`GDAGsim` 0.3 User Guide[*]

Darren J Wilkinson

School of Mathematics & Statistics
University of Newcastle

Abstract

`GDAGsim` is a 'C' library for the analysis of linear graphical models. It can carry out sampling and conditional sampling of Gaussian Directed Acyclic Graph (GDAG) models, and hence can be used for the efficient implementation of block-MCMC samplers for linear models. It can also be used for the Bayes linear analysis of Bayes linear graphical models. This document gives an introduction to the use of the software, and provides a reference for the library interface.

1 Introduction

The `GDAGsim` software is a library for 'C' programmers who want to analyse GDAG models. It makes heavy use of the GNU Scientific library (GSL), and the GSL interface to the standard Basic Linear Algebra Subprograms (BLAS). This documentation assumes familiarity with 'C', the GSL, the BLAS, and the GSL interface to the BLAS.

The software is currently only suitable for DAG models. Those interested in Gaussian Markov Random Fields (GMRFs), should refer to Harvard Rue's 'C' library, `GMRFsim`.

The current interface is only appropriate for describing *univariate* DAG models, though the computational back-end is general, and an extension of the interface for *multivariate* DAG models is planned.

The software works by building the (sparse) precision matrix for the latent (unobserved) variables conditional on any observations, and then computing with it. The previous implementation (0.1) stored the full matrix, and operated on it using dense matrix algorithms. This is *very* inefficient in terms of both storage and computation time. A sparse-matrix version (0.2) is now available, which uses the sparse-matrix algorithms from the *Meschach* numerical library. However, the sparse implementation has *exactly* the same interface as the old implementation, and assumes no knowledge of the *Meschach* library. The current system is practical for MCMC problems involving up to 10,000 latent (unobserved) variables. The author has used `GDAGsim` to obtain a small number of samples from, and conduct Bayes linear analyses of, problems containing over 100,000 latent variables (on an 8GB workstation), but current CPU speeds make fully blown MCMC analyses of such problems impractical. Versions of the library for massively parallel supercomputers are also planned (for the analysis of huge spatio-temporal models), but not in the very near future.

---

[*] © 2000–02, Darren J Wilkinson. Documentation last updated: August 21, 2002

2 DAG models

Gaussian DAG models are specified conditionally. Consider a DAG model for the multivariate Normal (MVN) random vector, $X = (X_0, X_1, \ldots, X_{n-1})'$ (note that `'C'`-style, 0 indexing is used throughout). Associated with the vector X is the graph $\mathcal{G} = (V, E)$, where $V = \{X_0, X_1, \ldots, X_{n-1}\}$ denotes a set of vertices, and $E = \{(v, v')|v, v' \in V, v \to v'\}$ is the set of ordered pairs denoting the set of directed edges in the graph. The *parents* of $v \in V$ are denoted $\mathrm{pa}(v) = \{v' \in V|(v', v) \in E\}$. The joint density of X factors according to the graph, $\mathcal{G}$ if

$$\pi(x) = \prod_{i=0}^{n-1} \pi(x_i|\mathrm{pa}(x_i)).$$

For MVN vectors, X, each conditional distribution is normal, so that

$$\pi(x_i|\mathrm{pa}(x_i)) = N(\alpha_i' x + b_i, 1/\tau_i),$$

where the n-dimensional vector, α_i has non-zero elements only in positions corresponding to the parents of X_i. Thus, a GDAG model can be constructed by specifying a precision, τ_i, a mean-shift, b_i, and the sparse-vector α_i, for each component, X_i.

3 Model construction

The two structs, `gdag` and `gsl_usv`, and the `GDAGsim` function templates are all given in the include file, `gdag.h`. All `GDAGsim` functions begin `gdag_`, minimising name-space pollution. If any sampling is to be done, a GSL random number stream must first be initialised with a call like

```
gsl_rng * r=gsl_rng_alloc(gsl_rng_mt19937);
```

This is a change from 0.2.

3.1 Sparse vectors

As mentioned in the previous section, associated with each variable is a sparse vector α, relating the variable to its parents. Thus, the `GDAGsim` library has a struct and associated functions for building sparse vectors. For example, to build the sparse vector $(0, 11, 0, 13, 14, 0, 0, \ldots)'$, which has 3 non-zero elements, the commands

```
gsl_usv * alpha=gdag_usv_alloc(3);
gdag_usv_add(alpha,1,11.0);
gdag_usv_add(alpha,3,13.0);
gdag_usv_add(alpha,4,14.0);
```

can be used. Such objects can be freed with

```
gdag_usv_free(alpha);
```

See the reference section for further details.

3.2 GDAG structure

All specifications relating to a given DAG model are stored in a `gdag` struct. In `GDAGsim`, the key number is the number of *latent* variables. That is, the number of Gaussian variables for which observations are not available. Observations are added once all of the latent variables have been declared, and do not affect the size of the `gdag` struct required. For example, to declare and clear a 3-variable structure, the command:

```
gdag * d=gdag_calloc(3);
```

should be used. The objects can be freed with

```
gdag_free(d);
```

3.3 Building latent structure

There are two basic kinds of variables; root nodes (which have no parents), and other nodes (which do have parents). To build the model

$$X_0 \sim N(1,1/2), \qquad X_1 \sim N(3,1/4), \qquad (X_2|X_0 = x_0, X_1 = x_1) \sim N(x_0 + 2x_1, 1),$$

use the commands

```
gdag_add_root(d,0,1.0,2.0);
gdag_add_root(d,1,3.0,4.0);
alpha=gdag_usv_alloc(2);
gdag_usv_add(alpha,0,1.0);
gdag_usv_add(alpha,1,2.0);
gdag_add_node(d,2,alpha,0,1);
gdag_usv_free(alpha);
```

3.4 Adding observations

Observed variables are added by specifying their conditional distributions, together with their observed values. Note that an index is not specified, since they are not added to the latent structure. For example, to add the observation $z = 20$, where

$$(Z|X_1 = x_1, X_2 = x_2) \sim N(x_1 + x_2 + 5, 2),$$

use the commands

```
alpha=gdag_usv_alloc(2);
gdag_usv_add(alpha,1,1.0);
gdag_usv_add(alpha,2,1.0);
gdag_add_observation(d,alpha,5,0.5,20);
gdag_usv_free(alpha);
```

4 Structure analysis

Once all latent variables have been specified, and any observations have been added to the structure, analysis of the resulting MVN distribution is possible. First the process function is called (which, *inter alia*, forms the Cholesky decomposition of the precision matrix).

```
gdag_process(d);
```

Now various analyses are possible. If the mean is required, this can be obtained with

```
gsl_vector * mean=gdag_mean(d);
```

If samples from the distribution are required, these can be obtained with

```
gsl_vector * sample=gdag_sim(r,d);
```

where `r` is a pre-defined GSL random number stream. *This is a change from 0.2*. If the log-likelihood of this sample is required, this can be obtained with

```
double ll=gdag_loglik(d);
```

If the marginal variance of the random quantity $\alpha'X$ is required, this can be obtained with

```
double var=gdag_var(d,alpha);
```

For other possibilities, consult the examples and the reference section.

5 Examples

5.1 Locally constant DLM

Consider the locally constant DLM,

$$(X_i|\Theta_i = \theta_i) \sim N(\theta_i, 1/\tau_{obs}), \qquad i = 0, 1, \dots n - 1$$
$$\Theta_0 \sim N(0, 1)$$
$$(\Theta_i|\Theta_{i-1} = \theta_{i-1}) \sim N(\theta_{i-1}, 1/\tau_{state}), \qquad i = 1, 2, \dots, n - 1$$

The example file `example_dlm.c` shows how to sample from the prior for Θ and X (which has $2n$ variables), and then uses the sample for sampling from the conditional distribution of $\Theta|X = x$ (which has n variables). A two-block MCMC sampler is implemented for the variance components τ_{state} and τ_{obs}.

5.2 Two-way ANOVA with random effects

Consider the model

$$Y_{ijk} = \mu + \alpha_i + \beta_j + \gamma_{ij} + \varepsilon_{ijk}, \quad i = 0, 1, \dots, p - 1, \ j = 0, 1, \dots, q - 1, \ k = 0, 1, \dots, r_{ij} - 1$$
$$\mu \sim N(0, 1/0), \quad \alpha_i \sim N(0, 1/\tau_\alpha), \quad \beta_j \sim N(0, 1/\tau_\beta), \quad \gamma_{ij} \sim N(0, 1/\tau_\gamma), \quad \varepsilon_{ijk} \sim N(0, 1/\tau_\varepsilon)$$

The example file `example_twa.c` accepts data of this form, and carries out a two-block MCMC sampler for the variance components and the mean, based on the joint sampling of the posterior for all $1 + p + q + pq$ latent Gaussian variables. To understand this program, it is helpful to understand the numbering of the latent variables. Variable μ is numbered 0, α_i is numbered $1 + i$, β_j is numbered $1 + p + j$ and γ_{ij} is numbered $1 + p + q + p * i + j$. The function `dep_usv` accepts parameters i and j, and returns a sparse vector with 1s in the four aforementioned positions. The file `foo.inf` contains information relating to the shape of the data. The first line contains p and q, and the rest of the file contains the values of r_{ij}. The file `foo.dat` contains the data itself.

6 Hints and tips

If you get the error `Error: Status mismatch in gdag_something`, it means that you are calling functions at the wrong time — *eg.* trying to call `gdag_sim` before calling `gdag_process`, *etc.*

If you get an error relating to positive definiteness problems during the Cholesky decomposition phase, then for some reason, the sparse matrix library thinks that your precision matrix isn't positive definite. Have you added any variables with zero or negative precisions? Have you added all of the variables and observations you should have done? Is your model really a DAG — are there any "loops"?! If you think your code is OK, then you are probably suffering from numerical problems. Usually this is caused by having precisions that are too small for the software to be able to cope with. Make them bigger — by rescaling variables if necessary. If you are using `GDAGsim` as part of an MCMC scheme, change the priors you are using for the variance components so as to make very small precisions less likely. Also, re-ordering the latent variables may help — see below.

Think carefully about the order in which you introduce variables into the problem. Currently, the sparse matrix library does no re-ordering of the rows and columns of the precision matrix before attempting to do a Cholesky decomposition. One day, I hope that it will, but until then, it is up to you to introduce your variables in such a way as to minimise the "fill-in" which occurs when the Cholesky factorisation takes place. The easiest way to do this is to try and minimise the "band-width" of the precision matrix. You do this by introducing nodes in such a way as to ensure that nodes don't depend on other nodes with a much lower number. For example, if you are building a spatio-temporal model, introducing the nodes in temporal order is vital — entering all the variables at one site, then all at another site, *etc.*, is disastrous!

If you think you have found a bug, please send me a patch which fixes it! Failing that, send me a complete program which I can (easily) run in order to reproduce the problem.

7 Library reference

7.1 `main`

`gdag * gdag_alloc(size_t n)`

This function allocates (but does not clear) a GDAG object (`gdag` struct) to hold n latent variables.

```
void gdag_set_zero(gdag * d)
```

Clears `d` ready for specification of the structure.

```
gdag * gdag_calloc(size_t n)
```

Allocates then clears a `gdag` struct ready for use.

```
void gdag_free(gdag * d)
```

Frees memory associated with `d`.

```
void gdag_add_root(gdag * d,size_t i,double mean, double precision)
```

Adds root variable numbered `i` to `d` with given mean and precision.

```
void gdag_add_node(gdag * d,size_t i,gdag_usv * alpha,
                   double b,double p)
```

Adds node numbered `i` to `d`, with conditional distribution

$$X_i | \operatorname{pa}(X_i) \sim N(\alpha' X + b, 1/p).$$

```
void gdag_add_observation(gdag * d,gdag_usv * alpha,
                          double b,double p,double x)
```

Adds observation x with conditional distribution as above to `d`.

```
void gdag_process(gdag * d)
```

Processes `d`, calculating, *inter alia*, the Cholesky decomposition of the precision matrix, and the mean vector of the structure.

```
void gdag_prior_process(gdag * d)
```

If the function `gdag_mloglik` is to be called, then this function must be called once all of the latent variables have been added to the structure, but *before* any of the observed variables have been added to the structure. The function computes and stores some quantities relating to the prior likelihood, which are needed for the computation of the marginal likelihood.

7.2 `sparse`

```
gdag_usv * gdag_usv_alloc(int nz)
```

Creates an unstructured sparse vector object (`gdag_usv` struct), with `nz` non-zero elements. All non-zero elements must be specified before the object can be used.

```
void gdag_usv_free(gdag_usv * usv)
```

Free memory associated with `usv`.

```
gdag_usv * gdag_usv_basis(int i)
```

Creates and returns a sparse vector with a 1 in the ith position.

```
void gdag_usv_add(gdag_usv * usv,int i,double x)
```

Puts x in the ith position of `usv`.

```
void gdag_dusaxpy(double alpha,gdag_usv * x,gsl_vector * y)
```

Sparse BLAS operation, which computes $\alpha x + y$, where x is sparse and y is dense. Result is stored in y.

```
void gdag_dussc(gdag_usv * x,gsl_vector * y)
```

Sparse BLAS operation, which scatters sparse x into dense y.

```
void gdag_dusdot(gdag_usv * x,gsl_vector * y,double * res)
```

Sparse BLAS operation, which computes the dot-product of x and y, storing the result in `res`.

```
void gdag_usv_dump(gdag_usv * usv)
```

Prints the contents of `usv` to standard output (mainly for debugging purposes).

7.3 `sim`

The syntax of all these commands has changed from 0.2. They all have a new first argument which is a GSL random number stream. This change of syntax is necessary to make the library thread-safe and hence ease the development of multi-threaded and parallel codes.

```
gsl_vector * gdag_sim(gsl_rng * r,gdag * d)
```

Samples from processed GDAG, `d`, and returns a pointer to the sampled vector using the stream `r`.

```
void gdag_vector_set_znorm(gsl_rng * r,gsl_vector * v)
```

Makes `v` a vector of standard normal random quantities.

```
double gdag_ran_gamma(gsl_rng * r,double a,double b)
```

Returns a $\Gamma(a,b)$ random quantity with mean a/b. N.B. The scale parameter is the reciprocal of that used by the corresponding GSL function.

```
double gdag_ran_gaussian(gsl_rng * r,double mu,double tau)
```

Returns a $N(\mu, 1/\tau)$ random quantity, with mean μ and variance $1/\tau$. N.B. This is parameterised differently to the corresponding GSL function.

```
double gdag_ran_uniform(gsl_rng * r,double l,double u)
```

Returns a $U(l,u)$ random quantity, with lower limit l and upper limit u.

```
int gdag_accept_p(gsl_rng * r,double p)
```

Returns TRUE with probability p. Note that if $p > 1$, it will return TRUE with certainty. The next function is more useful, however.

```
int gdag_accept_lp(gsl_rng * r,double l)
```

Returns TRUE with probability e^l. Note that if $l > 0$, it will return TRUE with certainty. Also note that e^l is not actually evaluated, thus preventing numerical problems. This function is very useful for Metropolis-Hastings and related algorithms — just evaluate $\log(A)$, where $\min(1,A)$ is the acceptance probability, and call this function with it.

7.4 `utils`

```
void gdag_dump(gdag * d)
```

Dumps `d` to standard output (mainly for debugging purposes).

```
size_t gdag_size(gdag * d)
```

Returns the number of latent variables in `d`.

```
size_t gdag_count(gdag * d)
```

Returns the number of latent variables added to the structure so far.

```
int gdag_status(gdag * d)
```

Returns the *status* of `d` (explained in `gdag.h`). Again, this is mainly for debugging purposes.

```
gsl_vector * gdag_mean(gdag * d)
```

Returns a pointer to the mean vector of the processed structure, `d`.

```
SPMAT * gdag_chol(gdag * d)
```

Returns a pointer to the Cholesky decomposition of the precision matrix — currently a Meschach sparse matrix. N.B. This function is unsupported! Use this only for debugging purposes, and only if you are familiar with Meschach. In future versions, this will return something different (eg. the handle of a Sparse BLAS matrix). You have been warned...

```
double gdag_var(gdag * d,gdag_usv * alpha)
```

Returns the variance of the quantity $\alpha'x$.

```
double gdag_ex_sq(gdag * d,gdag_usv * alpha)
```

Returns the expectation of the square of $\alpha'x$ (useful for implementing EM-style algorithms).

```
double gdag_loglik(gdag * d)
```

Returns the log-likelihood of the last sample generated by `gdag_sim`. This can sometimes be useful if the block samples are to be used as part of a Metropolis-Hastings updating scheme (but see below).

```
double gdag_mloglik(gdag * d)
```

Returns the marginal log-likelihood of the observed data (integrated over the distribution of the latent variables). This is especially useful for block-sampling schemes (due to the marginal likelihood formula that Chib refers to as the BMI). This is arguably the smartest bit of `GDAGsim`, but I don't have time to explain why here!

```
double gdag_vloglik(gdag * d,gsl_vector * v)
```

Returns the log-liklihood of the latent vector v. Again, this can be useful for block-samplers. NOTE THAT THE VECTOR v IS DESTROYED BY THIS FUNCTION!

```
void gdag_vector_dump(gsl_vector * v)
```

Prints a GSL vector to standard output.

```
void gdag_matrix_dump(gsl_matrix * m)
```

Prints a GSL matrix to standard output.

```
void gdag_vector_diff(gsl_vector * v)
```

Computes the one-step difference of the GSL vector v.

```
double gdag_sqr(double x)
```

Returns the square of x.