS: %d' % (prec,
self.params[p].chain[i].mean(), prec, self.params[p].chain[i].std(),
ESS(self.params[p].chain[i]))
            else:
                results[p]["chain"] = '%.*g (%.*g) ESS: %d' % (prec,
self.params[p].chain.mean(), prec, self.params[p].chain.std(),
ESS(self.params[p].chain))
            print results[p]["chain"]
            print 'Z-Score: ' + str(self.params[p].Zscore())

```

```

else:
    print '\n' + p + ': '
    if isinstance(self.params[p].rs, list):
        out = ''
        for i in range(0, len(self.params[p].rs)):
            out += '%.*g (%.*g) ' % (prec,
self.params[p].rs[i].Mean(), prec,
self.params[p].rs[i].StandardDeviation())
        results[p]["summary"]=out
        print results[p]["summary"]
    else:
        results[p]["summary"] = '%.*g (%.*g) ' % (prec,
self.params[p].rs.Mean(), prec, self.params[p].rs.StandardDeviation())
        results[p]["summary"]

```

We have now completed the *go* function and control returns to the *webtest* code. The *webtest* code continues as follows:

```

for p in m.params.keys():
    if m.params[p].monitor == True:
        t.objects[p] = sixway(p, m.params[p].chain)

if 'param_input' in i:
    t.objects['current_plot'] = i['param_input']
context['model']=m
datasets.datasets[t.EstObjects['outdata'].name] = t.resultdata(m)

if hasattr(t, 'outbug'):
    statstemplate=t.output()
if hasattr(t, 'outlatex'):
    latextemplate=t.outputlatex()

```

Firstly the *sixway* function that creates the graphical summaries for each parameter in the model that is monitored is called and these graphs are stored within the template instance. Next the current plot is assigned so that we know which parameter to display. The results from fitting the model are stored as an output dataset so that they can be used by other templates and finally we recreate the model specification and LaTeX outputs so that they appear on screen. Then we are done and the browser will show all the output. We next look at alternatives that rather than running the model directly, instead simply generate code to fit the model.

### 5.3 C code generation (XMLtoPureC)

In the last section we looked at how Stat-JR uses the built-in MCMC estimation engine to fit the model specified by a template to a particular dataset. Stat-JR can also output free-standing C++ code that can be copied and compiled in isolation. This code performs essentially the same estimation algorithm for the model/dataset combination. The advantage of this option is that the code can be scrutinised and modified (and we hope that possibly this will be a route to allow users in the future to increase the range of algorithms on offer).

When we were describing the *webtest* Python file we noted that under the *POST* method we were able to find the lines of code that are executed when the **Run** button is pressed. Similarly the code that is executed when the **Code** button is pressed can be found and is as follows:

```

if 'code' in i:
    web.header('Content-Type', 'text/plain')
    m = Model()
    m.modelspec = t.output()
    m.preccode = t.preccode()
    m.postccode = t.postccode()
    m.compilemodel()
    m.burnin = t.EstObjects['burnin']
    m.iterations = t.EstObjects['iterations']
    if 'seed' in t.EstObjects:
        m.seed = t.EstObjects['seed']
    t.applydata(m, context['variables'])
    if hasattr(t, 'customsteps'):
        for p in t.customsteps:
            print 'Custom step for parameter: ' + p
            m.params[p].tree = ''
    for p in t.monitor_list():
        m.params[p].monitor = True
    return m.genCPP()

```

You will notice here similarities to the code in the *go* function that we used when running a model. Basically we again form a model and set some of the attributes and call the *compilemodel* and *applydata* methods as described in sections 5.2.1-5.2.4. The difference here is we replace the calls to *run* and *summarise* with a returning of the *genCPP* method.

### 5.3.1 Gencpp

The gencpp method code is quite short:

```

def genCPP(self):
    self.code = stdtmpl.get_template('statlib.cpp').render(seed =
self.seed)
    self.code += "void iterate() {\n"
    self.code += self.preccode

    for k in self.params.keys():
        self.code += '// Update ' + k + '\n'
        self.code += XMLtoPureC(self.params[k].tree,
self.params[k].mode, self.params[k].algorithm).toC()
        #self.code += self.params[k].purec
        self.code += '\n'
    self.code += self.postccode
    self.code += "}\n"

    varnames = {}
    varnames["itcode"] = self.code
    varnames["params"] = self.params
    varnames["data"] = self.data
    varnames["burnin"] = self.burnin
    varnames["iterations"] = self.iterations
    varnames["seed"] = self.seed

    code = stdtmpl.get_template('standalone.cpp').render(**varnames)
    return code

```

Basically the method constructs the standalone C++ code for the model and here we see the call to *XMLtoPureC* that creates the C++ code to update a particular parameter from its tree. Note that all of this stuff is passed into the *standalone.cpp* file which contains code substitutions, the most important being **\${itcode}** which is all the updating steps for all parameters in the model as we will see later.

### 5.3.2 XMLtoPureC

This file is used to construct pure (standalone) C++ code and can be found in the *src/lib/EStat* subdirectory. The code in *XMLtoPureC* is basically very similar to the code in *XMLtoC* except it doesn't assume the parameters are objects in Python and so the code it produces is much easier to read and looks much more like the sort of C++ code that would be familiar to C++ programmers. We will now consider the *Regression1* example and look at the code produced.

### 5.3.3 Example code for our template

When the **Code** button is pressed, C++ code is generated and then displayed in the browser so that it can be saved and used elsewhere. Lots of the code at the start of the program is generic for all models. There are specific inputs of starting values and data in places but perhaps the most interesting chunk of code is the iteration loop produced by the **\${itcode}** substitution. To find this code you could search for *iterate* in the browser window. The code contains steps to update the four parameters. Of these parameters, tau, beta1 and beta0 have stochastic updates, whilst sigma is a derived quantity and therefore has a deterministic update. In this simple example it is instructive to look at the code below and compare with the mathematics displayed in the *Graph Node* window of the algebra system. The two sets of expressions should be identical to one written in mathematical form and the other in C++ code. The C++ code is as follows:

```
void iterate() {
// Update tau
{
    double sum0=0;
    for(int i=0; i<length(normexam); i++) {
        sum0+=pow(((normexam[int(i)]-(beta0*cons[int(i)]))-
(beta1*standlrt[int(i)])),2);
    }

    std::tr1::gamma_distribution<double>
gamma((0.001+(0.5*length(normexam))));
    tau = (1.0 / (0.001000+(sum0/2))) * gamma(eng);

}
// Update sigma
{
    sigma = (1/sqrt(tau));
}
// Update beta1
{
    double sum0=0;
    for(int i=0; i<length(normexam); i++) {
        sum0+=(standlrt[int(i)]*(normexam[int(i)]-
(beta0*cons[int(i)])));
    }
    double sum1=0;
    for(int i=0; i<length(normexam); i++) {
        sum1+=pow(standlrt[int(i)],2);
    }
}
```

```

    }

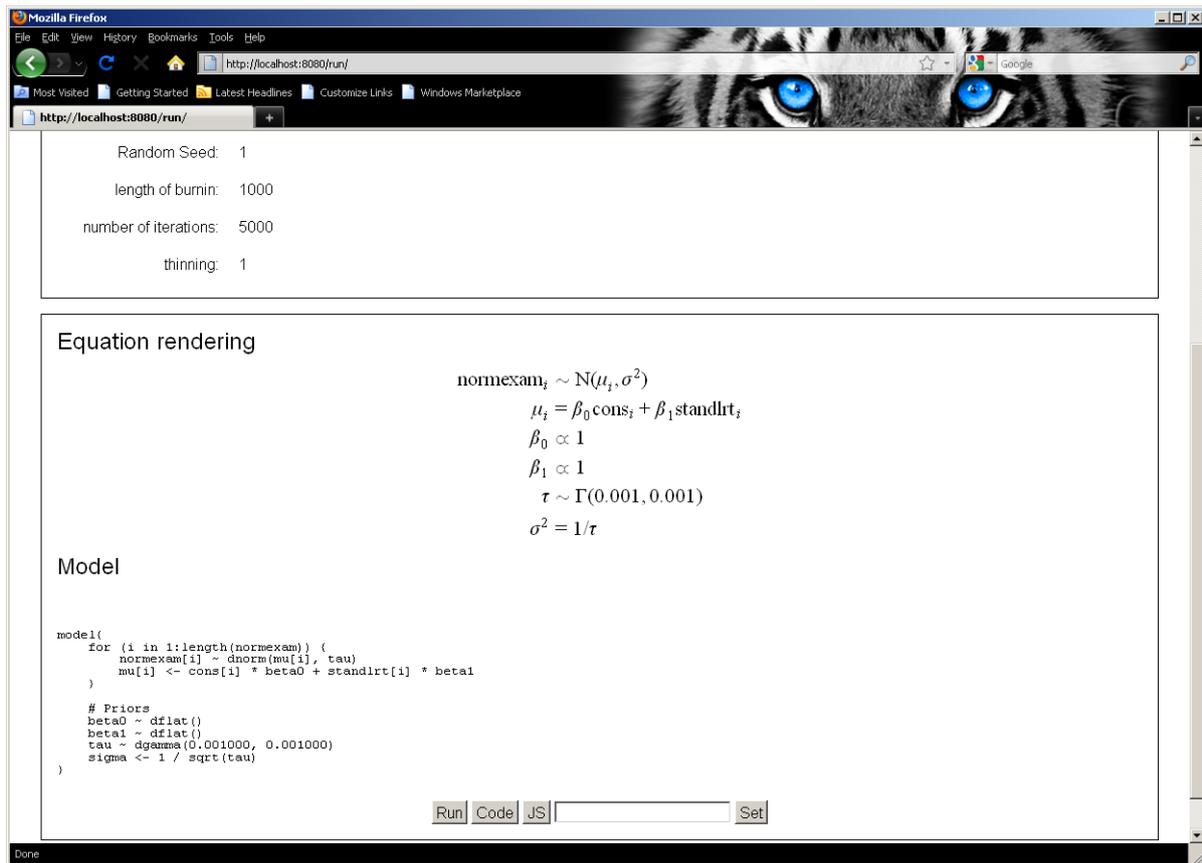
    std::tr1::normal_distribution<double>
normal(((tau*sum0)/(tau*sum1)), 1/sqrt((tau*sum1)));
    beta1 = normal(eng);
}
// Update beta0
{
    double sum0=0;
    for(int i=0; i<length(normexam); i++) {
        sum0+=(cons[int(i)]*(normexam[int(i)]-
(beta1*standlrt[int(i)])));
    }
    double sum1=0;
    for(int i=0; i<length(normexam); i++) {
        sum1+=pow(cons[int(i)],2);
    }
    std::tr1::normal_distribution<double>
normal(((tau*sum0)/(tau*sum1)), 1/sqrt((tau*sum1)));
    beta0 = normal(eng);
}
}

```

## 5.4 JavaScript generation (XMLtoJS)

In the last section we looked at how to output C++ code that could be taken away and reused. The engine in Stat-JR uses C++ for its number crunching as we have already noted. This will allow users to install Stat-JR on their own machine and host the software locally. If instead we wished to place Stat-JR on a server which users have remote access to, then all their computations would be performed on the server which is perhaps not very attractive. An alternative would be for the Stat-JR system to export code on to the local machine so that computations could be performed locally. This is the motivation behind the JavaScript generation facilities.

We will return to the *Regression1* example and after setting up the simple regression model the screen should look as follows with the **JS** button available:



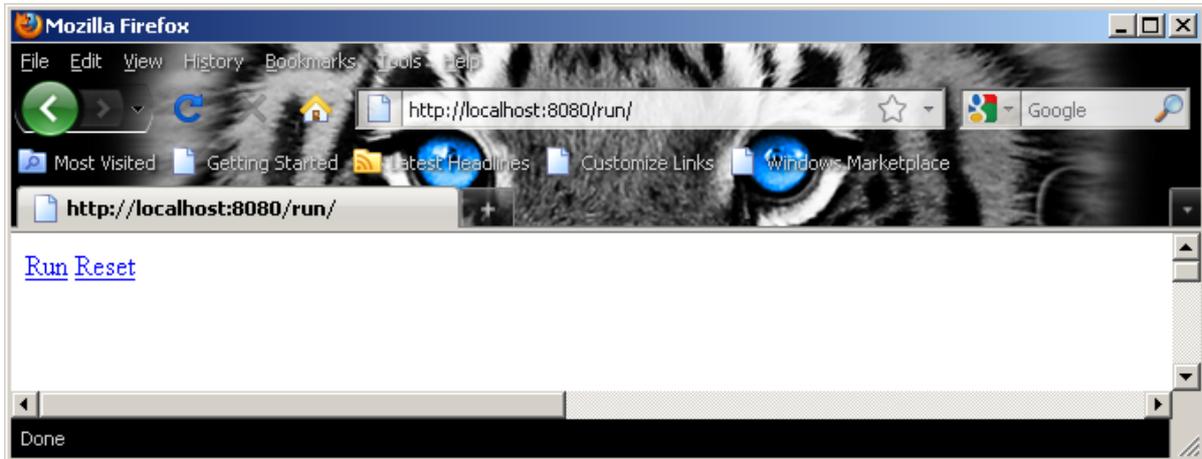
When we were describing the *webtest* Python file we noted that under the *POST* method we were able to find the lines of code that are executed when various buttons are pressed. Similarly the code that is executed when the *JS* button is pressed can be found within the *POST* method of *webtest* and is as follows:

```
if 'js' in i:
    m = Model()
    m.modelspec = t.output()
    m.prejcode = t.prejcode()
    m.postjcode = t.postjcode()
    m.compilemodel()
    m.burnin = t.EstObjects['burnin']
    m.iterations = t.EstObjects['iterations']
    t.applydata(m, context['variables'])
    if hasattr(t, 'customsteps'):
        for p in t.customsteps:
            print 'Custom step for parameter: ' + p
            m.params[p].tree = ''
    for p in t.monitor_list():
        m.params[p].monitor = True
    return render.jsrun(model = m.genJS())
```

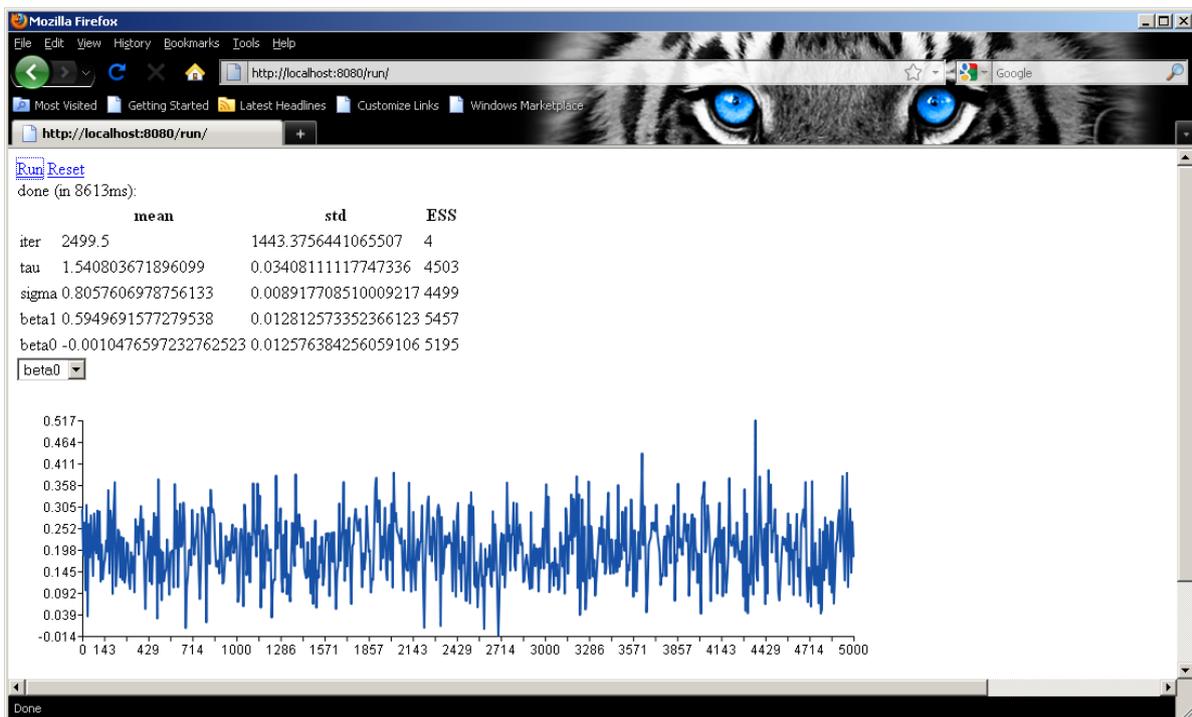
Again this code snippet is very similar to that used for running the model or for generating C code. There are references to *prejcode* and *postjcode* which are used for users to incorporate additional chunks of Java code (as is done for C++ in *preccode* and *postcode*). The main difference is that now we call the method *genJS* for the model and switch to the html template file *jsrun.html* (which is in the *html\_templates* subdirectory of the *webtest* directory).

### 5.4.1 Running the Regression1 example

The method *genJS* works in a similar way to method *genCPP* but instead of C++ code generates Javascript using the class *XMLtoJS* (code in *src/lib/EStat*). We will omit details here but show the output. The JavaScript screen looks as follows:



To run the script we simply press the **Run** button and we will get the following output (after potentially waiting for some time):



Note that it is best here to use a Firefox browser as the Javascript engine that comes with Internet Explorer can be slow and give lots of warning messages!

The output we see is a summary of each parameter and the chain for the first parameter with a pull-down list for other parameters. We have not fully integrated the JavaScript options in Stat-JR into

the rest of the system for this alpha release and so the output does not get sent to an output file and we do not get more extensive diagnostic plots etc.

### 5.4.2 JavaScript source code

We can also view the source code used in the JavaScript option by clicking on View/Page Source in the browser. This will give the JavaScript code in a separate window and once again we can look at the chunk of code for performing one iteration:

```
function iterate() {
// Update tau
{
    var sum0=0;
    for(var i=0; i<length(normexam); i++) {
        sum0+=Math.pow(((normexam[i]-(beta0*cons[i]))-
(beta1*standlrt[i])),2);
    }
    tau =
rgamma((0.001+(0.5*length(normexam))), (0.001000+(sum0/2)));
}
// Update sigma
{
    sigma = (1/Math.sqrt(tau));
}
// Update beta1
{
    var sum0=0;
    for(var i=0; i<length(normexam); i++) {
        sum0+=(standlrt[i]*(normexam[i]-(beta0*cons[i])));
    }
    var sum1=0;
    for(var i=0; i<length(normexam); i++) {
        sum1+=Math.pow(standlrt[i],2);
    }
    beta1 =
rnormal(((tau*sum0)/(tau*sum1)),1/Math.sqrt((tau*sum1)));
}
// Update beta0
{
    var sum0=0;
    for(var i=0; i<length(normexam); i++) {
        sum0+=(cons[i]*(normexam[i]-(beta1*standlrt[i])));
    }
    var sum1=0;
    for(var i=0; i<length(normexam); i++) {
        sum1+=Math.pow(cons[i],2);
    }
    beta0 =
rnormal(((tau*sum0)/(tau*sum1)),1/Math.sqrt((tau*sum1)));
}
}
```

Once again, although in a different computer language, it is easy to see that this code is doing the same algorithm as the C++ code and is using the distributions generated from the algebra system.

## 6. Including Interoperability

The Stat-JR package has its own new algebra system and estimation engine as illustrated in the last chapter. 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. In this chapter we 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 the four software packages that have thus far been considered for interoperability, namely WinBUGS, MLwiN, R and Stata.

### 6.1 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 by looking at the *MethodInput* attribute that we described briefly in chapter 4:

```
def MethodInput(self):
    self.EstObjects['EstM']=Boolean('Is estimation method by MCMC: ')
    if self.EstObjects['EstM']:
        self.EstObjects['Engine']=Text('Choose estimation engine -
eSTAT, WinBUGS, MLwiN, R: ', ['eSTAT', 'WinBUGS', 'MLwiN', 'R'])

        if self.EstObjects['Engine'] == 'MLwiN':
            self.EstObjects['FixedAlg']=Text('Select MCMC sampling
method for fixed effect- Gibbs, MH: ', ['Gibbs', 'MH'])
            self.EstObjects['VarAlg']=Text('Select MCMC sampling method
for residuals- Gibbs, MH: ', ['Gibbs', 'MH'])

            if self.EstObjects['Engine']=='WinBUGS':
                self.EstObjects['nchains']=Integer('number of chains: ')
            else :
                self.EstObjects['nchains']=1

            self.EstObjects['seed']=Text('Random Seed: ')
            self.EstObjects['burnin']=Integer('length of burnin: ')
            self.EstObjects['iterations']=Integer('number of iterations: ')
            self.EstObjects['thinning']=Integer('thinning: ')

        else :
            self.EstObjects['Engine']=Text('Choose estimation engine -
MLwiN, R, STATA: ', ['MLwiN', 'R', 'STATA'])
```

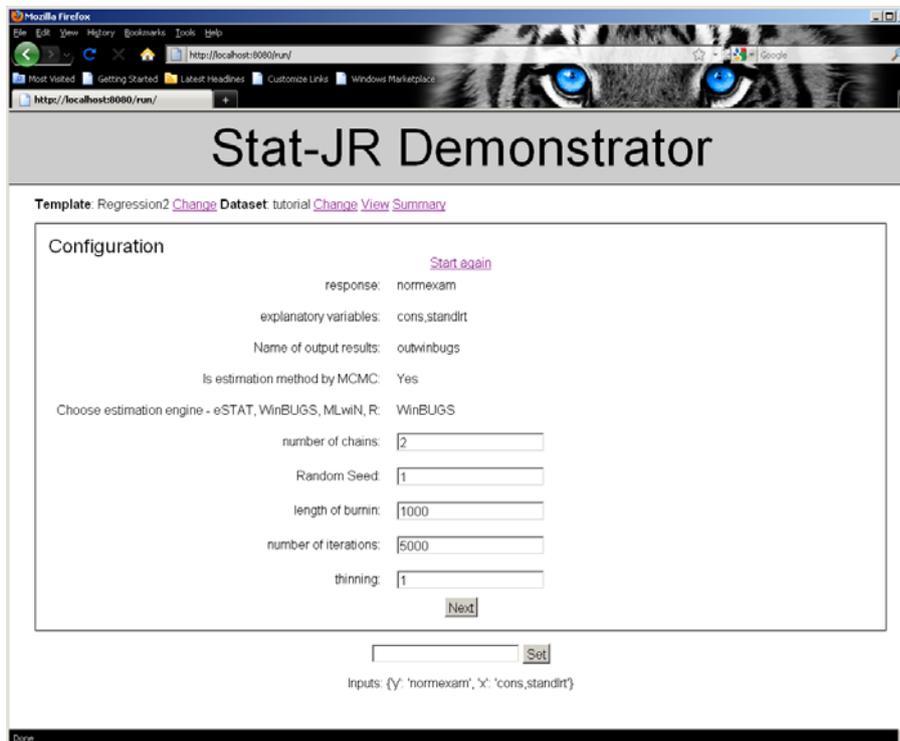
As we saw in section 4.1 this *MethodInput* function is called after the model inputs and you will see some additional inputs that were set to defaults in the *MethodInput* function in *templating.py*. Here we allow the user to input whether their method (*EstM*) is MCMC or not and then, conditional on the method, we offer a choice of packages that offer that method (*Engine*). There are some additional package specific inputs, for example MLwiN allows the user to select which method to be used for updating fixed effects and residuals whilst WinBUGS allows multiple chains from different starting points. We will look at these in more details in the sections for the specific packages.

## 6.2 WinBUGS and Winbugscript.py

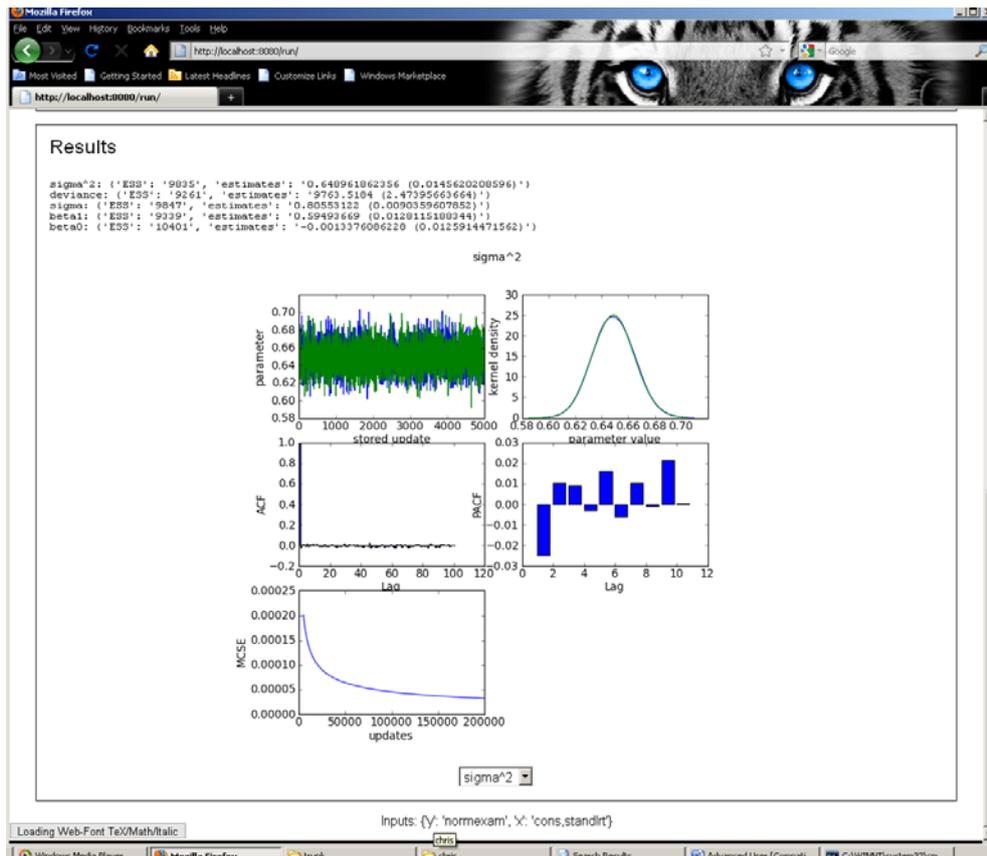
We will begin by looking at the WinBUGS package as the model code we have been creating for the Stat-JR engine has many similarities with WinBUGS code. We will begin by running the template and viewing the output. It should be noted that in order to run the WinBUGS engine Stat-JR needs to be able to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

```
[WinBUGS]
executable=C:\Documents and Settings\frwjb\WinBUGS14\WinBUGS14.exe
working_directory=C:\Documents and Settings\frwjb\WinBUGS14
```

If you wish to use this option you need to change these paths to point to WinBUGS on your machine and you will need write access to the WinBUGS directory (Note that you must set the working\_directory to be the WinBUGS directory). When you have done this restart the *webtest* program so that it uses the new settings. Select *Regression2* from the *template* choices and *tutorial* for the *dataset*. Next select the following inputs:



Clicking on **Next** will simply show the Model code and mathematical representation as we saw for *Regression1* with the in-built engine (Note this can be chosen by selecting eSTAT for the estimation engine). 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:



Here you will notice that we get results for each parameter (including the addition of the deviance and some reordering of the output). As we chose 2 chains you will also observe a green and blue output for both the chains and kernel density plots.

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. We will therefore need to investigate how the *webtest.py* code uses the changes to Estimation method input.

Within the *run* class in *webtest.py* (search for “class run” to find this) we will now no longer run the code following the statement

```
if t.EstObjects['Engine'].name=='eSTAT':
```

and instead we have the following code (note elif is Python for ‘else if’):

```

elif t.EstObjects['Engine'].name=='WinBUGS':
    m=goWinBUGS(t)
    if t.EstObjects['nchains']==1:
        for p in m.displayres.keys():
            t.objects[p]=sixway(p,m.displayres[p])
    else :
        for p in m.displayres.keys():
            t.objects[p]=sixwaymulti(p,m.displayres[p],t)
    datasets.datasets[t.EstObjects['outdata'].name]=m.combresults

```

```

if hasattr(t, 'WinBUGSMod'):
    statstemplate=t.WinBUGSMod()
elif hasattr(t,'outbug'):
    statstemplate=t.output()
if hasattr(t, 'outlatex'):
    latextemplate=t.outputlatex()

```

Here we see calls to a new function – *goWinBUGS* which we will look at next. You will also see that for multiple chains a different graphics function (*sixwaymulti*) is used. We will not go into details of how this works as it is generic to all templates. Finally you will see that there is an if statement:

```

if hasattr(t, 'WinBUGSMod'):
    statstemplate=t.WinBUGSMod()

```

which allows us to write our own function for interpreting the user inputs in terms of WinBUGS code. This function isn't present for this template as the code generated is close enough to standard WinBUGS code (aside from some standard substitutions that we will see later). A use of this function can be seen for example in the template *MVNormal1Level* which we will look at (using the built-in estimation engine) in section 12.3.

The *goWinBUGS* function is in *webtest.py* (higher up the code):

```

def goWinBUGS(t):
    m=WinBUGSScript()
    m.PrepareWBugsInputs(t)
    m.WriteScript(t)
    m.ExecuteScript()
    m.ImportWBUGSOutput(t)
    m.Summarise()
    return m

```

Here you will notice an initialisation of *m* to a class *WinBUGSScript* which can be found in `src\lib\EStat\external\WinBUGS` subdirectory . The following lines then call various methods for the class to perform all the operations required to fit the model using WinBUGS and convert the results into the format needed for Stat-JR. We will not describe each line of code in detail, but just give brief descriptions of the role of each function.

The first initialisation line will run the `__init__` method and thus construct a blank *WinBUGSScript* object. The next line calls the *PrepareWBugsInputs* method. This method is long and contains the bulk of the work in translating from the inputs from the template to files required for WinBUGS. For fitting a model in WinBUGS three files are required: a model file, a data file and an initial values file. An additional script file is required to actually get WinBUGS to use these three files to fit the model.

The method starts by setting up filenames for the various files to be created. The next section of code then interrogates the template to construct a list of data objects and a separate list of parameters. There is then a long section that constructs the data file. Some templates will have their own function `WinBUGSData` for this purpose but by default there is code here for generic templates. It should be noted that this code at present relies on specific naming of objects in the template e.g. *NumLevs* and *L2ID* which we will try to change later.

The data file is stored as a text string called *infile* which is extended as data items are added. At the top of the file will be scalar quantities like *N*, the length of the dataset. Then the data itself is added. The columns have to be added as comma separated lists housed within round brackets. You will notice special code for classification variables:

```

if p in self.clas:
    for q in self.data[p]:
        infile +=str("%E" %(q+1))
        k=k+1
        if (k<N):
            infile+=','
        else :
            infile+=')'

```

As WinBUGS requires classification units to begin with the number 1 (rather than 0 which is how STAT-JR stores them) all classification data items have 1 added to them.

Having written the text string the following three lines write the data out to file:

```

file=open(self.myInfile, 'w')
file.write(str(infile))
file.close()

```

The code then proceeds to write the model file. Here the file is stored as a text string labelled 'modspec' and the main purpose of the code is to do a few text substitutions for the 'for loop' lines as the length function is not standard WinBUGS code.

The initial values code follows next which is looped over for the number of chains to be used. The code is stored in a text string labelled 'inits' and there is some matching of variable names so that appropriate random starting values can be generated.

After creating the three input files the next method call in the *goWinBUGS* function is to write the script file and this is done via the *WriteScript* method. This method is somewhat shorter as it only has to write one file that runs within WinBUGS. Basically the function creates a script file that reads in the three input files, performs the burnin and then stores chains for all parameters for the main run. These chains are then output via the WinBUGS *CODA* function.

Having written the script the next line in *goWinBUGS* calls *ExecuteScript* which is very short:

```

def ExecuteScript(self):
    WinBugsApp=EStat.configuration.get('WinBUGS', 'executable')
    subprocess.call([WinBugsApp, "/par", "script.txt"])# with
script.txt being in the same directory as WinBug14.exe

```

This function runs WinBUGS in the background. Having run WinBUGS we need to import back in the chains from WinBUGS into Stat-JR and this is performed by the function *ImportWBUGSOutput* which takes the files output by WinBUGS and constructs chains in an appropriate format. Here the code needs to deal with thinning and multiple chains and to deal with precision parameters in a slightly different way. The final function that is called is the *Summarise* function which then constructs the text that will appear in the browser under Results.

We will limit our descriptions here but will endeavour to write further documentation on interoperability later. We next look at MLwiN.

### 6.3 MLwiN

MLwiN is another package with MCMC functionality but which can also fit multilevel models using classical statistical methods. In Stat-JR we offer the option of fitting models in MLwiN using either approach where available.

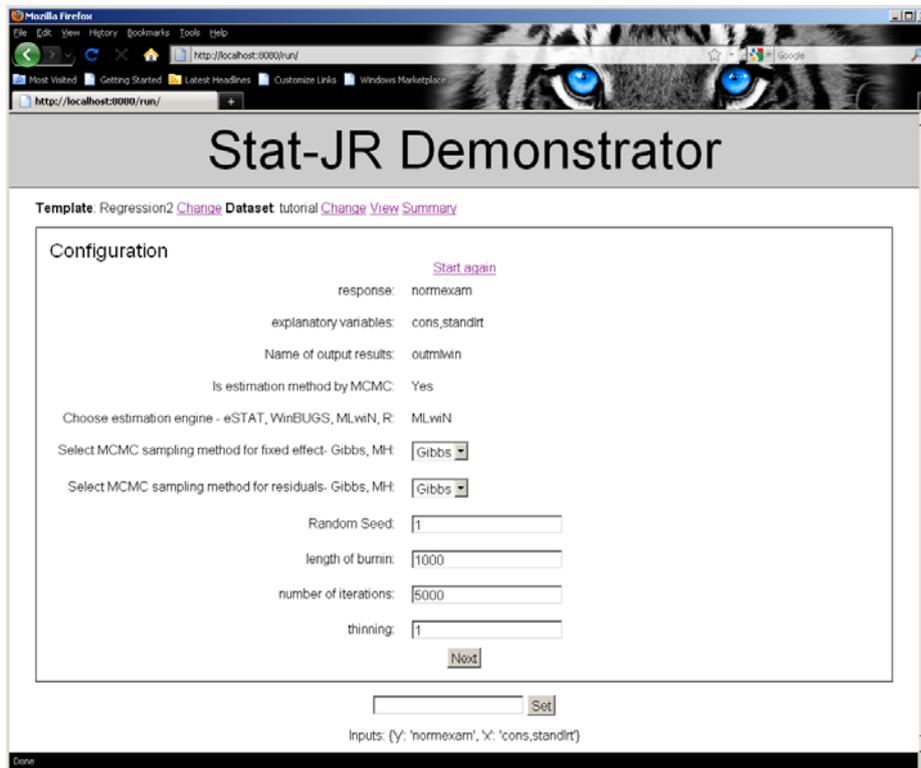
Having seen how WinBUGS links into Stat-JR we will now show the similarities and differences in how MLwiN links in. The first observation is that MLwiN doesn't use a model description language like Stat-JR or WinBUGS. It is also more restrictive in terms of which models it can fit which means that it will not be available for all templates (although MLwiN can be used in many of the templates we have written thus far). Although MLwiN has a GUI user interface it also has a macro language and it is this language that we 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:

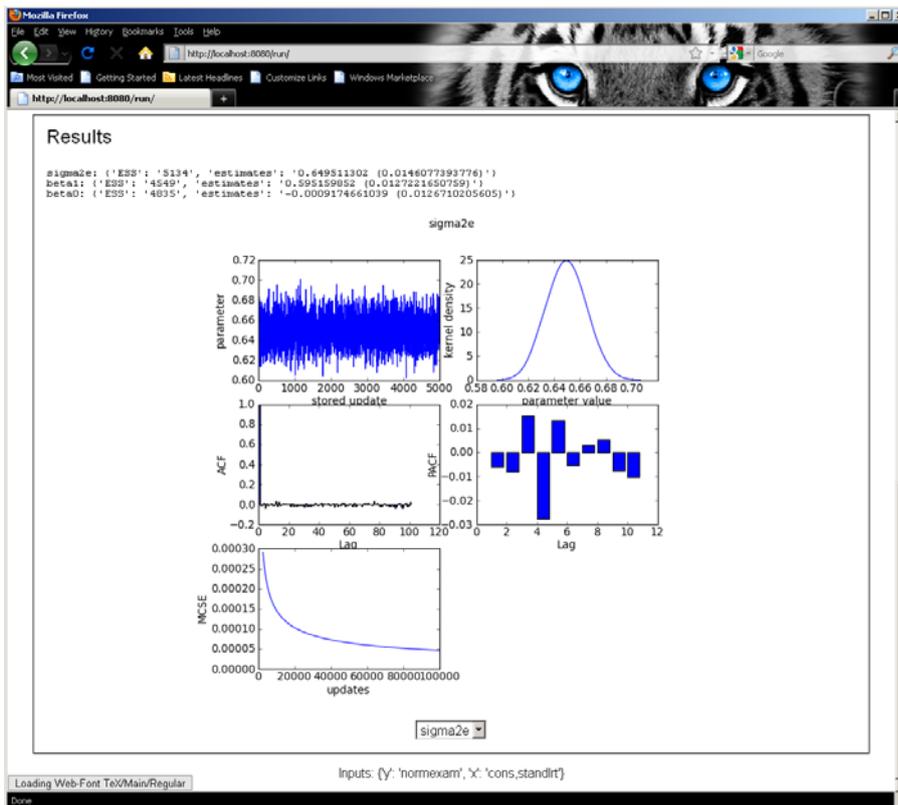
```
[MLwiN]
executable=C:\Program Files\MLwiN v2.22\mlwin.exe
```

Note, if you change the directory then you will need to restart Stat-JR.

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:



Click on the **Next** button and then the **Run** button to call MLwiN (in fact some MLwiN windows should appear and then disappear). The following output is shown:



Apart from the speed of estimation (which is much quicker than Stat-JR and WinBUGS) the results are very similar. We could as an alternative run the model not using MCMC by choosing the following inputs:

# Stat-JR Demonstrator

Template: Regression2 [Change Dataset](#) tutorial [Change View Summary](#)

Configuration

[Start again](#)

response: normexam

explanatory variables: cons,standit

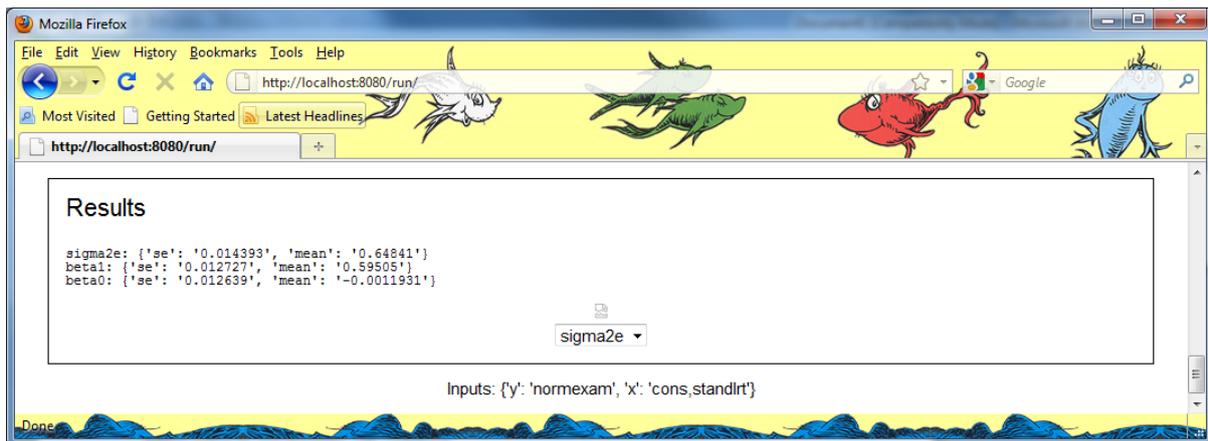
Name of output results: outmlwin

Is estimation method by MCMC: No

Choose estimation engine - MLwiN, R, STATA:

Inputs: {y: 'normexam', x: 'cons,standit'}

After clicking **Next** and **Run** you get almost instantaneous answers:



You will notice 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. If we explore `webtest.py` we will find that there is a chunk of code for when the estimation engine is set to 'MLwiN':

```

elif t.EstObjects['Engine'].name=='MLwiN':
    func = getattr(t, 'WriteMLwiNOut', None)
    if callable(func):
        m=goMLwiN(t)
...

```

Each template must have a method `WriteMLwiNOut` to be able to work with MLwiN and this is checked for prior to the call to `goMLwiN` which plays the role of the `go` function when using MLwiN.

The `goMLwiN` function consists of constructing an `MLwiNOutput` object that will be used to run the defined model using MLwiN. The definitions of the `MLwiNOutput` class are in the file `MLwiNOutput.py` in the subdirectory `\src\lib\EStat\external\MLwiN`. The code for the `goMLwiN` function is as follows:

```

def goMLwiN(t):

    m=MLwiNOutput()
    m.input(t)
    m.write(t)
    t.WriteMLwiNOut(m)
    m.script(t)
    m.execute()
    m.output(t)
    m.Summarise()
    return m

```

As with `goWinBUGS` we have a series of steps which involve constructing files to send to MLwiN, an execution of MLwiN function and postprocessing of the results. The `input` method is first called. This method constructs a datafile ready to send as input to MLwiN. The `write` method is called next. This method constructs the start of the macro that is needed to set up and run the model in MLwiN. The generic code for inputting the data and naming the columns (that is common to all templates) is

written by this method. The template specific *WriteMLwiNOut* method is then called to fill out the rest of the MLwiN macro. For the *Regression2* template the *WriteMLwiNOut* method is as follows:

```
def WriteMLwiNOut(self,MLwiNOut):
    MLwiNOut.macro += 'resp \'' + self.objects['y'] + '\'\n'
    MLwiNOut.macro += 'iden 1 \'' + self.objects['y'] + '\'\n'
    MLwiNOut.params={}
    MLwiNOut.macro += 'put ' + str(len(self.data[self.objects['y']])) + '
1 c'+str(len(MLwiNOut.data.keys()+1)+' \n'
    MLwiNOut.macro += 'name c'+str(len(MLwiNOut.data.keys()+1)+'
\'LevRes\'\n'
    MLwiNOut.macro += 'addt \'LevRes\'\n'
    MLwiNOut.macro += 'setv 1 \'LevRes\'\n'
    MLwiNOut.macro += 'fpar 0 \'LevRes\'\n'
    for i in range(len(self.objects['x'])):
        MLwiNOut.macro += 'addt \'' + self.objects['x'][i] + '\'\n'
    l=0
    for p in range(len(self.objects['x'])):
        MLwiNOut.params['beta'+str(l)]=1.0
        l=l+1
    MLwiNOut.params['sigma2e']=1.0
    # number of parameters
    MLwiNOut.numparams=self.objects['beta'].ncols+1
    MLwiNOut.nummodparams=len(MLwiNOut.params.keys())
```

Here the macro code sets up the response and level 1 id columns. A level 1 residuals (constant) column is then created and named and added at level 1 before all the predictors are added as fixed effects. Several attributes of the MLwiNOut object are also set here.

Having run the template specific code the next call is to the script method. This method completes the macro file by including code for running the model in MLwiN which is estimation method specific. The macro code (for MCMC) then saves the output in the line:

```
self.macro += 'dwrite c1101-c' +str(self.nummodparams+1101-1) + '\n'
self.macro += self.directory + self.objects['myModel'].name
+ 'MCMCChain.txt\n'
```

This is followed by lots of lines that create some fancy plots which currently reside in a temporary directory but should eventually link in when we implement the E-books functionality. Finally the worksheet is saved and MLwiN is exited and we save the string self.macro as a file using the following lines

```
f=open(self.MacroName, 'w')
f.write(str(self.macro))
f.close()
```

The next call in *goMLwiN* is to the *execute* method:

```
def execute(self):
    executable=EStat.configuration.get('MLwiN', 'executable')
    subprocess.call([executable, "/run",self.MacroName])
```

This will simply run MLwiN using the macro we have just constructed. Having run MLwiN the *goMLwiN* function finishes with calls to the *output* and *Summarise* functions which bring back the data into Stat-JR in an appropriate format and construct summary statistics respectively.

We will leave MLwiN here and move onto another package with some functionality for the use of both MCMC and classical estimation methods, R

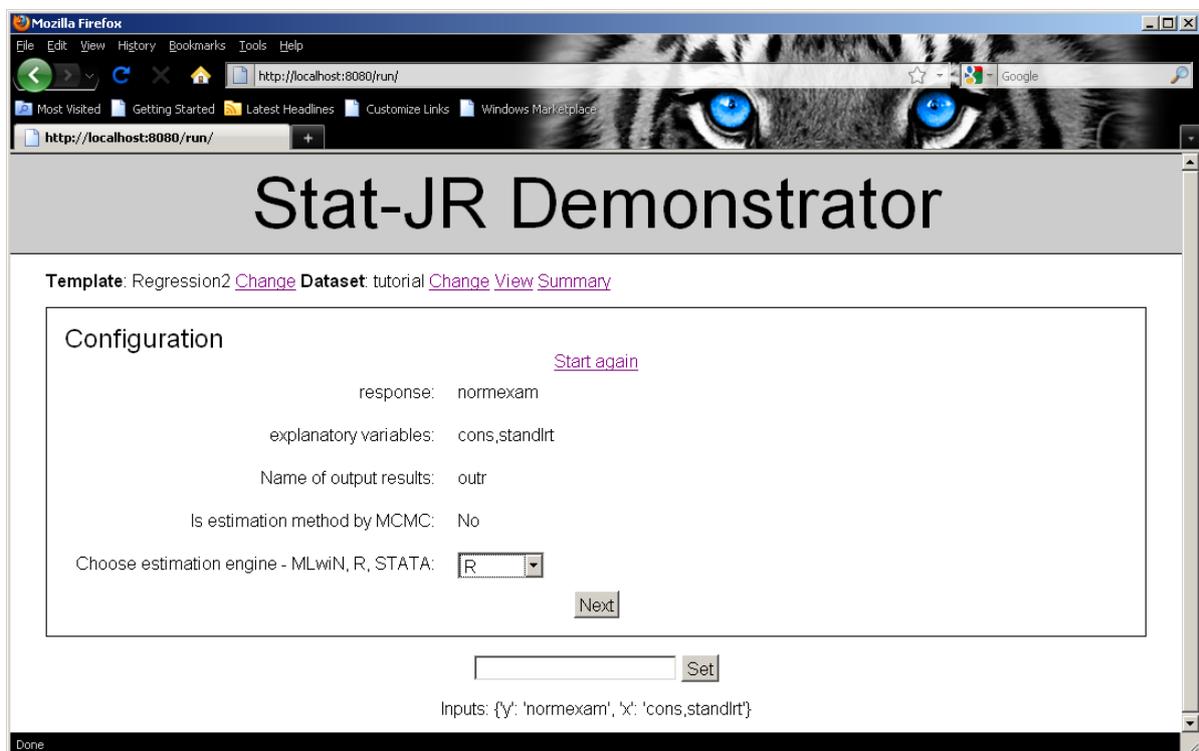
## 6.4 R

R is a general purpose statistics programme that consists of a framework of interlinking statistical commands that are often called packages. The R installation consists of a base package containing many of the standard statistical operations and to this can be added to 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 MCMC *glmm* package 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=C:\Program Files\R\R-2.12.0\bin\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 **Next** and **Run** will give the following output:

Results

```

modelout:
normexam      cons      standirt
Min.   :-3.6661000  Min.   :1  Min.   :-2.935000
1st Qu.:-0.6998100  1st Qu.:1  1st Qu.:-0.620710
Median : 0.0043218  Median :1  Median : 0.040499
Mean   :-0.0001179  Mean   :1  Mean   : 0.001810
3rd Qu.: 0.6787600  3rd Qu.:1  3rd Qu.: 0.619060
Max.   : 3.6661000  Max.   :1  Max.   : 3.016000

Call:
glm(formula = normexam ~ cons + standirt - 1, family = gaussian(identity),
    data = mydata)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.65617  -0.51847   0.01265   0.54397   2.97399

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
cons      -0.001195   0.012642  -0.095   0.925
standirt  0.595055   0.012730  46.744 <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for gaussian family taken to be 0.6487338)

Null deviance: 4049.4 on 4059 degrees of freedom
Residual deviance: 2631.9 on 4057 degrees of freedom
AIC: 9766.5

Number of Fisher Scoring iterations: 2

```

modelout

Inputs: {y: 'normexam', x: 'cons,standirt'}

Here we see the full output of the R command *glm*. If we wish to use MCMC estimation we would give the following inputs:

Stat-JR Demonstrator

Template: Regression2 [Change Dataset](#) tutorial [Change View Summary](#)

Configuration

[Start again](#)

response: normexam

explanatory variables: cons,standirt

Name of output results: outr

Is estimation method by MCMC: Yes

Choose estimation engine - eSTAT, WinBUGS, MLwiN, R: R

Random Seed:

length of burnin:

number of iterations:

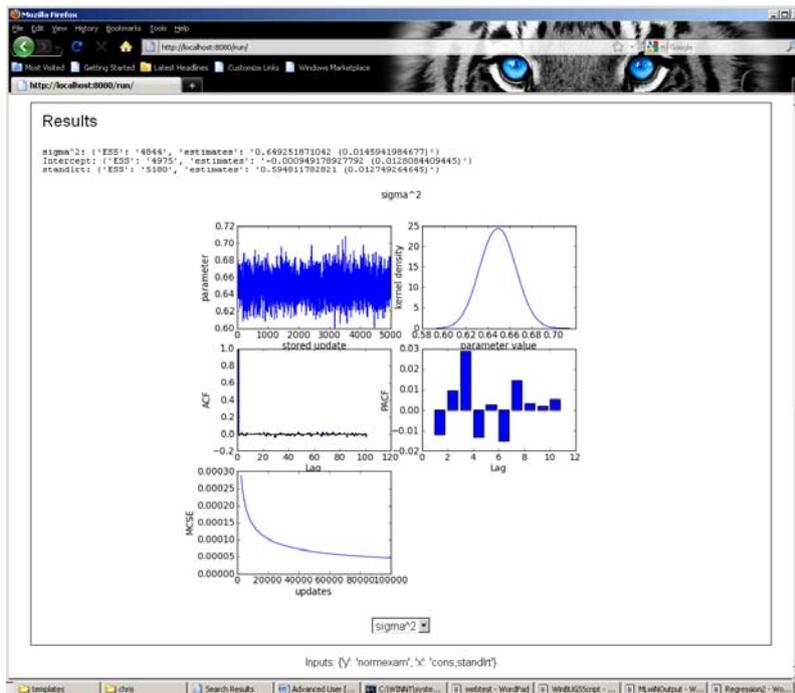
thinning:

Next

Set

Inputs: {y: 'normexam', x: 'cons,standirt'}

This will give the following outputs upon running:



The estimates and graphical output are very similar to those from eSTAT, WinBUGS and MLwiN (using MCMC).

As with other external packages, the *webtest* code interrogates the estimation engine as follows:

```
elif t.EstObjects['Engine'].name=='R':
    func = getattr(t, 'REstimationInput', None)
    if callable(func):
        m=goR(t)
```

...

In fact the interface to R has a similar format to MLwiN as R also requires a packaging up of the data into an appropriate format and a script file to be run. We require a method called *REstimationInput* to be present in the template to allow interface with R and, having checked for its presence, we run the *goR* function which has the following code:

```
def goR(t):
    m=RInterface()
    m.WriteData(t)
    m.Script(t)
    t.REstimationInput(m)
    m.Execute()
    if (m.OutStatus>1):
        print 'Problem at execution:'
        print 'Try running R-script' +str(m.myRScript)+' in R'
    else :
        m.DisplayOut(t)
        m.Summarise()
    return m
```

We have a similar order of processing as with MLwiN. This time an object of type *RInterface* is formed and the definition of this class of object is given in *RInterface.py* which can be found in the

\src\lib\EStat\external\R subdirectory.

After initialising the instance *m* we then call the *WriteData* method that packages up the data into a text file format ready to be fed into R via a script. The *Script* method is called next and this method begins the construction of the script file used within R to import data and fit the model. The *Script* method only writes the start of the script file which is generic code for all templates. The template's own method *REstimationInput* is then called and this then adds to the script file code specific to this template.

In the *Regression2* template this begins with a reallocation of the *outbug* attribute with the lines:

```
self.outbug='''
    ${y.name} ~ ${Rmmult(x.name)}
    '''
```

This is just so that the model can display the model part of the R statements. The R model is set up so that it can be used in R commands by removing the intercept (as it is usually present in the variable list) and doing some text substitutions. There are then two chunks of engine specific code for the classical and MCMC engines respectively. This code is longer than required as it produces some graphs that may be useful when the E-book extension to the Stat-JR software is written.

Finally we see the lines:

```
# write the script out
file=open(Robj.myRScript, 'w')
file.write(Robj.RScript)
file.close()
```

which copy the *Robj* string we have been constructing to a file which will be called from R.

The *execute* method is next called:

```
def Execute(self):
    self.executable=EStat.configuration.get('R', 'executable')
    command='%s --vanilla -f %s' %(self.executable,self.myRScript)
    self.OutStatus=subprocess.Popen(command).wait()
```

which, as one might expect, executes R with the script we have just constructed.

Having run R we have two final functions *DisplayOut* and *Summarise* which read back in the output and construct summary statistics respectively. We next consider the Stata package.

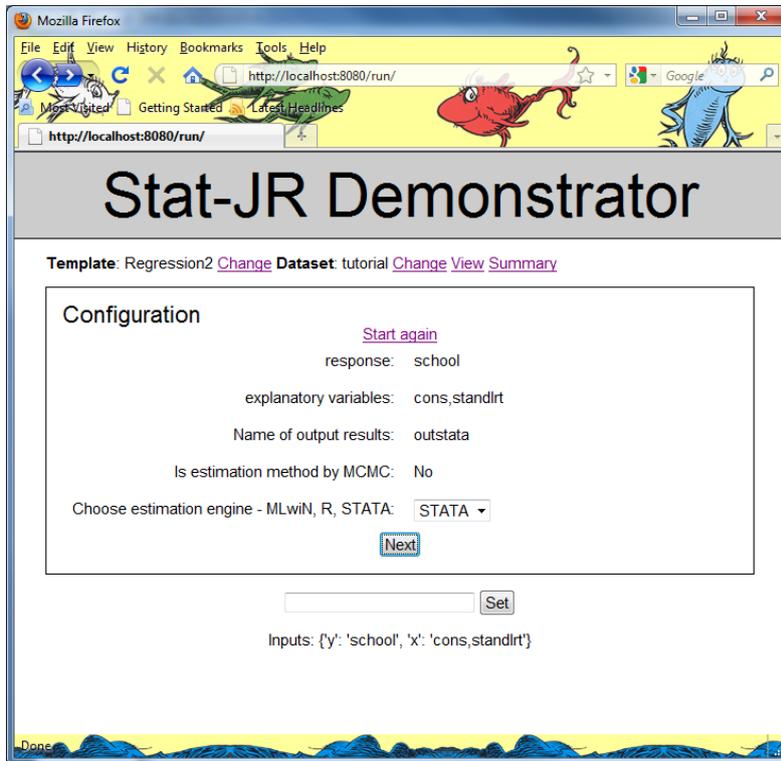
## 6.5 Stata

Stata is another popular statistical package but unlike MLwiN and R it will only allow classical estimation. This means that the code for interacting with Stata is somewhat shorter.

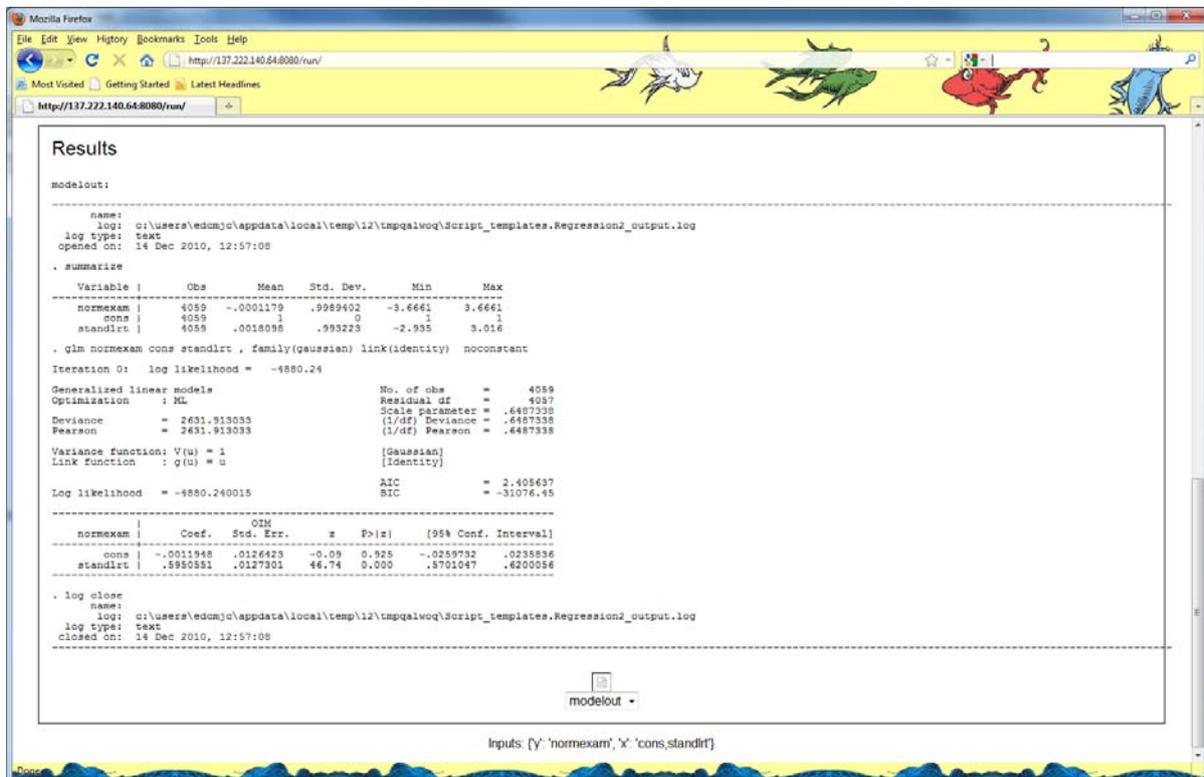
For our *Regression2* template we will utilise Stata's *glm* function for estimation. As with the other three programs we need to include details of the location of Stata on our machine in *settings.cfg* prior to running *webtest*. By default in the file this is as follows:

```
[STATA]
executable=C:\Program Files (x86)\Stata11\StataMP-64.exe
```

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



Clicking the **Next** button and the **Run** button will then run Stata and the output will be as follows:



As with the other estimation engines when we set the attribute Engine via the inputs then this is picked up in *webtest.py* by the following chunk of code:

```
elif t.EstObjects['Engine'].name=='STATA':
    func = getattr(t, 'STATAInput', None)
    if callable(func):
        m=goSTATA(t)
...

```

To run Stata the template must have an additional attribute entitled *STATAInput* which the above code checks for. If it is found then the *goSTATA* function is called which performs the equivalent of the *go* function when using Stata. The code for the *goSTATA* function is as follows:

```
def goSTATA(t):
    m=STATAInterface()
    m.WriteData(t)
    m.Script(t)
    t.STATAInput(m)
    m.Execute()
    m.DisplayOut()
    m.Summarise()
    return m

```

This code creates a *STATAInterface* object and the class definition of such objects is found in the file *STATAInterface.py* which can be found in the `\src\lib\EStat\external\STATA` subdirectory.

Having initialized an instance *m* of the *STATAInterface* class we next call the *WriteData* method. This method converts the data to be used in the model into a text file to be input into Stata. We next call the *Script* method which constructs the beginning of the Stata Do script – which is the method that Stata uses to work in batch mode and fire off a series of commands. The Script is continued via the template specific code which is done by interrogating the *STATAInput* method of the template object whose code is given below:

```
def STATAInput(self, STATAobj):

    family='gaussian'
    link='identity'

    STATAobj.DoScript+='glm '+self.objects['y'].name + ' '
    for p in self.objects['x'].name:
        STATAobj.DoScript+=p+ ' '

    STATAobj.DoScript+=', family('+family+') link('+link+') '

    STATAobj.DoScript+=' noconstant\n' #always remove the intercept

    STATAobj.DoScript+=' log close\n'

```

The model fitting is performed by the lines above and the lines below simply carry out some additional plotting functions that are linked with the template. These plots will for now reside in the temporary directory but may be linked in by the E-book extension to Stat-JR when it is written:

```
# produce diagnostic plots

```

```

        STATAobj.DoScript+='predict yhat, mu\npredict ehat,
response\npredict rhat, response standardized\n'
        STATAobj.DoScript+='scatter ehat yhat\n'
        dumfile=re.sub('\\\\\\', '/', STATAobj.DoScriptOut)
        dumfile2=re.sub('\.', '', dumfile)
        STATAobj.DoScript+='graph export
'+re.sub('_outputlog', 'ResivsFitted.png', dumfile2)+'", replace\n'
        STATAobj.DoScript+='egen rank=rank(rhat)\n'
        STATAobj.DoScript+='gen nscore=invnormal(rank/(_N+1))\n'
        STATAobj.DoScript+='scatter rhat nscore\n'
        STATAobj.DoScript+='graph export
'+re.sub('_outputlog', 'QQ_Plot.png', dumfile2)+'", replace\n'
        STATAobj.DoScript+='exit, clear STATA\n'

        # write the script out
        file=open(STATAobj.myDoScript, 'w')
        file.write(STATAobj.DoScript)
        file.close()

```

The final few lines finish creating the script as a text file and then the *execute* method of the *STATAInput* object is called as shown below to fire up Stata and run the code.

```

def Execute(self):
    self.executable=EStat.configuration.get('STATA', 'executable')
    command='%s %s' %(self.executable, self.myDoScript)
    self.OutStatus=subprocess.Popen(command).wait()

```

Having run the model there are two further functions to display the output and summarise the parameter estimates. This code is shown below:

```

def DisplayOut(self):
    self.out=open(self.DoScriptOut, 'r').read()
    print self.out

def Summarise(self):
    results={}
    if hasattr(self, 'out'):
        results['modelout']='\n\n'+self.out
    return results

```

This ends our 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 chapters we will not 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. We will extend the documentation on interoperability in later versions.

## 7. 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 look at a few examples of the templates along with their code and explain how they fit into the *webtest* interface.

### 7.1 Generate template (*generate.py*)

The first template we will look at is called *Generate* and is used for generating columns to add to a dataset. These columns can be constants, sequences, repeated sequences or random numbers. The *Generate* template has an *invars* function as shown below:

```
invars = '''
outcol = Text('Output column name: ')
type = Text('Type of number to generate: ', ['Uniform Random', 'Binomial
Random', 'Chi Squared Random', 'Exponential Random', 'Gamma Random',
'Normal Random', 'Poisson Random', 'Constant', 'Sequence', 'Repeated
sequence'])

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':
    value = Text('Value')

if type == 'Sequence':
    start = Integer('Starting Value')
    step = Integer('Step')

if type == 'Repeated sequence':
    max = Integer('Maximum number')
    repeats = Integer('Repeats per block')
'''
```

We see that there are two main attributes: a name for the additional column (*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 are no *outbug* or *outlatex* attributes. Instead the computations are performed within a *method* called *resultdata* which performs the required calculation and adds the column to the output.

The *resultdata* method is as follows:

```
def resultdata(self, m):
    datalen = len(self.data[self.data.keys()[0]])

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

    return self.data
```

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 a variable called *random* then the only lines to be executed are:

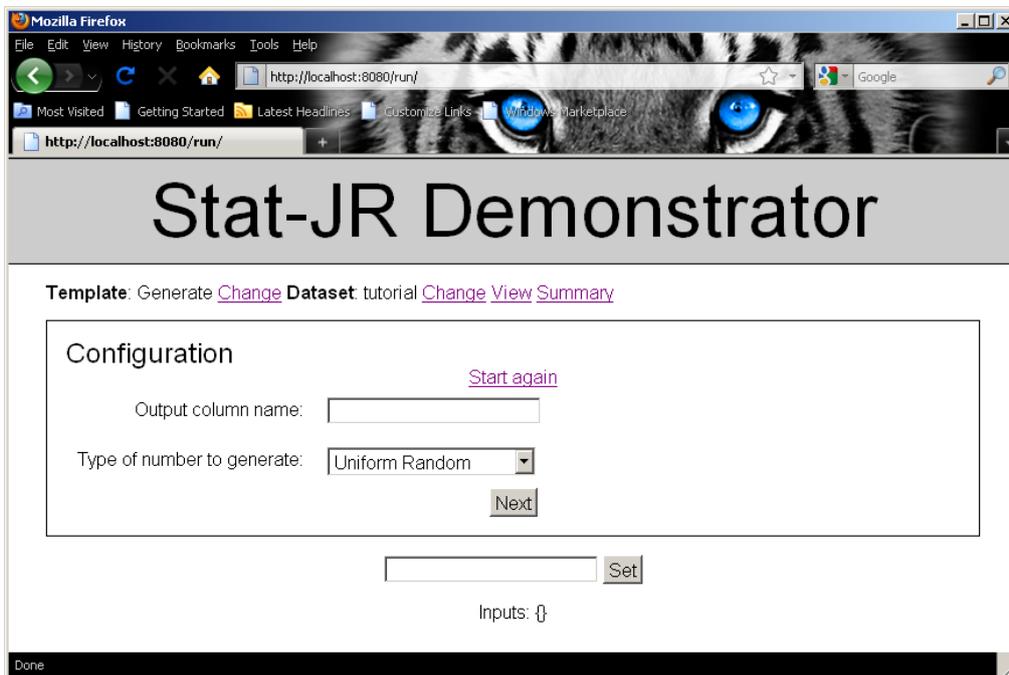
```
datalen = len(self.data[self.data.keys()[0]])
outvar = numpy.random.uniform(size = datalen)
self.data[self.objects['outcol']] = list(outvar)
return self.data
```

The first line in the above code calculates the length of the columns of data (by looking at the first column of data hence the [0] and assuming all columns are the same length) . The next line then

uses the numpy random generator to create the column of numbers in *outvar*. In the third line we link the column into the dataset and finally return the dataset to the output name we gave as an input.

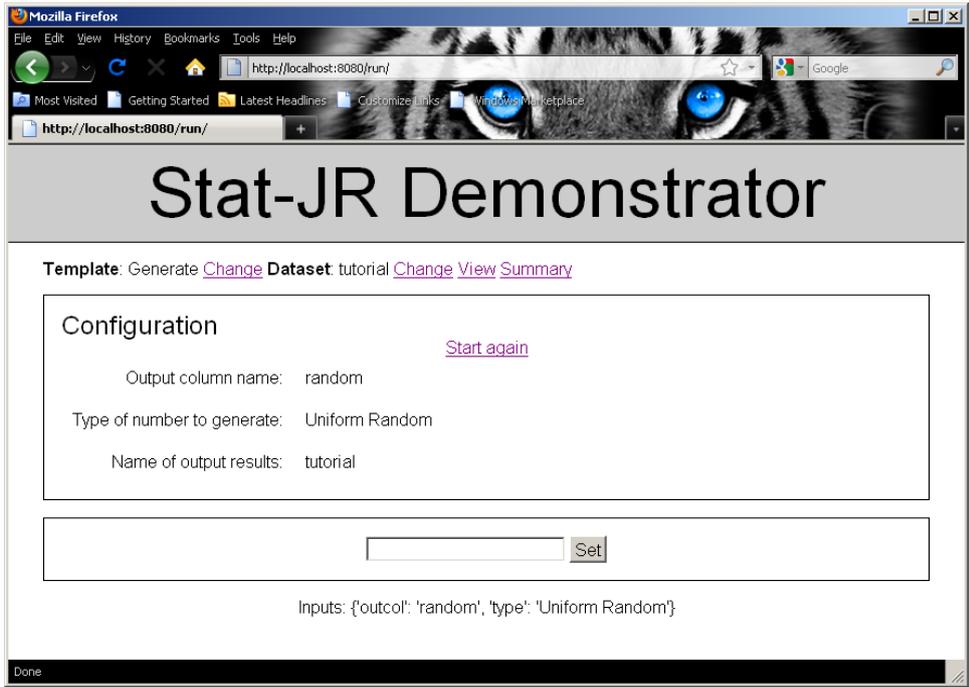
As this template doesn't have any exciting outputs we will not see much happen after execution. The *webtest.py* code knows when the template fits a model and displays the required output. As the *Generate* template is not a model template nothing is displayed. Let's look at an example of adding a vector of uniform random numbers to the tutorial dataset.

We firstly choose the *Generate* template from the pull down list and press the **set** button. Next we press the **Run** button to run the template. The template will look as follows:

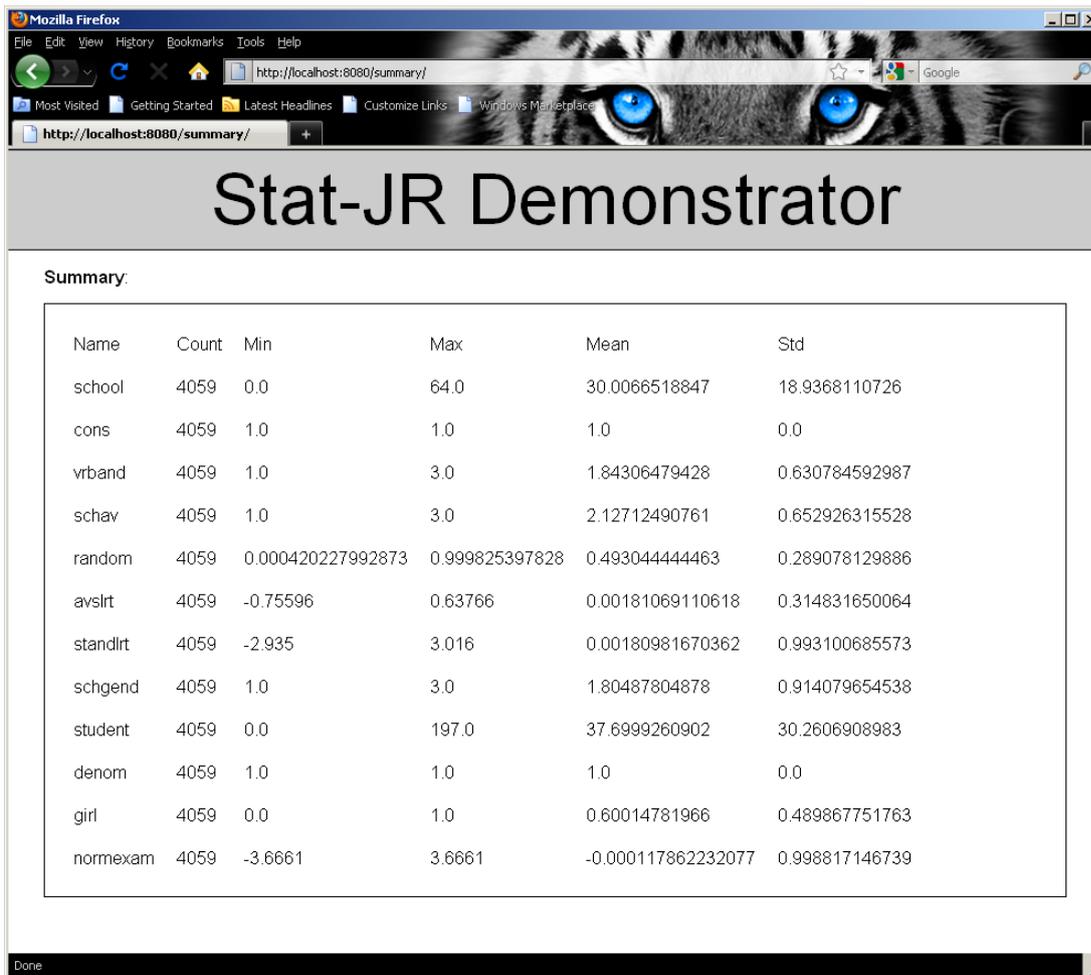


Now we type *random* for the output column name and stick with *Uniform Random* for the type. After clicking Next we are asked for a name for the output results, and here if we enter *tutorial* the new column will be appended to the dataset and the tutorial dataset (in memory) will have an additional 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 we click on the **Summary** button we will see the new tutorial dataset complete with column labelled random as shown below:



## Exercise 2

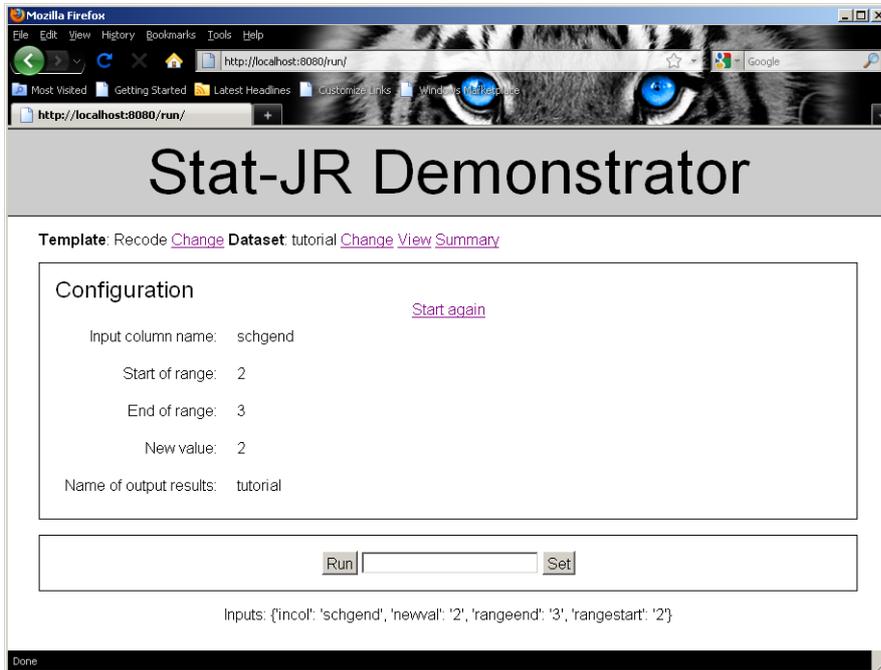
Try modifying this template so that it only offers the random number generators. Try expanding the inputs so for example the Normal random generator will allow a mean and a standard deviation, the Gamma has a scale parameter and the exponential has a rate parameter.

## 7.2 Recode template (recode.py)

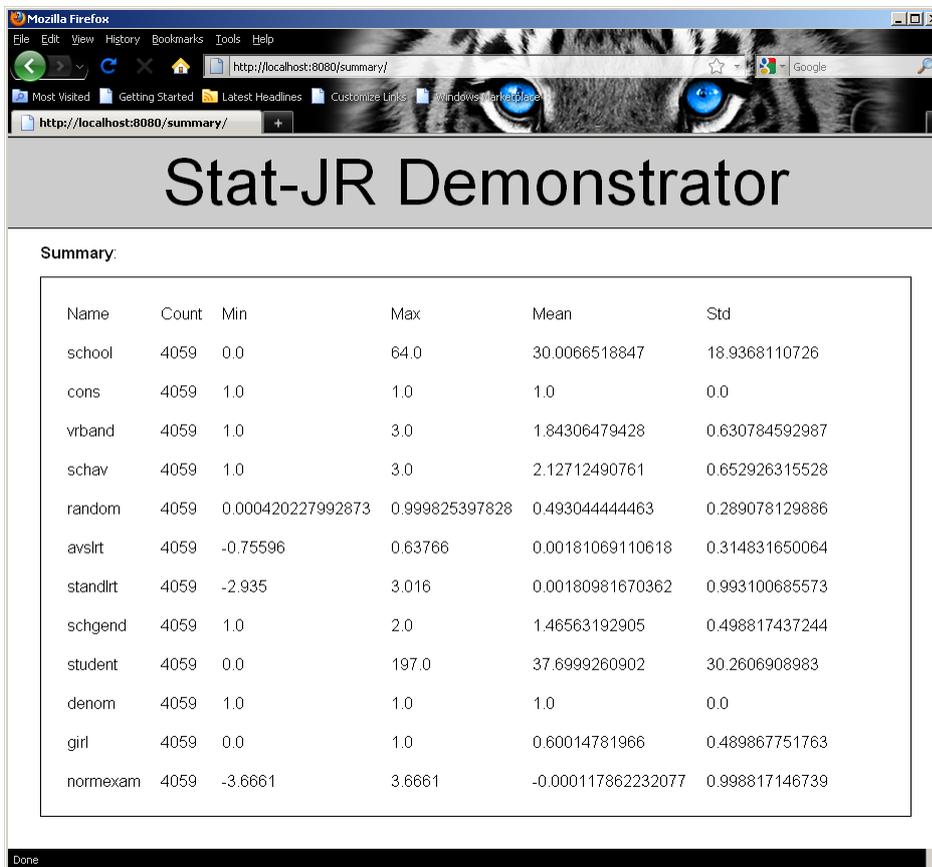
The *Generate* template allows the user to add new columns to their existing dataset. There are many templates that expand or manipulate a dataset and we will here look at a second template, the *Recode* template. The *Recode* template as the name suggests recodes values, in this case recoding values 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 we first select *Recode* from the template list and hit **set** 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 the **Summary** button shows a data summary:



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* (input variables) attribute and a *resultdata* method. These are both quite short:

```
invars = '''
incol = DataVector('Input column name: ')
rangestart = Text('Start of range: ')
rangeend = Text('End of range: ')
newval = Text('New value: ')
'''
```

Here the *invars* attribute has the four inputs that we saw when running the template. We now turn to the *resultdata* method:

```
def resultdata(self, m):
    # Copy data into numpy array for processing
    var = numpy.array(self.data[self.objects['incol']])

    var[(var >= float(self.objects['rangestart'])) & (var <=
float(self.objects['rangeend']))] = float(self.objects['newval'])

    self.data[self.objects['incol']] = list(var)
    return self.data
```

The *resultdata* 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  $\geq$  and  $\leq$  operators mean that the range includes its end points. Finally the modified version of *var* replaces the 'incol' original column in the data 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. choose another name for the recoded variable. Note that the code for the *Generate* template should help here.

## **7.3 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* (input variables) attribute:

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

Here *op* stores the operation to be performed i.e. allows the user to choose between averages and correlations whilst *vars* stores which columns to perform the operation on. This template has another different method, *tabledata*. The name *tabledata* is recognised as a special method by *webtest* and hence the results of running this template are displayed in the browser.

The code for *tabledata* (that will be called by *webtest.py*) is as follows:

```
def tabledata(self):
    if self.objects['op'] == 'averages':
        tabout = [['name', 'count', 'mean', 'sd']]
        for i in range(0, len(self.objects['vars'])):
            var = numpy.array(self.data[self.objects['vars'][i]])
            row = [self.objects['vars'][i], len(var), var.mean(),
var.std()]
                tabout.append(row)

        if self.objects['op'] == 'correlation':
            invars = numpy.empty([len(self.objects['vars']),
len(self.data[self.objects['vars'][0]])])
            for i in range(0, len(self.objects['vars'])):
                for j in range(0, len(self.data[self.objects['vars'][i]])):
                    invars[i, j] = self.data[self.objects['vars'][i]][j]

            corrs = numpy.corrcoef(invars)
            tabout = []
            row = ['']
            for j in range(0, len(self.objects['vars'])):
                row.append(self.objects['vars'][j])
            tabout.append(row)

            for i in range(0, len(self.objects['vars'])):
                row = [self.objects['vars'][i]]
                for j in range(0, len(self.objects['vars'])):
                    row.append(corrs[i, j])
                tabout.append(row)

        print tabout
        return tabout
```

You will see separate chunks of code for averages and correlations. The *average* code basically initializes a table output with a 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 is then stored in *row*, and we use 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. This row is then appended to the output, *tabout*.

The correlation code is slightly longer. We first 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 command `print tabout` is simply a debug line as this printing occurs in the debug window only. The *webtest* program knows to do something with the output as in the *build\_form* method the lines

```
elif hasattr(t, 'tabledata'):
    t.EstObjects['Engine'] = Text()
    t.EstObjects['Engine'].name = 'taboutput'
```

pick up the method *tabledata* and flag the template's name, *Engine*, as 'taboutput'; this is then picked up in the *Run* method where the lines:

```

elif t.EstObjects['Engine'].name=='taboutput':
    t.objects['outtab'] = t.tabledata()

```

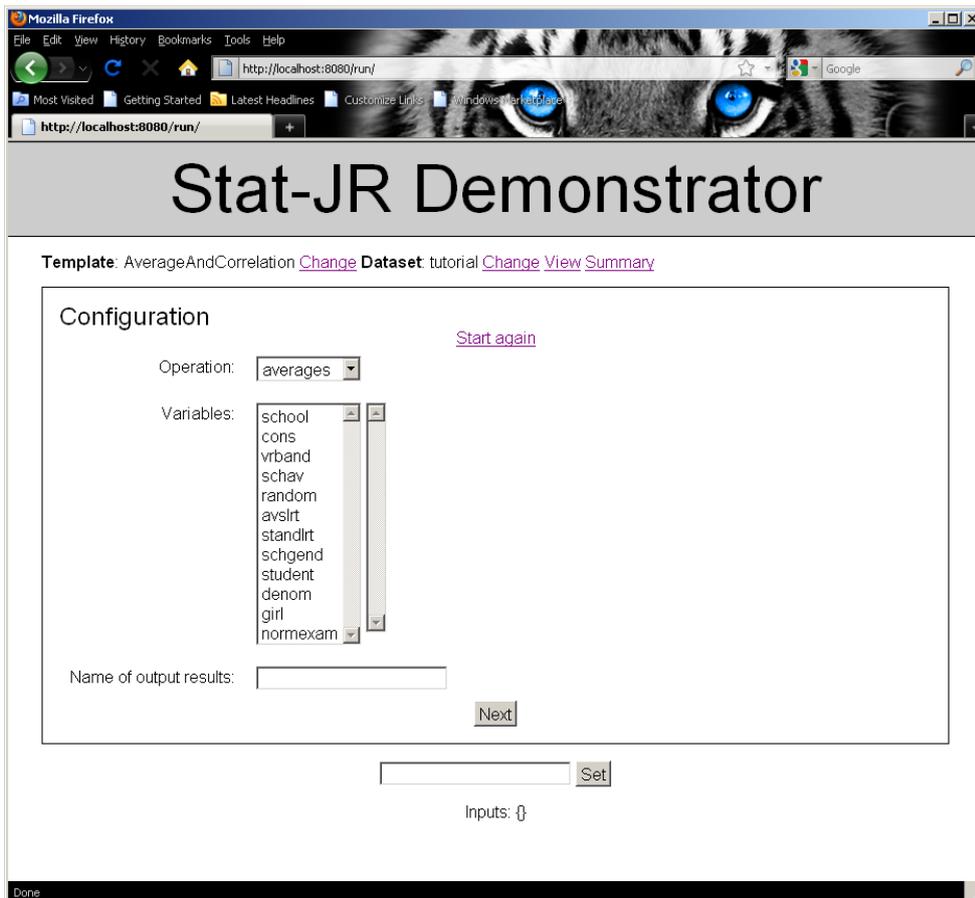
make a template object *outtab* which is checked for by the html template *run.html* with the lines:

```

% if template.objects.has_key('outtab'):
<% print str(template.objects['outtab'])%>
<table class="config">
% for i in range(0, len(template.objects['outtab'])):
<tr>
% for j in range(0, len(template.objects['outtab'][i])):
<td>
${template.objects['outtab'][i][j]}
</td>
% endfor
</tr>
% endfor
</table>
% endif

```

This places the output in the correct place. If we use this template with the *tutorial* dataset we first need to **set** and **run** it to get the default screen:



You can now try applying this template to some of the variables, specifying averages and correlation in turn. It is worth noting that you do not need to fill in the output results box as this is not used by this template.

Here is an example of averages:

The screenshot shows the Stat-JR Demonstrator interface in a Mozilla Firefox browser window. The page title is "Stat-JR Demonstrator". Below the title, there are navigation links: "Template: AverageAndCorrelation", "Change Dataset", "tutorial", "Change View Summary".

The "Configuration" section shows:

- Operation: averages
- Variables: avslit, standlit, girl, normexam
- Name of output results: (empty field)

Below the configuration is a "Set" button. The main content area displays a table of statistics:

name	count	mean	sd
avslit	4059	0.00181069110618	0.314831650064
standlit	4059	0.00180981670362	0.993100685573
girl	4059	0.60014781966	0.489867751763
normexam	4059	-0.000117862232077	0.998817146739

Inputs: {vars: 'avslit,standlit,girl,normexam', 'op': 'averages'}

and the correlations for the same four variables:

The screenshot shows the Stat-JR Demonstrator interface in a Mozilla Firefox browser window. The page title is "Stat-JR Demonstrator". Below the title, there are navigation links: "Template: AverageAndCorrelation", "Change Dataset", "tutorial", "Change View Summary".

The "Configuration" section shows:

- Operation: correlation
- Variables: avslit, standlit, girl, normexam
- Name of output results: (empty field)

Below the configuration is a "Set" button. The main content area displays a table of correlations:

	avslit	standlit	girl	normexam
avslit	1.0	0.317018286878	0.0407057856259	0.287936736229
standlit	0.317018286878	1.0	0.0532110422088	0.591649518248
girl	0.0407057856259	0.0532110422088	1.0	0.114602648345
normexam	0.287936736229	0.591649518248	0.114602648345	1.0

Inputs: {vars: 'avslit,standlit,girl,normexam', 'op': 'correlation'}

#### Exercise 4

Why not try to add an option to this template to give the standard error of the mean (i.e. the standard deviation of the sample mean as an estimate of the population mean) to do this you will

need to calculate a square root and this can be done using the *numpy.sqrt* function. Also try to allow the template to output both averages and correlations together for the same variables – you will need to include them both in the same table output. Remember to rename the template first!

## 7.4 XYPlot template

The 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 allows the user to plot one or more Y variables against an X variable on the same plot.

The template has an *invars* (input variables) attribute as shown below:

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

yaxislabel = Text('Y label: ')
xaxislabel = Text('X label: ')
'''
```

Here we have four inputs: the various Y variables and the corresponding X variable to plot against and then axes labels for each axes. For a graph template we use another special method, *graphdata*, that can be used to output an image file to the browser.

The *graphdata* code is as follows:

```
def graphdata(self, data):
    import numpy
    from matplotlib.figure import Figure
    import matplotlib.lines as lines
    from matplotlib.backends.backend_agg import FigureCanvasAgg
    import subprocess
    import tempfile
    import os
    fig = Figure(figsize=(8,8))
    ax = fig.add_subplot(100 + 10 + 1, xlabel =
str(self.objects['xaxislabel']), ylabel = str(self.objects['yaxislabel']))
    for n in self.objects['yaxis']:
        ax.plot(self.data[self.objects['xaxis']], self.data[n], 'x')
    canvas = FigureCanvasAgg(fig)
    directory = tempfile.mkdtemp() + os.sep
    canvas.print_figure(directory + 'xyplot.png', dpi=80)
    return directory + 'xyplot.png'
```

We have to firstly import several Python libraries in order to call the graphics function. We are using the *Figure* function from the *matplotlib* package. We then make a new plot (using the *Figure* command) before sticking on the axes labels; and 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.

Once again the *webtest* program looks for the method name to know to plot the figure. This is achieved in the *build\_form* function via the following code:

```
elif hasattr(t, 'graphdata'):
    t.EstObjects['Engine'] = Text()
```

```
t.EstObjects['Engine'].name = 'output'
```

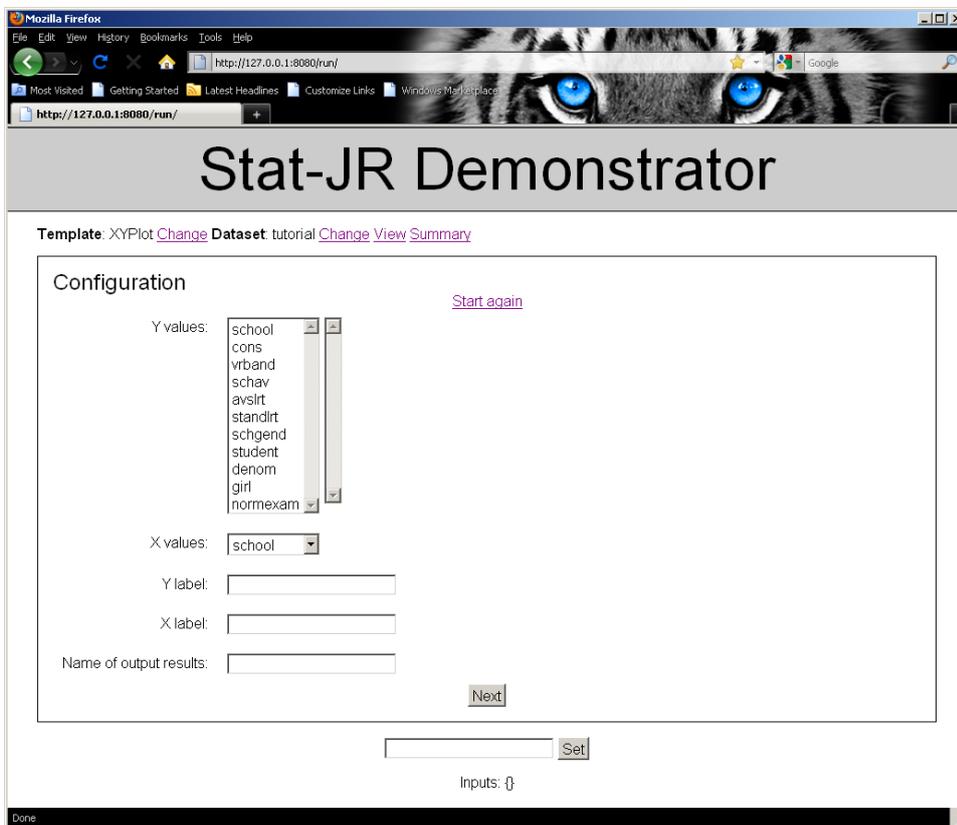
The *Engine* name being set to 'output' is picked up in the *Run* method in *webtest* with the following check

```
elif t.EstObjects['Engine'].name=='output':  
    t.objects['outimg'] = t.graphdata(context['variables'])
```

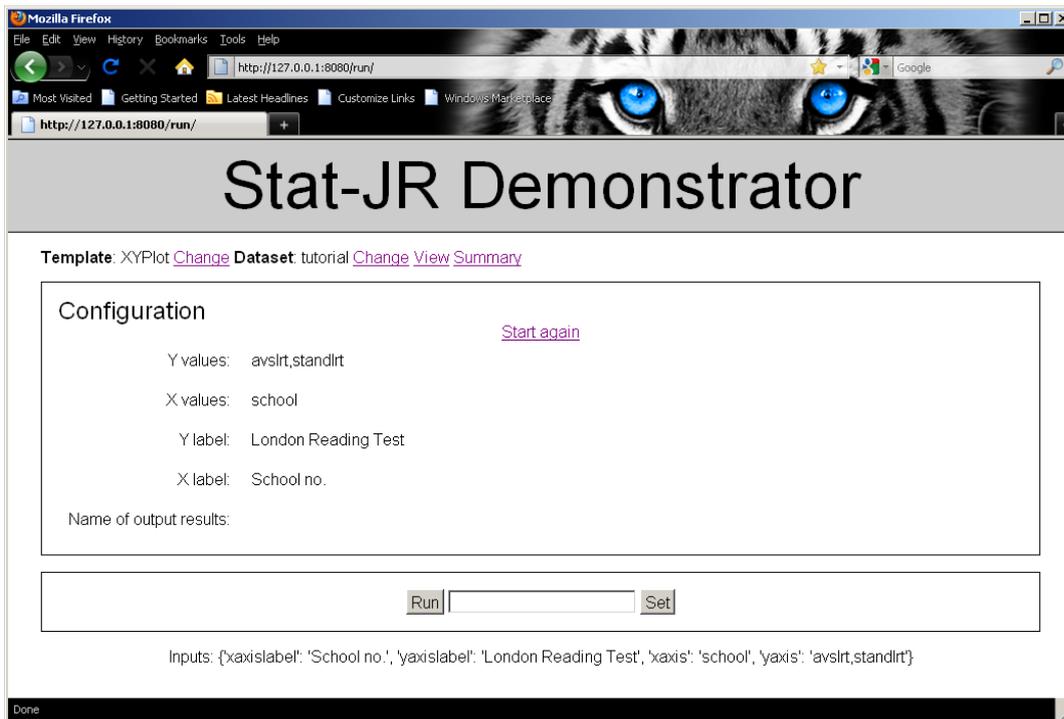
This sets an attribute *outimg* which is picked up in the html template file *run.html* which then does the following:

```
% if template.objects.has_key('outimg'):  
  
% endif
```

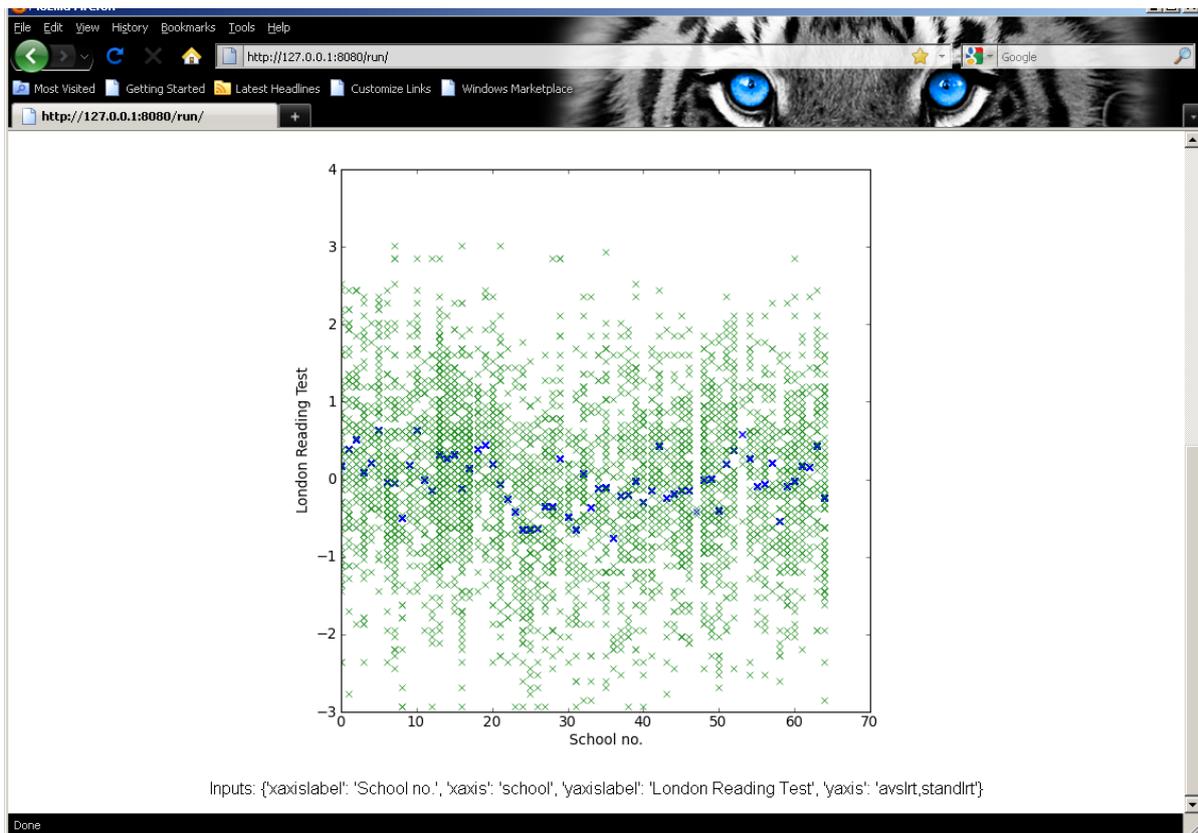
To see this template in action we will pick it from the *webtest* main screen (along with the *tutorial* dataset) and after running we will see the following in the browser:



Perhaps the simplest plot would be of *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:



Clicking on the **Run** button 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**

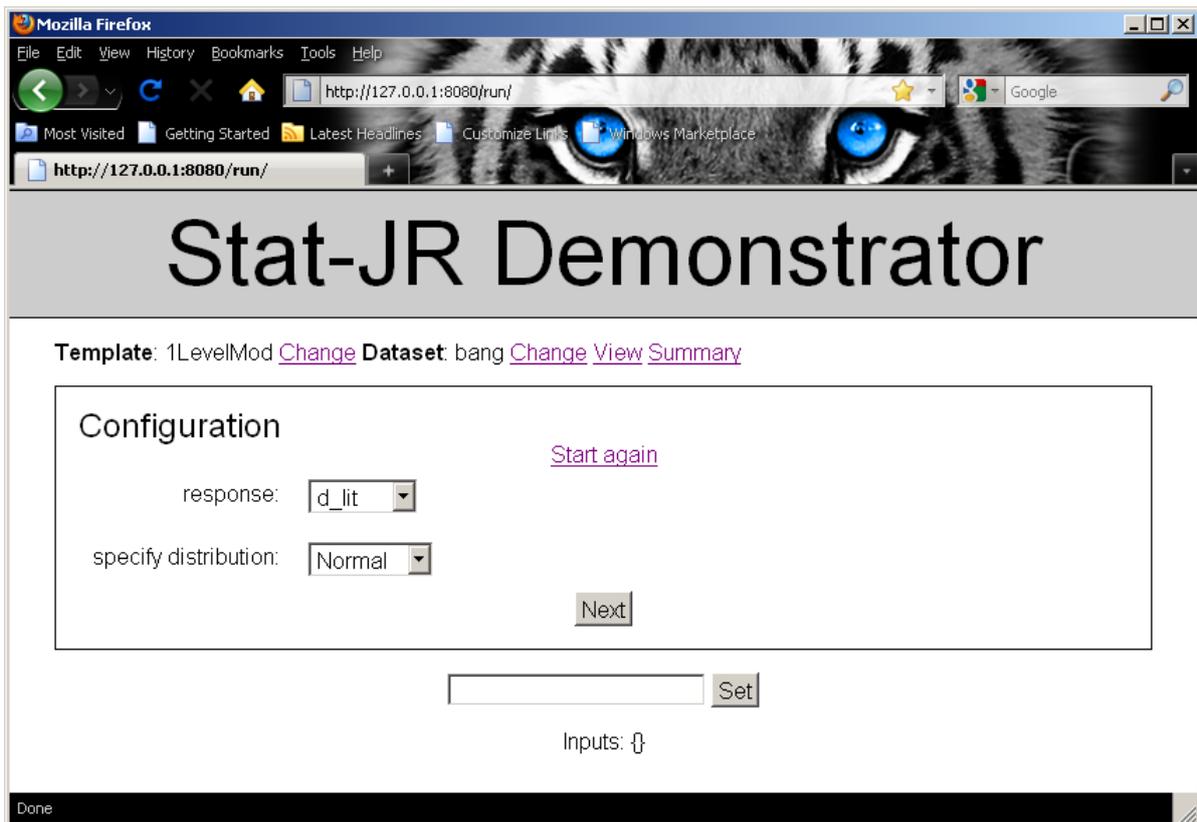
Simplify this template to allow only a single y variable. Try adding a main title to the graph and varying the symbol and colours. To do this you will need to look at the documentation for the plot command in matplotlib

([http://matplotlib.sourceforge.net/api/pyplot\\_api.html#matplotlib.pyplot.plot](http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.plot)). You can then maybe make this an option for the user to choose. Remember however to rename the template before you start!

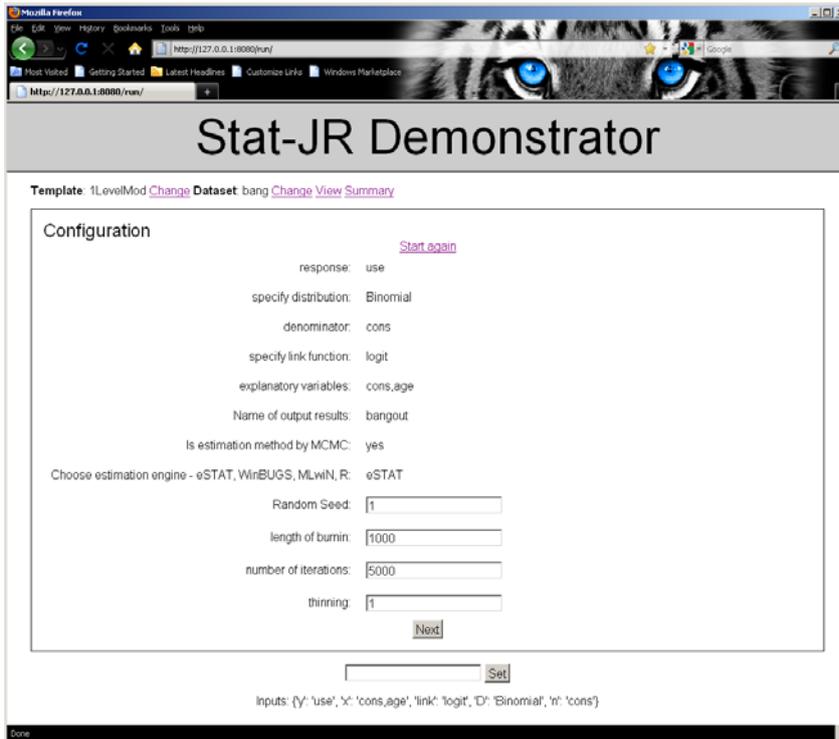
## 8. Single level models of all flavours – A logistic regression example

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

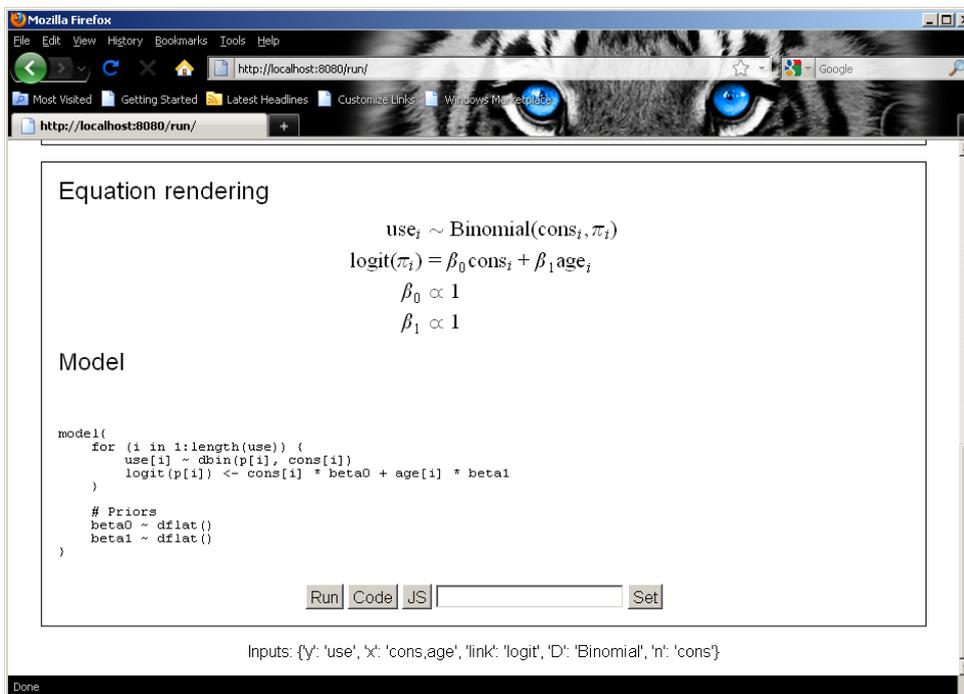
We will begin by looking at the template in action in *webtest*. The template should be able to fit all the models that *Regression1* fits, and so you could try repeating the earlier regression analysis, but here we will look at a logistic regression. From the main browser screen we need to set the *template* to be *1LevelMod* and the *dataset* to be *bang*, our example binary response dataset. Clicking on **Run** gives the following output in the browser:



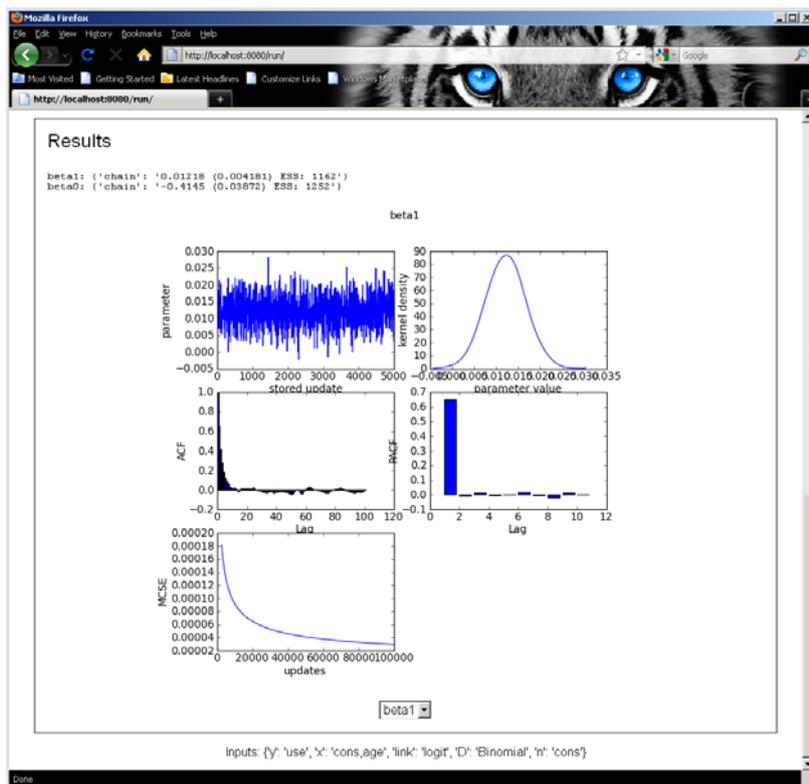
We will now set up the various inputs as shown on the following screen:



We are fitting a logistic regression to the response variable 'use' which indicates whether a Bangladeshi woman is using contraception at the time of the survey. We are regressing 'use' on age and using the Stat-JR built-in MCMC engine with some default settings for estimation. Clicking on **Next** will display the model:



We will next click on **Run** to run the model and then get the following results:



We see that the age coefficient is positive and statistically significant, indicating that older women are more likely to use contraceptives. We now 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 implements interoperability (this will be an extension of the code for *Regression2* in chapter 6).

## 8.1 Invars

The code for *invars* (input variables) is as follows:

```
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()
x = DataMatrix('explanatory variables: ')
beta = ParamVector()
beta.ncols = len(x)
'''
```

Compared to *Regression1* you will see that we have introduced an input *D* for distribution along with 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 (note that currently there is a bug in the algebra system so that probit and cloglog do not work correctly). We also see that for non-normal models there is no level 1 variance and so the quantities *tau* and *sigma* are not included.

## 8.2 Methodinput

This template has its own *Methodinput* method designed to allow the user to choose between the various possible engines to estimate the model , as shown below:

```
def MethodInput(self):
    self.EstObjects['EstM']=Boolean('Is estimation method by MCMC: ')
    if self.EstObjects['EstM']:
        self.EstObjects['Engine']=Text('Choose estimation engine -
eSTAT, WinBUGS, MLwiN, R: ', ['eSTAT', 'WinBUGS', 'MLwiN', 'R'])

        if self.EstObjects['Engine'] == 'MLwiN':
            self.EstObjects['FixedAlg']=Text('Select MCMC sampling
method for fixed effect- Gibbs, MH: ', ['Gibbs', 'MH'])

            if self.objects['D'] == 'Normal':
                self.EstObjects['VarAlg']=Text('Select MCMC sampling
method for residuals- Gibbs, MH: ', ['Gibbs', 'MH'])

        if self.EstObjects['Engine']=='WinBUGS':
            self.EstObjects['nchains']=Integer('number of chains: ')
        else :
            self.EstObjects['nchains']=1

        self.EstObjects['seed']=Text('Random Seed: ')
        self.EstObjects['burnin']=Integer('length of burnin: ')
        self.EstObjects['iterations']=Integer('number of iterations: ')
        self.EstObjects['thinning']=Integer('thinning: ')

        if self.EstObjects['Engine']=='R':
            if self.objects['D']=='Binomial':
                if self.objects['link'] != 'logit':
                    if self.objects['link']!='probit':
                        print 'other link functions for binomial
distribution not implemented, choose another engine'
                        self.MethodInput()

            else :
                self.EstObjects['Engine']=Text('Choose estimation engine -
MLwiN, R, STATA: ', ['MLwiN', 'R', 'STATA'])
```

We see the two inputs, *EstM* and *Engine* , that correspond to the first two method inputs in the example. There are then chunks of code for MLwiN and WinBUGS before the standard inputs for *seed*, *burnin*, *iterations* and *thinning*. There is then some code for R (Camille: Not sure why this is not before standard inputs?) and finally the *Engine* choices when MCMC is not chosen.

## 8.3 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]) <- \\
    % 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)
  % 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.

## 8.4 Outlatex

Finally the *outlatex* method now also contains conditional statements.

```

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

```

outlatex = r'''
\begin{aligned}
\mbox{\${y}}_i & \sim \mbox{Binomial}(\mbox{\${n}}_i, \pi_i) \\
\mbox{\${link}}(\pi_i) & = \\
  \$\mmlttex(x, r'\beta', 'i') \\
%for i in range(0, len(x)):
\beta_{\${i}} & \propto 1 \\
%endifor
\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 another filename e.g. *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 chapter with your new template.

## 9. Including categorical predictors

The template *1LevelMod* can handle many response types but treats all predictor variables as if they are continuous. This means that if we have a categorical predictor, for example school gender in the tutorial dataset, we will need to perform some data manipulation to construct the dummy variables that are used for a categorical predictor before using this template. We will here look at an alternative template that has built-in functionality for constructing these categorical variables within the model 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. We then demonstrate its use with the *tutorial* dataset.

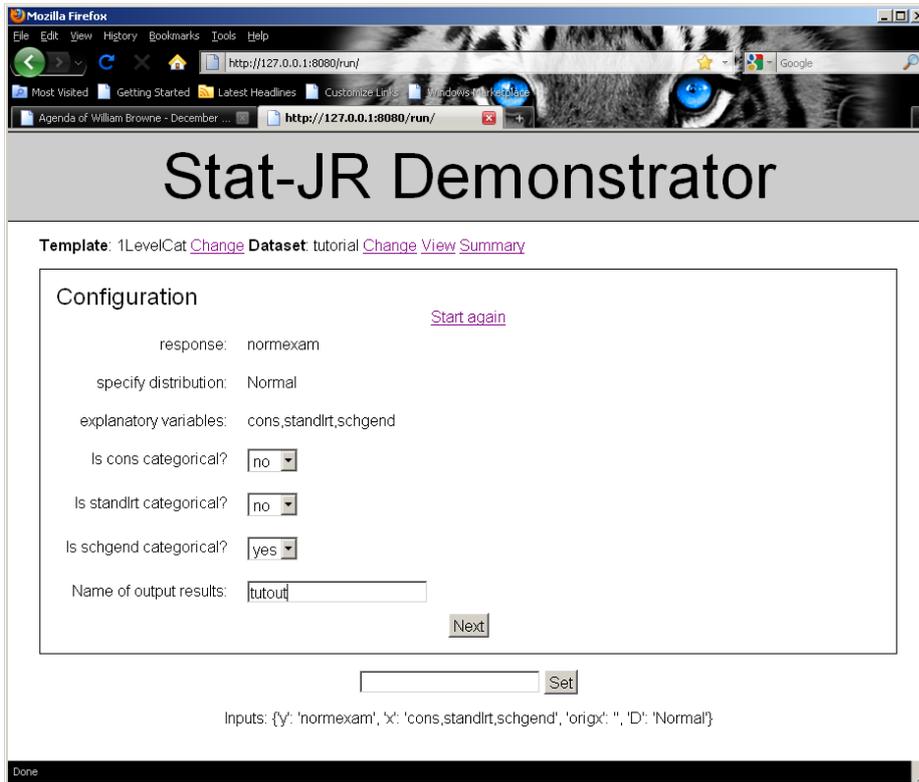
The *invars* function is as follows:

```
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()
x = DataMatrix('explanatory variables: ')
for var in x:
    context[var + '_cat'] = Boolean('Is ' + var + ' categorical? ')
origx = Text(value = [])
beta = ParamVector()
'''
```

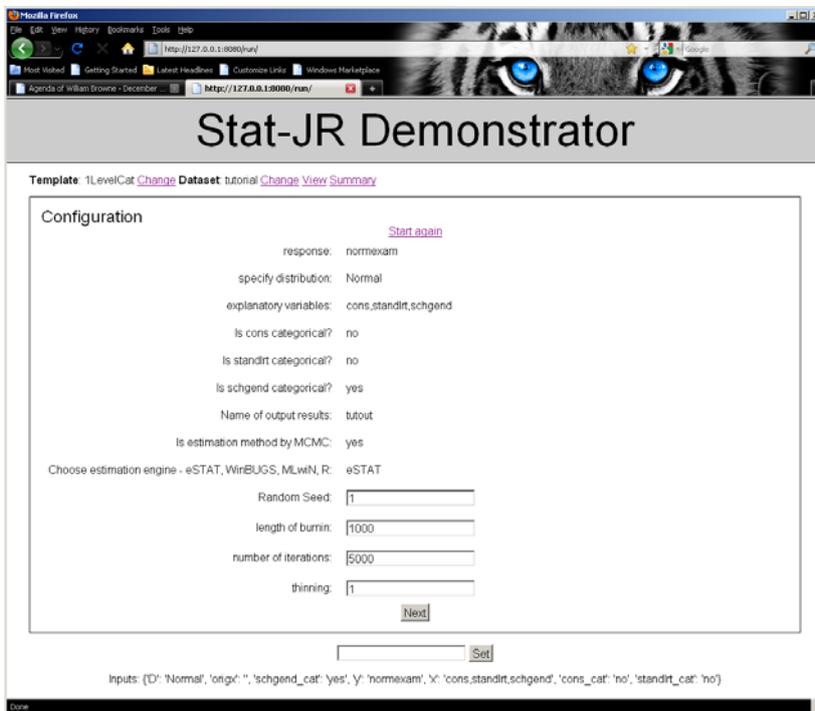
This section of code is the same as that in *1LevelMod* up to the point that *x* is input. We next see a for loop that includes the use of the context statement to construct attribute names that are a combination of text and variable names. If, for example, *x* contains the three-variable list ['cons', 'standlrt', 'schgend'] then the context statements will concatenate the the three variables with the text string '\_cat' to create three new (Boolean) variables 'cons\_cat', 'standlrt\_cat' and 'schgend\_cat' which will store whether the variables are categorical or not. The line

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

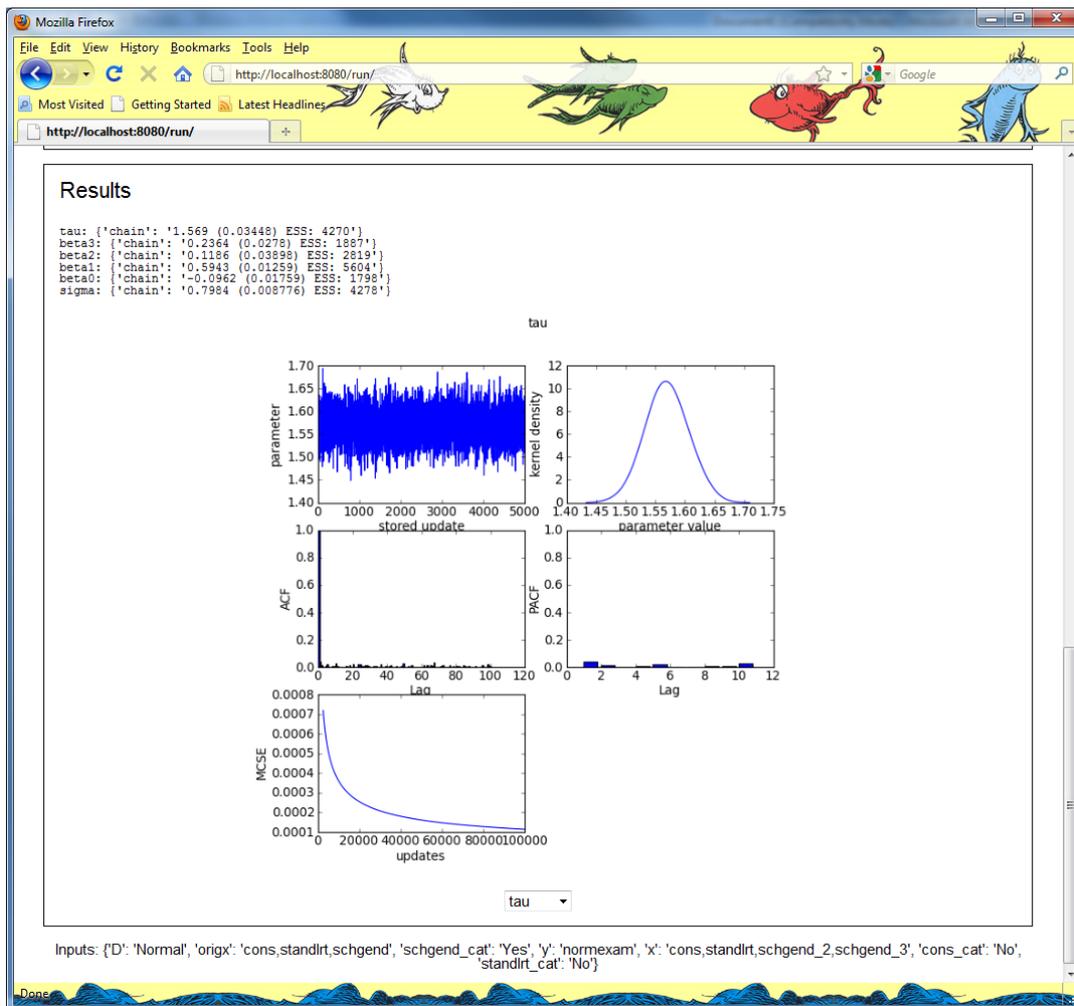
will be used to store the original *x* variables prior to manipulating the categorical variables. By setting its value in the assignment (i.e. value = []) we will not get an input widget appearing in the browser. Let us now demonstrate fitting a model with a categorical predictor. Choose *1LevelCat* from the template list and *tutorial* as the dataset. Firstly we will choose the inputs as follows (remembering that the 'Name of output results' input is the name given to where the resulting chains will be stored):



Click on the **Next** button and set the estimation options as follows:



If we press **Next** again we will get the *outbug* and *outlatex* output as follows:



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 which we mentioned briefly when describing the generic template definition in *templating.py*. The generic definition of the method *preparedata* basically attaches the data to the template, but we can write template-specific versions of the method that replace this generic function and allow preprocessing of the data. In this case the code is as below:

```

def preparedata(self, data):
    self.data = data
    for var in self.objects['x']:
        self.objects['origx'].name.append(var) # Save user's original
selection
    del self.objects['x'][:]
    for var in self.objects['origx'].name:
        if self.objects[var + '_cat']:
            uniqvals = list(set(self.data[var]))
            uniqvals.sort()
            uniqvals.remove(uniqvals[0])
            for i in uniqvals:
                self.data[var + '_' + str(int(i))] =
list((numpy.array(self.data[var])[:, :] == i).astype(float))
                self.objects['x'].name.append(var + '_' + str(int(i)))

```

```

        del self.data[var]
    else:
        self.objects['x'].name.append(var)
self.objects['beta'].ncols = len(self.objects['x'])
return self.data

```

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 conditional (the if statement) on a particular variable being categorical does some processing. The lines

```

uniquvals = list(set(self.data[var]))
uniquvals.sort()
uniquvals.remove(uniquvals[0])

```

find all unique values in the categorical predictor which are then stored in *uniquvals*. We then sort these into ascending order before removing the first (*uniquvals[0]*) as it will play the role of the base category in the model.

We then have a second loop over this list of *uniquvals* where we create the dummy variables. The lines

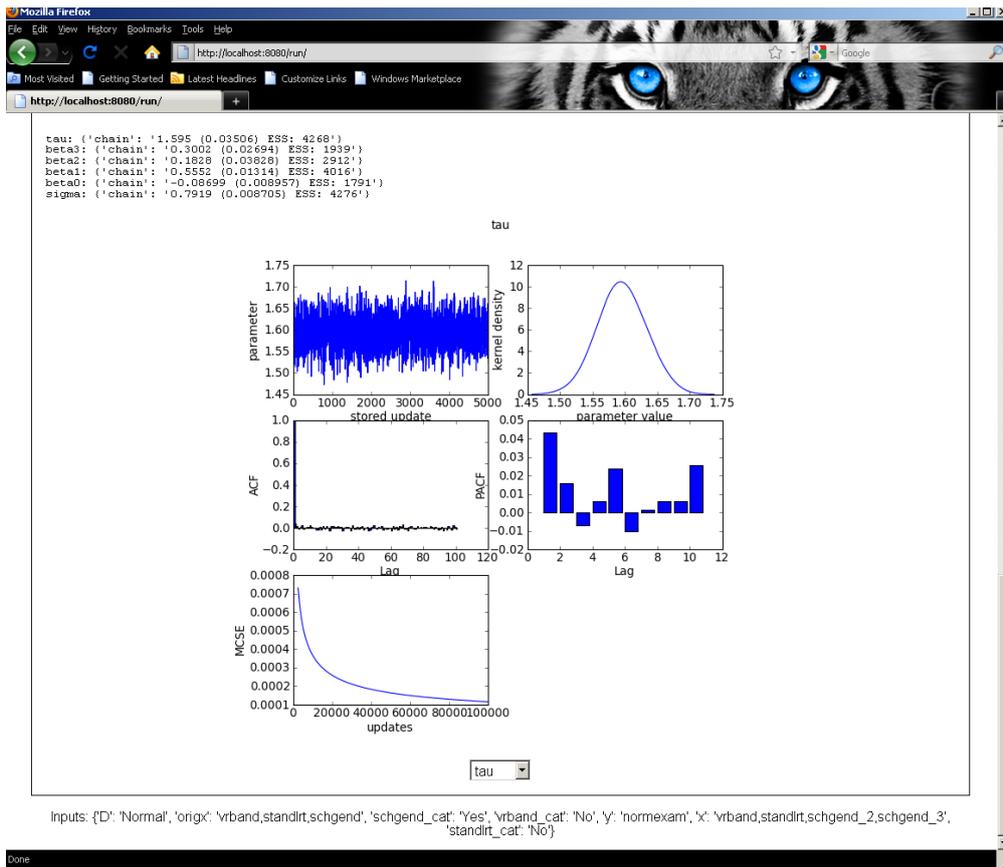
```

self.data[var + '_' + str(int(i))] =
    list((numpy.array(self.data[var])[:,] == i).astype(float))
self.objects['x'].name.append(var + '_' + str(int(i)))

```

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 fit the model with the dummy variables for school gender, we press the **Run** button and get the following results:



This completes this chapter and is the last single-level model we will meet for a while. Another extension would be to allow the inclusion of interactions in the model. This has been done in the template *1LevelwithInteractions*. 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.

## 10. Multilevel models

We now move to templates for models for more complex data structures. In this chapter 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. (one level of clustering) and then secondly a more general template that will fit models with any number of levels of clustering whether nested or crossed. Note here that these templates allow only random intercepts.

### 10.1 2LevelMod template

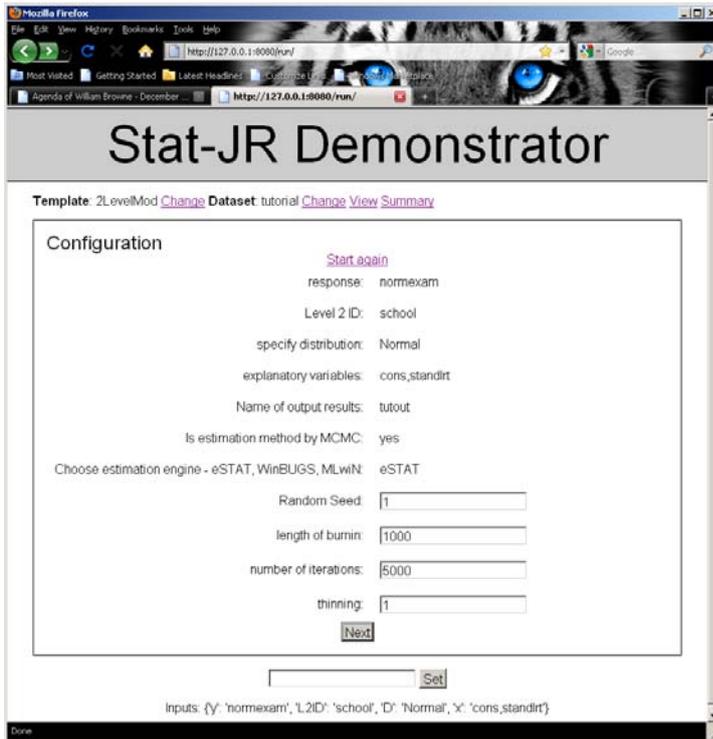
We will begin our investigation of *2LevelMod* by looking at its *invar* function:

```
invars = '''
y = DataVector('response: ')
L2ID = DataVector('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 = ParamScalar()
    sigma = ParamScalar()
if D == 't':
    tau = ParamScalar()
    sigma = ParamScalar()
    #df = ParamScalar()
u = ParamVector()
tau_u = ParamScalar()
sigma_u = ParamScalar()

x = DataMatrix('explanatory variables: ')
beta = ParamVector()
beta.ncols = len(x)
'''
```

If you compare this with the *invars* function for *1LevelMod* you will see we have added one additional input, *L2ID*, to allow the user to input the column containing the level 2 identifiers. We have also included three additional model parameters *u*, *tau\_u* and *sigma\_u* (the level 2 residuals, their precision and standard deviation respectively).

We can try out an example of these inputs by selecting the template *2LevelMod* and the dataset *tutorial* and specifying the following inputs:



Clicking on **Next** will bring up the maths and model code:

Equation rendering

$$\text{normexam}_i \sim N(\mu_i, \sigma^2)$$

$$\mu_i = \beta_0 \text{cons}_i + \beta_1 \text{standlrt}_i + u_{\text{school}[i]}$$

$$u_{\text{school}[i]} \sim N(0, \sigma_u^2)$$

$$\beta_0 \propto 1$$

$$\beta_1 \propto 1$$

$$\tau \sim \Gamma(0.001, 0.001)$$

$$\sigma^2 = 1/\tau$$

$$\tau_u \sim \Gamma(0.001, 0.001)$$

$$\sigma_u^2 = 1/\tau_u$$

Model

```

model {
  for (i in 1:length(normexam)) {
    normexam[i] ~ dnorm(mu[i], tau)
    mu[i] <- cons[i] * beta0 + standlrt[i] * beta1 + u[school[i]] * cons[i]
  }

  for (j in 1:length(u)) {
    u[j] ~ dnorm(0, tau_u)
  }

  # Priors
  beta0 ~ dflat()
  beta1 ~ dflat()

  tau ~ dgamma(0.001000, 0.001000)
  sigma <- 1 / sqrt(tau)

  tau_u ~ dgamma(0.001000, 0.001000)
  sigma_u <- 1 / sqrt(tau_u)
}

```

Run Code JS Set

Inputs: {y: 'normexam', 'L2ID': 'school', 'D': 'Normal', 'X': 'cons,standlrt}

Here we see a mathematical representation of the model, created in *outlatex*, along with the model code in *outbug*.

We saw in the last chapter the use of *preparedata* and this method is used here as well:

```

def preparedata(self, data):
    self.data = data
    self.objects['u'].ncols =
-1*len(numpy.unique(self.data[self.objects['L2ID']].name))
    return self.data

```

All we use *preparedata* for, apart from the default operation of copying the data to the template, is to assess the number of unique level 2 identifiers and hence the length of the level 2 residual vector, *u*.

Let's look 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])
  ${link}(p[i]) <- \\
  % if offset:
${n}[i] + \\
  % endif
  % endif
  % if D == 't':
dnorm(mu[i], tau)
  mu[i] <- \\
  % endif
${mmult(x, 'beta', 'i')} + u[${L2ID}[i]] * cons[i]
}
for (j in 1:length(u)) {
  % if D == 't':
  u[j] ~ dt(0, tau_u, df)
  % else:
  u[j] ~ dnorm(0, tau_u)
  % endif
}
# 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)
% endif
% if D == 't':
tau ~ dgamma(0.001000, 0.001000)
sigma <- 1 / sqrt(tau)
#df ~ dunif(2, 200)
df <- 8
% endif
tau_u ~ dgamma(0.001000, 0.001000)
sigma_u <- 1 / sqrt(tau_u)
}
'''

```

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  $u[{\$}\{L2ID.name\}[i]] * cons[i]$  has been appended to the linear predictor. Note that due to a bug in the algebra system we need to include the multiplication by  $cons[i]$  otherwise the posterior is incorrectly calculated. The chunk of code

```

for (j in 1:length(u)) {
  % if D.name == 't':
  u[j] ~ dt(0, tau_u, df)
  % else:

```

```

    u[j] ~ dnorm(0, tau_u)
  % endif
}

```

specifies the random effect distribution, and finally the code

```

tau_u ~ dgamma(0.001000, 0.001000)
sigma_u <- 1 / sqrt(tau_u)

```

specifies the prior distribution for the precision of the random effects. The *outlatex* function is adapted in very similar ways and so for brevity we omit this code here.

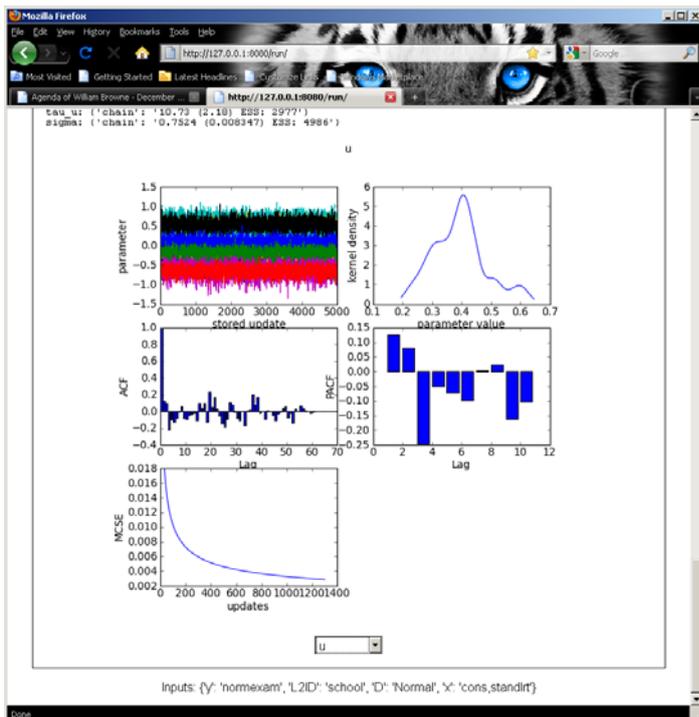
Running this template produces a large amount of output because there are 65 level 2 residuals. An extract of the output is as follows:

```

tau: {'chain': '1.767 (0.0392) ESS: 5004'}
sigma_u: {'chain': '0.31 (0.03178) ESS: 2961'}
u: {'chain': '
0.374 (0.09149) ESS: 1682
0.5033 (0.1037) ESS: 2545
...
...
-0.1679 (0.08954) ESS: 1464'}
beta1: {'chain': '0.5633 (0.01254) ESS: 3984'}
beta0: {'chain': '0.003276 (0.0395) ESS: 252'}
tau_u: {'chain': '10.73 (2.18) ESS: 2977'}
sigma: {'chain': '0.7524 (0.008347) ESS: 4986'}

```

This template also stores chains for each of the 65 random effects  $u$ . As usual we also get the MCMC plots which for  $u$  is kind of strange as it includes all 65 chains together so should probably be ignored!



## Exercise 7

The *2LevelCat* template extends *2LevelMod* to allow for categorical predictors. Try adapting this template so that it allows the user to incorporate interactions. Remember to save your new template under a different name. (You will find it helpful to look at *1LevelwithInteractions* first.)

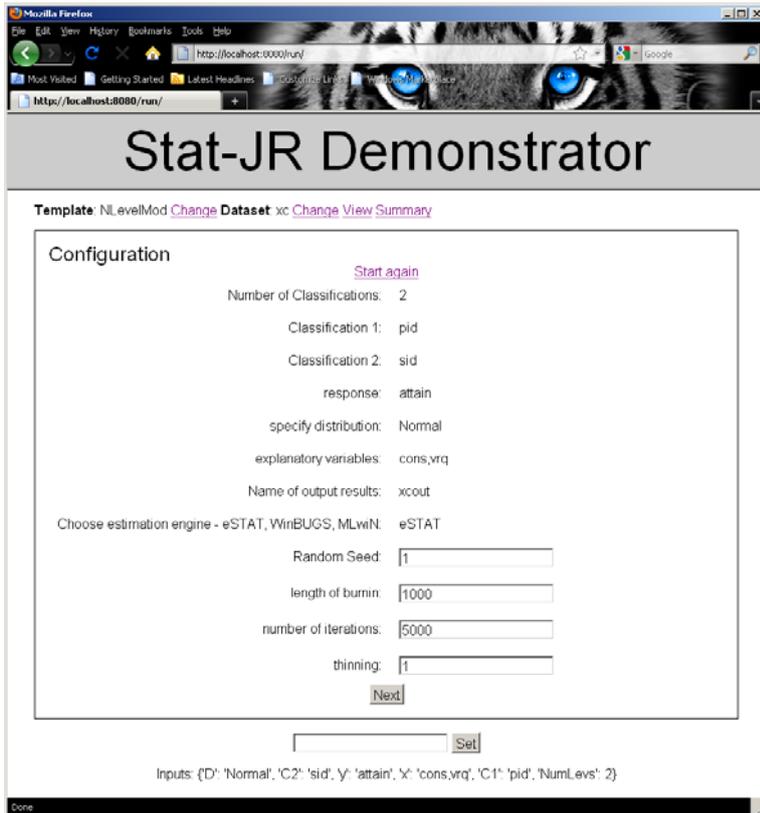
## 10.2 NLevelMod template

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*:

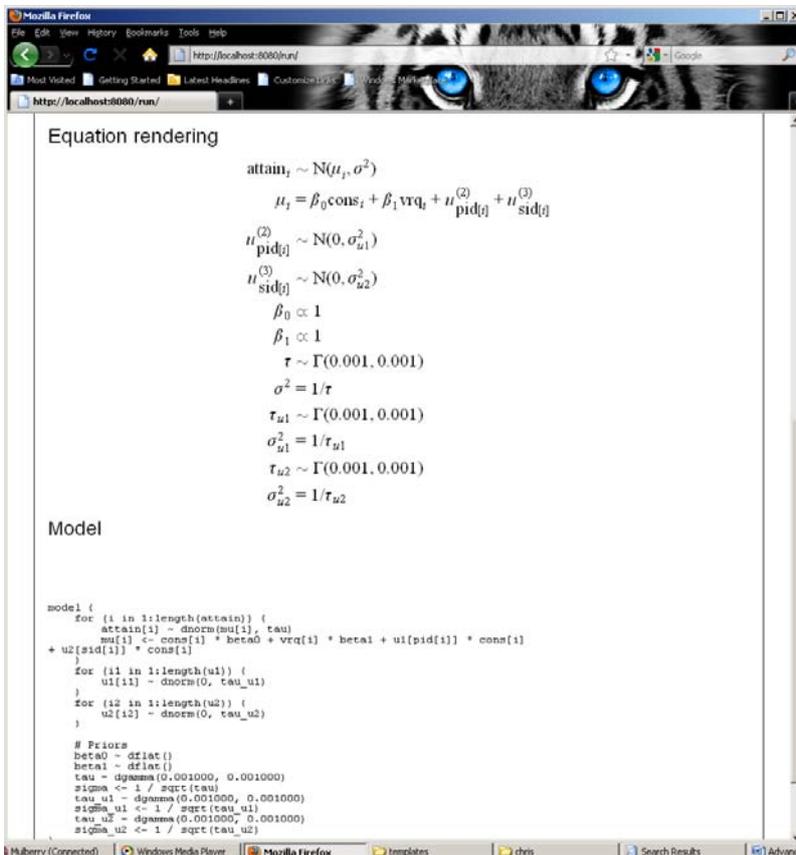
```
invars = '''
NumLevs = Integer('Number of Classifications: ')
for i in range(0, int(NumLevs)):
    context['u' + str(i + 1)] = ParamVector()
    context['tau_u' + str(i + 1)] = ParamScalar()
    context['sigma_u' + str(i + 1)] = ParamScalar()
    selstr = 'Classification ' + str(i + 1) + ': '
    context['C' + str(i + 1)] = DataVector(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':
    tau = ParamScalar()
    sigma = ParamScalar()
x = DataMatrix('explanatory variables: ')
beta = ParamVector()
beta.ncols = len(x)
'''
```

We have needed to replace the code for inputting the level 2 identifier with code to input the number of classifications (levels of clustering). We have then looped over the number of classifications constructing both the names of the columns that contain the classification vectors (which are labelled *C1*, *C2* ...) and the new parameters associated with each classification (*u1*, *tau\_u1* and *sigma\_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 text string for the input 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 one classification as we are not considering level 1 here.

We now demonstrate use of this template for a cross-classified example with two higher classifications. Using a dataset from Fife in Scotland we look at the impact of both primary school and secondary school on the attainment of children at age 16. 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:



We can see that the model code is similar to that for the 2 level model in *2LevelMod*. Again we have a *preparedata* function which transfers data to the template but also calculates the lengths of the various sets of random effects:

```
def preparedata(self, data):
    self.data = data
    numlevs = int(self.objects['NumLevs'])

    for i in range(0, numlevs):
        self.objects['u' + str(i + 1)].ncols = -
1*len(numpy.unique(self.data[self.objects['C' + str(i + 1)]]))

    return self.data
```

The model code is created by the *outbug* attribute with the following code:

```
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
        ${mmult(x, 'beta', 'i')} \
        % for i in range(0, numlevs):
+ u${i + 1}[$${context['C' + str(i + 1)}][i]] * cons[i]
        % endfor
    }
    % for 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

# 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)
% endif
% for i in range(0, numlevs):
```

```

    tau_u${i + 1} ~ dgamma(0.001000, 0.001000)
    sigma_u${i + 1} <- 1 / sqrt(tau_u${i + 1})
  % endfor
}
'''

```

Here we introduce the use of local variable *numlevs*. The *outbug* attribute is a text string with substitutions. If we wish to include a Python statement 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 the upper bound in loops 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 a mixture of WinBUGS like model code and Python code. For example, consider the following code for our cross-classified model:

```

% for 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

```

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 expanding the outside `%for` (Python) gives:

```

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 :

## Results

```

sigma_u2: {'chain': '0.1194 (0.06839) ESS: 227'}
sigma_u1: {'chain': '0.5297 (0.05856) ESS: 847'}
tau: {'chain': '0.235 (0.005764) ESS: 4434'}
u1: {'chain': '
0.08162 (0.2647) ESS: 3523
-0.0314 (0.4495) ESS: 4729
0.2171 (0.485) ESS: 4880
0.1411 (0.435) ESS: 4760
...
0.07879 (0.5157) ESS: 5203'}
tau_u1: {'chain': '3.698 (0.841) ESS: 829'}
tau_u2: {'chain': '220.6 (366.8) ESS: 281'}
u2: {'chain': '
0.0405 (0.1133) ESS: 1699
0.007032 (0.1079) ESS: 2274
...
-0.03756 (0.1093) ESS: 1549
-0.1284 (0.1489) ESS: 452'}

```

```
beta1: {'chain': '0.1596 (0.00279) ESS: 52'}
beta0: {'chain': '-9.979 (0.2814) ESS: 52'}
sigma: {'chain': '2.064 (0.02532) ESS: 4429'}
```

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 (*NLevelwithRS*) which allows random slopes in chapter 12. 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 the *NLevelMod* template to allow categorical predictors as in *1LevelCat*. You could also try adding the option to include interactions. Remember to save your template under a new name.

## **11. Using the Preccode/PreJcode methods**

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 chapter 6 details of how the system can interact with third-party software. In this chapter (and in fact the following three chapters) we will see how through the inclusion of additional C++ and/or Javascript code the user can increase the set of models and algorithms 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 eventually be rectified. As the names suggest the *preccode* and *preJcode* functions will involve writing C++ or Javascript code and so some knowledge of these languages would be useful. The examples given here will, however, allow the user with some modification to use similar code for their examples. We begin in this chapter with a simple example of a 1 level probit regression model.

### **11.1 The ProbitRegression 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).

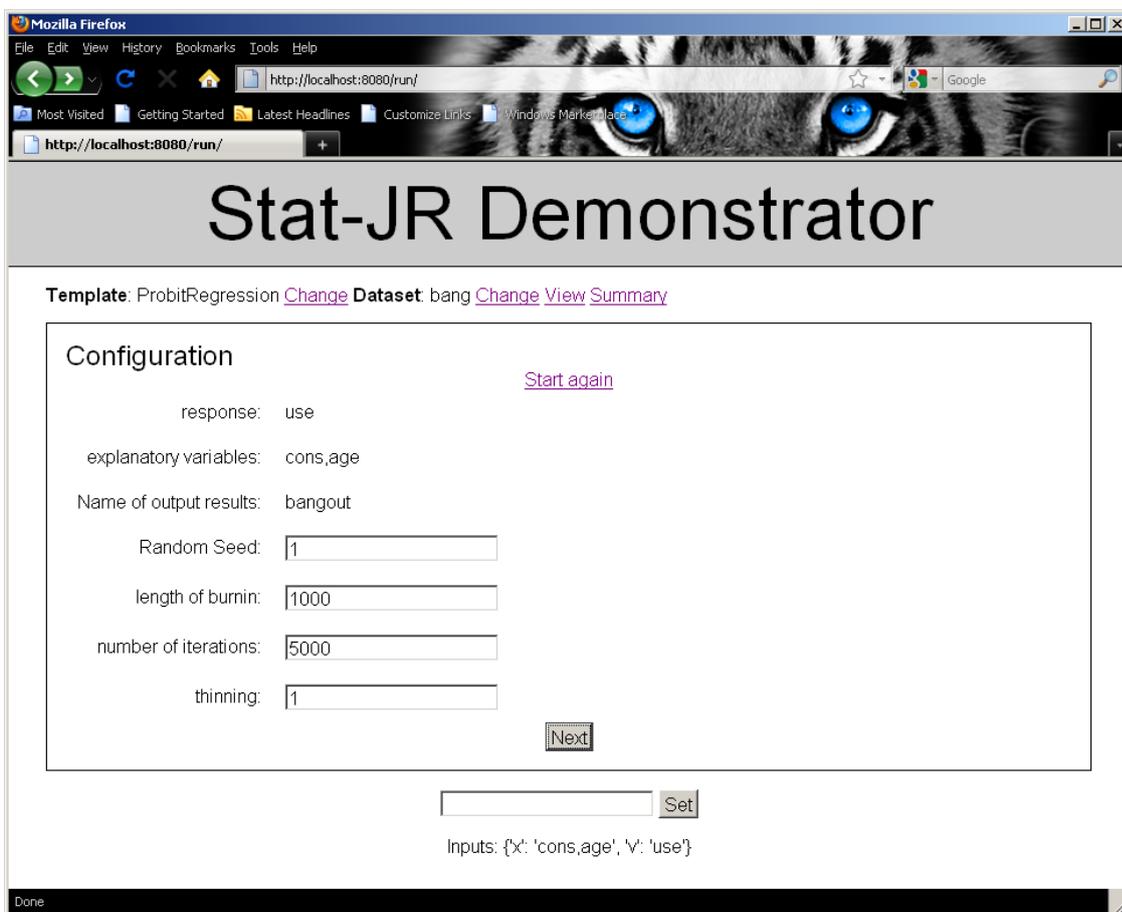
We can use this latent variable representation of the probit model in MCMC estimation by generating the latent continuous response variable as part of the algorithm. Having generated the latent variable, we then have a normal response model for these variables which is easy to fit. The

*ProbitRegression* template fits a probit regression using this technique and we will add the step to update the continuous response variables via the *preccode/prejcode* methods. It should be noted that the logistic regression can also use a latent variable representation. In this the latent variables follow a standard logistic distribution though this distribution is not as easy to deal with as the normal.

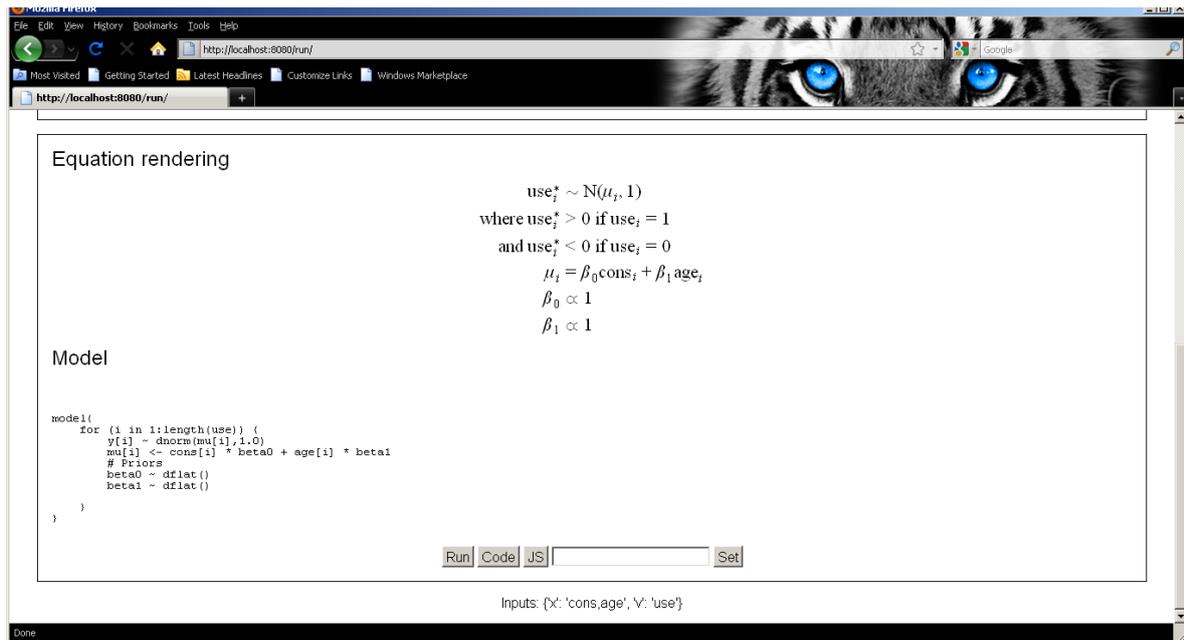
We will start as usual by looking at the *invars* attribute which is quite short:

```
invars = '''
y = ParamVector()
v = DataVector('response: ')
x = DataMatrix('explanatory variables: ')
beta = ParamVector()
beta.ncols = len(x)
'''
```

You will see that the column containing the 0/1 response is actually stored as *v* in this template as we will derive *y* as the underlying continuous response. As always it helps to demonstrate the template with an example so we will fit a probit regression model chapter to the Bangladeshi dataset (i.e. replace the logit link of chapter 7 with the probit link). Select *ProbitRegression* from the *template* list and *bang* from the *dataset* list and then fill in the template as shown below:



Clicking on the **Next** button we see the model described mathematically:



The latent continuous response variable, written as  $y$  in the model code, is denoted by  $use^*$  in the model equation. 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.

```
outbug = '''
model{
  for (i in 1:length(${v})) {
    y[i] ~ dnorm(mu[i], 1.0)
    mu[i] <- ${mmult(x, 'beta', 'i')}
    # Priors
    % for i in range(0, x.ncols()):
    beta${i} ~ dflat()
    % endfor
  }
}
'''
```

The code is fairly straightforward so the interesting part is the step generating  $y$  to make this the correct model. We have a *preparedata* function:

```
def preparedata(self, data):
    self.data = data
    self.objects['y'].ncols = -1*len(data[self.objects['v'].name])
    return data
```

which ensures that the continuous response vector  $y$  is the same length as the observed binary response vector  $v$ . We will now look at the three methods in this template that we haven't looked at before.

## 11.2 monitor\_list method

In all the templates we have looked at so far all parameters in the model have had their whole chains stored. This can mean that the program uses a large amount of memory if the number of

parameters is large, and in this template we have a parameter for each data point (the latent continuous response). We would therefore like the option to not store every value. This is done via the *monitor\_list* method which has the following code:

```
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
```

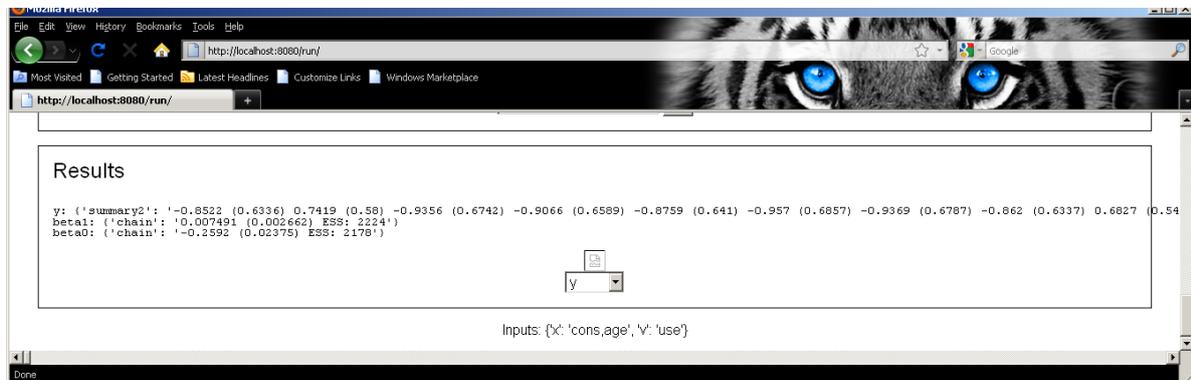
Here we have a method that goes through all the parameters in the model and as long as they are not equal to 'y' they are appended to the monitor list. To see how this feeds through to the system in *webtest* you will recall that the *go* function is the main driver and is called thus:

```
m = go(t, context['variables'], t.EstObjects['burnin'],
t.EstObjects['iterations'], thinning, seed, t.monitor_list, callback)
```

You will see that the *monitor\_list* method is called as one of the arguments to the *go* function. We then see within the *go* function that this argument (which has the name *mon* within the function) is used as follows:

```
for p in mon():
    m.params[p].monitor = True
```

and so only those parameters in the list are monitored. The *run* function will then treat parameters differently in terms of their monitor status. To see this in practice we will continue our example and press the **Run** button. The results will look as follows:



For *y* we only store summary statistics for each element of the *y* vector and hence just output a posterior mean and standard deviation for each element. This is also why no graphs appear for *y* below the results.

### 11.3 preccode method

We have seen that the code works but we now need to look at how the step for updating the latent *y* variable is incorporated into the code. This is done via the *preccode* function which for this template is as follows:

```

def preccode(self):
    extracode = '''
<%!
    def mmult(names, var, index):
        out = ""
        count = 0
        for name in names:
            if count > 0:
                out += ' + '
            out += 'double(' + name + '[' + index + ']) * ' + var +
str(count)
                count += 1
        return out
%>
double mean;
for(int i=0;i<length(y);i++)
{
    mean = ${mmult(x.name, 'beta', 'i')};
    if(${v.name}[i] <= 0)
        y[i] = rtnormal(mean,1,2,0,0);
    else
        y[i] = rtnormal(mean,1,1,0,0);
}
'''
    return MakoTemplate(extracode).render(**self.objects)

```

This function could have been even shorter except we need to include in here a definition for the mmult function that is used to construct the linear predictor, this time as C++ code.

The actual C++ code is:

```

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

```

Here we see that the code involves looping over all data points using a for loop and, for each point, we evaluate 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 rtnormal function. Here rtnormal takes 5 arguments: the mean, the standard deviation, the type of truncation (1 = left, 2 = right, 3 = both), and finally the left and right truncation values.

To see this in action we will use the **Code** button so click on **Start Again** and fill in the inputs for the model as shown earlier in the chapter. Then press the **Code** button to see the generated code. The generated C++ code is long and so again we will just display the iteration loop:

```

void iterate() {

```

```

double mean;
for(int i=0;i<length(y);i++)
{
    mean = double(cons[i]) * beta0 + double(age[i]) * beta1;
    if(use[i] <= 0)
        y[i] = rtnormal(mean,1,2,0,0);
    else
        y[i] = rtnormal(mean,1,1,0,0);
}
// Update y
// Update beta1
{
    double sum0=0;
    for(int i=0; i<length(use); i++) {
        sum0+=(age[int(i)]*(y[int(i)]-(beta0*cons[int(i)])));
    }
    double sum1=0;
    for(int i=0; i<length(use); i++) {
        sum1+=pow(age[int(i)],2);
    }
    std::tr1::normal_distribution<double> normal((sum0/sum1),
1/sqrt(sum1));
    beta1 = normal(eng);
}
// Update beta0
{
    double sum0=0;
    for(int i=0; i<length(use); i++) {
        sum0+=(cons[int(i)]*(y[int(i)]-(beta1*age[int(i)])));
    }
    double sum1=0;
    for(int i=0; i<length(use); i++) {
        sum1+=pow(cons[int(i)],2);
    }
    std::tr1::normal_distribution<double> normal((sum0/sum1),
1/sqrt(sum1));
    beta0 = normal(eng);
}
}
}

```

Here we see that the **precode** part, as the name suggests, appears at the start of the iteration loop. This is important as the *y* variable needs initialising before the other parameters are updated, and updating it first ensures *y* is positive when *use* is 1 and negative when *use* is 0.

## 11.4 precode method

In section 5.4 we discussed the alternative code generation of JavaScript that can be run in the local browser. If we wish our template to work with JavaScript as well we need to create an equivalent *precode* method which for this template is as follows:

```

def prejcode(self):
    extracode = ''
<%!
    def mmult(names, var, index):
        out = ""
        count = 0
        for name in names:

```

```

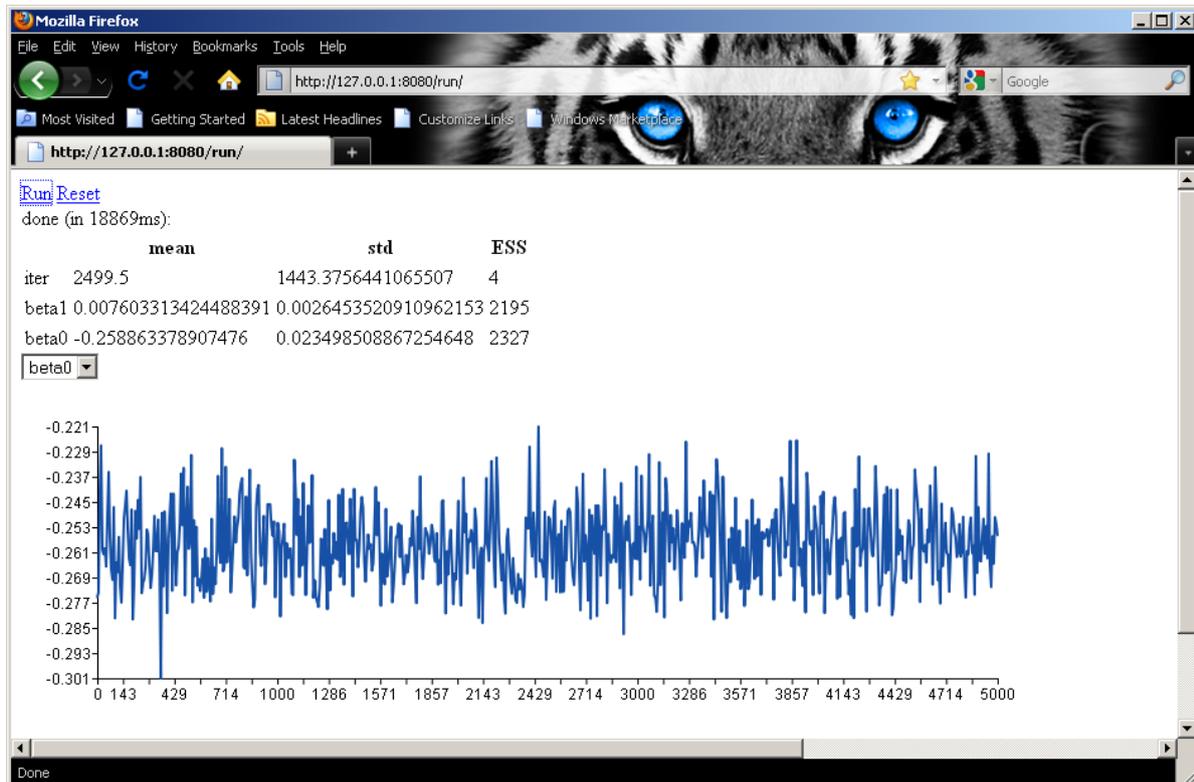
        if count > 0:
            out += ' + '
        out += name + '[' + index + '] * ' + var + str(count)
        count += 1
    return out

%>
for(var i=0;i<length(y);i++)
{
    var mean = ${mmult(x.name, 'beta', 'i')};
    if(${v.name}[i] <= 0)
        y[i] = rtnormal(mean,1,2,0,0);
    else
        y[i] = rtnormal(mean,1,1,0,0);
}
'''
return MakoTemplate(extracode).render(**self.objects)

```

In fact Javascript and C++ code are pretty similar and so you will see only a few differences here, for example the use of var instead of int or double when defining types of variable.

If you click on the JS button and then the **Run** button this code will be incorporated into the Javascript that is run and the output is as follows:



If you are interested in looking at the code you can select view page source and the code will be displayed. We will not show the code here but you will see the similarities with the C++ code used in the main engine.

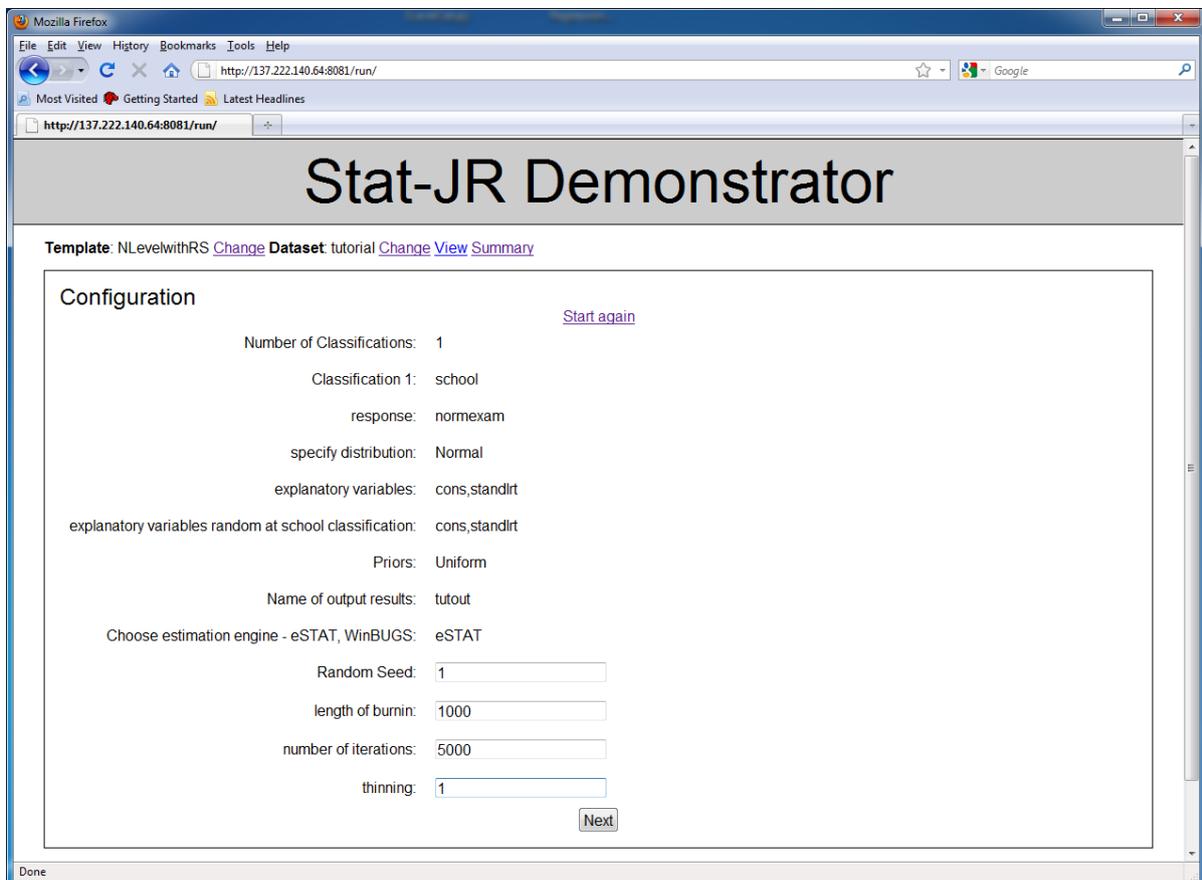
As a more complicated example of using the *preccode* function we also have a template for ordered probit regression models (*OrderedProbitRegression.py*). We do not have space in this guide to go into details for this template but such models also include multiple threshold parameters (rather than a single intercept term as in the binary case).

## 12. Multilevel models with Random slopes and the inclusion of Wishart priors

One limitation of the algebra system in its current form is that it treats all parameters as scalars. This means, for example, that for the *Regression1* template where we started the set of beta parameters are all updated individually through univariate normal steps. We will investigate the implications of this in chapter 13. In chapter 10 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 rather than scalar parameters and so our model code diverges from standard WinBUGS model code here. 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 *NLevelwithRS*.

### 12.1 An example with random slopes

Select *NLevelwithRS* from the *template* list and *tutorial* from the *data* list. Then choose the inputs as follows:



The screenshot shows a Mozilla Firefox browser window displaying the Stat-JR Demonstrator web interface. The browser's address bar shows the URL `http://137.222.140.64:8081/run/`. The page title is "Stat-JR Demonstrator". Below the title, there is a navigation bar with links for "Template: NLevelwithRS", "Change Dataset: tutorial", "Change View", and "Summary". The main content area is titled "Configuration" and contains the following settings:

- Number of Classifications: 1
- Classification 1: school
- response: normexam
- specify distribution: Normal
- explanatory variables: cons,standlrt
- explanatory variables random at school classification: cons,standlrt
- Priors: Uniform
- Name of output results: tutout
- Choose estimation engine - eSTAT, WinBUGS: eSTAT
- Random Seed: 1
- length of burnin: 1000
- number of iterations: 5000
- thinning: 1

A "Next" button is located at the bottom right of the configuration area. A "Start again" link is also present in the top right of the configuration section.

The *invars* function for this template is quite long as we see below:

```

    invars = ''
NumLevs = Integer('Number of Classifications: ')
for i in range(0, int(NumLevs)):
    selstr = 'Classification ' + str(i + 1) + ': '
    context['C' + str(i + 1)] = DataVector(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':
    tau = ParamScalar()
    sigma = ParamScalar()
x = DataMatrix('explanatory variables: ')
beta = ParamVector()
beta.ncols = len(x)

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()
    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)] = ParamVector()
        context['omega_u'+str(i+1)].ncols = -((num+1)*num)/2
        context['d_u'+str(i+1)] = ParamVector()
        context['d_u'+str(i+1)].ncols = -((num+1)*num)/2
        context['priors' + str(i)] = Text('Priors: ', ['Uniform',
'Wishart'])
        if context['priors' + str(i)] == 'Wishart':
            context['R' + str(i)] = Text('R matrix: ')
            context['v' + str(i)] = Integer('Degrees of Freedom:')
...

```

The template begins 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 make extensive use of the *context* function to construct the variable names and that there are different parameters for classifications according to the number of random effects. 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 for a classification. Finally in this case there are two possible priors and for the

(informative) Wishart priors a prior variance matrix estimate (beginning with R) and degrees of freedom (beginning with v) parameter are required.

Having completed our inputs we now need to click on Run to see what the model code looks like:

The screenshot shows a web browser window with a Jupyter Notebook interface. The notebook content is as follows:

$$\text{normexam}_i \sim N(\mu_i, \sigma^2)$$

$$\mu_i = \beta_0 \text{cons}_i + \beta_1 \text{standlrt}_i + u_{0,\text{school}[i]}^{(2)} \text{cons}_i + u_{1,\text{school}[i]}^{(2)} \text{standlrt}_i$$

$$\begin{pmatrix} u_{0,\text{school}(i)}^{(2)} \\ u_{1,\text{school}(i)}^{(2)} \end{pmatrix} \sim N \left[ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \Omega_u^{(2)} \right]$$

$$\Omega_u^{(2)} \propto 1$$

$$\beta_0 \propto 1$$

$$\beta_1 \propto 1$$

$$\tau \sim \Gamma(0.001, 0.001)$$

$$\sigma^2 = 1/\tau$$

**Model**

```

model {
  for (i in 1:length(normexam)) {
    normexam[i] ~ dnorm(mu[i], tau)
    mu[i] <- cons[i] * beta0 + standlrt[i] * beta1 + u0_0[school[i]] * cons[i]
    + u1_0[school[i]] * standlrt[i]
  }

  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()
  }

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

```

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:

```

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()
}

```

The `dnormal2a` distribution has as its first two arguments the two responses. Next we get the two means and then the three parameters that make up the precision matrix. As the algebra system expects all parameters to appear on the lefthand side we complete our workaround for a multivariate Normal distribution by including the two `dflat` statements; these 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 handside. The definition of `dnormal2a` does not depend on the LHS and as they do not appear anywhere on the RHS the algebra system does not recognise them as parameters and so does not attempt to find their posteriors.

The code for creating the model code is in *outbug* but does not contain anything very new that needs reporting here. The *outlatex* code might interest those trying to learn LaTeX as it contains a section to produce the multivariate Normal line as follows:

```

\left(
\begin{array}{l}
% for i in range(0, len(context['x'+str(lev+1)])):
u^{\{(\text{\lev + 2})\}}_{\{i\}},\{\text{context['C' + str(lev + 1)]}\}(i)}
% if i != len(context['x'+str(lev+1)]) -1:
\\
% endif
% endfor
\end{array}
\right) & \sim \mbox{N}
\left[ \left(
\begin{array}{l}
% for i in range(0, len(context['x'+str(lev+1)])):
0
% if i != len(context['x'+str(lev+1)]) -1:
\\
% endif
% endfor
\end{array}
\right), \Omega^{\{(\text{\lev + 2})\}}_{\{u\}} \right] \\
% endif
% endfor

```

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 viewed by right-clicking on the LaTeX and selecting show source. It looks as follows:

```

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

```

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

## 12.2 Precode for NlevwithRS

We will look at the *preccode* in sections. The *preccode* is 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:

```

        extracode = '''
            {
<% numlevs = int(NumLevs) %>\\
                static int itnum = -1;
                itnum += 1;
% for i in range(0, numlevs):
<% n = len(context['x'+str(i + 1)]) %>\\
% if n > 1 :
                // Note currently using a uniform prior for variance matrix
                struct _gmatrix *sb${i + 1} = matrix_blank(${n},${n});
                for(int i = 0; i < length(u0_${i}); i++) {
% for j in range(0, n):
% for k in range(0, n):
                    sb${i + 1}->e[${j}][${k}] += double(u${j}_${i}[i]) *
double(u${k}_${i}[i]);
% endfor
% endfor
                }

```

This first section stores the number of levels (*numlevs*) for looping purposes and, within the loop, the number of random effects is constructed (as *n*) because for classifications with only one set of random effects no further action is required because the algebra system has already evaluated the correct posterior. We next construct a matrix variable *sb1* which initially stores the crossproduct matrix of the residuals before moving to the next chunk of code:

```

% if context['priors' + str(i)] == 'Uniform':
        int vw${i+1} = length(u0_${i})-${n + 1};
        if (itnum == 0) {
% for j in range(0, n):
            sb${i + 1}->e[${j}][${j}] += 0.0001;
% endfor
        }
% endif
% if context['priors' + str(i)] == 'Wishart':
        int vw${i+1} = length(u0_${i}) + ${context['v' + str(i)]};
% endif

% if context['priors' + str(i)] == 'Wishart':
<%
import numpy
Rmat = numpy.empty([n, n])
count = 0
for j in range(0, n):
    for k in range(0, j + 1):
        Rmat[j, k] = float(context['R' + str(i)].name[count])
        Rmat[k, j] = Rmat[j, k]
        count += 1
%>

<% count = 0 %>
% for j in range(0, n):
% for k in range(0, n):
        sb${i+1}->e[${j}][${k}] += ${str(Rmat[j][k] *
float(context['v' + str(i)]))};

```

```

% endfor
% endfor
% endif

```

In this chunk of code we have different blocks depending on the type of prior distribution. 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 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:

```

        matrix_sym_invinplace(sb${i+1});
        struct _gmatrix *va${i+1} = rwishart(vw${i+1},sb${i+1});
        matrix_free(sb${i+1});
<% count = 0 %>\\
% for j in range(0, n):
% for k in range(j, n):
        d_u${i+1}[$count] = va${i+1}->e[$j][$k];
<% count += 1 %>\\
% endfor
% endfor

        matrix_sym_invinplace(va${i+1});
<% count = 0 %>\\
% for j in range(0, n):
% for k in range(j, n):
        omega_u${i + 1}[$count] = va${i+1}->e[$j][$k];
<% count += 1 %>\\
% endfor
% endfor

        matrix_free(va${i+1});

%endif
%endifor
    }
    ...
return MakoTemplate(extracode).render(**self.objects)

```

In this last chunk of code we invert the *sb1* parameter before drawing the new precision matrix which we give the place holder *va1*. The last chunk then involves copying these values to the vector *d\_u1* and the inverse matrix to the vector *omega\_u1*. To see the code that the *preccode* method generates for our example we can click on the **code** button and search for the *iterate* function. This can be seen here: {

```

static int itnum = -1;
itnum += 1;
// Note currently using a uniform prior for variance matrix
struct _gmatrix *sb1 = matrix_blank(2,2);
for(int i = 0; i < length(u0_0); i++) {
    sb1->e[0][0] += double(u0_0[i]) * double(u0_0[i]);
    sb1->e[0][1] += double(u0_0[i]) * double(u1_0[i]);
    sb1->e[1][0] += double(u1_0[i]) * double(u0_0[i]);
    sb1->e[1][1] += double(u1_0[i]) * double(u1_0[i]);
}
int vw1 = length(u0_0)-3;
if (itnum == 0) {
    sb1->e[0][0] += 0.0001;
    sb1->e[1][1] += 0.0001;

```

```

    }
    matrix_sym_invinplace(sbl);
    struct _gmatrix *val = rwishart(vw1,sbl);
    matrix_free(sbl);
    d_ul[0] = val->e[0][0];
    d_ul[1] = val->e[0][1];
    d_ul[2] = val->e[1][1];
    matrix_sym_invinplace(val);
    omega_ul[0] = val->e[0][0];
    omega_ul[1] = val->e[0][1];
    omega_ul[2] = val->e[1][1];
    matrix_free(val);
}

```

We can finally run the template by clicking the back arrow on the browser and then clicking on the **Run** button. The results appear as follows:

### Results

```

tau: {'chain': '1.806 (0.04131) ESS: 5138'}
u0_0: {'chain': '
0.3747 (0.09285) ESS: 1532
0.4694 (0.1057) ESS: 1723
...

-0.1498 (0.09254) ESS: 1364'}
u1_0: {'chain': '
0.1274 (0.06977) ESS: 3378
0.1697 (0.07278) ESS: 2869
...

-0.0006234 (0.06879) ESS: 3210'}
omega_ul: {'chain': '
0.1036 (0.02194) ESS: 2737
0.02069 (0.008522) ESS: 1741
0.0183 (0.005671) ESS: 1214'}
d_ul: {'chain': '
13.81 (3.635) ESS: 1524
-16.56 (8.738) ESS: 965
82.24 (29.96) ESS: 838'}
beta1: {'chain': '0.5565 (0.02155) ESS: 603'}
beta0: {'chain': '-0.01124 (0.04278) ESS: 209'}
sigma: {'chain': '0.7443 (0.00852) ESS: 5141'}

```

and as usual we get MCMC output. There is again a similar *prejcode* function that will allow you to run this model in JavaScript. It is very similar to the *preccode* function so we do not go into details here.

### Exercise 9

We have not written a *2LevelwithRS* template so try adapting the *NLevelwithRS* template so that it only allows one higher classification. This exercise will be in essence a merging of features of two templates, *2levelmod* and *NlevelwithRS*, and will test your understanding of the various chunks of code.

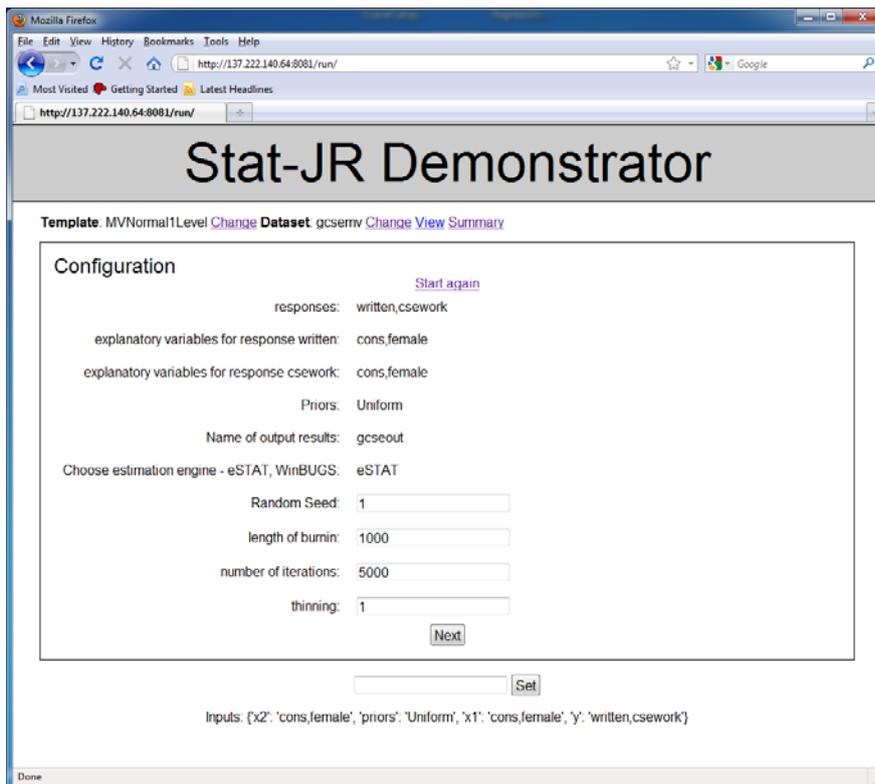
## 12.3 Multivariate Normal response models

Having established a method of including multivariate distributions for use with random slopes in the precode 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, *MVNormal1Level*. 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 precode will generate two steps: one for the variance matrix of the responses and an initial step to set up the missing responses. The invars function is as follows:

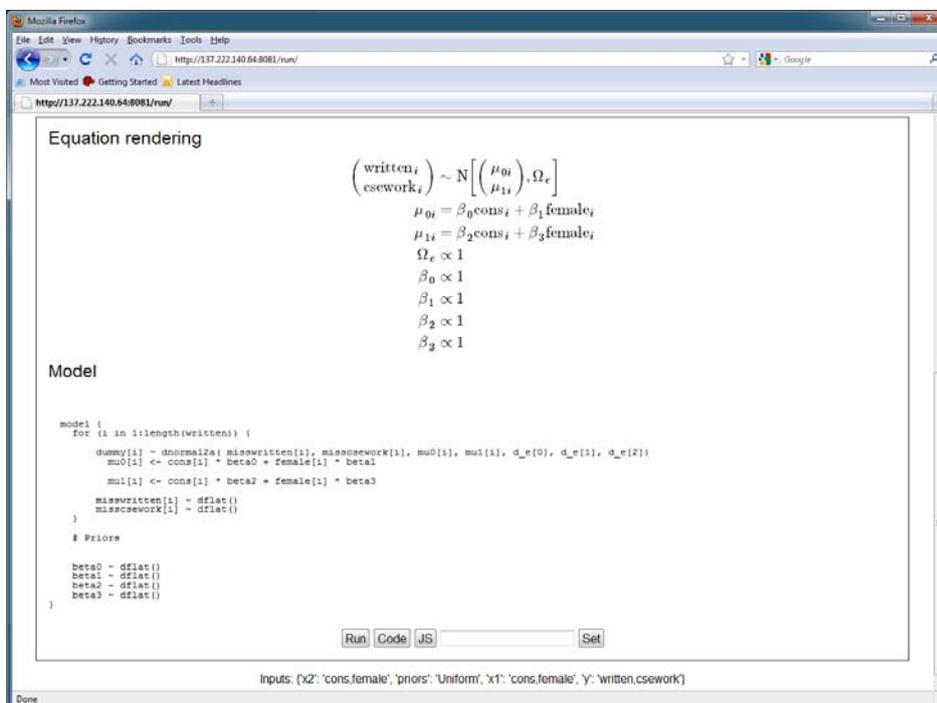
```
    invars = '''
y = DataMatrix('responses: ')
lenbeta = 0
for i in range(0, len(y)):
    context['x'+str(i+1)] = DataMatrix('explanatory variables for response
' + y[i] + ': ')
    lenbeta += len(context['x'+str(i+1)])
    context['miss'+y[i]] = ParamVector()
n = len(y)
if n == 1:
    tau = ParamScalar()
    sigma = ParamScalar()
else:
    omega_e = ParamVector()
    omega_e.ncols = -((n+1)*n)/2
    d_e = ParamVector()
    d_e.ncols = -((n+1)*n)/2
    priors = Text('Priors: ', ['Uniform', 'Wishart'])
    if priors == 'Wishart':
        R = Text('R matrix: ')
        v = Integer('Degrees of Freedom:')
beta = ParamVector()
beta.ncols = lenbeta

'''
```

You will notice that we construct parameter vectors that are a combination of the string 'miss' and y variable names input and these will be used in the model. Let us run the *MVNormal1Level* template with the *gcsemv* dataset that contains two responses for secondary school pupils, a written and a coursework test score. We will set up the inputs as follows:



We allow the two responses to both depend on one predictor, a dummy for gender (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 *precode* function. The model output is as follows:



We see again the use of the *dnormal2a* function and also that we have included *dflat* statements for both the *misswritten* and *misscsework* responses to let the algebra system know that these are

parameters. We will not look in detail at the *outbug* method as we can see the output it produces on the screen.

There is a *monitor\_list* method to avoid storing chains for the missing data:

```
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 *preparedata* method that is used to set the length of the missing data vector to equal the original response vector:

```
def preparedata(self, data):
    self.data = data
    for i in range(0, len(self.objects['y'])):
        self.objects['miss'+self.objects['y'][i]].ncols = -
1*len(data[self.objects['y'][i]])
    return self.data
```

We next turn our attention to the *precode* function.

### 12.3.1 The precode function

We will deal with the code of the precode function in sections. 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 for each response:

```
extracode = '''
<%!
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']) %>\\
```

We next have some code that is run in the first iteration which sets any non-missing responses in the parameter vector that will be updated to their value in the actual data. These will then not be updated in the main code:

```

        static int itnum = -1;
        itnum += 1;

        if (itnum == 0){
            for(int i = 0; i < length(miss${y[0]}); i++) {
%for var in range(0,n):
                if(${y[var]}[i] > (-9.999e29))
                    miss${y[var]}[i] = ${y[var]}[i];

%endfor
            }
        }

```

Next we have code 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  $u$ ); this crossproduct needs to be constructed which is done in the initial code using the `mmult2` function:

```

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

% endfor
% endfor
        }

```

Once constructed the remaining code follows the same pattern as for random slopes and so for brevity we omit this code here (it can be viewed in *MVNormal1Level.py*).

We have not yet mentioned how the missing data are updated, and this is currently done in a slightly undesirable way. Buried deep in the `toC` functions of `XMLtoC`, `XMLtoPureC` and `XMLtoJS` are two little bits of code thus:

```

% 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 conditional statements around the the update step for that parameter. This reliance on the parameter name is undesirable and we will hopefully have a better method for making such algorithmic changes in later releases. It is helpful at this point to look at the code generated for this example. To do this click on the **code** button and scroll down to the *iterate* method. Here we see the step for the variance matrix *omega\_e* at the top:

```
{
    static int itnum = -1;
    itnum += 1;

    if (itnum == 0){
        for(int i = 0; i < length(misswritten); i++) {
            if(written[i] > (-9.999e29))
                misswritten[i] = written[i];
            if(csework[i] > (-9.999e29))
                misscsework[i] = csework[i];
        }
        // Note currently using a uniform prior for variance matrix
        struct _gmatrix *sb = matrix_blank(2, 2);
        for(int i = 0; i < length(misswritten); 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->e[0][0] += e0 * e0;
            sb->e[0][1] += e0 * e1;
            sb->e[1][0] += e1 * e0;
            sb->e[1][1] += e1 * e1;
        }
        if (itnum == 0) {
            sb->e[0][0] += 0.0001;
            sb->e[1][1] += 0.0001;
        }
        matrix_sym_invinplace(sb);
        int vw = length(misswritten) - 3;
        struct _gmatrix *va = rwishart(vw, sb);
        matrix_free(sb);
        d_e[0] = va->e[0][0];
        d_e[1] = va->e[0][1];
        d_e[2] = va->e[1][1];
        matrix_sym_invinplace(va);
        omega_e[0] = va->e[0][0];
        omega_e[1] = va->e[0][1];
        omega_e[2] = va->e[1][1];
        matrix_free(va);
    }
}
```

and towards the end the steps for the missing data:

```
// Update misswritten
    for(int i=0; i<length(misswritten); i++){
        {

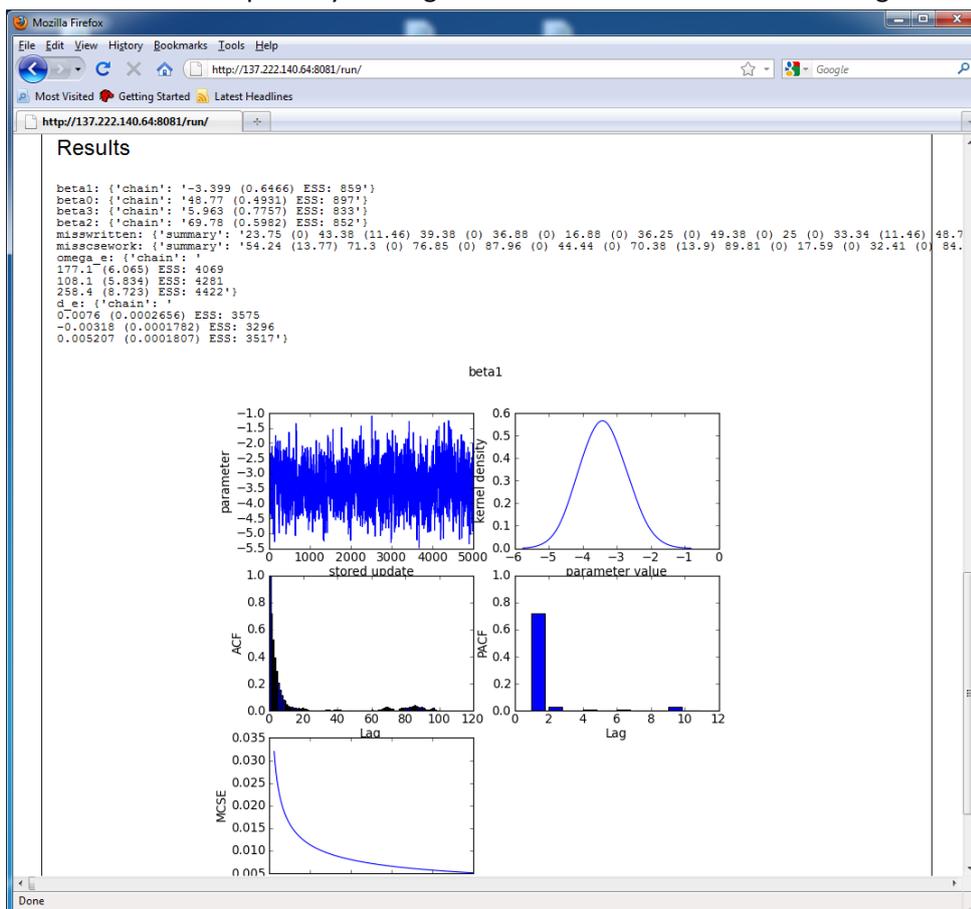
if (written[i] <= -9.999e29) {
```

```

        std::tr1::normal_distribution<double>
normal(((0.5*((d_e[int(0)]*((2*beta0*cons[int(i)])+(2*beta1*female[int(i)]))
))+((2*d_e[int(1)]*((-
misscsework[int(i)]+(beta2*cons[int(i)]+(beta3*female[int(i)])))))/d_e[in
t(0)]), 1/sqrt(d_e[int(0)]));
        misswritten[i] = normal(eng);
    }
    }
}
// Update misscsework
for(int i=0; i<length(misscsework); i++){
    {
if (csework[i] <= -9.999e29) {
        std::tr1::normal_distribution<double>
normal(((0.5*((2*d_e[int(1)]*((-
misswritten[int(i)]+(beta0*cons[int(i)]+(beta1*female[int(i)]))))+(d_e[int
(2)]*((2*beta2*cons[int(i)]+(2*beta3*female[int(i)])))))/d_e[int(2)]),
1/sqrt(d_e[int(2)]));
        misscsework[i] = normal(eng);
    }
    }
}
}
}

```

We can run the template by clicking back in the browser and then clicking on the **Run** button:



You will see that for the variables with missing data, the data items that are not missing have standard deviation zero (or very small values due to rounding) as their values should not change

from iteration to iteration; for example, we can see that the first written score and second coursework scores were observed.

We have extended these multivariate normal modelling templates to more levels and to include random slopes. They will also form the basis of the REALCOM templates we are currently writing to mimic 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, C++ and JavaScript). It is hoped that with advances in the algebra system the reliance on the *preccode/prejcode* functions will be reduced 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 open source nature of the Stat-JR system.

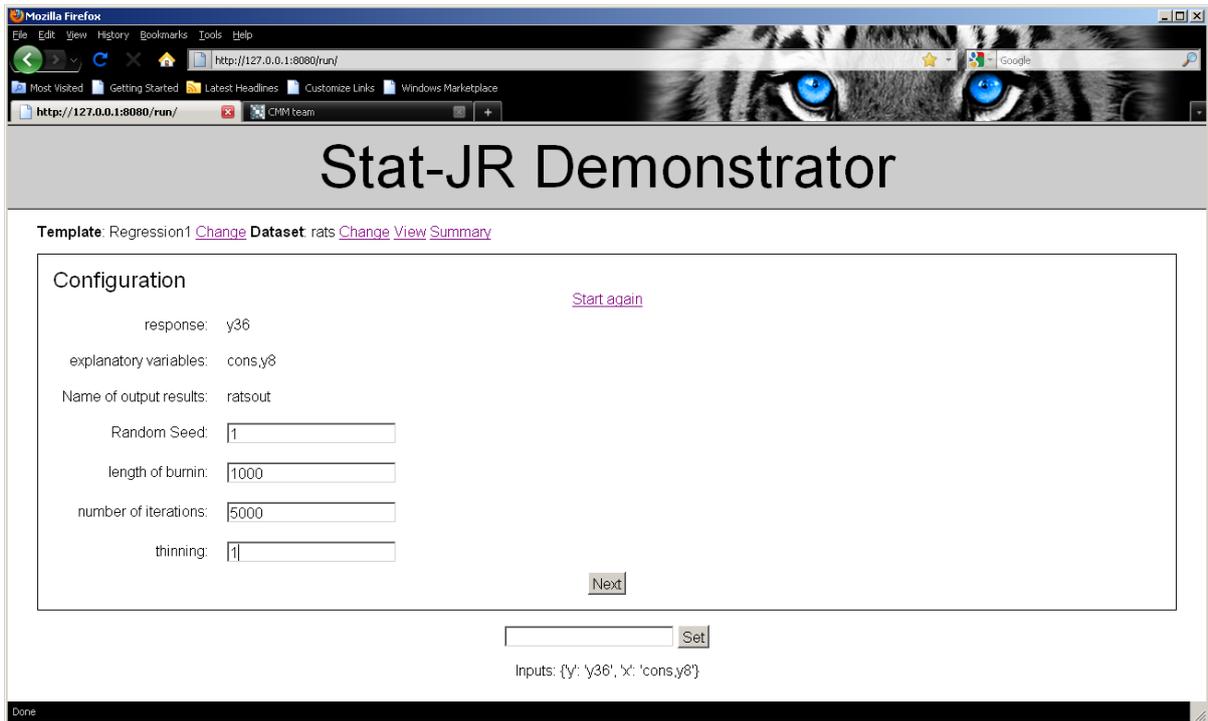
We will finish this documentation by considering two more examples of getting more from the MCMC estimation engine.

### 13. Improving mixing (1LevelBlock and 1LevelOrthogParam)

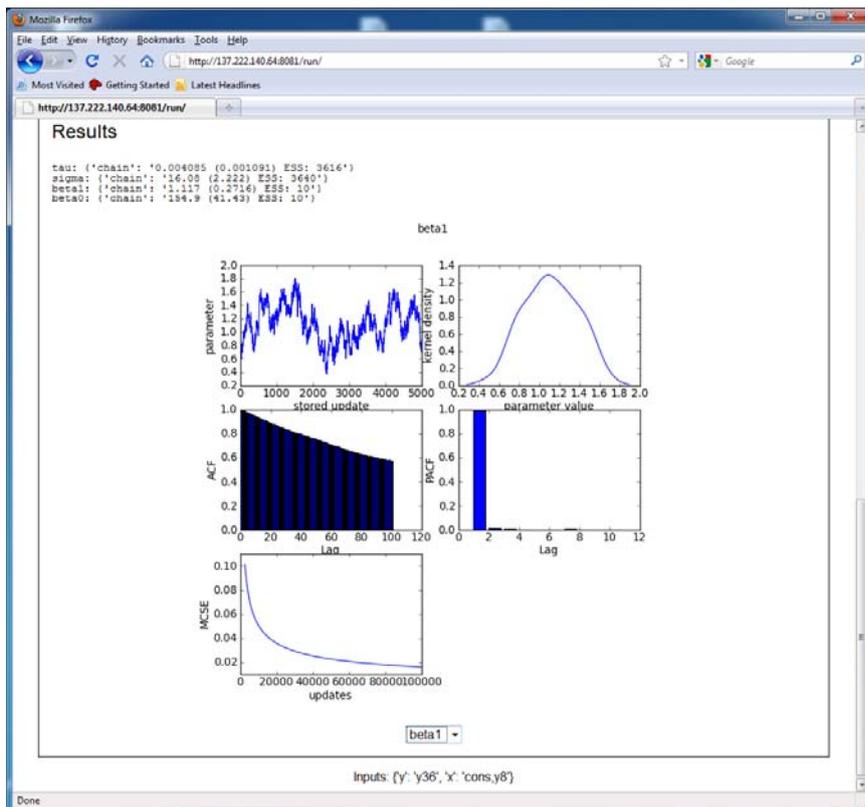
In this chapter we will return once again to our first template *Regression1* but use it on a different dataset, *rats*. This dataset contains the weights of 30 laboratory rats taken at weekly intervals from 8 days old. We will regress their final weight at 36 days on their initial weight at 8 days old.

#### 13.1 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 MCMC plot so that we have *beta1* visible we will see the following:



We see that both the regression coefficients have very small effective sample sizes and the chain we observe is not mixing well. Aside from being a small dataset, a difference between the *rats* and the *tutorial* dataset is that the variables have not been centred, unlike *standlrt* in the tutorial dataset. This means that the joint posterior distribution of  $\beta_0$  and  $\beta_1$  has a large correlation and so if we update these parameters 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 *precode*. 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* attribute requiring the user to specify whether or not to block the fixed effects. Otherwise the code is little changed from the template *1LevelMod* template apart from the additional *precode* method which we consider now.

The first chunk of code is as follows:

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

Here we see for the first time the *preccode* function doing something other than writing a text string. This code is additionally informing the code generator that custom steps (i.e. user written steps) are to be used for the beta parameters when the block updating option (*mv*) is selected. The remainder of the code updates the beta vector which has mean (in matrix form)  $(X^T X)^{-1} X^T y$  and variance  $(X^T X)^{-1}$  times the residual variance. The code is as follows:

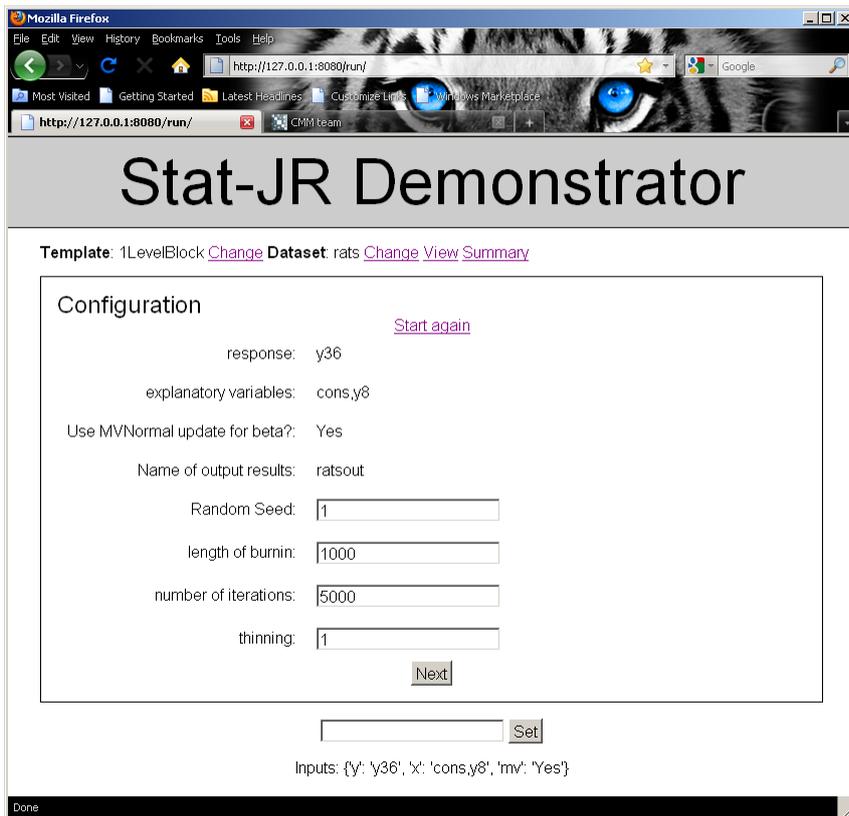
```

extracode = '''
    {
<% nbeta = len(context['x']) %> \\
        static int itnum = -1;
        itnum += 1;
        static struct _gmatrix *xtxinv = matrix_blank(${nbeta},
${nbeta});
        static struct _gmatrix *mean = matrix_blank(${nbeta}, 1);
        // Setting up constant terms for beta step
        if (itnum == 0) {
            struct _gmatrix *xty = matrix_blank(${nbeta}, 1);

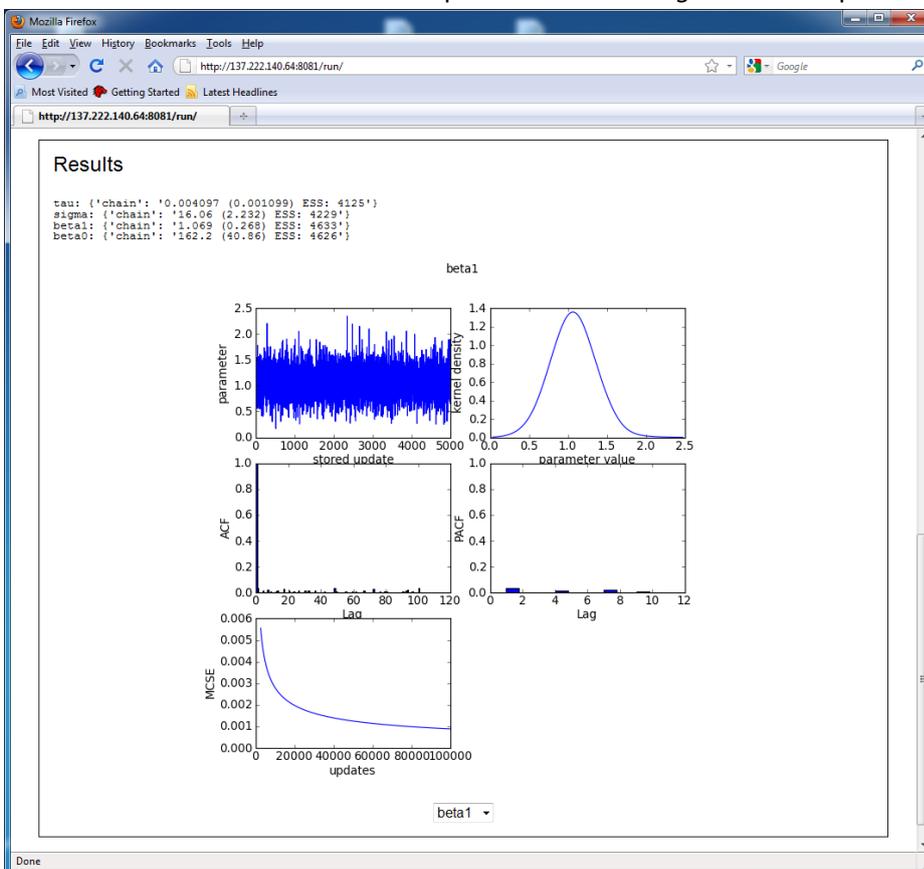
            for (int i = 0; i < length(${context['x']}[0]); i++) {
% for j in range(0, nbeta):
                xty->e[${j}][0] += double(${context['x']}[j][i]) *
double(${context['y']}[i]);
% for k in range(0, nbeta):
                xtxinv->e[${j}][${k}] +=
double(${context['x']}[j][i]) * double(${context['x']}[k][i]);
% endfor
% endfor
            }
            matrix_sym_invinplace(xtxinv);
            for (int j = 0; j < ${nbeta}; j++) {
                for (int k = 0; k < ${nbeta}; k++) {
                    mean->e[j][0] += xtxinv->e[j][k] * xty-
>e[k][0];
                }
            }
            matrix_free(xty);
        }
        // Multivariate step for beta
        struct _gmatrix *variance = matrix_blank(${nbeta},
${nbeta});
        for (int j = 0; j < ${nbeta}; j++) {
            for (int k = 0; k < ${nbeta}; k++) {
                variance->e[j][k] = xtxinv->e[j][k] / tau;
            }
        }
        struct _gmatrix *rand = rmultnormal(mean, variance);
        matrix_free(variance);
% for j in range(0, nbeta):
        beta[${j}] = rand->e[${j}][0];
        py_beta[${j}] = PyFloat_FromDouble(beta[${j}]);
        PyDict_SetItemString(raw_locals, "beta[${j}]", py_beta[${j}]);
        Py_DECREF(py_beta[${j}]);
% endfor
        matrix_free(rand);
    }
'''
return MakoTemplate(extracode).render(**self.objects)

```

Choose the *1LevelBlock* template and the *rats* dataset and set up the inputs as follows:



Running the template by pressing the **Next** and **Run** buttons results in the following output. Note that we have selected *beta1* for comparison with the *Regression1* output.



We can see that this method has given much better mixing for all parameters! 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

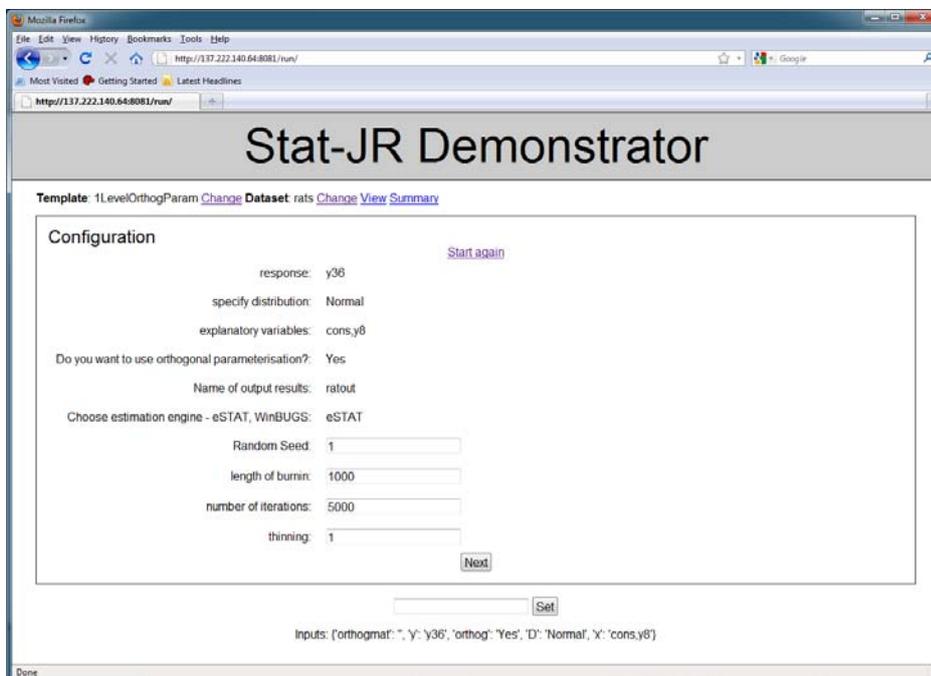
The alternative approach to blocking variables that are correlated is to reparameterise the parameters to a configuration that are less correlated. We will achieve this by using an orthogonal parameterisation for the fixed effects rather than the standard parameterisation.

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 one additional input in *invars* which is used to find out whether or not to use an orthogonal parameterisation.

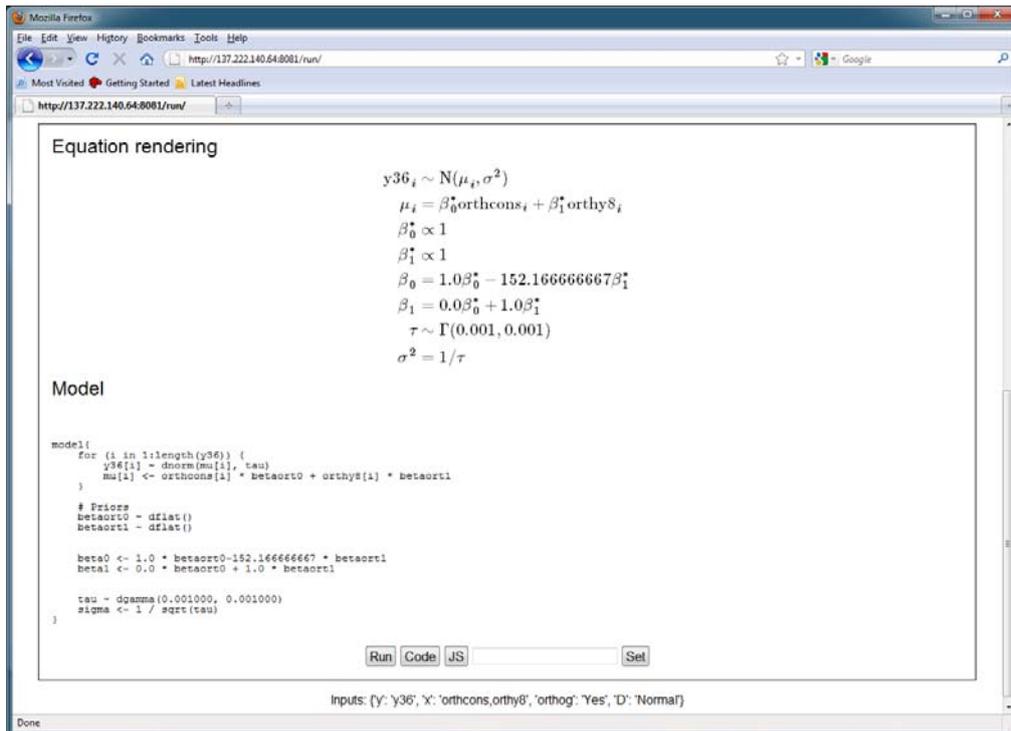
This can be seen in the following lines

```
orthog = Boolean('Do you want to use orthogonal parameterisation?: ')
if orthog:
  betaort = ParamVector()
  betaort.ncols = len(x)
  orthogmat = 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 will give the following output for the model:



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 of how we construct orthogonal vectors we refer the reader to Browne et al. (2009) but note that a function 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 following *preparedata* function.

```

def preparedata(self, data):
    self.data = data

    if self.objects['orthog']:
        self.objects['orthogmat'] = []

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

        #(tmp, om) = numpy.linalg.qr(numpy.mat(orth))
        om = orthog(orth)
        for i in om.flat:
            self.objects['orthogmat'].append(str(i))

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

        self.objects['x'][:] = map(lambda n: 'orth' + n,
self.objects['x'])

```

```
return self.data
```

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'. 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 as a prefix to the original x variables names. Finally the map function replaces the original x variable names with these new orthogonal variable names before the data are 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)
    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
%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} <- \\
  % for j in range(0, beta.ncols):
${orthogmat[count]} * betaort${j}\\
  % if j == (beta.ncols - 1):

% else:
```

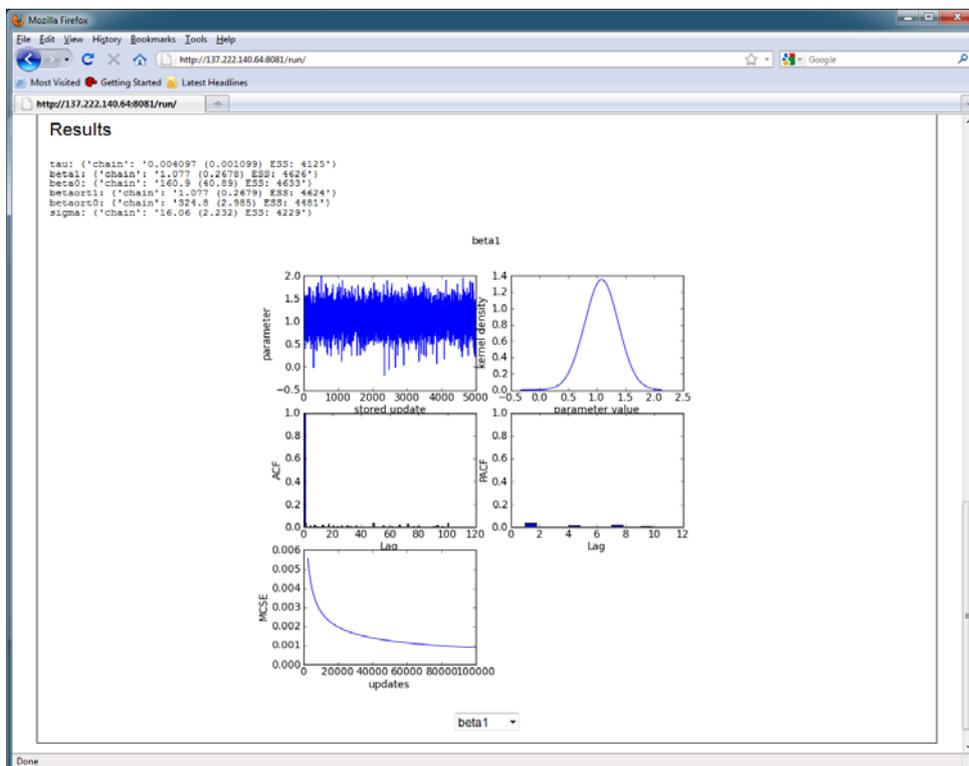
```

% if float(orthogmat[count+1]) >= float(0.0) :
+ \\
%endif
% endif
<% count += 1 %>\\
% endfor
% endfor
% endif
% if D == 'Normal':
tau ~ dgamma(0.001000, 0.001000)
sigma <- 1 / sqrt(tau)
% endif
}
'''

```

Here we see that a different *mmult* function is used for the orthogonal parameterisation and priors are given for *betaort* rather than *beta* in this case. Finally there is code 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*:



We again see good mixing of the chains and very similar estimates to the blocking approach. The other advantage of this orthogonal approach is 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**

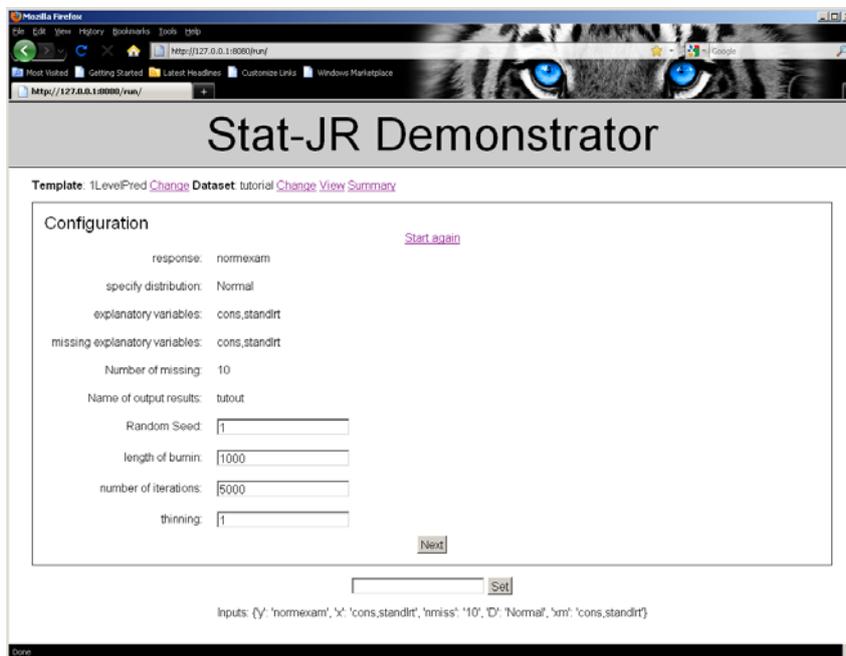
Convert this template so that it is analogous to the *Regression1* template but uses the orthogonal parametrisation. Remember to rename the template.

## 14. Out of sample predictions

Most of the statistical modelling templates we have thus far created are primarily being used to estimate a model. We might be interested in using the fitted model to predict responses for units outside the sample. Using a simulation-based approach, it is easy to get confidence intervals about these predictions at the same time as estimating the model. In a classical setting out of sample prediction would occur after a model is fitted. In MCMC the predictions are estimated simultaneously with the model fit so it is important that although the model fit is used in the prediction that the reverse is not true, and that the fit of the model to the data should not depend on the out of sample predictions. WinBUGS has a method to do this with its cut function and we have developed a similar method which we will demonstrate here.

### 14.1 The 1LevelPred template - using the zxfd trick

We will illustrate our approach on a 1 level model which we can fit using the *1LevelPred* template. We will choose this template along with the *tutorial* dataset. 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 tutorial dataset (the 10 has to be input as the *Number of missing* input). Note if you want to predict for 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. We therefore select the following inputs:

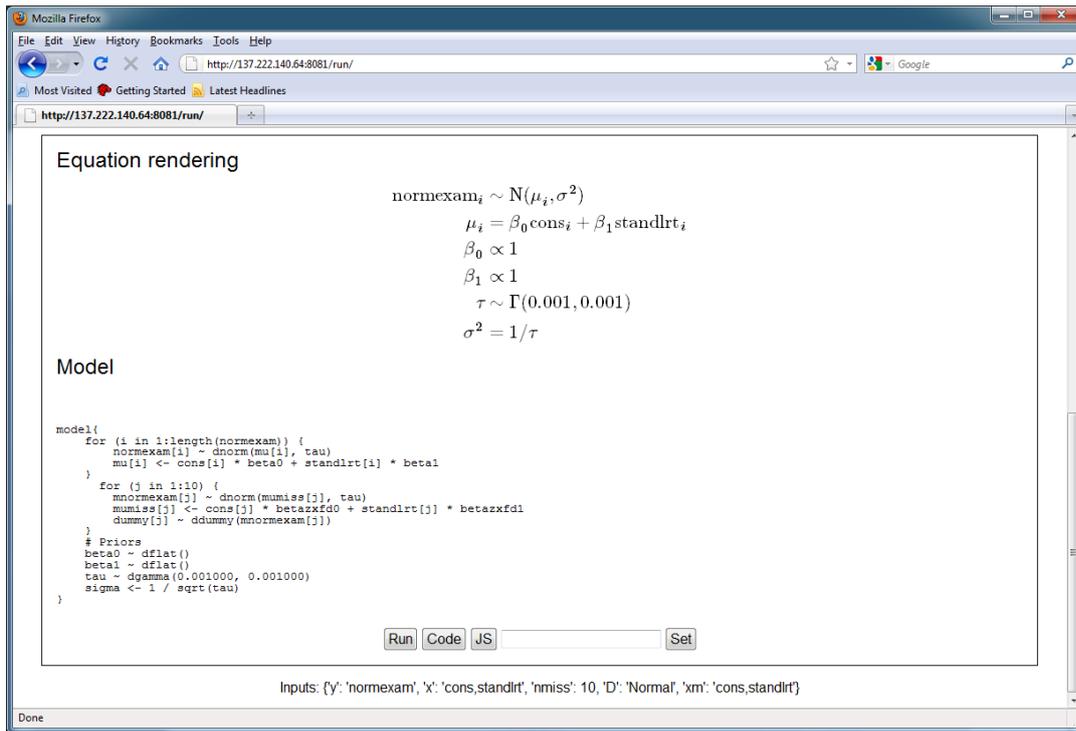


The screenshot shows the Stat-JR Demonstrator web interface. The browser window title is "Stat-JR Demonstrator". The URL is "http://127.0.0.1:8080/jur/". The page content includes a "Configuration" section with the following settings:

- response: normexam
- specify distribution: Normal
- explanatory variables: cons\_standlrt
- missing explanatory variables: cons\_standlrt
- Number of missing: 10
- Name of output results: tutout
- Random Seed: 1
- length of burnin: 1000
- number of iterations: 5000
- thinning: 1

There are "Next" and "Set" buttons at the bottom of the configuration section. Below the configuration section, the inputs are listed as: Inputs: {'Y': 'normexam', 'X': 'cons\_standlrt', 'nmiss': '10', 'D': 'Normal', 'xi': 'cons\_standlrt'}

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

```
mumiss[j] <- cons[j] * betazxfd0 + standlrt[j] * betazxfd1
```

has the strange string *zxfd* placed in the middle of the two parameter names. This is our way of performing the cut function used in WinBUGS. As the predictors in this line are not  $beta_0$  and  $beta_1$  then this line will not influence the posterior of the fixed effect. The posterior for  $mnormexam$  will be calculated but this is only because we include the line

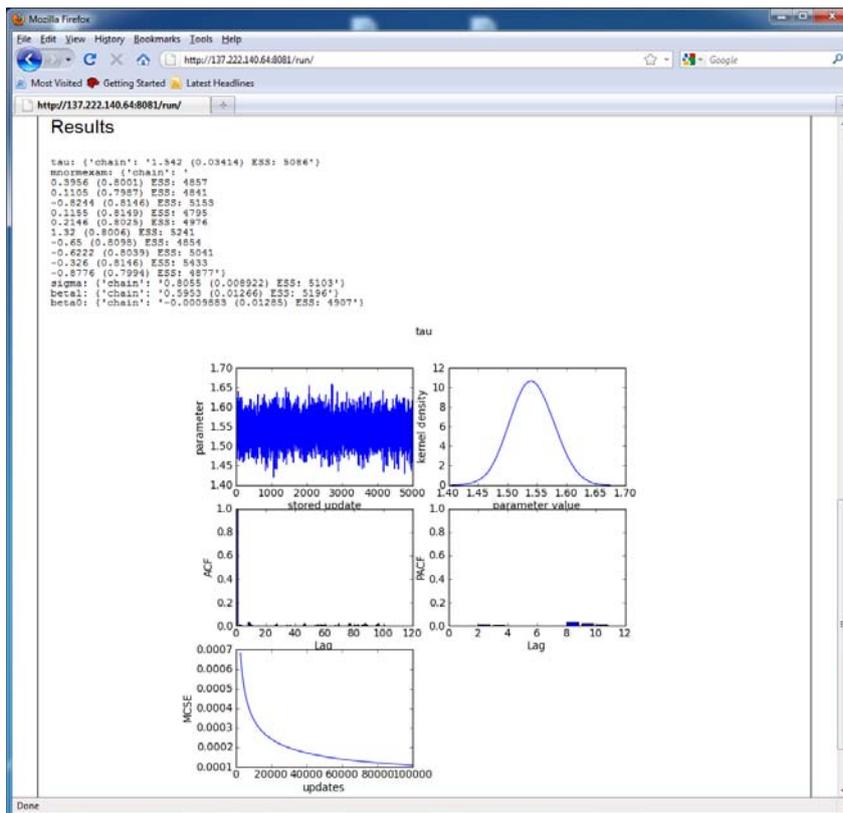
```
dummy[j] ~ ddummy(mnormexam[j])
```

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  $betazxfd0$  and  $betazxfd1$ . Of course in practice we want these replaced by the correct  $beta_0$  and  $beta_1$  and this is done in the bowels of the code generator with the lines:

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

which can be found in *XMLtoC.py* and other such files.

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 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 *1LevelCat* template to allow for out of sample predictions. Remember to save the new template under a different name.

## 15. Solutions to the exercises

Rather than fill many pages with Python code we have placed potential solutions to each of the exercises in a solutions directory on the e-STAT website. 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

**Exercise 6** 1LevelLogit.py

**Exercise 7** 2levelcatinteractions.py

**Exercise 8** nlevelcatint.py

**Exercise 9** 2levelwithRS.py

**Exercise 10** orthogregression.py

**Exercise 11** 1levelcatpred.py