Extending JAGS: A tutorial on adding custom distributions to JAGS (with a diffusion model example)

Dominik Wabersich[1,2] and Joachim Vandekerckhove[1]
[1]Department of Cognitive Sciences, University of California, Irvine
[2]Department of Psychology, University of Tübingen, Germany

Abstract

We demonstrate how to add a custom distribution into the general-purpose, open-source, cross-platform graphical modeling package JAGS ("Just Another Gibbs Sampler"). JAGS is intended to be modular and extensible, and modules written in the way laid out here can be loaded at runtime as needed and do not interfere with regular JAGS functionality when not loaded. Writing custom extensions requires knowledge of `C++`, but installing a new module can be highly automatic, depending on the operating system. As a basic example, we implement a Bernoulli distribution in JAGS. We further present our implementation of the Wiener diffusion first passage time distribution, which is freely available via `https://sourceforge.net/projects/jags-wiener/`.

Keywords: Custom distributions; JAGS; Bayesian; diffusion model; HDM

Introduction

JAGS ("Just Another Gibbs Sampler"; Plummer, 2003) is a free software package for analysis of Bayesian models. JAGS uses a suite of Markov chain Monte Carlo methods—general-purpose stochastic simulation methods—to draw samples from the joint posterior distribution of the parameters of a Bayesian model. These samples can then be used to draw inference regarding the model parameters and model fit. JAGS uses a dialect of the BUGS language (Thomas, Spiegelhalter, & Gilks, 1992; Lunn, Jackson, Best, Thomas, & Spiegelhalter, 2012) to express directed acyclic graphs, a mathematical formalism to define joint densities. What makes JAGS particularly interesting, however, is that it is designed

Correspondence concerning this article may be addressed to JV joachim@uci.edu or DW dominik.wabersich@gmail.com. This project was partially supported by a grant from the National Science Foundation's Measurement, Methods, and Statistics panel to JV, and a travel grant from German Academic Exchange Service (PROMOS) to DW. We are endebted to Martyn Plummer for helpful comments on this manuscript and helping us with compilation issues. We also thank two anonymous reviewers and the action editor for constructive comments on an earlier draft of this article.

to be extensible with user-defined functions, monitors, distributions, and samplers.[1]

The goal of the present paper is to provide a how-to guide on writing custom extensions for JAGS. While JAGS is envisioned as a flexible and extensible (modular) framework and contains an infrastructure for customization, at the time of writing no tutorials, technical manuals, or other sources on how to do that were available.

In the section titled *Steps to extending JAGS with a new module*, we describe in detail how custom distributions may be implemented in JAGS. This section will assume some knowledge of the `C++` programming language (and object-oriented programming) on the part of the reader. Throughout, we use the Bernoulli distribution as a didactic example, but JAGS can accommodate all other types of probability distributions (continuous and discrete, univariate and multivariate) with little extra effort.

In the second part of the paper, we present a module implementing a distribution of particular interest to the cognitive science community. The *JAGS Wiener module* (JWM) adds the first passage time distribution of a drift diffusion process to JAGS. We also provide two sanity checks of the JWM: an extensive simulation study, showing good recovery, and an application to a previously analyzed data set, showing parallel results.

Steps to extending JAGS with a new module

In this section, we detail the steps required to write a new module for JAGS. JAGS modules are library files that are located in the JAGS `/modules` directory. These modules can be loaded during JAGS runtime in order to extend its functionality with more functions and distributions or even sampling algorithms or monitors.

Modules are written in `C++`. In order to write a custom module that provides a new distribution, we need to define two new `C++` *classes*: one for the module itself, and one for the distribution we will use. Throughout, we will display much of the required code, as well as templates for files used in building and compiling. The code can be copied from this text, or can be found on our SourceForge archive (and can be used as a template for new modules). The Appendix contains a quick reference table to the required steps.

We use the Bernoulli distribution as an example to illustrate the basics of extending JAGS because the functions that define this distribution are relatively easy to write without the need for calling advanced functions from extra libraries.[2] Using the naming scheme[3] from JAGS, we will name our module class `BERNModule` and the distribution class `DBern`.

[1] Another desirable feature of JAGS is that it is truly free software, as it is open-source and released under a free, copyleft license—it is "free-as-in-speech" as opposed to merely "free-as-in-beer". The advantages of open-source scientific software are many: not only does it enable researchers to check underlying functions for accuracy and appropriateness, but it also contributes to the exchange of information and reproducibility, even across platforms. In the particular case of JAGS, its modular and open-source nature enables the potential formation of a contributor community not unlike that of the R system.

[2] Note, however, that JAGS includes a version of R's math library, `JRmath.h`, providing many basic functions that can be useful in writing extensions (e.g., there is a function for the normal density and distribution). It is also possible to link to third-party libraries such as the LAPACK and BLAS libraries, for optimized performance.

[3] In the JAGS naming scheme, a new distribution is given a short name (in our case Bern) and a long name (in our case Bernoulli). Module classes will be called `<SHORTNAME>Module`, distribution classes will be called `D<shortname>`, the module namespace will be called `<longname>`. Naming schemes for filenames are given in the Appendix.

Step 1: Defining a module class

We start by creating a specific *module class* that is a child of the base `Module` class, which JAGS provides. To make a new module class, create a new file like the one displayed in Box 1—this is our `Bernoulli.cc` module file. Place it in `src/`, a subfolder of your working directory for this project.

In the first lines, we reference two header files containing pre-made functions we will use. Apart from the JAGS header file `Module.h` (which is available through JAGS), we will include the Bernoulli distribution class file, which we will create at a later point.

The new functions and classes we create need to be defined within a single *namespace*—a named environment that is used to group programmatic entities that are frequently used together. The third line of Box 1 names the namespace.

Then, we define the module class, which we will call `BERNModule`. `BERNModule` is a subclass of `Module` with *public inheritance*, and needs to define two functions—both are required.

Constructor and destructor for the `BERNModule` *class.* In Box 1, the constructor function `BERNModule()` is called whenever an object of our `Bernoulli` class is instantiated.

The constructor function `BERNModule()` needs to do two things. First, it needs to instantiate a generic Module by calling the constructor function of the parent `Module` class. It calls that generic constructor with a single argument: the name of the module (as a string). Second, our new module's constructor needs to call the `insert` function with as input a new instance of the class describing the new distribution (in our case, `DBern`, which is detailed below).

The destructor function `~BERNModule()` needs to remove the instance. To do so, we use a `for` loop over all instances of the module.[4]

The last line of Box 1 actually calls the new module's constructor function and instantiates one `BERNModule`.

The module class `BERNModule` is now complete. The second step in creating the JAGS module is to define the `DBern` scalar distribution, which is inserted with the constructor call in Box 1.

Step 2: Defining a scalar distribution

A new scalar distribution for JAGS is implemented through a new `C++` class. The new class will need, in addition to its own constructor function, the following four specific functions:

- `logDensity`, a function to calculate and return the log-density
- `randomSample`, a function to draw and return random samples
- `typicalValue`, a function to return typical values

[4]For the present example, it is not strictly required to loop over many instances because the constructor function only ever makes a single instance, but in general it is possible for a module to instantiate more than one distribution on loading, so we write a more general destructor function that it will remove all instantiated objects on unloading. Our current code will always loop over only one element, the instantiated `DBern` object.

```
#include <Module.h>            // include JAGS module base class
#include <distributions/DBern.h> // include Bernoulli distribution class

namespace Bernoulli { // start defining the module namespace

  // Module class
  class BERNModule : public Module {
    public:
      BERNModule(); // constructor
      ~BERNModule(); // destructor
  };

  // Constructor function
  BERNModule::BERNModule() : Module("Bernoulli") {
    insert(new DBern); // inherited function to load objects into JAGS
  }

  // Destructor function
  BERNModule::~BERNModule() {
    std::vector<Distribution*> const &dvec = distributions();
    for (unsigned int i = 0; i < dvec.size(); ++i) {
      delete dvec[i]; // delete all instantiated distribution objects
    }
  }

} // end namespace definition

Bernoulli::BERNModule _Bernoulli_module;
```

Box 1: The `Bernoulli.cc` module definition file.

- `checkParameterValue`, a function to check if the given parameter values are in the allowed parameter space

We now create two new files in a subdirectory called `src/distributions/`. Our files will be `src/distributions/DBern.h`, which will contain the class prototype and `src/distributions/DBern.cc`, which will contain the actual computational implementation.

DBern.h. Box 2 contains our `src/distributions/DBern.h`. Near the top of the code, we include the (JAGS) parent class `ScalarDist` in order to be able to use it as a base class and inherit from it. Since we will be adding this to the namespace of our new module, we use the same name for it here as in Box 1. What follows in Box 2 are the prototypes of (1) the constructor function (which has the same name as the class), (2) the

```
#ifndef DBERN_H_
#define DBERN_H_
#include <distribution/ScalarDist.h> // JAGS scalar distribution base
    class

namespace Bernoulli {

class DBern : public ScalarDist // scalar distribution class
{
  public:
    DBern(); // constructor
    double logDensity(double x, PDFType type,
                      std::vector<double const *> const &parameters,
                      double const *lower, double const *upper) const;
    double randomSample(std::vector<double const *> const &parameters,
                        double const *lower, double const *upper,
                        RNG *rng) const;
    double typicalValue(std::vector<double const *> const &parameters,
                        double const *lower, double const *upper) const;
    /** Checks that pi lies in the open interval (0,1) */
    bool checkParameterValue(std::vector<double const *> const &
        parameters) const;
    /** Bernoulli distribution is discrete valued */
    bool isDiscreteValued(std::vector<bool> const &mask) const;
};

}
#endif /* DBERN_H_ */
```

Box 2: The Bernoulli scalar distribution class header file DBern.h.

four required functions (which will be identical for other scalar distributions[5]), and (3) the function `isDiscreteValued`, which will tell JAGS that the function has a discrete domain.

The `logDensity` function takes five input arguments:

1. The first argument (`double x`) contains the data point at which the function is to be calculated.

2. The second argument, of the JAGS-specific type `PDFType` (defined in Distribution.h, which is a parent of ScalarDist.h), defines different ways of calculating the log density. During JAGS runtime, depending on *how* the node is used, this argument can take different values. Possible values are `PDF_FULL` (for when the node is used for the full evaluation of the likelihood, when the parameters and sampled values are not constant),

[5]These functions are implemented as `virtual` in the base class and need to be defined in the child class, with the same name and arguments. It is possible to introduce new functions here.

`PDF_PRIOR` (for when parameters are constant, so that terms that depend on them can be omitted from the calculations), and `PDF_LIKELIHOOD` (used when the sampled value is constant, so that terms that depend on it can be omitted from the calculations). The `PDFType` can simply be ignored (as it is in our example).

3. The third argument (`std::vector<double const *> const &parameters`) is a reference to a vector that contains pointers for the parameters of the distribution. In our case, the vector consists of a pointer to `pi` (the probability parameter of the Bernoulli distribution; see Eq. 1).

4. The final two arguments (`double const *lower` and `double const *upper`) are pointers to the lower and upper boundary of the distribution in case of truncated sampling (see `lib/graph/ScalarStochasticNode.cc` in the JAGS code to find the call to the `logDensity` function). In our example these are not used, and these arguments are ignored.

The arguments used in the other functions have similar definitions. One final input argument, which appears only in the `randomSample` function, is the JAGS-specific `RNG` type argument (`RNG *rng`). It points to an `rng` object that provides the following functions to generate random numbers:

```
rng->uniform()
rng->exponential()
rng->normal()
```

These random number generating functions can be used to drive samplers for any distribution.

DBern.cc. Now we need to implement the four functions and the constructor function prototyped in Box 2. The distribution function of a Bernoulli distributed random variable X, used as the basis for the computations, is in Equation 1.

$$\begin{aligned} P(X = 0) &= (1 - \pi) \\ P(X = 1) &= \pi \end{aligned} \tag{1}$$

Note that, while these four functions need to be present in the code, not all are strictly speaking required to run an analysis. If drawing random samples from the distribution is not a functionality your analyses require, it is possible to let `randomSample` return `JAGS_NAN`. This will cause JAGS to throw an error if a model is defined that requires random sampling from the new distribution, but it does not affect using the new node as a likelihood. Similarly, if `typicalValue` is not implemented, JAGS won't run a sampling process if no starting values are supplied, but work otherwise. The `checkParameterValue` function is a prerequisite for `logDensity`, however.

Further, as the Bernoulli distribution is a discrete valued distribution, we redefine the function `isDiscreteValued` and let it return `true` (implementation of that particular function is not necessary for continuous distributions: it will be inherited from the base class and return `false` by default).

The implementations of the required functions are provided, with comments, in Boxes 3, 4, and 5.

Constructor and destructor for the `DBern` *class.* Finally, we need to set up constructor and destructor functions for our `ScalarDist` class. The constructor function calls the constructor for `ScalarDist` with three input arguments (as defined in the JAGS header file `distribution/ScalarDist.h`). The first input argument is `std::string const &name` and is used to give the distribution node a name, which can later be used in the JAGS model file to define a node with this distribution (in this example, the distribution node will be called "dbern2"). The second input argument of the constructor is `unsigned int npar` and is used to define the number of arguments the new distribution node can take (that is one parameter, the probability π, for this example). The third argument of the constructor is `Support support`, a JAGS-specific variable that defines the support of our new distribution. In our case, `support` is `DIST_PROPORTION` indicating a distribution that spans from 0 to 1. Other valid options for this argument are `DIST_POSITIVE` (for a distribution that has support only on the positive real half-line), `DIST_UNBOUNDED` (for a distribution that spans the entire real line) and `DIST_SPECIAL` (for other domains).

Step 3: Building the module

To configure and build our module, we used the GNU Autotools and libtool[6] (these are the same tools used in building the JAGS project), according to the steps below.

We will give a brief overview, together with all the information necessary to organize the build process of the module described in this article. However, it cannot be guaranteed that these building methods will stay the same over time[7], so that the build process may be slightly different on newer machines and operating systems. However, build processes are central to software development and are typically very well documented for new architectures and operating systems.

Step 3.1: Creating a file structure. First, make sure that there exists a project directory with in it a subdirectory called `src/`, and that all the source files (that contain the `C++` code for the module) are present in the `src/` subdirectory. For the given example, that would be:

- The module main file: `src/Bernoulli.cc`
- In the sub-subdirectory `src/distributions/`, the distribution class files: `DBern.h` and `DBern.cc`

Now create one `configure.ac` file in the main directory of the project and one `Makefile.am` in every directory of the project (including all subdirectories). The `configure.ac` file will contain information on how to correctly configure the project. The `Makefile.am` files contain the necessary arguments for building, including libraries to link to and the paths to the include directories (and more).

As the newly created module uses functions from the JAGS library, it needs to be linked to the appropriate libraries and JAGS' include files need to be available (i.e., JAGS needs to be installed properly).

[6]GNU Autotools and libtool are well-documented, manuals are available via `http://www.gnu.org/software/autoconf/`, `http://www.gnu.org/software/automake/`, `http://www.gnu.org/software/libtool/`

[7]Fortunately, major changes to these kinds of tools are very rare—it is in the interest of the developers to keep this process simple and relatively stable over time.

```
#include <config.h>   // system configuration file, created by Autoconf
                      // and defined in configure.ac
#include "DBern.h"    // header file, containing class prototype
#include <rng/RNG.h>  // provides random functions
#include <util/nainf.h> // provides na and inf functions etc.

#include <cmath>      // library for standard math operations

using std::vector;    // vector is used in code
using std::min;       // min is used in code
using std::max;       // max is used in code

#define PROB(par) (*par[0]) // makes code more readable

namespace Bernoulli { // module namespace

// The constructor function:
// also calls the constructor of the base class.
// The base class constructor takes as input arguments:
// 1 - the name of the distribution node, as used in the BUGS code
// 2 - the number of arguments of that node
// 3 - distribution type
DBern::DBern() : ScalarDist("dbern2", 1, DIST_PROPORTION)
{}

// missing functions, shown in Box 4, go here

}
```

Box 3: The DBern.cc file. Note that we need to include rng/RNG.h and util/nainf.h from the JAGS library, to provide the RNG struct and the JAGS_* constants, as well as the jags_* functions. cmath.h is needed for standard math operations.

```
// this function checks if the probability parameter of the Bernoulli
// distribution lies between 0 and 1, and returns false if it doesn't
bool DBern::checkParameterValue (vector<double const *> const &
    parameters) const
{
    return (PROB(parameters) >= 0.0 && PROB(parameters) <= 1.0);
}

// this function calculates the log(density)
// at a given value (data point),
// for given parameter values
// (in this case for a given probability pi)
double DBern::logDensity(double x, PDFType type,
       vector<double const *> const &parameters,
       double const *lbound, double const *ubound) const
{
    double d = (x ? PROB(parameters) : 1 - PROB(parameters));
    return (d == 0) ? JAGS_NEGINF : log(d);
}

// this function is used for drawing random samples for given parameter
// values (in this case for a given probability pi)
double DBern::randomSample(vector<double const *> const &parameters,
        double const *lbound, double const *ubound,
        RNG *rng) const
{
    return (rng->uniform() < PROB(parameters)) ? 1 : 0;
}

// this function returns a typical value of the distribution
// (in this case for a Bernoulli distribution with given probability pi)
double DBern::typicalValue(vector<double const *> const &parameters,
        double const *lbound, double const *ubound) const
{
    return (PROB(parameters) > 0.5) ? 1 : 0;
}
```

Box 4: Functions from the DBern.cc file.

```
// this redefined function is needed, as the Bernoulli distribution is
// discrete
bool DBern::isDiscreteValued(vector<bool> const &mask) const
{
    return true;
}
```

Box 5: Further functions from the DBern.cc file.

Step 3.2: Creating the configure.ac *file.* Box 6 and Box 7 together show the configure.ac file. All the functions used in this file are documented in the documentation of Autoconf (see Footnote 6). This file contains instructions on how to prepare the module for building (e.g., which compiler to use). Much in the file can remain unchanged.

The AC_INIT directive takes as its first argument the package name, as second argument the package version, as third argument a contact email address (for bug reports) and as fourth argument the name for the .tar file. JAGS_MAJOR and JAGS_MINOR should be edited to the current JAGS version that is used. AC_CONFIG_SRCDIR should contain a path to a (any) file of the package. The AC_CONFIG_FILES directive at the end tells the configuration process where to create Makefiles using the Makefile.am files as a template.[8] The libltdl directory and its Makefile.am will be created automatically. If one follows all directions given here, nothing else has to be changed.

Step 3.3: Creating several Makefile.am *files.* Box 8 shows the Makefile.am file in the main directory. The configuration process will use a subdirectory called m4/, which needs to be created manually. The file also needs to reference the building subdirectories: the src/ directory containing the actual source, and a libtool/ directory, which will be created automatically.

Box 9 shows the Makefile.am file in the src/ directory. It contains the instructions to produce the module library.

The first directive, SUBDIRS, points to the subdirectories to be used; for JAGS modules this will typically be distributions. The next directive defines the JAGSMODULE libtool library to be created. The library file will be named *<longname>*.la (e.g., Bernoulli.la). The next directive defines where to find the library source code. The following directive gives the C++ compiler additional options. The -I path option passes to the compiler the path of JAGS' include directory (a subdirectory of the system's include path indicated by $(includedir)).

The next set of directives tell the compiler which object code (i.e., compiled code) should be linked to the libraries. For the Bernoulli module this is the object code for the Bernoulli distribution and the JAGS library. The -ljags argument searches for the JAGS

[8] The Makefiles files are system-specific and necessary for the build process, whereas the Makefile.am files are system-independent. They contain directives to generate appropriate Makefiles for the system to build the source.

```
1  dnl Process this file with Autoconf to produce a configure script.
2
3  AC_PREREQ([2.68])
4
5  AC_INIT([JAGS-BERN],[1.1],[emailof@auth.or],[JAGS-BERN-MODULE])
6  JAGS_MAJOR=3
7  JAGS_MINOR=3
8  AC_SUBST(JAGS_MAJOR)
9  AC_SUBST(JAGS_MINOR)
10
11 AC_CANONICAL_HOST
12 dnl The following lines check if the required files exist
13 AC_CONFIG_SRCDIR([src/distributions/DBern.cc])
14 AC_CONFIG_MACRO_DIR([m4])
15 dnl The configure process creates a header file called config.h
16 AC_CONFIG_HEADERS([config.h])
17 AM_INIT_AUTOMAKE
18
19 dnl libtool and ltdl configuration
20 LT_PREREQ(2.2.6)
21 LT_CONFIG_LTDL_DIR([libltdl])
22 LT_INIT([dlopen disable-static win32-dll])
23 LTDL_INIT([recursive])
24
25 dnl Indicate C++
26 AC_PROG_CXX
```

Box 6: The first part of the `configure.ac` file (file continues in Box 7). Lines beginning with `dnl` ("delete to new line") denote comments.

library in the system library directory (note that since Windows organizes its libraries differently and the JAGS library will be in a slightly different location, we need to add a Windows-specific part here). Had we used additional external libraries, then we could add thes here as well. For example, we could link to the `JRmath.h` library, with `-ljrmath` (and `-ljrmath-0` in the Windows part).

Box 10 shows the `Makefile.am` file of the `src/distributions/` directory. It contains the instructions to build the Bernoulli distribution code, which will then be included in the package library. The sub-library `distributions/Bernoullidist.la` is named on line 1 of Box 10. This name must be matched in the `src/Makefile.am` (line 9 of Box 9). In Box 10, the directive `noinst_LTLIBRARIES` will cause the sub-library to be built, but not installed (i.e., the code will be compiled but not copied to a system location). The sub-library will be incorporated in the module library by `src/Makefile.am`. As before, the `Bernoullidist_la_CPPFLAGS` directive takes arguments to be passed to the compiler. In

```
dnl Optionally, reference the Rmath library
AC_DEFINE(MATHLIB_STANDALONE, 1, [Define if you have standalone R math
    library])

case "${host_os}" in
  mingw*)
    win=true ;;
  *)
    win=false ;;
esac
AM_CONDITIONAL(WINDOWS, test x$win = xtrue)

jagsmoddir=${libdir}/JAGS/modules-${JAGS_MAJOR}
AC_SUBST(jagsmoddir)

AC_CONFIG_FILES([
  Makefile
  libltdl/Makefile
  src/Makefile
  src/distributions/Makefile
])
AC_OUTPUT
```

Box 7: The second part of the `configure.ac` file (continued from Box 6).

```
ACLOCAL_AMFLAGS = -I m4
SUBDIRS = libltdl src
```

Box 8: The `Makefile.am` file.

this case, it passes the include directories where the header files for the code can be found.

With all the files organized as described in this section, we can now proceed to configure and build the module using the steps below. We first provide the steps for building on a unix-like (i.e., Mac or linux) environment, in which building is much easier than under Windows systems. Unfortunately, building on Windows systems is a tedious and involved affair. In particular, the `autoreconf -fvi` command on a Windows system is not possible with standard emulators like MinGW. It is, however, possible to perform the first two steps below on a Mac/linux system, then creating a source .tar file, copy and extract that on a Windows machine, and then using the MinGW environment with msys to configure and build the source. We will outline the full building process for Mac/linux systems, and then describe the steps needed to compile for Windows systems.

Step 3.4a: Building the module (Mac/linux). To proceed in a Mac/linux system, follow these steps:

```
SUBDIRS = distributions

jagsmod_LTLIBRARIES = Bernoulli.la

Bernoulli_la_SOURCES = Bernoulli.cc

Bernoulli_la_CPPFLAGS = -I$(includedir)/JAGS

Bernoulli_la_LIBADD = distributions/Bernoullidist.la
if WINDOWS
Bernoulli_la_LIBADD += -ljags-$(JAGS_MAJOR)
else
Bernoulli_la_LIBADD += -ljags
endif

Bernoulli_la_LDFLAGS = -module -avoid-version
if WINDOWS
Bernoulli_la_LDFLAGS += -no-undefined
endif
```

Box 9: The src/Makefile.am file.

```
noinst_LTLIBRARIES = Bernoullidist.la

Bernoullidist_la_CPPFLAGS = -I$(top_srcdir)/src -I$(includedir)/JAGS

Bernoullidist_la_LDFLAGS = -no-undefined -module -avoid-version

Bernoullidist_la_SOURCES = DBern.cc

noinst_HEADERS = DBern.h
```

Box 10: The src/distributions/Makefile.am file.

• `autoreconf -fvi`: this command will generate a number of auxiliary files that are necessary for the configuring and building process.

• `./configure`: this command configures the source package for building on your system.

• `make`: this command compiles the source code into system-specific object code.

• `make install`: this command creates local copies of the object code, placing it in the correct locations in the filesystem (this step typically requires administrator/superuser privileges). In our case, this command will copy the module library to an appropriate location in the system where JAGS can find it and load it.[9] Alternatively, this can be done manually by copying the libraries to the JAGS directory that contains the other module libraries.

Additionally, after running the first two commands (the configuration process), one can easily create source .tar files with `make dist-gzip` or `make dist-bzip2` (or whichever format one prefers and is supported by the Makefile routines).

Step 3.4b: Building the module (Windows). Building under Windows follows a similar process, with some added steps. First, create the Windows libraries.

We start with installing MinGW (MinGW installer including msys) and the TDM-GCC Compiler Suite, which can be obtained via `http://www.mingw.org` and `http://tdm-gcc.tdragon.net`.[10] In the rest of this paragraph, it will be assumed that MinGW and TDM-GCC are installed in their default directories.

Second, delete all `*.dll.a` files in the TDM-GCC directory, to force the compiler to link to the static libraries (the `*.dll` files). This is necessary to build libraries that will work on systems that don't have TDM-GCC.

Third, change the path in the file `C:\mingw\msys\1.0\etc\fstab` from `C:\mingw /mingw` to `C:\MinGW64 /mingw`.[11] This is necessary in order to use the TDM-GCC compilers instead of the standard MinGW compilers.

Now the actual configuration and building process can commence. The module needs to locate the JAGS include files and the JAGS libraries. As Windows has no standard path where to look for these files, this needs to be done manually. Edit the code in Box 11 to reflect the paths to the JAGS libraries (with the `-L` option) and the JAGS include files (with the `-I` option). The code is given for a standard installation of JAGS 3.3.0.

Start msys (the MinGW shell), extract the source .tar file in any directory, navigate to that directory by using the `cd` command and run the appropriate commands given in Box 11. Executing these commands will create a number of files in the `src/.libs/` directory. Now copy those files that are named after your module and ending in `.dll`, `.dll.a` and `.la` to your JAGS modules directory (where all the other module libraries are located). You can

[9] Unfortunately, this command does not work under Windows. It is possible, however, to create installers for Windows machines that include precompiled binaries and copy them to the correct locations. One easy-to-use third-party tool to create such installers is NSIS (`http://nsis.sourceforge.net`).

[10] Note that when compiling a module it is necessary to use the same tool chain as was used to compile JAGS, which at the time of writing means TDM-GCC v4.6.1. Newer versions of TDM-GCC will probably cause errors (and note also that older versions of TDM-GCC will by default update themselves during installation unless the user unchecks that option). For future releases of JAGS, check the JAGS Manuals for information on which compiler version was used.

[11] These are the default paths. They might be different, depending on where the software was installed.

```
# For building 32bit binaries
CXX="g++ -m32" \
  ./configure LDFLAGS="-L/c/Progra~1/JAGS/JAGS-3.3.0/i386/bin" \
  CXXFLAGS="-I/c/Progra~1/JAGS/JAGS-3.3.0/include"
make

# For building 64bit binaries
CXX="g++ -m64" \
  ./configure LDFLAGS="-L/c/Progra~1/JAGS/JAGS-3.3.0/x64/bin" \
  CXXFLAGS="-I/c/Progra~1/JAGS/JAGS-3.3.0/include"
make
```

Box 11: Building the source code under Windows. Note that if you build libraries for both 32bit and 64bit you need to run `make clean` between the two building processes.

also use these files to create an installer (e.g., with NSIS; see Footnote 9).

After installing the new module, JAGS will recognize a new stochastic node, to be used as follows: `x ~ dbern2(pi)`, where the string "dbern2" is defined by us in Box 3 (line 23) and its parameter is interpreted through line 13.

Extending the module with logical nodes

Adding a custom distribution is only one of several possible extensions a JAGS module can provide. One other possible extension is the creation of logical nodes (i.e., deterministic functions, rather than stochastic ones; logical nodes are useful for transforming variables within a JAGS model file, and are always preceded by an assignment operator `<-`.).

Here we demonstrate how to supplement our Module example with a logical node that calculates the Bernoulli log-density at a given data point for a given parameter set.[12] To do so, we will write a *scalar function*. Much like the scalar distribution, the `ScalarFunction` class requires the implementation of two functions:

- `evaluate`: the function which does the calculations and returns the result.
- `checkParameterValue`: a function to check if the parameter values are in the domain of the function.

In the `src/` directory of our module, add a subdirectory `functions/` and in it the two files: `LogBernFun.h` and `LogBernFun.cc`. Box 12 and Box 13 show examples of such files.

After the function class is ready, we need to add it to the `Bernoulli.cc` module by loading it with the constructor and deleting it with the destructor (as in Box 1). See Box 14 for the lines to add.

[12] Of course, it is possible to write a module that contains *only* a logical node without tacking it on to another module.

```
1  #ifndef BERN_FUNC_H_
2  #define BERN_FUNC_H_
3
4  #include <function/ScalarFunction.h> // the base class used
5
6  namespace Bernoulli { // module namespace
7
8  class LogBernFun : public ScalarFunction
9  {
10   public:
11     LogBernFun(); // the constructor function
12
13     // the two necessary functions that have to be implemented
14     bool checkParameterValue(std::vector<double const *> const &args)
           const;
15     double evaluate(std::vector<double const *> const &args) const;
16 };
17
18 }
19
20 #endif /* BERN_FUNC_H_ */
```

Box 12: The src/functions/LogBernFun.h file.

Finally, add an appropriate Makefile.am to the newly created src/functions/ directory. In src/functions/Makefile.am, the sub-library is named (here: Bernoullifunc.la). Also add the name of that sub-library to src/Makefile.am like this: `bernoulli_la_LIBADD += functions/Bernoullifunc.la` (in order to incorporate the sub-library in the common Bernoulli library). Additionally, add the name of the subdirectory (/functions) to the `SUBDIRS` directive in the first line of src/Makefile.am, and add the appropriate directives to generate the Makefile file in the configure.ac file. The required Makefile.am will be similar to that in Box 10 with only small edits (replacing the filenames and the name for the sublibrary in the first line and in the following lines, where it is part of the directive). In the configure.ac file, merely add a line to the last directive to tell Autoconf to create a Makefile in the src/functions/ directory.

Now, once the module is installed, JAGS will recognize a new logical node that can be used as follows: `p <- logbern(x,pi)`, where the name of the node is defined in Box 13 (line 18), and the order of the parameters is defined in lines 10 and 11. This line, when used in a JAGS model file, will store in the variable `p` the log-likelihood under a Bernoulli distribution of data point `x` given parameter `pi`. Storing the log-likelihood at every iteration of the Gibbs sampler can be useful for convergence checks or to compute model fit statistics in the Bayesian framework.

Note that there exist still more ways to extend JAGS. One can, for example, write custom sampling algorithms, or multivariate distributions—fully describing the installation

```
#include <config.h>
#include "LogBernFun.h" // class header file
#include <util/nainf.h> // provides na and inf functions

#include <cmath> // basic math operations

using std::vector; // vector is used in the code
using std::string; // string is used in the code

#define x(par) (*args[0])
#define prob(par) (*args[1])

namespace Bernoulli { // module namespace

// constructor function, also calls the constructor of the base class
// with 2 arguments: the name of the logical node, as used later
// in a model file and the number of arguments
LogBernFun::LogBernFun() :ScalarFunction("logbern", 2)
{}

// checks if the parameter pi lies between 0 and 1
bool LogBernFun::checkParameterValue(vector<double const *> const &args)
    const
{
  return (x(par) == 0.0 || x(par) == 1.0
          && (prob(par) <= 1.0 && prob(par) >= 0.0));
}

// does the computation that the logical node is supposed to do
double LogBernFun::evaluate(vector<double const *> const &args) const
{
  double d = ( (x(par) == 1) ? prob(par) : 1-prob(par) );
  return (d == 0) ? JAGS_NEGINF : log(d);
}

}
```

Box 13: The src/functions/LogBernFun.cc file.

```
// add this to the constructor function
// to instantiate a function object when the module is loaded
  insert(new LogBernFun);

// add this to the destructor function
// to remove the instantiated funtion object when the module is unloaded
  vector<Function*> const &fvec = functions();
  for (unsigned int i = 0; i < fvec.size(); ++i) {
    delete fvec[i];
  }
```

Box 14: The code to add to `src/Bernoulli.cc` file.

procedure of every possible extension would be beyond the scope of a single article. However, the open-source nature of JAGS allows enterprizing researchers to study the implementation of various components of the program[13] (we would recommend the multivariate normal distribution, `DMNorm.cc`, as an example of a multivariate distribution, and the code for the slice sampler, `Slicer.cc`, as a sampler example).

The generic method of creating modules presented here will work for more sophisticated components as well. The main difference between the workflow outlined here and that for a new component will be in the time that needs to be invested in studying the JAGS framework for the appropriate function class.

The next section focuses on the installation—from a end user point of view—of a custom stochastic node that we believe to be of use for the cognitive science community.

The JAGS Wiener module

Few cognitive models have had as much success as the diffusion model for two-choice response times (see Wagenmakers, 2009, for a review). Recently, there has been a boom of research actually applying the diffusion model, since practical, user-friendly software has become available (Vandekerckhove & Tuerlinckx, 2007; Voss & Voss, 2007; Wagenmakers, Van Der Maas, & Grasman, 2007; Wiecki, Sofer, & Frank, 2011). The extension into hierarchical Bayesian modeling provides the most flexible modeling framework yet (Vandekerckhove, Tuerlinckx, & Lee, 2011). However, flexible implementations of the hierarchical diffusion model (HDM) are currently limited to WinBUGS (Lunn, Thomas, Best, & Spiegelhalter, 2000), which is not truly free software (although it is *gratis*[14]) and only runs natively on Windows systems. The present paper hence addresses a collateral need by providing an HDM implementation in an open-source, platform-independent framework.

The JWM is an extension for JAGS and is designed to integrate seamlessly with the existing JAGS platform. It extends JAGS to recognize `dwiener`, the first passage time density of a drift diffusion process as a new stochastic node with four parameters: the boundary separation α, the nondecision time τ, the initial bias β and the drift rate δ. The

[13] In fact, studying the source code is how the present methods came about.

[14] There exists, however, an open-source port of WinBUGS: OpenBUGS (`http://www.openbugs.info`).

Symbol	Parameter	Interpretation
α	Boundary separation	Speed-accuracy trade-off (high α means high accuracy)
β	Initial bias	Bias for either response ($\beta > 0.5$ means bias towards response 'A')
δ	Drift rate	Quality of the stimulus (close to 0 means ambiguous stimulus)
τ	Nondecision time	Motor response time, encoding time (high means slow encoding, execution)

Table 1:: The four main parameters of the Wiener diffusion model, with their substantive interpretations (reprinted with permission from Vandekerckhove, 2009).

psychological interpretations of the four Wiener distribution parameters are summarized in Table 1, but for a more detailed description of the assumptions of the Wiener diffusion model we refer to Vandekerckhove et al. (2011).

Installing and using the module is straightforward, as described below.

Installation

In order to install and run the JWM, a recent version of JAGS is required.[15] JAGS can be freely downloaded from `http://mcmc-jags.sourceforge.net/`, but is also available as a package for various Linux distributions that use the RPM Package Manager, as well as Debian and Ubuntu. For the purposes of the installation procedure, we will assume that a recent version of JAGS is already available.

Because JAGS is designed with the capacity for extension in mind and it is capable of dynamically loading libraries, installing a module does not compromise the regular functioning of JAGS. To activate the extensions, modules have to be loaded explicity by JAGS. To allow the program to locate the new libraries, they need to be installed according to the instructions below.

We have split up the installation instructions between a procedure for Windows systems and one for Linux and Mac systems.

Windows systems. We provide precompiled libraries for Windows that can be readily downloaded from the SourceForge repository. The installation proceeds along the following steps:

1. Obtain the Windows installer from SourceForge at: `https://sourceforge.net/projects/jags-wiener/files`.

2. Execute the installer and make sure to save the libraries at the correct position so that JAGS will be able to find the module. The correct location will be `C:\Program Files\JAGS\JAGS-3.3.0\i386\modules\` (for JAGS Version 3.3.0, the Version number in the path changes for older or newer releases) on 32-bit systems and `C:\Program Files\JAGS\JAGS-3.3.0\x64\modules\` (for JAGS Version 3.3.0) on 64-bit systems (possibly replacing the JAGS root directory with its actual installation directory). The installer just needs the JAGS root directory to install the libraries at the correct position.

[15] The module is written and tested using JAGS version 3.3.0, but works for versions $\geq$ 3.0.0. Note that our compiled binaries may not be compatible with future versions of JAGS.

Linux and Mac systems, compiling from source. The general way to install our module on Linux and Mac systems is to compile the source for your system and then install them. For this operation, some general knowledge of GNU Tools and using a console interface (i.e., the terminal) is required. The following instructions are linux- and Mac-specific. The installation instructions will assume that JAGS is already installed, so that it is possible to link correctly to the JAGS library.

For those interested in compiling under Windows, please note that the MinGW environment with mysys needs to be installed, as well as the TDM-GCC compiler suite. Please see the previous instructions of the subsection titled **Step 3: Building the module** and the `README.win` file, present in the win source directory, for further instructions on compiling under Windows.

1. Get the JWM tar archive, containing the source code, from SourceForge at `https://sourceforge.net/projects/jags-wiener/files` and extract it.[16]

2. `cd` into the directory containing the source code.

3. In case you are compiling the source from a cloned repository, run `autoreconf -fvi` before configuring (not necessary if you use the stable .tar file release).

4. Configure and compile the source code for your system with the following commands in a terminal window: `./configure && make`. When this is done, install the libraries on your system with the following command, which usually requires root privileges: `sudo make install`.

The JWM should now be installed on your system.

Ubuntu Linux, with the Advanced Packaging Tool. If you use Ubuntu Linux, you can alternatively install the JWM with the Advanced Packaging Tool (APT). The authors maintain an online repository (a personal package archive, or PPA, called *cidlab*) from which the module can be downloaded. Adding the PPA, updating APT, and installing the module is achieved with this code:

```
sudo apt-add-repository ppa:cidlab/jwm
sudo apt-get update
sudo apt-get install jags-wiener-module
```

Testing the installation

The successful installation can be confirmed by loading the Wiener module in JAGS. In order to do that, bring up a terminal or command window and start JAGS. Then do the following in the JAGS interface:

```
$ load wiener
Loading module: wiener: ok
```

Alternatively, R users can use the rjags package to interface with JAGS. Loading the module in R works like this:

```
> library(rjags)
```

[16] Instead of the tar archive, you can also get the current developers' version from the web-based mercurial repository, hosted on SourceForge at `https://sourceforge.net/projects/jags-wiener/code`. For normal usage, it is generally recommended to use the stable release, that is, the .tar file.

```
Loading required package: coda
Loading required package: lattice
linking to JAGS 3.3.0
module basemod loaded
module bugs loaded
> load.module("wiener")
module wiener loaded
```

We have also made available a version of the MATJAGS MATLAB-to-JAGS interface that loads extra modules, as specified. The MATJAGS interface is a single MATLAB file that can be downloaded from the SourceForge repository. Documentation is provided within that file.

Using the JAGS Wiener module

With the JWM installed, the `dwiener` stochastic node is now available for use in a model definition. To define that a certain node is distributed according to a Wiener distribution with the four parameters given in Table 1, the following stochastic node can be used:

```
x ~ dwiener(alpha,tau,beta,delta)
```

This syntax is almost identical to that used in the `wiener.odc` extension to WinBUGS as presented in the supplemental material to Vandekerckhove et al. (2011). The main differences to that implementation are that (a) the diffusion coefficient s is set to 1 in our JAGS implementation (rather than 0.1 in the WinBUGS one) and (b) the JAGS implementation takes as third input argument the *relative bias* parameter β (rather than the absolute starting point $\zeta_0 = \alpha\beta$ in the WinBUGS version). Setting the diffusion coefficient s to 1 instead of 0.1 is computationally more efficient and further yields a more natural interpretation of the drift rate parameter (because it now has a range similar to that of a standard normally distributed variate). To convert to a different evidence scale (i.e., a different value for s), multiply the obtained[17] drift rate and boundary separation parameters by s. We use the β parameter instead of the ζ_0 starting point primarily because β has an interpretation on an absolute scale (the unit scale), whereas ζ_0 can only be interpreted relative to the boundary separation α.

Some of the parameters have limited domains: $\alpha, \tau > 0$ and $0 < \beta < 1$. Note that the distribution is implemented as a univariate distribution. To use the distribution, choice response time data should be coded in such a way that error response times (or whichever response type is associated with the lower boundary) are given *negative* values. Specifically:

$$x = \begin{cases} RT & \text{if correct} \\ -RT & \text{if error} \end{cases}$$

For a worked example on how the module can be used, see the examples directory in the JWM repository on SourceForge, or download a zipfile of the example from SourceForge at `https://sourceforge.net/projects/jags-wiener/files`.

[17] Alternatively, s can be coded directly in the BUGS file, using the stochastic node as follows: `y ~ dwiener(alpha/s,tau,beta,delta/s)`.

Testing of the JAGS Wiener Module

In this section, we report results of some of the extensive testing of our JAGS Wiener Module, which we created and built according to the instructions above. The first part describes results from a numerical simulation experiment, while the second part describes an application to a benchmark data set.

Parameter recovery simulations

To affirm the accuracy of our implementation of the Wiener distribution, we ran a comprehensive numerical experiment. Using the DMAT toolbox (Vandekerckhove & Tuerlinckx, 2008) we generated data from known parameter sets and then used JAGS to recover the parameters. In all cases, we generated $1,000$ data sets with three conditions, which differed in their drift rates only. The three drift rates were always $(-3.0, 1.0, 4.0)$. The boundary separation was either 0.6, 1.2, or 3.0, the bias was either 0.3, 0.5, or 0.7, and the nondecision time was always 0.4s, yielding a total of nine distinct parameter sets. Using these parameters, we generated data sets with 200 data points per condition. We then used JAGS to estimate parameters. Our estimate was the posterior mean obtained after running four chains for $2,000$ iterations and discarding the first $1,000$ as burnin and using no thinning, for a total of $4,000$ samples.

In only one case (0.01%) was the $\hat{R} < 1.05$ criterion for convergence (see Gelman, Carlin, Stern, & Rubin, 2004) violated. This case was discarded. In general, recovery was good. Figure 1 shows the distributions of each parameter's estimates by parameter set. Though some parameters in some conditions, particularly the drift rate parameter, show some variability, no systematic biases are evident for any parameter.

Benchmark data

Data set and theoretical framework. To further illustrate the functionality and utility of our module, we applied a Bayesian hierarchical diffusion model analysis to benchmark data from Vandekerckhove, Panis, and Wagemans (2007, data used with permission). The data come from nine participants, who performed a visual detection task (i.e., they reported whether or not a change occured in a figure in a temporal 2AFC task). The difficulty of this task was manipulated in a 2-by-2 factorial design, resulting in four experimental conditions plus one control condition (where no change in the figure occured). The dependent variables X (binary, 1 if the correct response was given, 0 otherwise) and T (the reaction time) were recoded into a single variable $Y = (2X - 1)T$, preserving all the information in the bivariate data. For a more detailed description on the data and the research question, we refer the reader to Vandekerckhove et al. (2007).

Vandekerckhove et al. (2011) have previously used the same data to show the advantages of a hierarchical diffusion model in a Bayesian framework. Here the focus lies on demonstrating how we use the same theoretical framework with the presented, open-source software. Moreover, we will do our analysis with two different models, differing in the exact specification of the hierarchical structure. For a more detailed description of the model framework, see Vandekerckhove et al. (2011).

Model definitions. Following Vandekerckhove et al. (2011), we assumed an unbiased diffusion process and set $\beta = .5$ accordingly. We further allowed the boundary separation

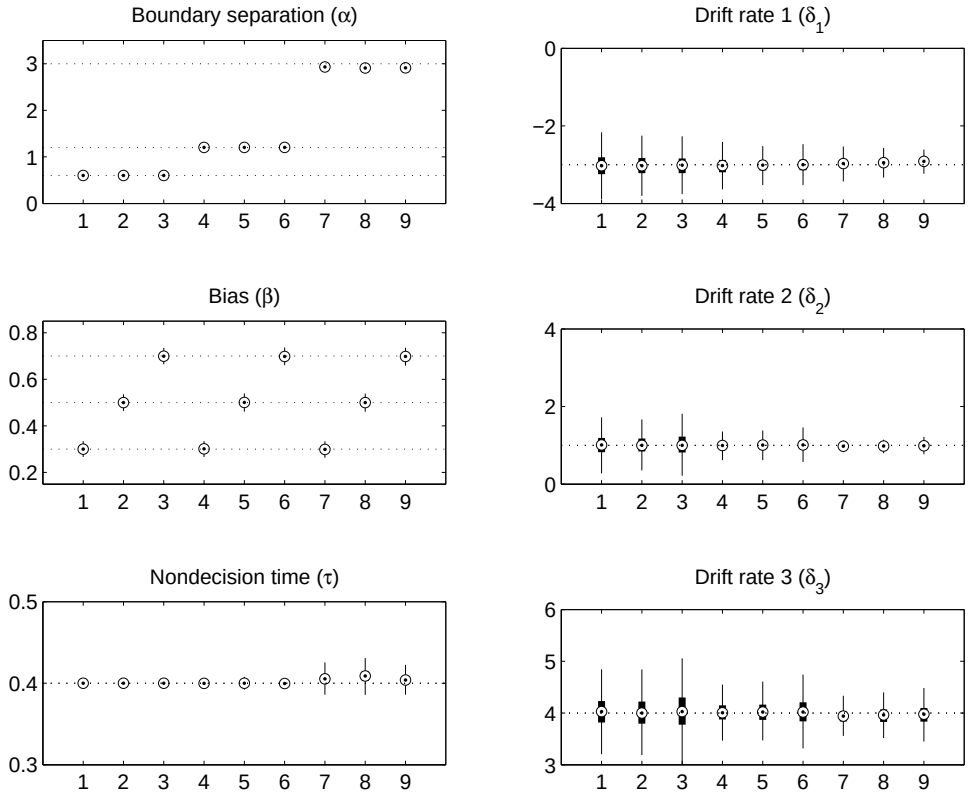

Figure 1. : Results of the parameter recovery simulations. In all panels, the nine levels refer to the nine different parameter sets. True parameter values are given in the text. Values are recovered as posterior means, and the plots display the population of parameter sets over 1,000 simulated data sets. Circles indicate mean recovered values, whiskers span 1.5 times the interquartile range, and the edges of the boxes are the 25th and 75th percentiles. The results indicate very good recovery, with slightly more variability in the drift rate estimates than in the other parameters, especially in the conditions with low boundary separations.

parameter α to differ between persons as a random effect. In other words, every person p is allowed to have their own boundary separation parameter $\alpha_{(p)}$, which is a draw from a joint population distribution: $\alpha_{(p)} \sim N(\mu_\alpha, \sigma_\alpha^2)$.

The nondecision time $\tau_{(pij)}$ is also allowed to differ between persons p, conditions i and trials j. It is a random variable with mean $\theta_{(p)}$ and variance $\chi^2_{(p)}$, both of which are themselves seen as random variables, differing between persons, with respective population means μ_χ and μ_θ and variances σ^2_χ and σ^2_θ: $\tau_{(pij)} \sim N\left(\theta_{(p)}, \chi^2_{(p)}\right)$ and $\theta_{(p)} \sim N\left(\mu_\theta, \sigma^2_\theta\right)$ and $\chi_{(p)} \sim N\left(\mu_\chi, \sigma^2_\chi\right)$.

In the first model, $\mathcal{M}_1$, we will not estimate μ_χ and σ_χ, but set them to the fixed values $\mu_\chi = .35$ and $\sigma_\chi = .125$, casting these as fixed effects rather than random ones.[18]

The drift rate parameter $\delta_{(pij)}$ is also allowed to differ between trials, conditions and persons. Moreover, it is cast as a random variable with a condition-by-person-specific mean $\nu_{(pi)}$ and person-specific variance $\eta^2_{(p)}$. The mean $\nu_{(pi)}$ is itself a random parameter with mean $\mu_{\nu(i)}$ and variance $\sigma^2_{\nu(i)}$, which both differ between conditions as fixed effects. The variance $\eta^2_{(p)}$ differs between persons and is a random variable with the population mean μ_η and variance σ^2_η. In the $\mathcal{M}_1$ model, we will not estimate μ_η and σ_η, but set them to 3.5 and 3.5, respectively.

The second model ($\mathcal{M}_2$) differs from the first ($\mathcal{M}_1$) only in that μ_χ, σ_χ, μ_η and σ_η are free parameters whose values are estimated by JAGS. Figure 2 shows a graphical model representation of the second model. A graphical representation of the first model would look identical, but for the removal of the four now fixed nodes.

Results—technical. We ran three chains with 50,000 iterations each for the $\mathcal{M}_1$ model and three chains with 250,000 iterations each for the $\mathcal{M}_2$ model. For no parameters did the potential scale reduction factor $\hat{R}$ exceed 1.02 in $\mathcal{M}_1$. In $\mathcal{M}_2$, η_μ and η_σ exceeded an acceptable value with 1.8 and 2.88, respectively, whereas all other parameters, and the deviance (i.e., the log-posterior), had $\hat{R}$-values of approximately 1.

The Monte Carlo chains of $\mathcal{M}_1$ showed high autocorrelation and poor mixing, prompting us to subsample the chains by a factor of 1,000 in addition to using a burn-in period of 1,000, resulting in 150 independent samples. The overall shape of the thinned chains was satisfactory, based on visual examination. Figure 3 shows the deviance chain of the easy model before and after burning and thinning, with the chain after burning and thinning showing good mixing.

Similar results held for the more complex model $\mathcal{M}_2$: High autocorrelation and unconverged chains at the beginning were removed by burning the chains with 20,000 and thinning them by 2,000, resulting in 348 independent samples. Figure 4 shows a typical chain of $\mathcal{M}_2$. Two parameters showed poor mixing even after burning and thinning: η_μ and η_σ. However, the total deviance of the model did not show convergence issues, suggesting that these two parameters converge poorly due to the modest impact they have on the model's fit to the data (i.e., they are poorly identified by the data).

[18]The theoretical difference between fixed and random effects is subtle. We refer to De Boeck (2008) and Gelman and Hill (2007) as good starting points regarding the issue. For our purposes, it suffices to know that, while random effects are particularly useful as generalization tools, they occasionally carry an additional computational burden that may not be warranted if the particular parameter is not of core interest. This is why we provide an example for each scenario.

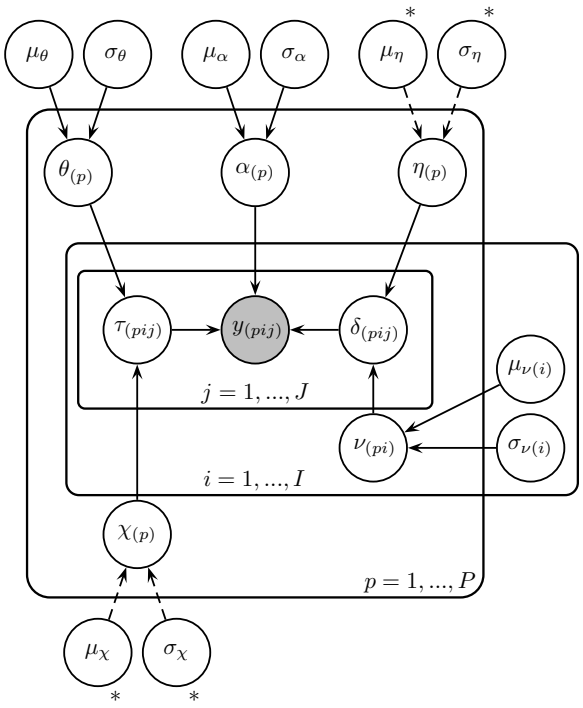

Figure 2. : The graphical model for the $\mathcal{M}_2$ model. The nodes that are indicated with stars and connected with dashed arrows— μ_χ, σ_χ, μ_η and σ_η —are set to fixed values in the $\mathcal{M}_1$ model.

Finally, Table 2 shows the summary statistics of the drift rate population distributions for the $\mathcal{M}_1$ model and the $\mathcal{M}_2$ model. As the table shows, the parameters differ between conditions. The outcome of our analysis with JAGS did not substantially differ from the results reported Vandekerckhove et al. (2011).

Summary

JAGS is an open-source software package for the analysis of graphical models, and is written with extensibility in mind. Additionally, open-source software is advantageous for academic research because of its permanency and accessibility. We provide step-by-step instructions on how to implement custom distributions and logical nodes in JAGS. It is hoped that our documenting the process of extending JAGS will contribute to the formation of a collaborative community that will extend the usefulness of JAGS even more.

We implement the first passage time distribution of a Wiener diffusion model as a worked example, and provide the resulting module as freely downloadable package.

References

De Boeck, P. (2008). Random item IRT models. *Psychometrika, 73*(4), 533–559.

Gelman, A., Carlin, J., Stern, H., & Rubin, D. (2004). *Bayesian data analysis.* New York: Chapman & Hall/CRC.

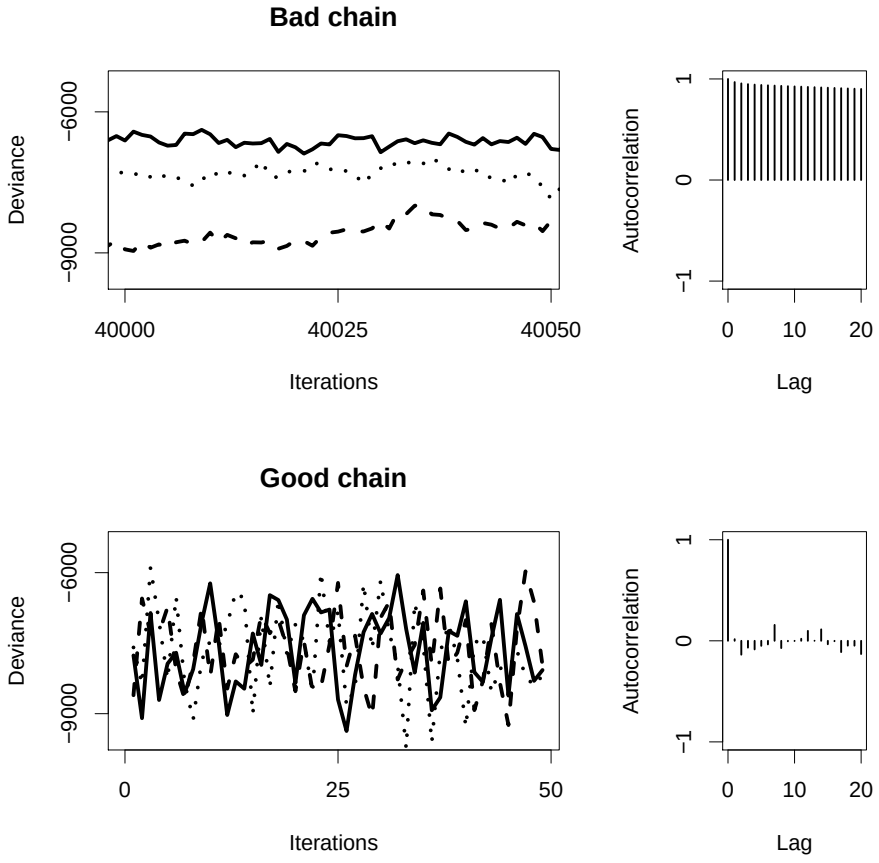

Figure 3. : Examples of a good and a bad chain. The upper left panel shows the deviance chain, as it was sampled. The lower left panel shows the deviance chain after thinning and burning. The two panels on the right show autocorrelation plots before and after thinning, respectively.

Gelman, A., & Hill, J. (2007). *Data analysis using regression and multilevel/hierarchical models*. Cambridge, MA: Cambridge University Press.

Lunn, D., Jackson, C., Best, N., Thomas, A., & Spiegelhalter, D. (2012). *The BUGS Book: A practical introduction to Bayesian analysis*. CRC Press.

Lunn, D., Thomas, A., Best, N., & Spiegelhalter, D. (2000). WinBUGS—A Bayesian modelling framework: concepts, structure, and extensibility. *Statistics and computing*, *10*(4), 325–337.

Plummer, M. (2003). *JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling*.

Thomas, A., Spiegelhalter, D., & Gilks, W. (1992). BUGS: A program to perform Bayesian inference using Gibbs sampling. *Bayesian statistics*, *4*, 837–842.

Vandekerckhove, J. (2009). *Extensions and applications of the diffusion model for two-choice response times*. Unpublished doctoral dissertation, University of Leuven.

Vandekerckhove, J., Panis, S., & Wagemans, J. (2007). The concavity effect is a compound of local and global effects. *Attention, Perception, & Psychophysics*, *69*(7), 1253–1260.

Vandekerckhove, J., & Tuerlinckx, F. (2007). Fitting the Ratcliff diffusion model to experimental

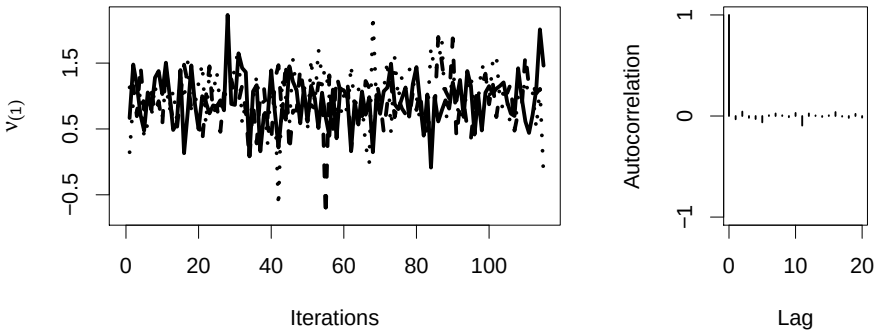

Figure 4. : Trace and autocorrelation plot for the first drift rate mean of the $\mathcal{M}_2$ model.

			$\mathcal{M}_1$		$\mathcal{M}_2$	
	Type	Quality	$\mu_{\nu(i)}$	$\sigma_{\nu(i)}$	$\mu_{\nu(i)}$	$\sigma_{\nu(i)}$
ν_1	1	0	1.0078	0.3508	0.9442	0.3987
ν_2	0	0	−0.4347	0.5615	−0.3925	0.5510
ν_3	1	1	3.0505	0.5353	2.8907	0.5535
ν_4	0	1	0.4699	0.4667	0.4099	0.4304
ν_5			3.6166	0.3698	3.3758	0.4099

Table 2:: Estimates (posterior means) of the five population means (μ) and standard deviations (σ) of the drift rate parameters for the two models.

data. *Psychonomic Bulletin & Review*, *14*(6), 1011–1026.

Vandekerckhove, J., & Tuerlinckx, F. (2008). Diffusion model analysis with MATLAB: A DMAT primer. *Behavior Research Methods*, *40*(1), 61–72.

Vandekerckhove, J., Tuerlinckx, F., & Lee, M. (2011). Hierarchical diffusion models for two-choice response times. *Psychological Methods*, *16*(1), 44.

Voss, A., & Voss, J. (2007). Fast-dm: A free program for efficient diffusion model analysis. *Behavior Research Methods*, *39*(4), 767–775.

Wagenmakers, E. (2009). Methodological and empirical developments for the Ratcliff diffusion model of response times and accuracy. *European Journal of Cognitive Psychology*, *21*(5), 641–671.

Wagenmakers, E., Van Der Maas, H., & Grasman, R. (2007). An EZ-diffusion model for response time and accuracy. *Psychonomic Bulletin & Review*, *14*(1), 3–22.

Wiecki, T. V., Sofer, I., & Frank, M. J. (2011). Fitting drift-diffusion models in a hierarchical Bayesian framework: methods and applications. In *Frontiers in Neuroinformatics Conference Abstract: 4th INCF Congress of Neuroinformatics.* NIPS.

Appendix
JAGS custom distribution quick reference table

	Step	Files involved	Contains
1.	Define a module class	`src/<longname>.cc`	Constructor and destructor
2.	Define scalar distribution	`src/distributions/D<shortname>.h`	Function prototypes
		`src/distributions/D<shortname>.cc`	Computational implementation
			Defines name of new stochastic node in BUGS
3.1.	Create directory stucture	`m4/` (directory)	Nothing
3.2.	Create configuration file	`configure.ac`	Configuration instructions as per Autoconf
3.3.	Create Make files	`Makefile.am`	Compilation instructions
		`src/Makefile.am`	Compilation instructions
		`src/distributions/Makefile.am`	Compilation instructions
3.4.	`autoreconf -fvi && ./configure`	(generates files)	Produces auxiliary files suitable for system
	`make && make install`*	(generates files and compiles)	Produces binary files
			Copies binaries to correct system locations

Note: Replace *longname* and *shortname* with the long and the short form of the distribution name (e.g., *Bernoulli* and *Bern*). *: On linux systems. See text for Windows procedure.