JAGS Developers Manual

Martyn Plummer

November 7, 2010

Contents

Chapter 1

Introduction

This is currently a collection of notes on working with the JAGS source. It will eventually grow into an explanation of how to extend the capabilities of JAGS by writing new modules.

Chapter 2

Working with the CVS repository

The JAGS source code is held in a CVS repository. Clear instructions on how to access the CVS source are given at the project web site `http://sourceforge.net/projects/mcmc-jags`. To access the instructions, click the link marked "develop" then the "Code" tab and select "CVS" from the pull-down menu.

You need a complete installation of GNU autotools (autoconf, automake, and libtool) to work with the CVS source, since all non-essential files have been stripped out of the repository. You must build local versions of these files by changing directory into the top-level source directory and typing

```
autoreconf -fi
```

Your source tree is then ready to work with.

The CVS repository also excludes some `C++` source files that are included in the source tarball. These files are re-created in the build tree by the GNU tools `flex` and `bison`. You must also have these tools installed if you are using the CVS repository. Note that the standard unix versions of these tools – `lex` and `yacc` – are not sufficient, and you must have an up-to-date version of `flex`.

Once you have checked out a CVS tree, you can keep it up to date with

```
cvs update -Pd
```

You may occasionally need to rerun the autoreconf function when files are added, removed, or moved within the repository.

I recommend keeping one or more build directories that are separate from the source directory. I have several build directories for JAGS configured in different ways: one standard one for testing the BUGS examples, one with no optimization for debugging, another statically linked one for profiling, and so on.

Chapter 3

Testing the Installation

The classic bugs are available in the CVS module “examples”. They can also be downloaded in a tarball from the JAGS home page. There are two sub-directories: “vol1” and “vol2”. Within each sub-directory you can test the installation with

```
make check
```

To test a subset of examples, set the environment variable `EXAMPLES`:

```
make check EXAMPLES="blocker bones"
```

If you are not using a GNU system, you may need to use GNU make (`gmake`).

You need to have R installed in order to check the output of JAGS against the benchmarks. If you have the `rjags` package installed, then you may also test the rjags package with

```
make Rcheck
```

Chapter 4

Directory structure

The JAGS source is divided into three main directories: `lib`, `modules`, and `terminal`. The `lib` directory contains the JAGS library, which contains all the facilities for defining a Bayesian graphical model in the BUGS language, running the Gibbs sampler and monitoring the sampled values. The JAGS library is divided into several convenience libraries

sarray which defines the basic SArray class, modelled on an S language array, and its associated classes.

function which defines the interface for functions and the `FuncTab` class that allows you to reference them by name.

distribution which defines the interface for distribution and the `DistTab` class that allows you to reference them by name.

graph which defines the various Node classes used by JAGS when constructing a Bayesian graphical model, as well as the `Graph` class which is a container for nodes.

sampler which defines the interface for Samplers, which update stochastic nodes in the graph.

model which defines all the classes needed to create a model, including monitor classes.

compiler which contains the Compiler class and a number of supporting classes designed for an efficient translation of a BUGS-language description the model into a `Graph`.

rng which defines the interface for random number generators (RNGs) and the factories that create them.

util which contains some utility functions used in the rest of the JAGS library.

The `Console` class provides a clean interface to the JAGS library. The member functions of the `Console` class conduct all of the operations one may wish to do on a Bayesian graphical model. They are designed to catch any exceptions thrown by the library and print an informative message to either an output stream or an error stream, depending on the result.

The `modules` directory contains the source code for JAGS modules, which contain concrete classes corresponding to the abstract classes defined in the JAGS library.

The `terminal` directory contains the source code for a reference front end for the JAGS library, which uses the Stata-like syntax described in the user manual

Chapter 5

Debugging and Profiling

Debugging and profiling tools are essential for finding bugs and bottlenecks in the code. The most important tools are `gdb`, `valgrind`, `gprof` and `oprofile`.

5.1 Debugging with gdb

JAGS can be run from within the GNU debugger `gdb` by typing

```
jags -d gdb
```

To run a script, type

```
r <scriptname>
```

at the gdb prompt.

Debugging of optimized C++ code is not easy, especially when using code from the Standard Template Library (STL). Unless you speak fluent STL, you will need to work with a non-optimized build of JAGS. Using `gcc` this is done with the following build flags.

```
CXXFLAGS="-g -O0"
CFLAGS="-g -O0"
```

It is helpful to keep a separate non-optimized build directory for occasions when you need to use a debugger.

It is not possible to set a break point in a module before it has been dynamically loaded. To do so, run JAGS by typing "r" at the `gdb` prompt, then control-C to return to the gdb prompt after the modules have been loaded.

5.2 Debugging with valgrind

Valgrind (`www.valgrind.org`) is a memory profiler and debugger. To run JAGS through valgrind, type

```
jags -d valgrind <script-file>
```

If you need to pass options to valgrind, enclose these in quotes

```
jags -d 'valgrind --leak-check=full' <scriptfile>
```

JAGS will run very slowly inside valgrind, and will use more memory, so its use should be limited to small test programs.

5.3 Profiling with gprof

The GNU profiler `gprof` does not debug dynamic libraries. It is therefore not very useful for a standard installation of JAGS, since almost all of the functionality is contained in the jags library, the jrmath library, and the modules. However, you can build a statically linked version in which the libraries and modules are folded into the executable `jags-terminal`. To build this version of JAGS, with profiling information for `gprof`, use the following configure options:

```
CXXFLAGS="-g -O2 -pg" \
CFLAGS="-g -O2 -pg" \
/path/to/JAGS/configure --disable-shared
```

Whenever JAGS is run, it will create a file `gmon.out` in the working directory that can be used for profiling with `gprof`.

5.4 Profiling with oprofile

Oprofile (`oprofile.sourceforge.net`) is a linux-based profiler that runs as a daemon. Unlike `gprof` it does not require any special configuration options, and can be used to debug dynamic libraries.

You must be root to start the profiler

```
opcontrol --no-vmlinux
opcontrol --start
```

Then, as a normal user, you may run a model and dump the profiling information to file with

```
opcontrol --dump
```

To see how much time JAGS is spending in the functions in a module type

```
opreport -l /usr/local/lib/JAGS/modules/bugs.so | less
```

The opreport command gives copious information, so you will need to redirect the output to a file or, as in this example, a pager. The same command works for the main JAGS library

```
opreport -l /usr/local/lib/libjags.so
```

More detailed profiling information can be obtained with the `opannotate` command, provided that JAGS has been compiled with debugging symbols. The command

```
opannotate --source /usr/local/lib/JAGS/modules/bugs.so | less
```

reconstructs the source code and gives annotations in a column in the left hand side counting the number of samples in each function, block or line. This can be useful for finding bottlenecks in the code.

Oprofile will continue to accumulate samples from multiple runs of JAGS, although the output of the `opreport` and `opannotate` commands will not change until you dump the data again with `opcontrol --dump`. If you do not wish to see the cumulative samples from multiple runs – for instance if you have modified the JAGS code and want to check that a previous bottleneck has been removed – then you can clear the existing data collection by typing, as root

```
opcontrol --reset
```