

CISST Software Build Instructions

Version 0.5.0, 0.7.0, 0.9.0

Anton Deguet, Peter Kazanzides

Copyright © 2005-2006 Johns Hopkins University (JHU). All rights reserved.

Contents

1	Introduction	2
2	Required software	2
3	Creating a local directory structure	4
4	Compiling the libraries	4
5	CMake configuration options	5
5.1	Library selection	5
5.2	Building static or shared libraries	5
5.3	Numerical methods	6
5.4	Python interface	6
5.5	Test programs	6
5.6	Example programs	7
6	Result of the build process	7
7	Using the CISST Package with C++	7
7.1	Using the native build environment	8
7.1.1	Special instructions for Windows	8
7.2	Using CMake	9
7.3	Setting paths to shared libraries	10
7.3.1	Setting paths on Windows	10
8	Using the CISST Package with Python	12
8.1	Extending the Python interpreter	12
8.2	Embedding the Python interpreter	13

1 Introduction

This document provides instructions for compiling and using the open source *cisst package* on Unix (Linux, Mac OS X, cygwin) and Windows operating systems/environments. Currently, this package consists of the following C++ libraries with compatible Python interfaces:

- cisstCommon** Common infrastructure for the rest of the package such as logging, error and exception handling, class registries.
- cisstVector** Vectors and matrices (fixed size and dynamic), quaternions and transformations in 2D and 3D.
- cisstNumerical** C++ interface to some of the LAPACK and Lawson-Hanson numerical methods.
- cisstInteractive** The Interactive Research Environment (IRE) – an embedded, Python-based interactive shell with GUI features.

The package also includes several examples and tests for these libraries. Other libraries (e.g., for real-time support, device interfaces, robot control) are in development and will be released as open source software in the future. For convenience, the documentation is provided exclusively in its “compiled” form (pdf, ps and html formats). In the future, the *cisst package* may include the documentation source files (primarily L^AT_EX files) and associated build instructions.

The *cisst* build process requires the CMake makefile generator (see www.cmake.org). CMake is an interactive tool that allows the user to specify certain options for the build process, as described in Section 5. It uses the compiler and linker to generate the libraries and executables and (if configured) uses SWIG (www.swig.org) to generate the Python interfaces.

The directory structure should consist of separate source and build trees, as described in Section 3. The source tree contains all the source code and the build tree contains all the files generated by CMake and by the programs it invokes (e.g., compiler, linker, SWIG). In future versions of the *cisst package*, there will also be an install tree for Unix systems and possibly for Windows systems.

2 Required software

The build process depends on and requires the following external software packages:

CMake: version 2.4.2 or later, available from www.cmake.org.

Compiler: The *cisst* libraries are compatible with the following compilers/linkers and operating systems:

- Visual Studio .NET 2003, Visual Studio .Net 2005 (and Visual Studio Express) on Windows
- gcc 3.3, 3.4, 4.0 on Linux (i386)
- gcc 3.3., 3.4, 4.0 on Mac OS X (with gmake or XCode, PowerPC or Intel)
- Intel C++ 7.1, 9.0 (and probably later) on Linux
- gcc 3.3 (and probably later) on cygwin

The following packages are optional in the build process. Their necessity depends on the configuration that is chosen via `CMake`.

- cisstNetlib:** the `cisstNumerical` library uses a customized version of the LAPACK and the Lawson-Hanson libraries, which are available from: www.cisst.org/resources/software/cnetlib. Be sure to choose the correct binary distribution for your build environment. Note that `cisstNetlib` is an improved (thread-safe) replacement for the `cnetlib` library that was introduced with `cisst` Version 0.1.0. Although `cnetlib` is still available, users should upgrade to `cisstNetlib`. `cisst` 0.3.0 requires `cisstNetlib` 2006-01-28 and `cisst` 0.5.0 requires `cisstNetlib` 2006-09-15 or later.
- SWIG:** Available from www.swig.org; version 1.3.28 or later is required if the `cisst` libraries will be wrapped for use with Python.
- Python:** Available from www.python.org; required if the `cisst` libraries will be wrapped for use with Python. Python 2.3 (or later) is required, Python 2.4 is known to work, and Python 2.5 requires SWIG 1.3.31 or later.
- CppUnit:** Available from cppunit.sourceforge.net; required for compiling and running the test programs.

If the Interactive Research Environment (IRE) is used, the following package will be required at runtime (in addition to the Python interpreter listed above):

- wxPython:** Available from www.wxPython.org; version 2.5 or later is required. This is a Python interface to the wxWidgets cross-platform GUI development package. It is already installed on Macintosh OS X 10.4 or later. For other operating systems, it must be downloaded and installed to the `Python Lib/site-packages` directory, after Python is installed. If downloading a pre-built binary, be sure to download the version that is built for the version of Python installed on your system.

<pre> cisst -- source -- libs -- examples '-- tests '-- Linux-gcc3.4 -- libs -- examples '-- tests </pre>	<pre> source -- cisst -- libs -- examples '-- tests '-- otherpackage '-- other build -- cisst -- libs -- examples '-- tests '-- otherpackage '-- other </pre>
--	--

Example #1

Example #2

Figure 1: Two sample directory structures

3 Creating a local directory structure

The local directory structure for the *cisst* package should consist of separate source and build trees. It is suggested that the build tree indicate the operating system and compiler in the path name, for example: `.../Linux-gcc3.4/` or `.../Windows/VC71/`. However, the specific choice of directory structure is left to the user. Figure 1 shows two possible directory structures. The left directory structure keeps all the *cisst* package files under the `cisst` parent directory. This is the structure assumed in this document. The right directory structure organizes all the source code (for *cisst* and other packages) under a single `source` directory and all the compiler output under a single `build` directory (which could be given a more informative name).

Regardless of the choice of directory structure, the first step is to get the source code into the source directory. This can be achieved by unpacking the distribution archive (`.zip` or `.tar.gz`) to the root of the source directory (e.g., `.../cisst/source`).

Important Note: If upgrading an earlier *cisst* installation (e.g., Version 0.1.0 or Version 0.3.0), it is necessary to remove the existing build tree and create a new one. This is due to the directory structure changes introduced in Version 0.5.0.

4 Compiling the libraries

After setting up the source tree as described above, compile the libraries as follows, based on your operating system:

Unix: Go to the build directory and run `cmake`, giving it the path to the source tree. For example, for the directory structure given above (#1):

```
cd /cisst/Linux-gcc3.4
cmake ../source
```

Set your options (see 5), hit `c` to process the current configuration until all the required variable definitions and conflicts are resolved. Then hit `g` to generate the makefiles and quit. In your build directory, you now have a Makefile and you can run `make`.

Windows: Run `CMakeSetup`, the interactive configuration utility provided by `CMake`. Set the “Where is the source code” to your source directory (e.g., `/cisst/source`) and “Where to build the binaries” to your build directory (e.g., `/cisst/Windows-VC71`). Set your options (see 5), solve all the conflicts and generate the projects. In your build directory you now have a solution file called `cisst.sln` which can be opened with Visual C++.

5 CMake configuration options

The user must set several `CMake` options to specify which elements of the *cisst package* are to be compiled and to provide the path to external software tools and libraries.

5.1 Library selection

Individual libraries can be enabled or disabled via the corresponding `CMake` option `BUILD_LIBS_libname`, where `libname` is `cisstCommon`, `cisstVector`, `cisstNumerical` or `cisstInteractive`. By default, the `cisstCommon` and `cisstVector` libraries are enabled. If `cisstNumerical` is enabled, then `cisstNetlib` or `cnetlib` must also be enabled, as described in Section 5.3. If `cisstInteractive` is enabled, the path to `SWIG` and `Python` must be specified, as described in Section 5.4.

5.2 Building static or shared libraries

Both Windows and Unix support static (`.lib`, `.a`) and shared (`.dll`, `.so`) libraries. By default, the *cisst package* is built as static libraries. Set the `CISST_BUILD_SHARED_LIBS` option to `ON` to build shared libraries. Note that on some operating systems (such as Windows and Mac OS X) shared libraries are required in order to enable the Python wrapping. This requirement is checked and enforced by `CMake`. Also, if the *cisst package* is built as a shared library on Windows, it is necessary to define the `CISST_DLL` preprocessor symbol when compiling code that links with *cisst*, as described in Section 7.1.1.

Finally, on all operating systems, using shared libraries requires to set your paths correctly, as described in Section 7.3.

5.3 Numerical methods

The numerical methods provided by `cisstNumerical` require the `cisstNetlib` (recommended) or `cnetlib` library to be installed on the system. The user can enable or disable the use of `cisstNetlib` (`cnetlib`) by setting the CMake option `CISST_HAS_CISSTNETLIB` (`CISST_HAS_CNETLIB`) to be `ON` or `OFF`, respectively. If this option is `ON`, the user will have to provide the name of the directory where `cisstNetlib` (`cnetlib`) has been installed.

It seems redundant to use two CMake flags (`BUILD_LIBS_cisstNumerical` and `CISST_HAS_CISSTNETLIB` or `CISST_HAS_CNETLIB`), but `cisstNumerical` includes some methods, such as a Gauss-Jordan inverse for fixed-size matrices, that do not depend on `cisstNetlib` or `cnetlib`.

Note that for Mac OS X, the current distribution of `cisstNetlib` can not be used to link against the *cisst* libraries if they are compiled as shared libraries. This is the only case for which it is recommended to use `cnetlib`. This is a known problem that we intend to address before the next release.

5.4 Python interface

Automated wrapping of the C++ libraries for Python is enabled by turning `ON` the `CISST_HAS_SWIG_PYTHON` option. If this option is set, the user will have to provide the path to the SWIG directory (`SWIG_DIR`) and to the SWIG executable (`SWIG_EXECUTABLE`). In addition, the user will have to provide the path to the Python include directory, `PYTHON_INCLUDE_PATH`, and to the Python libraries `PYTHON_LIBRARY` and (for Windows) `PYTHON_DEBUG_LIBRARY`.

Note that the Python binary distribution for Windows does not include the Python debug library (e.g., `python24.d.lib` for Python Version 2.4), so this file (and the associated `dll`) must be created locally by compiling the Python source distribution. Alternatively, the user can decide to compile the software only in release mode.

As noted in Section 5.2, CMake may require the shared library option to be set in order to enable the Python wrapping. Whether you have to use shared libraries or not, the environment variable `PYTHONPATH` should probably be set based on the user configuration, so please be sure to read Section 7.3.

The Python wrapping can be used without the Interactive Research Environment (IRE) provided by `cisstInteractive`. For example, it is possible to start the Python interpreter and load the wrapped *cisst* libraries, as described in Section 8.1.

5.5 Test programs

The `BUILD_TESTS` variable is `OFF` by default. Turn it `ON` to enable building of the test programs. If this option is set, the user will have to provide the path to CppUnit. Note

that the test programs for each library can be individually enabled/disabled via the `BUILD_TESTS_libname` CMake variables.

It is strongly recommended to compile CppUnit and the *cisst package* with the same compiler.

5.6 Example programs

The `BUILD_EXAMPLES` variable is `OFF` by default. Turn it `ON` to enable building of the example programs. Note that individual example programs can be selected by setting the `BUILD_EXAMPLES_name` variable to `ON`. Some example programs may require software packages in addition to those listed in Section 2.

6 Result of the build process

Once the build process is complete, the directory structure should look something like the following (assuming that the examples and tests were enabled):

```

cisst
|-- source (the source tree)
|   |-- libs
|   |-- examples
|   '-- tests
'-- Linux-gcc3.4 (the build tree)
    |-- examples
    |   |-- bin (examples executables)
    |   '-- lib (examples specific libraries)
    |-- tests
    |   |-- bin (tests executables)
    |   '-- lib (tests specific libraries)
    '-- libs
        '-- lib (cisst libraries and Python wrappers)

```

Note that with Visual C++ under Windows, each `bin` and `lib` sub-directory will contain `Debug` and `Release` sub-directories, corresponding to whether the build was performed in debug or release mode, respectively.

7 Using the CISST Package with C++

This section describes how to use the *cisst package* with your C++ software. Two options are presented: i) building your software in the native environment and ii) using CMake to

configure your software. In addition, this section includes information about setting up the environment so that shared libraries (if used) can be located at runtime.

7.1 Using the native build environment

If you use your native build environment (e.g., Visual Studio for Windows, Makefiles for Unix), it is necessary to specify the paths to the include (header) files and to the library files for the compiler and linker, respectively.

The include path must include the header files in the source tree as well as the header files in the build tree, which for the example directory structure are:

```
.../cisst/source/libs/include  
.../cisst/Linux-gcc3.4/libs/include
```

The latter path includes header files that are generated during the build process of the *cisst package*, but are nevertheless necessary to incorporate *cisst* in your software. If other packages, such as `cisstNetlib`, are used, it is also necessary to set the path to their include files.

The library path must be to the build tree:

```
.../cisst/Linux-gcc3.4/libs/lib
```

Note that in a typical build of the *cisst package* under Windows, there will be `Debug` and `Release` directories for the library. The library user must point to the correct one. The simplest solution is to use the Visual Studio variable `$(INTDIR)` in the “Additional Library Directories” entry in the project settings, as in the following example:

```
...\\cisst\\Windows-VC71\\libs\\lib\\$(INTDIR)
```

If other packages, such as `cisstNetlib`, are used, it is also necessary to set the path to their static library files.

7.1.1 Special instructions for Windows

If the *cisst* libraries were compiled as shared libraries (dll’s), it is necessary to define the `CISST_DLL` preprocessor symbol in the compiler’s command line, typically by adding the option `/DCISST_DLL`. This option is necessary for the compiler to generate code that looks for the function definitions in the DLL rather than in a static library.

Also, pay attention to the fact that by default, the Visual Studio Solution file generated by CMake sets the *Runtime Library* setting (under *C/C++...Code Generation*) to “Multi-threaded DLL” or “Multi-threaded Debug DLL.” Your application must use the same value for this attribute. Therefore, you must do one of the following:

- Set your project *Runtime Library* setting to “Multi-threaded DLL” or “Multi-threaded Debug DLL.”
- Change the runtime library specified for the *cisst package*. The relevant CMake variables are `CMAKE_C_FLAGS_RELEASE` and `CMAKE_C_FLAGS_DEBUG`, which currently specify the runtime libraries via the `/MD` and `/MDd` flags, respectively.

7.2 Using CMake

The most portable way to include the *cisst package* in your software is by configuring your software with CMake. To simplify this process, during the configuration of the *cisst* libraries (with CMake), a file named `cisstBuild.cmake` was created in the build directory. This file can be included in your `CMakeLists.txt` to set the useful paths. For example, if you are creating a library called `newLibrary` from source files `newLibrary.cpp` and `newLibrary.h` as well as a program called `newProgram` from source file `main.cpp`, insert the following lines into your `CMakeLists.txt`:

```

1  # See license at http://www.cisst.org/cisst/license.txt
2  # Find the cisst settings
3
4  project (myProject)
5  cmake_minimum_required (VERSION 2.6)
6
7  find_package (cisst REQUIRED)
8
9  if (cisst_FOUND)
10     # Modify CMake configuration to use cisst
11     include (${CISST_USE_FILE})
12
13     # Add a new library
14     add_library (newLibrary newLibrary.h newLibrary.cpp)
15
16     # Add a new program and link against the new library
17     add_executable (newProgram main.cpp)
18     target_link_libraries (newProgram newLibrary)
19
20     # cisst_target_link_libraries will check that the libraries are compiled
21     # and set the correct link options
22     cisst_target_link_libraries (newProgram cisstCommon cisstVector cisstNumerical)
23 endif (cisst_FOUND)

```

When running the CMake configuration utility, you will see an entry labeled `CISST.CMAKE`, which you should set to the actual path (absolute or relative) of the file `cisstBuild.cmake`, which was generated in the *cisst* build directory. The `INCLUDE` command imports all the definitions made while building *cisst* into the current CMake session.

If you are using `cisstNumerical` in combination with `cisstNetlib` or `cnetlib`, you must add the include path for the header files and link your program with the library. For example, to use `cisstNetlib`, your `CMakeLists.txt` must contain:

```
INCLUDE_DIRECTORIES(${CISSTNETLIB_INCLUDE_DIR})
TARGET_LINK_LIBRARIES(newProgram ${CISSTNETLIB_LIBRARIES})
```

This example of CMake configuration file (`CMakeLists.txt`) is provided as part of the `cisst` package along with the C++ code for `newLibrary` and `newProgram` in `examples/separateCMake`.

7.3 Setting paths to shared libraries

If shared libraries are used, it is also necessary to make sure that the operating system can find them at run-time. This can be achieved by adding the appropriate path (e.g., `.../cisst/Windows-VC71/libs/lib/release`) to the environment variable `PATH` (for Windows), `LD_LIBRARY_PATH` (for Unix/Linux), or `DYLD_LIBRARY_PATH` (for OS X) or by copying the `.dll/.so` files to a directory already in the search path (such as `\Windows\system32` for Windows or `/usr/local/lib` for the Unix/Linux/OS X).

To simplify this process, different scripts are generated during the configuration (i.e. based on your CMake settings). These scripts allow to set the required paths based on your system and configuration (`PATH`, `LD_LIBRARY_PATH` and `PYTHONPATH`).

Unix: If you are using `csh` or `tcsh`, you can source `cisstvars.csh` in the build tree. If you are a `bash` or `sh` user, type `. cisstvars.sh`.

Windows: There are multiple ways to set the environment variables with Windows, but none of them are ideal in all situations. The following section presents some options.

7.3.1 Setting paths on Windows

On Windows, environment variables (including paths) are stored in the registry. The registry contains both *System* and *User* environment variables; the final setting of an environment variable is the concatenation of the *System* and *User* settings. The standard method for manually setting environment variables is to use the Control Panel, but the exact procedure depends on the version of Windows. Although it is possible to set or change an environment variable from a command shell using `SET` (either directly or from within a batch file), this change only applies within that command shell (i.e., locally) – the registry is not updated. Microsoft provides a command-line program, `SETX`, that can change environment variables in the registry, but it is not included in the default Windows installation. Similarly, the command line program `PATHMAN` manipulates the `PATH` environment variable in the registry,

but is also not included in the default Windows installation. Finally, CMake includes some capabilities for updating the Windows registry.

The recommended approach depends on whether the path change can be local (within the current shell and programs invoked from that shell) or whether it must be persistent (within the registry).

For local path changes, the following scripts are provided:

- `cisstvsvars.bat` sets the environment variables required to run Visual Studio.
- `cisstvars.bat` sets the different paths based on your `cisst` configuration. This script requires a parameter to indicate the build type, e.g. `cisstvars Release`.
- `visual-studio-release.bat`, `visual-studio-debug.bat`, ... These scripts set the variables for Visual Studio (by calling `cisstvsvars.bat`) and the `cisst` package (by calling `cisstvars.bat`) before loading Visual Studio. Once Visual Studio is started, remember to choose the correct Build Configuration (i.e. Release, Debug, ...).

For persistent path changes, the recommended approach is to define new environment variables, `CISST_PATH` and `CISST_PYTHONPATH`, for the paths required by the `cisst` software. Then, it is necessary to make a one-time update to the `PATH` and `PYTHONPATH` environment variables, by adding `%CISST_PATH%` and `%CISST_PYTHONPATH%`, respectively. This one-time change should be made using the standard Windows Control Panel. There are several methods for setting the `CISST_PATH` and `CISST_PYTHONPATH` variables in the registry:

- Use CMake to update the registry by turning ON the `CISST_PATH_REGISTER` option. This will require `CISST_PATH_CONFIGURATION` to be set (e.g., to `Release`, `Debug`, etc.). When the OK button is pressed, CMake will update `CISST_PATH` and `CISST_PYTHONPATH` in the registry. Unfortunately, due to a Windows quirk, it is necessary to take further action to finalize the registry update. One option is to open the Control Panel, edit the environment variables, and then press OK. Rebooting the system should also work.
- Start a command shell and call `cisstvars.bat` (with appropriate build type parameter, such as `Release`) to define `CISST_PATH` and `CISST_PYTHONPATH` in the local environment. Use the Control Panel to manually update the registry environment variables or, if `SETX` is available, update them by typing the following:

```
SETX CISST_PATH %CISST_PATH%  
SETX CISST_PYTHONPATH %CISST_PYTHONPATH%
```

In general, changing the Windows registry will not affect the local settings in any command shells that are already open, so it will be necessary to close them and open new ones. One exception is for the second method presented above, where both the local environment (due to running `cisstvars.bat`) and the registry will be correct.

8 Using the CISST Package with Python

There are two options for using *cisst* with Python. The first, described in Section 8.1, is to extend the Python interpreter by loading the wrapped *cisst* libraries. Conceptually, this produces an interactive environment that is similar to Matlab (though much less extensive!). The second option, described in Section 8.2, is to embed the Python interpreter into your C++ software. In both cases, it is possible to use the standard Python interpreter or to use the Interactive Research Environment (IRE), which is a GUI-based Python development environment provided by the `cisstInteractive` library. For more information about the IRE, consult the [Quick start for `cisstInteractive`](#).

In either case, the Python interpreter must be able to locate the wrapped library files. The recommended solution (for Windows and Unix) is to set the `PYTHONPATH` environment variable to the `libs/lib` directory in the *cisst* library build tree. For Windows, `PYTHONPATH` should include both the `libs/lib` directory (if using the IRE) and either the `Debug` or `Release` sub-directory. If running a program that uses its own wrapped library file, add the appropriate path to `PYTHONPATH`. For instance, to run the `pythonEmbeddedIRE` example program, add `examples/lib` (`examples/lib/Debug` or `examples/lib/Release` for Windows) to `PYTHONPATH`. For more information about setting paths, see Section 7.3.

8.1 Extending the Python interpreter

If the *cisst* libraries have been wrapped for Python, it is possible to load them into a Python shell as follows:

```
from cisstCommonPython import *
from cisstVectorPython import *
from cisstNumericalPython import *
```

As a quick test, it should be possible to create a vector of 3 elements and manipulate it as follows:

```
x = vct3()           # Create a vector of 3 elements
x.SetAll(10)        # Set all elements to 10
x[0] = 1            # Set the first element to 1
print x             # Should display:  1.00000  10.00000  10.00000
```

Note that the IRE can be used instead of the standard Python interpreter to take advantage of its GUI features. This is accomplished by providing the following commands to the Python interpreter (e.g., as a command-line argument):

```
import irepy; irepy.launch()
```

8.2 Embedding the Python interpreter

It is also possible to embed the Python interpreter into a C++ program and to share objects between C++ and Python. Sharing of objects is accomplished via an *object registry* provided by the `cisstCommon` library. A simple example is presented in the `pythonEmbedded` example program.

The `cisstInteractive` library facilitates the embedding of a GUI-based Python shell, called the Interactive Research Environment (IRE), in C++ programs. The IRE provides a more complete development environment, including a GUI that features a list of C++ objects (in the object registry), a list of global (Python) variables, the command history, and a logger output window. The `pythonEmbeddedIRE` example program provides a demonstration of embedding the IRE in a C++ program; the [Quick start for `cisstInteractive`](#) provides a detailed explanation of this example.