Custom distributions and functions in JAGS

Dominik Wabersich[1,2] **and Joachim Vandekerckhove**[1]

[1]Department of Cognitive Sciences, University of California, Irvine
[2]Department of Psychology, University of Tübingen

Extending JAGS

Background

JAGS ("Just Another Gibbs Sampler", Plummer, 2003) is a free and open-source software package for the analysis of Bayesian graphical models. Because it is open-source, built to be modular, and written in C++, JAGS can be extended with a variety of distributions (e.g., cognitive models), deterministic functions (e.g., matrix operations), monitors (e.g., model evaluation statistics), and samplers (e.g., Hamiltonian MC)—all while using the same graphical modeling language.

Motivation

Because it is so extensible, we believe that JAGS has excellent potential for becoming a community-built resource for cognitive scientists (and others), and has the potential to support a wide user and contributor base. We aim to contribute to the formation of such a contributor base.

Our project

We demonstrate how to extend JAGS with custom functionality. We provide distributions and functions for use in cognitive science, and have written the first technical manual on JAGS extensions (Wabersich & Vandekerckhove, in press). Here we provide an overview of the steps to extending JAGS, and implement a diffusion model as an example.

JAGS wiener module

Info

Wiener diffusion model with four parameters:

- Drift rate δ
- Initial bias β
- Boundary separation α
- Non-decision time τ

$\Rightarrow$ Implementation of several model ideas possible through use of this node in BUGS (e.g., Ratcliff diffusion model).

Web

`https://sourceforge.net/projects/jags-wiener/`

BUGS example code

```
model {
  alpha ~ dunif(0.001,3)
  tau ~ dunif(0,1)
  beta <- 0.5

  for (c in 1:C) {
    delta[c] ~ dnorm(0, 1)

    for (i in 1:N) {
      y[i,c] ~ dwiener(alpha,tau,beta,delta[c])
    }
  }
}
```

Further Information

Where to go?

CIDLAB: `http://www.cidlab.com`
Contact: dominik.wabersich@gmail.com or joachim@uci.edu

References

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

Wabersich, D., & Vandekerckhove, J. (in press). Extending JAGS: A tutorial on writing JAGS modules. *Behavior Research Methods*.

How To

Step 0: familiarize

- Familiarize with JAGS functionality
  - ∗ Ability to dynamically load and unload modules
- Familiarize with the JAGS source
- Familiarize with JAGS module creation process

Step 1: create module (class)

Create a module code file **myModule.cc** that contains the module class and its prototypes. The constructor and destructor functions of this class have to load/unload the module functionality (e.g., the added distributions).

Step 2: add distributions/functions

Add the distribution or function by writing a distribution or function header file and code file, e.g. **DWiener.h** and **DWiener.cc**. The header file contains the function prototypes and the code file contains the actual implementation of the calculations. This is also where the new (stochastic/deterministic) nodes for use in the BUGS-language model definition get defined (i.e., the core functionality of the module).

Step 3:

Configuration

- JAGS uses Autoconf, Automake and Libtool
  - ∗ These tools make the configuration and building process easier
  - ∗ Certain file and directory structure is necessary
- Create directory/file structure as shown on the right
- Create configure.ac
  - ∗ Configuration instructions for Autoconf
- Create Automake makefiles
  - ∗ Makefile.am in every directory with building instructions

Structure

```
Makefile.am
configure.ac
m4
src
├─ Makefile.am
├─ myModule.cc
├─ distributions
│   ├─ Makefile.am
│   ├─ DWiener.h
│   └─ DWiener.cc
└─ functions
    ├─ Makefile.am
    ├─ Fexpl1.h
    ├─ Fexpl1.cc
    ├─ Fexpl2.h
    └─ Fexpl2.cc
```

Step 4: Build

- Build, install and your module is ready for usage!
  - ∗ Create necessary files, so autoreconf does not complain: *touch AUTHORS README NEWS ChangeLog*
  - ∗ Configure
    - · *autoreconf -fvi && ./configure*
  - ∗ Build and Install
    - · *make && make install*
- To successfully build a library
  - ∗ All Makefile.am files and the configure.ac file have to contain the right directives
  - ∗ All code files have to be correct and the necessary functions have to be implemented
- The build process builds a library that, when copied to the right place (usually done through installation) can be loaded in JAGS

Windows Info

- Compiling with **MinGW** Environment and **TDM-GCC** Compilers
  - ∗ Use exact same Compiler Version as the used JAGS version
  - ∗ See JAGS Manual for Version Information
  - ∗ Enforce static linking: Delete all *.dll.a files in TDM-GCC directory
- Recommended to use **tarball**, created with Linux
- Compiled module files in src/.libs: example.dll, example.dll.a, example.la
- Windows installers can be created by using **NSIS**
  $\rightarrow$ `http://nsis.sourceforge.net`

module class: **myModule.cc**

```
#include <Module.h>
#include <distributions/DWiener.h>
#include <functions/Fexpl1.h>
#include <functions/Fexpl2.h>

namespace example { // module namespace

// JAGS module class
class EXPLModule : public Module {
  public:
    EXPLModule();
    ~EXPLModule();
};

// constructor (executed when loading the module)
EXPLModule::EXPLModule() : Module("example")
{
  // JAGS functions to load classes that contain
  // the module functionality
  insert(new DWiener);
  insert(new Fexpl1);
  insert(new Fexpl2);
}
// destructor (executed when unloading the module)
EXPLModule::~EXPLModule() {
  std::vector<Distribution*> const &dvec = distributions();
  for (unsigned int i=0;i<dvec.size();++i) {
    delete dvec[i];
  }
}

}
example::EXPLModule _example_module;
```

distribution class2: **DWiener.h**

```
#ifndef DWIENER_H_
#define DWIENER_H_
#include <distribution/ScalarDist.h>

namespace example { // module namespace

class DWiener : public ScalarDist
{
  public:
    DWiener();

    // functions that need to be implemented
    // calculates logDensity at value x for given parameters
    double logDensity(double x, PDFType type,
      std::vector<double const *> const &parameters,
      double const *lower, double const *upper) const;
    // provides a random sample with given parameters
    double randomSample(std::vector<double const *> const
      &parameters, double const *lower,
      double const *upper, RNG *rng) const;
    // provides a typical value for a given set of parameters
    double typicalValue(std::vector<double const *> const
      &parameters, double const *lower,
      double const *upper) const;
    // checks if given parameter set is evaluable
    bool checkParameterValue(
      std::vector<double const *> const &parameters) const;

    // self added function for calculation of logDensity
    double wiener_logDensity(double x, PDFType type,
      std::vector<double const *> const &parameters) const;
};

}
#endif /* DWIENER_H_ */
```

distribution class: **DWiener.cc**

```
#include <config.h>
#include "DWiener.h"
#include <util/nainf.h>
#include <cmath>

using std::vector;

namespace example { // module namespace

// ScalarDist takes 3 arguments:
// - name of the distribution (as used in BUGS later)
// - number of parameters
// - type of distribution
DWiener::DWiener() : ScalarDist("dwiener", 4, DIST_UNBOUNDED) {}

// The most important arguments of this function are:
// - x: the value where the logDensity shall be evaluated
// - parameters: pointer to a vector containing the parameters
//   --> can be accessed by *parameters[0],
double DWiener::logDensity(double x, PDFType type,
  vector<double const *> const &parameters,
  double const *lbound, double const *ubound) const
{
  // actual numerical implementation elsewhere in code
  double d = wiener_logDensity(x, type, parameters);
  return d;
}
// ...(implementation of remaining functions)
```