A generic MCMC implementation in C++ based around specification of the model DAG

Graeme K. Ambler*

October 21, 2003

Abstract

The beginnings of a generic implementation of Markov chain Monte Carlo methods are discussed. At this stage, the only possible models are those in the Normal-Normal and Normal-Inverse Gamma class, however the framework is quite general, and may be extended as required to any other conjugate relationships. It is also possible to extend the framework to Metropolis-Hastings samplers when priors are not conjugate.

1 Introduction

The BUGS system has made it simple to specify Bayesian models simply and sample from the appropriate posterior distribution. However, a problem with BUGS is that it is much slower (sometimes a couple of orders of magnitude) than hand-coded software written in relatively low-level languages such as C or Fortran. There is need for a "middle way", providing samplers with (1) speed comparable with hand-coded algorithms, and (2) a method of specification close to that of BUGS in terms of ease-of-use. The purpose of this document is to introduce such a method. In Section 2 we introduce the method of specification, and in Section 3 we go on to discuss how to extend the framework when there are more complicated relationships between the variables. In Section 4 we discuss an example taken from the BUGS example files, before drawing some conclusions about the effectiveness of the method in Section 5. Some technical details about the workings of the Normal-Normal and Normal-Inverse Gamma classes are discussed in Appendix A. Finally, a complete listing of the rats example discussed in Section 4 is given in Appendix B. Source code for the library files needed to run the examples is available from the author on request.

2 Model specification

Let us first consider a very simple model. Suppose that we have some data, y_i, for $i = 1, \ldots, 10$ which is Normally distributed with unknown parameters μ and σ^2. A reasonable Bayesian model would be to use a Normal prior for the data, with a Normal hyperprior on the mean and a Gamma hyperprior on the precision of the data as follows:

$$\begin{aligned} y_i &\sim N(\mu, \sigma^2), \\ \mu &\sim N(0, 10^6), \\ \sigma^{-2} &\sim \Gamma(0.001, 0.001). \end{aligned}$$

The DAG for this model is shown in Figure 1.

This model would be specified as follows:

```
1: double mu=0;
2: double tau=1;
3:
4: double p_mu=0;
5: double p_tau=0.000001;
```

Address for correspondence: Department of Mathematics, University of Bristol, University Walk, Bristol, BS8 1TW, U.K. Email: graeme@ambler.me.uk

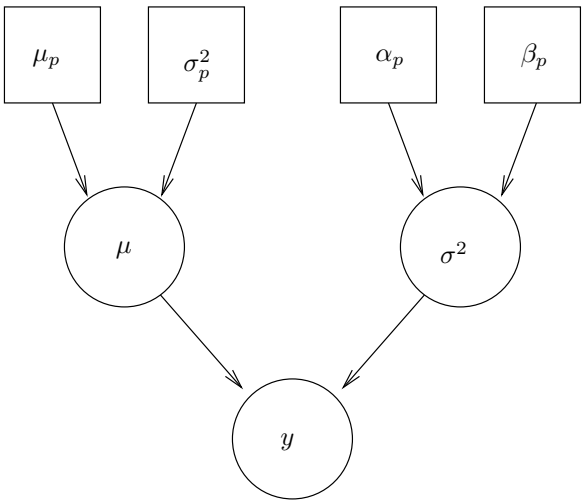

Figure 1: The DAG for a simple model.

```
 6: typedef NormalMu<Const,Const> Mu_T;
 7: Mu_T Mu(&p_mu,&p_tau,Fixed,Fixed);
 8:
 9: double p_alpha=0.001;
10: double p_beta=0.001;
11: typedef GammaTau<Const,Const> Tau_T;
12: Tau_T Tau(&p_alpha,&p_beta,Fixed,Fixed);
13:
14: Normal<Mu_T,Tau_T> Root(&mu,&tau,Mu,Tau);
15:
16: for(int i=0; i<MaxIter; ++i){
17:   Root.Update(&data,&rand);
18:   output << mu << '\t' << 1/sqrt(tau) << '\n';
19: }
```

I have not shown the input routine for reading in the data values, the definition of the random number generator `rand`, or the declaration of the output file, as these are trivial. It will be instructive to work through this code extract.

```
 1: double mu=0;
 2: double tau=1;
```

These lines declare the two model parameters μ and $\tau = \sigma^-2$, giving them default values.

```
 4: double p_mu=0;
 5: double p_tau=0.000001;
 9: double p_alpha=0.001;
10: double p_beta=0.001;
```

We declare the hyperparameters. One feature of this method is that it is necessary to declare *all* of the parameters in this way, *even the constants*.

```
 6: typedef NormalMu<Const,Const> Mu_T;
 7: Mu_T Mu(&p_mu,&p_tau,Fixed,Fixed);
```

Declare that `mu` has a Normal distribution with constant parameters `p_mu` and `p_tau`.

```
11: typedef GammaTau<Const,Const> Tau_T;
12: Tau_T Tau(&p_alpha,&p_beta,Fixed,Fixed);
```

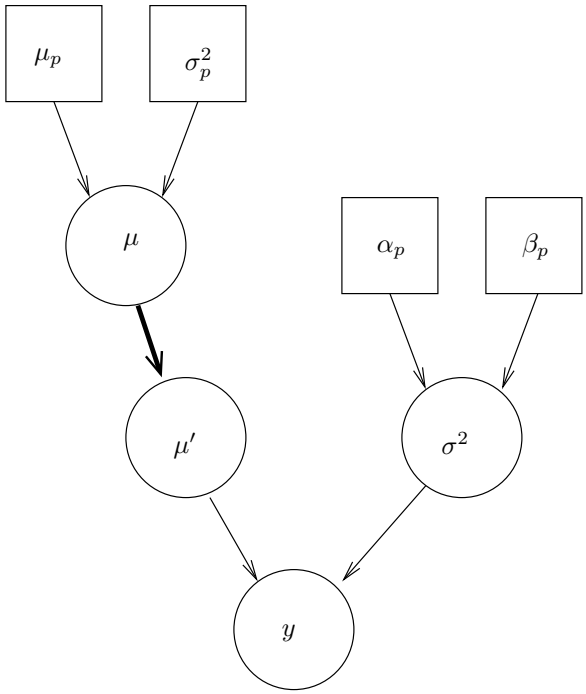

Figure 2: The DAG for a simple model with a formula node.

Declare that `tau` has a Gamma distibution with constant parameters `p_alpha` and p_beta.

```
14: Normal<Mu_T,Tau_T> Root(&mu,&tau,Mu,Tau);
```

Declare that the root of the DAG (the data) has a Normal distribution with parameters `mu` and `tau`.

```
16: for(int i=0; i<MaxIter; ++i){
17:   Root.Update(&data,&rand);
18:   output << mu << '\t' << 1/sqrt(tau) << '\n';
19: }
```

Run an MCMC simulation of the posterior distribution with `MaxIter` iterations, and output the values of `mu` and $1/\sqrt{\texttt{tau}}$ $(= \sigma)$ for analysis.

3 Formula nodes

Suppose that we now have a more complex model, where there is a deterministic relationship between some of the parameters. In this case, we must introduce a *formula node* into the DAG. We consider a trivial example here to introduce the method. A more concrete model is considered in the example discussed in Section 4.

Suppose now that

$$\begin{aligned} y_i &\sim N(\mu', \sigma^2), \\ \mu &= \mu' + 5, \\ \mu &\sim N(0, 10^6), \\ \sigma^{-2} &\sim \Gamma(0.001, 0.001). \end{aligned}$$

The DAG for this model is shown in Figure 2.

The specification of this model is more complex due to the presence of the formula node in the DAG. In Section 2 we glossed over some of the details, like where in a program the code extract would go. In this case, we cannot do this, as the structure of the source file is more important. A listing for this model is:

```
 1: class Formula;
 2:
 3: typedef NormalMu<Const,Const,double,double,Formula> Mu_T;
 4: typedef GammaTau<Const,Const,double,double,Formula> Tau_T;
 5:
 6: class Formula : public MuForm<Mu_T>{
 7: public:
 8:   Formula(Mu_T* Mu, double* mu, double* tau)
 9:     : MuForm<Mu_T>(Mu,mu) { Mu->SetParents(this,tau); }
10:   Formula& operator=(double mu) { *mu_=mu; return *this; }
11:   double GetResid(double data,int i,int j) { return data-5; }
12:   double GetMu(int i) { return *mu_+5; }
13: };
14:
15: int main()
16: {
17:   double mu=0;
18:   double tau=1;
19:   double p_alpha=0.001;
20:   double p_beta=0.001;
21:   double p_mu=0;
22:   double p_tau=0.000001;
23:   Mu_T Mu(&p_mu,&p_tau,Fixed,Fixed);
24:   Tau_T Tau(&p_alpha,&p_beta,Fixed,Fixed);
25:
26:   Formula MyFormula(&Mu,&mu,&tau);
27:
28:   Normal<Formula,Tau_T,Formula> Root(&MyFormula,&tau,MyFormula,Tau);
29:   for(int i=0; i<MaxIter; ++i){
30:     Root.Update(&data,&rand);
31:     output << mu << '\t' << 1/sqrt(tau) << '\n';
32:   }
33: }
```

This code extract skips over the same things as the first one does. We will focus on the differences between this and the first code extract.

The first thing we do is to declare that there is a formula node in the model:

```
 1: class Formula;
```

Secondly, we must now issue the `typedef` commands at the beginning:

```
 3: typedef NormalMu<Const,Const,double,double,Formula> Mu_T;
 4: typedef GammaTau<Const,Const,double,double,Formula> Tau_T;
```

There is also a change to the structure of these definitions: They take extra template parameters. The general form of the template parameters is given by

```
<Prior on Par1,Prior on Par2,Par1 type,Par2 type,Child1 type,Child2 type>
```

with the last four defaulting to type `double`. For example, in the case of `Mu_T`, this works out as:

`Prior on Par1` The prior on μ_p, which is that μ_p is a constant.

`Prior on Par2` The prior on σ_p^2, which is that σ_p^2 is a constant.

`Par1 type` The type of μ_p, which is `double`

`Par2 type` The type of σ_p^2, which is `double`

`Child1 type` The type of μ', which is a `Formula`

`Child2 type` The type of $\tau(=\sigma^{-2})$, which is `double`.

The way that default arguments work is that only *trailing* arguments can take a default value, so because we need to specify that the fifth template parameter is not the default, we must specify the third and fourth as well, even though these are the defaults.

```
 6: class Formula : public MuForm<Mu_T>{
 7: public:
 8:   Formula(Mu_T* Mu, double* mu, double* tau)
 9:     : MuForm<Mu_T>(Mu,mu) { Mu->SetParents(this,tau); }
10:   Formula& operator=(double mu) { *mu_=mu; return *this; }
11:   double GetResid(double data,int i,int j) { return data-5; }
12:   double GetMu(int i) { return *mu_+5; }
13: };
```

Line 6 begins the formula definition by declaring that it follows the same pattern as a default formula, `MuForm`. This allows us to take advantage of some default values and specify less ourselves. Lines 8 and 9 define a *constructor* for the Formula class, which passes some arguments to the default formula and sets the parents of the formula node to be `this` (a key word for the Formula class itself) and `tau`. Line 10 defines the assignment operator, so that we can update the value of `mu` (this is a standard form and is likely to be the same in many examples). Line 11 defines the residual for use in calculating the full conditional of `mu`. See Appendix A. Line 12 returns the formula for μ' for use in calculating the full conditional of `tau`. Again, see Appendix A. The only part of this definition which varies with the formula for μ' are the parts in braces in lines 11 and 12.

```
26:   Formula MyFormula(&Mu,&mu,&tau);
```

Declares that `MyFormula` is our `Formula`.

```
28:   Normal<Formula,Tau_T,Formula> Root(&MyFormula,&tau,MyFormula,Tau);
```

Declares that the root of the DAG (the data) is Normally distributed with mean given by a `Formula` and precision `tau`. Notice that the formula is passed as both the value of the mean and as the node itself. This is why we must overload the assignment operator '`=`' in line 10.

4 Rats example

We now move on to a more complex example, taken from the BUGS example files. Thirty young rats were weiwhere where ghed weekly for five weeks and those weights recorded. The model is essentially a random effects linear growth curve:

$$\begin{aligned} Y_{ij} &\sim \text{Normal}(\alpha_i + \beta_i(x_j - \bar{x}), \tau_c) \\ \alpha_i &\sim \text{Normal}(\alpha_c, \alpha_\tau) \\ \beta_i &\sim \text{Normal}(\beta_c, \beta_\tau) \end{aligned}$$

where i are rats, j are weeks and τ is again the *precision* of a Gaussian distribution. The hyperparameters α_c, α_τ, β_c, β_τ and τ_c are given "noninformative" priors:

$$\begin{aligned} \alpha_c &\sim \text{Normal}(0, 10^{-6}) \\ \alpha_\tau &\sim \text{Gamma}(0.001, 0.001) \\ \alpha_c &\sim \text{Normal}(0, 10^{-6}) \\ \alpha_\tau &\sim \text{Gamma}(0.001, 0.001) \\ \tau_c &\sim \text{Gamma}(0.001, 0.001) \end{aligned}$$

The DAG for this model is shown in Figure 3.

We list the code and then explain the nonstandard parts. Again, we have left out things like reading in the data. There is a full (and more spaced-out) version of the code in Appendix B.

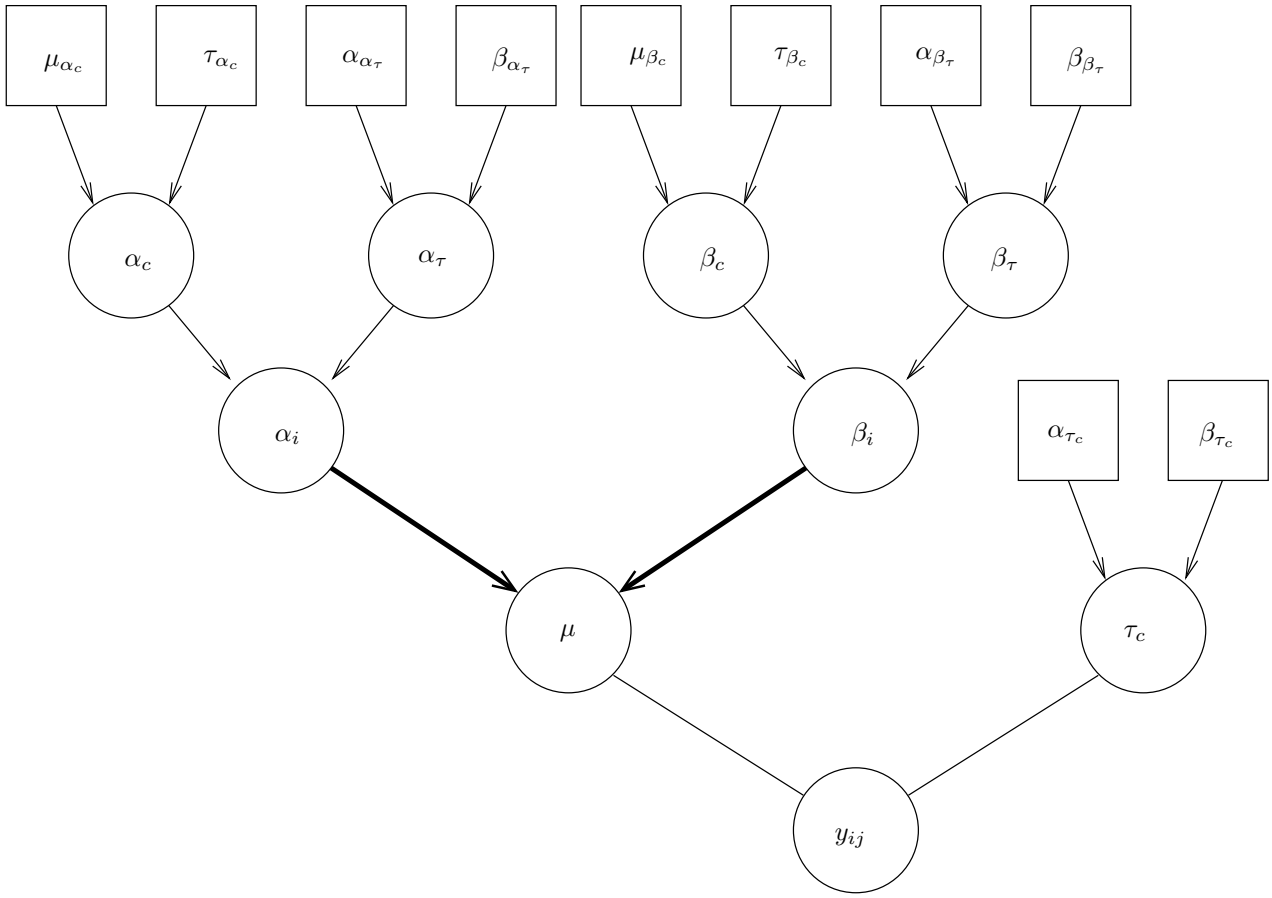

Figure 3: The DAG for the rats model.

```
 1: class Formula; class AlphaFormula; class BetaFormula;
 2:
 3: typedef GammaTau<Const,Const> AlphaT_T;
 4: typedef NormalMu<Const,Const> AlphaC_T;
 5: typedef GammaTau<Const,Const> BetaT_T;
 6: typedef NormalMu<Const,Const> BetaC_T;
 7: typedef NormalMu<AlphaC_T,AlphaT_T,double,double,AlphaFormula> Alpha_T;
 8: typedef NormalMu<BetaC_T,BetaT_T,double,double,BetaFormula> Beta_T;
 9: typedef GammaTau<Const,Const,double,double,Formula> TauC_T;
10:
11: class Formula{
12: protected:
13:   Alpha_T* Alpha_; Beta_T* Beta_;
14:   array* alpha_; array* beta_;
15:   array* x_; double SumXSq_;
16: public:
17:   Formula(Alpha_T* Alpha, Beta_T* Beta,
18:    array* alpha, array* beta, array* x_resid, double SumXSq)
19:     : Alpha_(Alpha), Beta_(Beta), alpha_(alpha), beta_(beta), x_(x_resid),
20:       SumXSq_(SumXSq) {}
21:   void SetParents(...) {}
22:   template<class T> void Update(T* data, Random* rand)
23:   { Alpha_->Update(data,rand); Beta_->Update(data,rand); }
24:   double GetMu(int i,int j) { return (*alpha_)[i]+(*beta_)[i]*(*x_)[j]; }
```

```
25: };
26: class AlphaFormula : public Formula{
27: public:
28:   AlphaFormula(Formula& F) : Formula(F) {}
29:   double& operator[](int i) { return (*alpha_)[i]; }
30:   double GetResid(double data,int i,int j)
31:   { return data-(*beta_)[i]*(*x_)[j]; }
32:   double GetConst(int n) { return n; }
33:   int size() { return alpha_->size(); }
34: };
35: class BetaFormula : public Formula{
36: public:
37:   BetaFormula(Formula& F) : Formula(F) {}
38:   double& operator[](int i) { return (*beta_)[i]; }
39:   double GetResid(double data,int i,int j)
40:   { return (data-(*alpha_)[i])*(*x_)[j]; }
41:   double GetConst(int n) { return SumXSq_; }
42:   int size() { return beta_->size(); }
43: };
44:
45: int main(){
46:   array alpha(30); array beta(30);
47:   for(int i=0; i<30; ++i) { alpha[i]=250; beta[i]=6; }
48:   double sum_x_sq=490;
49:   array x_resid(5); int temp=-14;
50:   for(int i=0; i<5; ++i) { x_resid[i]=temp; temp+=7; }
51:   double alpha_c=150; double alpha_tau=1;
52:   double beta_c=10; double beta_tau=1;
53:   double tau_c=1;
54:   double p_mu=0; double p_tau=0.000001;
55:   double p_alpha=0.001; double p_beta=0.001;
56:   AlphaC_T AlphaC(&p_mu,&p_tau,Fixed,Fixed);
57:   AlphaT_T AlphaT(&p_alpha,&p_beta,Fixed,Fixed);
58:   BetaT_T BetaT(&p_alpha,&p_beta,Fixed,Fixed);
59:   BetaC_T BetaC(&p_mu,&p_tau,Fixed,Fixed);
60:   TauC_T TauC(&p_alpha,&p_beta,Fixed,Fixed);
61:   Alpha_T Alpha(&alpha_c,&alpha_tau,AlphaC,AlphaT);
62:   Beta_T Beta(&beta_c,&beta_tau,BetaC,BetaT);
63:
64:   Formula MyFormula(&Alpha,&Beta,&alpha,&beta,&x_resid,sum_x_sq);
65:   AlphaFormula AlphaF(MyFormula); Alpha.SetParents(&AlphaF,&tau_c);
66:   BetaFormula BetaF(MyFormula); Beta.SetParents(&BetaF,&tau_c);
67:
68:   Normal<Formula,TauC_T,Formula> Root(&MyFormula,&tau_c,MyFormula,TauC);
69:   for(int i=0; i<MaxIter; ++i){
70:     Root.Update(&data,&rand);
71:     output << 1/sqrt(tau_c) << "\n"
72:         << alpha_c - xbar*beta_c << "\n"
73:         << beta_c << "\n";
74:   }
75: }
```

The code extract begins by declaring that there are some formulas to define.

```
1: class Formula; class AlphaFormula; class BetaFormula;
```

Since this is a non-trivial formula, we cannot take such good advantage of the defaults provided as we could in Section 3, so it is necessary to define 3 classes. We then declare the structure of the upper nodes in the DAG:

```
 3: typedef GammaTau<Const,Const> AlphaT_T;
 4: typedef NormalMu<Const,Const> AlphaC_T;
 5: typedef GammaTau<Const,Const> BetaT_T;
 6: typedef NormalMu<Const,Const> BetaC_T;
 7: typedef NormalMu<AlphaC_T,AlphaT_T,double,double,AlphaFormula> Alpha_T;
 8: typedef NormalMu<BetaC_T,BetaT_T,double,double,BetaFormula> Beta_T;
 9: typedef GammaTau<Const,Const,double,double,Formula> TauC_T;
```

Next, we define the Formulas. Since this is one of the more complex sections of code, we postpone discussion of that section until the end, and continue on to the parameter declarations:

```
46:   array alpha(30); array beta(30);
47:   for(int i=0; i<30; ++i) { alpha[i]=250; beta[i]=6; }
48:   double sum_x_sq=490;
49:   array x_resid(5); int temp=-14;
50:   for(int i=0; i<5; ++i) { x_resid[i]=temp; temp+=7; }
51:   double alpha_c=150; double alpha_tau=1;
52:   double beta_c=10; double beta_tau=1;
53:   double tau_c=1;
54:   double p_mu=0; double p_tau=0.000001;
55:   double p_alpha=0.001; double p_beta=0.001;
```

There is nothing special about this section of code, we are simply declaring the parameters and giving them default values. One thing to note is that although we have many constant values, we have declared only 4 (lines 54 and 55), as many of them take the same numerical value — in other words, although it is necessary to declare the constant parameters, it is quite possible to use the same declarations in several places if the numerical values are the same. Obviously this "short-cut" is only possible with the constants. One other thing to note is that we store only $x - \bar{x}$, rather than x itself (in the variable named `x_resid`. We also store the value of the sum of squares of $x - \bar{x}$, which we will use later.

We now initialise the nodes of the DAG we began defining in lines 3–9:

```
56:   AlphaC_T AlphaC(&p_mu,&p_tau,Fixed,Fixed);
57:   AlphaT_T AlphaT(&p_alpha,&p_beta,Fixed,Fixed);
58:   BetaT_T BetaT(&p_alpha,&p_beta,Fixed,Fixed);
59:   BetaC_T BetaC(&p_mu,&p_tau,Fixed,Fixed);
60:   TauC_T TauC(&p_alpha,&p_beta,Fixed,Fixed);
61:   Alpha_T Alpha(&alpha_c,&alpha_tau,AlphaC,AlphaT);
62:   Beta_T Beta(&beta_c,&beta_tau,BetaC,BetaT);
```

Again, we leave the next 3 lines until the end, when we discuss the formula node. Finally we define the root of the DAG, run the Markov chain, and output the quantities of interest:

```
68:   Normal<Formula,TauC_T,Formula> Root(&MyFormula,&tau_c,MyFormula,TauC);
69:   for(int i=0; i<MaxIter; ++i){
70:     Root.Update(&data,&rand);
71:     output << 1/sqrt(tau_c) << "\n"
72:         << alpha_c - xbar*beta_c << "\n"
73:         << beta_c << "\n";
74:   }
```

Having glossed briefly over the parts of the code which should be familiar from the previous two sections, we now come to the more complex matter of how we deal with the formula node. There are 3 classes involved here. The first is the general node. The latter two are specialisations of that node, one for each of the node's parents. This is a general pattern.

```
11: class Formula{
```

We cannot inherit from the template class `MuForm<>` this time, as the formula node has two stochastic parents. It is certainly possible to define a similar template node for formulas with two stochastic parents which would reduce the code here considerably (and the author would be justified in doing so, as it is a common scenario). However, it is instructive to consider what is necessary when a formula does not follow a standard pattern anticipated by the provider of the library.

```
12: protected:
13:   Alpha_T* Alpha_; Beta_T* Beta_;
14:   array* alpha_; array* beta_;
15:   array* x_; double SumXSq_;
```

We begin by declaring the data members and parent nodes of the formula node. Line 13 declares the parent nodes of the formula. It is necessary for every node to contain pointers to its parent nodes so that we may traverse the DAG recursively from the root updating each stochastic node to perform a sweep. Lines 14 and 15 declare the components of the formula. `SumXSq_` is a constant needed when updating `Beta` which is simply the sum of squares of $x - \bar{x}$.

```
17:   Formula(Alpha_T* Alpha, Beta_T* Beta,
18:    array* alpha, array* beta, array* x_resid, double SumXSq)
19:     : Alpha_(Alpha), Beta_(Beta), alpha_(alpha), beta_(beta), x_(x_resid),
20:       SumXSq_(SumXSq) {}
```

This is the *constructor* of the node. We pass it the values of the data members and parent nodes and it initialises them. The {} means "do nothing".

```
21:   void SetParents(...) {}
```

This is simple housekeeping. Each node stores not only the data members of its parents (e.g. node `AlphaC` stores the values 0 and 0.000001), but also the data members of its child's parents (α_c and α_τ). These are passed to it by its child when its child is initialised by calling the `SetParents` function for each of its parents in turn. Thus there must always be such a member function for each node, which is why we define it in line 21. We will come back to this subject when we discuss lines 65 and 66.

```
22:   template<class T> void Update(T* data, Random* rand)
23:   { Alpha_->Update(data,rand); Beta_->Update(data,rand); }
```

The `Update` function is called by a node's parents. It has two responsibilities:

1. Call Update for each of its parents in turn in order to recursively update every stochastic node in the DAG.
2. Update itself.

The second of these responsibilities is not applicable to a formula node, as it is not stochastic in nature. The function is a `template` as we did not want to have to worry about what type the data would be.

```
24:   double GetMu(int i,int j) { return (*alpha_)[i]+(*beta_)[i]*(*x_)[j]; }
```

The `GetMu` function returns the value of the formula for which the class is being written. See Appendix A for details.

```
26: class AlphaFormula : public Formula{
27: public:
28:   AlphaFormula(Formula& F) : Formula(F) {}
29:   double& operator[](int i) { return (*alpha_)[i]; }
30:   double GetResid(double data,int i,int j)
31:   { return data-(*beta_)[i]*(*x_)[j]; }
32:   double GetConst(int n) { return n; }
33:   int size() { return alpha_->size(); }
```

We now define the way that the formula relates to `Alpha`. Line 26 declares that `AlphaFormula` inherits from `Formula` (a common way of reading this line is "class `AlphaFormula` is a `Formula`"). Line 28 defines the constructor, which takes a `Formula` so that it may contruct it's parent correctly. Line 29 defines the subset operator "[]" so that if you were to call `AlphaF[i]` you would get α_i. This is analogous to defining the assignment operator "`=`" in Section 3, as it means that the "back end" can call `AlphaF[i]=<value>` to update α_i for each i. In general, if a node is an `array`, you must define the subset operator and if it is a single `double` you must define the assignment operator. If a node is an `array` you must also define the `size()` function, which is done in line 33. Lines 30–32 define the `GetResid` and `GetConst` functions. See Appendix A for details. The function `GetConst` was not discussed in Section 3 because the default from `MuForm<>` was used.

```
35: class BetaFormula : public Formula{
36: public:
37:   BetaFormula(Formula& F) : Formula(F) {}
38:   double& operator[](int i) { return (*beta_)[i]; }
39:   double GetResid(double data,int i,int j)
40:   { return (data-(*alpha_)[i])*(*x_)[j]; }
41:   double GetConst(int n) { return SumXSq_; }
42:   int size() { return beta_->size(); }
```

This is basically the same as `AlphaFormula`, except that we see a rare instance where the constant is not the value passed. See Appendix A for details.

5 Conclusions

It is hoped that the method described in the preceeding sections describes the beginnings at least of a "middle way" between the simple format of BUGS and the speed of hand-coded methods. Although the definition of formula nodes is a little more complex than one would like, I am confident that with a little familiarity it will be seen that the majority of code is very repeatable and not overly long. Much of this could be automated by writing a simple tool, as only the data members and the `Get*` functions vary between formulae, which require the calculation of full conditionals in order to fill in the details (though there is a pattern to these as discussed in Appendix A, so it is not usually necessary to complete the full calculation). The framework we have presented here is naturally extensible, and in a future report technical details of how to go about extending the framework to both other conjugate families and models where the priors are not conjugate will be given.

A Technical details for formula nodes

There are three key function calls which allow formula nodes to be incorporated into the DAG. They are `GetResid`, `GetConst` and `GetMu`. The reason for these is as follows. Let

$$\begin{aligned} A_i &\sim \text{Normal}(C + DE_i, \tau), \\ C &\sim \text{Normal}(\mu_C, \tau_C), \\ D &\sim \text{Normal}(\mu_D, \tau_D), \\ \tau &\sim \text{Gamma}(\alpha, \beta), \end{aligned}$$

where $i = 1, \ldots, N$, the E_i may or may not be constants, and the second parameter in all Normal distributions in this section is the *precision*. Then

$$\begin{aligned} C|A, D, \tau, \mu_C, \tau_C &\sim \text{Normal}\left(\frac{\mu_C\tau_C + \tau\sum_{i=1}^N (A_i - DE_i)}{\tau_C + \tau N}, \tau_C + \tau N\right) \text{ and} \\ D|A, C, \tau, \mu_D, \tau_D &\sim \text{Normal}\left(\frac{\mu_D\tau_D + \tau\sum_{i=1}^N (A_i - C)E_i}{\tau_D + \tau\sum_{i=1}^N E_i^2}, \tau_D + \tau\sum_{i=1}^N E_i^2\right). \end{aligned}$$

The purpose of the `GetResid` function is to return the result of a single term of the first sum. The purpose of the `GetConst` function is to return the multiplier of τ in the precision. Its parameter is the value of N, so in the first case we simply return the value passed.

Finally,

$$\tau|A, C, D, E \sim \text{Gamma}\left(\alpha + \frac{N}{2}, \beta + \frac{1}{2}\sum_{i=1}^N (A_i - (C + DE_i))^2\right).$$

The purpose of `GetMu` is to return the value of $C + DE_i$ for a given value i.

B Full listing of the rats example

```
#ifndef LOCAL_RAND_IS_INCLUDED
```

```
#define LOCAL_RAND_IS_INCLUDED
#include "rand.hh"
#endif               // LOCAL_RAND_IS_INCLUDED

#include <iostream>
#include <fstream>

#include "dist.hh"

class Formula;
class AlphaFormula;
class BetaFormula;

typedef GammaTau<Const,Const> AlphaT_T;
typedef NormalMu<Const,Const> AlphaC_T;
typedef GammaTau<Const,Const> BetaT_T;
typedef NormalMu<Const,Const> BetaC_T;
typedef NormalMu<AlphaC_T,AlphaT_T,double,double,AlphaFormula> Alpha_T;
typedef NormalMu<BetaC_T,BetaT_T,double,double,BetaFormula> Beta_T;
typedef GammaTau<Const,Const,double,double,Formula> TauC_T;

// Now for the formula part:
class Formula{
protected:
  Alpha_T* Alpha_;
  Beta_T* Beta_;
  array* alpha_;
  array* beta_;
  array* x_;
  double SumXSq_;
public:
  Formula(Alpha_T* Alpha, Beta_T* Beta,
  array* alpha, array* beta, array* x_resid, double SumXSq)
    : Alpha_(Alpha), Beta_(Beta), alpha_(alpha), beta_(beta), x_(x_resid),
      SumXSq_(SumXSq) {}
  void SetParents(...) {}
  template<class T> void Update(T* data, Random* rand)
  { Alpha_->Update(data,rand); Beta_->Update(data,rand); }
  double GetMu(int i,int j) { return (*alpha_)[i]+(*beta_)[i]*(*x_)[j]; }
};

class AlphaFormula : public Formula{
public:
  AlphaFormula(Formula& F) : Formula(F) {}
  double& operator[](int i) { return (*alpha_)[i]; }
  double GetResid(double data,int i,int j)
  { return data-(*beta_)[i]*(*x_)[j]; }
  double GetConst(int n) { return n; }
  int size() { return alpha_->size(); }
};

class BetaFormula : public Formula{
public:
  BetaFormula(Formula& F) : Formula(F) {}
  double& operator[](int i) { return (*beta_)[i]; }
  double GetResid(double data,int i,int j)
  { return (data-(*alpha_)[i])*(*x_)[j]; }
```

```
  double GetConst(int n) { return SumXSq_; }
  int size() { return beta_->size(); }
};

int main()
{
  Const Fixed;
  Random rand(192492);

  int MaxIter=1000000;

  int variables=30;
  int samples=5;

  // Open the data file and read in the data
  double tempdata;
  std::ifstream input("data.txt");
  if(!input)
    {
      std::cerr << "Error!  Cannot open data file!\n";
      exit(1);
    }
  array2D data;
  data.resize(variables);
  if(samples>0){
    for(int i=0; i<variables; ++i)
      {
data[i].resize(samples);
for(int j=0; j<samples; ++j)
  {
    if(!(input >> tempdata)){
      std::cerr << "Error reading value "
<< i*samples+j << "!\n";
      exit(1);
    }
    data[i][j]=tempdata;
  }
      }
  }
  input.close();

  // Define parameters and pick some vaguely sensible values for
  // the initial values.
  array x_resid(samples);
  array alpha(variables);
  array beta(variables);
  for(int i=0; i<variables; ++i){
    alpha[i]=250;
    beta[i]=6;
  }
  double temp[5]={8.0,15.0,22.0,29.0,36.0};
  double xbar=22;
  double sum_x_sq=0;
  for(int i=0; i<samples; ++i)
    {
      x_resid[i]=temp[i]-xbar;
      sum_x_sq+=x_resid[i]*x_resid[i];
```

```
    }
  double alpha_c=150;
  double beta_c=10;
  double alpha_tau=1;
  double beta_tau=1;
  double tau_c=1;

  std::ofstream output("output.txt");

  // Now define the model node by node, starting at the topmost level.
  // Since prior hyperparameter are identical for beta as for alpha,
  // we don't bother defining both sets.
  // We cannot do that with the dummy Const<> classes, however.  We need
  // one for each constant in the model.

  double p_alpha=0.001;
  double p_beta=0.001;
  double p_mu=0;
  double p_tau=0.000001;

  AlphaT_T AlphaT(&p_alpha,&p_beta,Fixed,Fixed);

  AlphaC_T AlphaC(&p_mu,&p_tau,Fixed,Fixed);

  BetaT_T BetaT(&p_alpha,&p_beta,Fixed,Fixed);

  BetaC_T BetaC(&p_mu,&p_tau,Fixed,Fixed);

  Alpha_T Alpha(&alpha_c,&alpha_tau,AlphaC,AlphaT);

  Beta_T Beta(&beta_c,&beta_tau,BetaC,BetaT);

  Formula MyFormula(&Alpha,&Beta,&alpha,&beta,&x_resid,sum_x_sq);
  AlphaFormula AlphaF(MyFormula); Alpha.SetParents(&AlphaF,&tau_c);
  BetaFormula BetaF(MyFormula); Beta.SetParents(&BetaF,&tau_c);

  TauC_T TauC(&p_alpha,&p_beta,Fixed,Fixed);

  Normal<Formula,TauC_T,Formula >
    Root(&MyFormula,&tau_c,MyFormula,TauC);

  for(int i=0; i<MaxIter; ++i)
    {
      Root.Update(&data,&rand);
      output << 1/sqrt(tau_c) << "\n"
     << alpha_c - xbar*beta_c << "\n"
     << beta_c << "\n";
    }
}
```