

JAGS Version 3.3.0 installation manual

Martyn Plummer Bill Northcott

1 October 2012

JAGS is distributed in binary format for Microsoft Windows, Mac OS X, and most Linux distributions. The following instructions are for those who wish to build JAGS from source. The manual is divided into three sections with instructions for Linux/Unix, Mac OS X, and Windows.

1 Linux and UNIX

JAGS follows the usual GNU convention of

```
./configure
make
make install
```

which is described in more detail in the file `INSTALL` in the top-level source directory. On some UNIX platforms, you may be required to use GNU make (`gmake`) instead of the native make command. On systems with multiple processors, you may use the option `-j` to speed up compilation, *e.g.* for a quad-core PC you may use:

```
make -j4
```

1.1 Configure options

At configure time you also have the option of defining options such as:

- The names of the C, C++, and Fortran compilers.
- Optimization flags for the compilers. JAGS is optimized by default if the GNU compiler (`gcc`) is used. If you are using another compiler then you may need to explicitly supply optimization flags.
- Installation directories. JAGS conforms to the GNU standards for where files are installed. You can control the installation directories in more detail using the flags that are listed when you type `./configure --help`.

1.1.1 Configuration for a 64-bit build

By default, JAGS will install all libraries into `/usr/local/lib`. If you are building a 64-bit version of JAGS, this may not be appropriate for your system. On Fedora and other RPM-based distributions, for example, 64-bit libraries should be installed in `lib64`, and on Solaris, 64-bit libraries are in a subdirectory of `lib` (*e.g.* `lib/amd64` if you are using a x86-64 processor), whereas on Debian, and other Linux distributions that conform to the FHS, the correct installation directory is `lib`.

To ensure that JAGS libraries are installed in the correct directory, you should supply the `--libdir` argument to the configure script, *e.g.*:

```
./configure --libdir=/usr/local/lib64
```

It is important to get the installation directory right when using the `rjags` interface between R and JAGS, otherwise the `rjags` package will not be able to find the JAGS library.

1.1.2 Configuration for a private installation

If you do not have administrative privileges, you may wish to install JAGS in your home directory. This can be done with the following configuration options

```
export JAGS_HOME=$HOME/jags #or wherever you want it
./configure --prefix=$JAGS_HOME
```

For more detailed control over the installation directories type

```
./configure --help
```

and read the section “Fine-tuning of the installation directories.”

With a private installation, you need to modify your PATH environment variable to include ‘\$JAGS_HOME/bin’. You may also need to set LD_LIBRARY_PATH to include ‘\$JAGS_HOME/lib’ (On Linux this is not necessary as the location of libjags and libjrmath is hard-coded into the JAGS binary).

1.2 BLAS and LAPACK

BLAS (Basic Linear Algebra System) and LAPACK (Linear Algebra Pack) are two libraries of routines for linear algebra. They are used by the multivariate functions and distributions in the bugs module. Most unix-like operating system vendors supply shared libraries that provide the BLAS and LAPACK functions, although the libraries may not literally be called “blas” and “lapack”. During configuration, a default list of these libraries will be checked. If configure cannot find a suitable library, it will stop with an error message.

You may use alternative BLAS and LAPACK libraries using the configure options `--with-blas` and `--with-lapack`

```
./configure --with-blas="-lmyblas" --with-lapack="-lmylapack"
```

If the BLAS and LAPACK libraries are in a directory that is not on the default linker path, you must set the LDFLAGS environment variable to point to this directory at configure time:

```
LDFLAGS="-L/path/to/my/libs" ./configure ...
```

At runtime, if you have linked JAGS against BLAS or LAPACK in a non-standard location, you must supply this location with the environment variable LD_LIBRARY_PATH, *e.g.*

```
LD_LIBRARY_PATH="/path/to/my/libs:${LD_LIBRARY_PATH}"
```

Alternatively, you may hard-code the paths to the blas and lapack libraries at compile time. This is compiler and platform-specific, but is typically achieved with

```
LDFLAGS="-L/path/to/my/libs -R/path/to/my/libs
```

1.2.1 Multithreaded BLAS and LAPACK

Some high-performance computing libraries offer multi-threaded versions of the BLAS and LAPACK libraries. Although instructions for linking against some of these libraries are given below, this should not be taken as encouragement to use multithreaded BLAS. Testing shows that using multiple threads in BLAS can lead to significantly *worse* performance while using up substantially more computing resources.

1.3 GNU/Linux

GNU/Linux is the development platform for JAGS, and a variety of different build options have been explored, including the use of third-party compilers and linear algebra libraries.

1.3.1 Fortran compiler

The GNU FORTRAN compiler changed between gcc 3.x and gcc 4.x from `g77` to `gfortran`. Code produced by the two compilers is binary incompatible. If your BLAS and LAPACK libraries are linked against `libgfortran`, then they were built with `gfortran` and you must also use this to compile JAGS.

Most recent GNU/Linux distributions have moved completely to gcc 4.x. However, some older systems may have both compilers installed. Unfortunately, if `g77` is on your path then the configure script will find it first, and will attempt to use it to build JAGS. This results in a failure to recognize the installed BLAS and LAPACK libraries. In this event, set the `F77` variable at configure time.

```
F77=gfortran ./configure
```

1.3.2 BLAS and LAPACK

The **BLAS** and **LAPACK** libraries from Netlib (<http://www.netlib.org>) should be provided as part of your Linux distribution. If your Linux distribution splits packages into “user” and “developer” versions, then you must install the developer package (*e.g.* `blas-devel` and `lapack-devel`).

Suse Linux Enterprise Server (SLES) does not include BLAS and LAPACK in the main distribution. They are included in the SLES SDK, on a set of CD/DVD images which can be downloaded from the Novell web site. See http://developer.novell.com/wiki/index.php/SLES_SDK for more information.

1.3.3 ATLAS

On Fedora Linux, pre-compiled atlas libraries are available via the `atlas` and `atlas-devel` RPMs. These RPMs install the atlas libraries in the non-standard directory `/usr/lib/atlas` (or `/usr/lib64/atlas` for 64-bit builds) to avoid conflicts with the standard `blas` and `lapack` RPMs. To use the atlas libraries, you must supply their location using the `LDLFLAGS` variable (see section 1.2)

```
./configure LDLFLAGS="-L/usr/lib/atlas"
```

Runtime linking to the correct libraries is ensured by the automatic addition of `/usr/lib/atlas` to the linker path (see the directory `/etc/ld.so.conf.d`), so you do not need to set the environment variable `LD_LIBRARY_PATH` at run time.

1.3.4 AMD Core Math Library

The AMD Core Math Library (`acml`) provides optimized BLAS and LAPACK routines for AMD processors. To link JAGS with `acml`, you must supply the `acml` library as the argument to `--with-blas`. It is not necessary to set the `--with-lapack` argument as `acml` provides both sets of functions. See also section 1.2 for run-time instructions.

For example, to link to the 64-bit `acml` using gcc 4.0+:

```
LDFLAGS="-L/opt/acml4.3.0/gfortran64/lib" \  
./configure --with-blas="-lacml -lacml_mv"
```

The `acmv_mv` library is a vectorized math library that exists only for the 64-bit version and is omitted when linking against 32-bit `acml`.

On multi-core systems, you may wish to use the threaded `acml` library (See the warning in section 1.2.1 however). To do this, link to `acml_mp` and add the compiler flag ‘`-fopenmp`’:

```
LDFLAGS="-L/opt/acml4.3.0/gfortran64_mp/lib" \  
CXXFLAGS="-O2 -g -fopenmp" ./configure --with-blas="-lacml_mp -lacml_mv"
```

The number of threads used by multi-threaded `acml` may be controlled with the environment variable `OMP_NUM_THREADS`.

1.3.5 Intel Math Kernel Library

The Intel Math Kernel library (MKL) provides optimized BLAS and LAPACK routines for Intel processors. MKL is designed to be linked to executables, not shared libraries. This means that it can only be linked to a static version of JAGS, in which the JAGS library and modules are linked into the main executable. To build a static version of JAGS, use the configure option ‘`--disable-shared`’.

MKL version 10.0 and above uses a “pure layered” model for linking. The layered model gives the user fine-grained control over four different library layers: interface, threading, computation, and run-time. Some examples of linking to MKL using this layered model are given below. These examples are for GCC compilers on `x86_64`. The choice of interface layer is important on `x86_64` since the Intel Fortran compiler returns complex values differently from the GNU Fortran compiler. You must therefore use the interface layer that matches your compiler (`mkl_intel*` or `mkl_gf*`).

For further guidance, consult the MKL Link Line advisor at <http://software.intel.com/en-us/articles/intel-mkl-link-line-advisor>.

Recent versions of MKL include a shell script that sets up the environment variables necessary to build an application with MKL.

```
source /opt/intel/composerxe-2011/mkl/bin/mklvars.sh intel64
```

After calling this script, you can link JAGS with a sequential version of MKL as follows:

```
./configure --disable-shared \  
--with-blas="-lmkl_gf_lp64 -lmkl_sequential -lmkl_core -lpthread"
```

Note that `libpthread` is still required, even when linking to sequential MKL.

Threaded MKL may be used with:

```
./configure --disable-shared \  
--with-blas="-lmkl_gf_lp64 -lmkl_gnu_thread -lmkl_core -liomp5 -lpthread"
```

The default number of threads will be chosen by the OpenMP software, but can be controlled by setting `OMP_NUM_THREADS` or `MKL_NUM_THREADS`. (See the warning in section 1.2.1 however).

1.3.6 Using Intel Compilers

JAGS has been successfully built with the Intel Composer XE compilers. To set up the environment for using these compilers call the ‘`compilervars.sh`’ shell script, *e.g.*

```
source /opt/intel/composerxe-2011/bin/compilervars.sh intel64
```

Then call the configure script with the Intel compilers:

```
CC=icc CXX=icpc F77=ifort ./configure
```

1.3.7 Using Clang

JAGS has been built with the clang compiler for C and C++ (version 3.1). The configuration was

```
LD="llvm-ld" CC="clang" CXX="clang++" ./configure
```

In this configuration, the gfortran compiler was used for Fortran and the C++ code was linked to the GNU standard C++ library (`libstdc++`) rather than the version supplied by the LLVM project (`libc++`).

1.4 Solaris

JAGS has been successfully built and tested on the Intel x86 platform under Solaris 11 using the Sun Studio 12.3 compilers.

```
./configure CC=cc CXX=CC F77=f95 \  
CFLAGS="-xO3 -xarch=sse2" \  
FFLAGS="-xO3 -xarch=sse2" \  
CXXFLAGS="-xO3 -xarch=sse2"
```

The Sun Studio compiler is not optimized by default. Use the option ‘`-xO3`’ for optimization (NB This is the letter “O” not the number 0) In order to use the optimization flag ‘`-xO3`’ you must specify the architecture with the ‘`-xarch`’ flag. The options above are for an Intel processor with SSE2 instructions. This must be adapted to your own platform.

To compile a 64-bit version of JAGS, add the option ‘`-m64`’ to all the compiler flags.

Solaris provides two versions of the C++ standard library: `libCstd`, which is the default, and `libstlport4`, which conforms more closely to the C++ standard. JAGS may be linked to the `stlport4` library by adding the option ‘`-library=stlport4`’ to `CXXFLAGS`.

The configure script automatically detects the Sun Performance library, which implements the BLAS/LAPACK functions.

2 Mac OS X

There have been big changes in Apple's developer tools in the last couple of years. Currently (this may change at any time) the core suite of tools are contained in Xcode. This is now a single app, which contains the command line tools in '`Xcode.app/Contents/Developer/usr`', instead of installing them in '`/usr`' as was the previous arrangement. The most recent version of Xcode (4.5) can be downloaded free from the App Store. This version will only run on Lion 10.7.x and Mountain Lion 10.8.x. If you are trying to build on an earlier version of MacOS, you will need to download an older version of Xcode from the Mac Dev Center at <http://developer.apple.com>. This is no longer free. You will need to sign up on the Mac Developer Program which costs \$99 per year. The program gives you access to many more resources including an Apple signed developer certificate which can be used to sign your code. The instructions that follow assume you are using MacOS 10.7 or 10.8 and have the free Xcode from the App Store.

2.1 Required tools

If you wish to build from a released source package i.e. '`JAGS-3.3.0.tar.gz`', you will need Xcode and the gfortran package which you can find by following the "tools directory" link on the "R for Mac OS X" page on CRAN. You will also need to install the command line tools into `/usr`. This is done within Xcode from Preferences → Downloads → Components. This setup should be able to build the JAGS sources and also source packages in R. All the necessary libraries such as BLAS and LAPACK are included in the standard MacOS install.

2.2 Prepare the source code

Move the downloaded '`JAGS-X.X.X.tar.gz`' package to some suitable working space on your disk and double click the file. This will decompress the package to give a folder called '`JAGS-X.X.X`', where '`X.X.X`' is the version number. Open the Terminal app from `/Applications/Utilities`. This gives you a UNIX shell know as bash. In the Terminal window after the `$` prompt type '`cd`' followed by a space. In the Finder drag `JAGS-X.X.X` folder into the Terminal window and hit return. If this worked for you, typing '`ls`' followed by a return will list the contents of the JAGS folder.

2.3 Set up the environment

Executable files on MacOS can contain code for more than one CPU architecture: `i386`, `x86_64` or `ppc`. Executables with more than one architecture are described as 'fat.' The instructions below will build a fat binary of JAGS containing 32-bit `i386` and 64-bit `x86_64` code which will run on 10.6, 10.7 and 10.8. This is compatible with current CRAN distributions of R and the `rjags` module.

In your Terminal window type the following instructions which set up the environment for the build. (Note that there is no space after '`platform`' and the '`\`' must not be followed by any spaces)

```
export CC=/usr/bin/clang
export CXX=/usr/bin/clang++
export CFLAGS="-g -Os -mmacosx-version-min=10.6 -isysroot \\
```

```
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform\  
/Developer/SDKs/MacOSX10.8.sdk \  
-arch i386 -arch x86_64"  
export CXXFLAGS="-g -Os -mmacosx-version-min=10.6 -isysroot \  
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform\  
/Developer/SDKs/MacOSX10.8.sdk \  
-arch i386 -arch x86_64"  
export FFLAGS="-g -Os -mmacosx-version-min=10.6 -isysroot \  
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform\  
/Developer/SDKs/MacOSX10.8.sdk -arch i386 -arch x86_64"  
export LDFLAGS="-mmacosx-version-min=10.6 -arch i386 -arch x86_64"
```

2.4 Configuration

To configure the package type:

```
../configure --disable-dependency-tracking --with-included-ltdl
```

This instruction should complete without reporting an error.

2.5 Compile

To compile the code type:

```
make -j 8
```

The number '8' indicates the number of build threads that should be run. In general this is best as twice the number of CPU cores in the computer. Most current Macs have four core CPUs. If your machine is different, you may want to change the number in the instruction. Again, this instruction should complete without errors.

2.6 Install

Finally to install JAGS you need to be using an account with administration privileges. Type:

```
sudo make install
```

This will ask for your account password and install the code ready to run as described in the User Manual.

You need to ensure `/usr/local/bin` is in your `PATH` in order for 'jags' to work from a shell prompt.

2.7 Older versions of MacOS

For instructions for building on Tiger or for older versions of R see previous versions of this manual.

2.8 Tips for developers and advanced users

2.8.1 Compilers

Older versions of Xcode used the gcc compiler suite. Apple abandoned the gcc project when GPL3 was introduced. Current Apple compilers are based on LLVM. They are invoked as clang and clang++. Although there appear to be gcc and gxx, these are actually gcc-llvm with an out of date gcc 4.2 front end on LLVM. These compilers are now deprecated. While there are versions of current gcc available for MacOS, they do not recognise the Apple specific -arch flag. This leads to a number of problems.

2.8.2 Developer tools

Now that Apple do not use gcc, they have no use for many of the auxiliary GNU tools required to build it. So the Xcode does not contain any of the autotools (autoconf, automaker and libtool). If you want to work on code from the JAGS repository, you will need to build and install these. Other GNU tools like bison and make are available but may be very out of date because of the GPL3 issue. Binaries of cvs, subversion and git are included in Xcode, Mercurial needs to be installed separately if required.

3 Windows

These instructions use MinGW, the Minimalist GNU system for Windows. You need some familiarity with Unix in order to follow the build instructions but, once built, JAGS can be installed on any PC running windows, where it can be run from the Windows command prompt.

3.1 Preparing the build environment

You need to install the following packages

- The TDM-GCC compiler suite for Windows
- MSYS
- NSIS, including the AccessControl plug-in

We used the TDM-GCC compilers based on the MinGW-w64 project (<http://tdm-gcc.tdragon.net>). This distribution was chosen because it allows us to build a version of JAGS that is statically linked against the gcc runtime library. This, in turn, is necessary to have a functional rjags package on Windows. We also tried Rtools (<http://www.murdoch-sutherland.com/Rtools>). Although the resulting JAGS library is functional, it is not compatible with R: loading the rjags package causes R to crash on exit.

TDM-GCC has a nice installer, available from Sourceforge (follow the links on the main TDM-GCC web site). Ensure that you download the MinGW-w64/sjlj version as this is capable of producing both 32-bit and 64-bit binaries.

Select a “Recommended C/C++” installation and customize it by selecting the Fortran compiler, which is not installed by default. After installation, to force the compiler to use static linking, delete any import libraries (files ending in ‘.dll.a’ in the TDM-GCC tree).

MSYS (the Minimal SYStem) is part of the MinGW project. It provides a bash shell for you to build Unix software. Download the MinGW installer from <http://www.mingw.org>. We used ‘mingw-get-inst-20120426.exe’. Run the installer and select “MSYS Basic System”. There is no need to install the “MinGW Developer Toolkit” if you are working with a release tarball of JAGS. It is not necessary to install any of the compilers that come with MinGW as we shall be using the TDM versions. To make MSYS use the TDM-compilers, edit the file ‘c:/mingw/msys/1.0/etc/fstab’ to read

```
c:\MinGW64\ /mingw
```

MSYS creates a home directory for you in ‘c:/mingw/msys/1.0/home/username’, where `username` is your user name under Windows. You will need to copy and paste the source files for LAPACK and JAGS into this directory.

The Nullsoft Scriptable Install System (<http://nsis.sourceforge.net>) allows you to create a self-extracting executable that installs JAGS on the target PC. These instructions were tested with NSIS 2.46. You must also install the AccessControl plug-in for NSIS, which is available from http://nsis.sourceforge.net/AccessControl_plug-in. The plug-in is distributed as a zip file which is unpacked into the installation directory of NSIS.

3.1.1 Building LAPACK

Download the LAPACK source file from <http://www.netlib.org/lapack> to your MSYS home directory. We used version 3.4.1.

You need to build LAPACK twice: once for 32-bit JAGS and once for 64-bit JAGS. The instructions below are for 32-bit JAGS. To build 64-bit versions, repeat the instructions with the flag ‘-m32’ replaced by ‘-m64’ and start in a clean build directory. Note that you cannot cross-build 64-bit BLAS and LAPACK on a 32-bit Windows system. This is because the build process must run some 64-bit test programs.

Launch MSYS, labelled as “MinGW shell” on the Windows Start Menu, and unpack the tarball.

```
tar xfvz lapack-3.4.1.tgz
cd lapack-3.4.1
```

Copy the file ‘INSTALL/make.inc.gfortran’ to ‘make.inc’ in the top level source directory. Then edit ‘make.inc’ replacing the following lines:

```
FORTRAN = gfortran -m32
LOADER = gfortran -m32
```

Type

```
make blaslib
make lapacklib
```

The compilation process is slow. Eventually, it will create two static libraries ‘librefblas.a’ and ‘liblapack.a’. These are insufficient for building JAGS: you need to create dynamic link library (DLL) for each one.

First create a definition file ‘libblas.def’ that exports all the symbols from the BLAS library

```
dlltool -z libblas.def --export-all-symbols librefblas.a
```

Then link this with the static library to create a DLL (‘libblas.dll’) and an import library (‘libblas.dll.a’)

```
gcc -m32 -shared -o libblas.dll -Wl,--out-implib=libblas.dll.a \
libblas.def librefblas.a -lgfortran
```

Repeat the same steps for the LAPACK library, creating an import library (‘liblapack.dll.a’) and DLL (‘liblapack.dll’)

```
dlltool -z liblapack.def --export-all-symbols liblapack.a
```

```
gcc -m32 -shared -o liblapack.dll -Wl,--out-implib=liblapack.dll.a \
liblapack.def liblapack.a -L./ -lblas -lgfortran
```

3.2 Compiling JAGS

Unpack the JAGS source

```
tar xfvz JAGS-3.3.0.tar.gz
cd JAGS-3.3.0
```

and configure JAGS for a 32-bit build

```
CC="gcc -m32" CXX="g++ -m32" F77="gfortran -m32" \
./configure LDFLAGS="-L/path/to/import/libs/ -Wl,--enable-auto-import"
```

where `‘/path/to/import/libs’` is a directory that contains the 32-bit import libraries (`‘libblas.dll.a’` and `‘liblapack.dll.a’`). This must be an *absolute* path name, and not relative to the JAGS build directory.

After the configure step, type

```
make win32-install
```

This will install JAGS into the subdirectory `‘win/inst32’`. Note that you must go straight from the configure step to `make win32-install` without the usual step of typing `make` on its own. The `win32-install` target resets the installation prefix, and this will cause an error if the source is already compiled.

To install the 64-bit version, clean the build directory

```
make clean
```

reconfigure JAGS for a 64-bit build:

```
CC="gcc -m64" CXX="g++ -m64" F77="gfortran -m64" \
./configure LDFLAGS="-L/path/to/import/libs/ -Wl,--enable-auto-import"
```

Then type

```
make win64-install
```

This will install JAGS into the subdirectory `‘win/inst64’`.

With both 32-bit and 64-bit installations in place you can create the installer. Normally you will want to distribute the blas and lapack libraries with JAGS. In this case, put the 32-bit DLLs and import libraries in the sub-directory `‘win/runtime32’` and the 64-bit DLLs and import libraries in the sub-directory `‘win/runtime64’`. They will be detected and included with the distribution.

Make sure that the file `‘makensis.exe’`, provided by NSIS, is in your PATH. For a typical installation of NSIS, on 64-bit windows:

```
PATH=$PATH:/c/Program\ files\ \(\x86\)/NSIS
```

Then type

```
make installer
```

After the build process finishes, the self extracting archive will be in the subdirectory `‘win’`.