

ERC CISST Software

Quick start for cisstInteractive (IRE)

Peter Kazanzides

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

Contents

1	Introduction	2
2	Sharing C++ Objects with Python	2
2.1	Defining the class	2
2.2	Storing an object in the registry	3
2.3	Wrapping the class for Python	3
3	Embedding the IRE	4
3.1	IRE thread created in C++	5
3.1.1	Launching the IRE (C++ thread)	5
3.1.2	Waiting for the IRE to initialize (C++ thread)	6
3.1.3	Checking whether the IRE is active (C++ thread)	6
3.1.4	Granting execution time to the IRE (C++ thread)	7
3.1.5	Cleaning up (C++ thread)	7
3.2	IRE thread created in Python	7
3.2.1	Launching the IRE (Python thread)	7
3.2.2	Waiting for the IRE to initialize (Python thread)	8
3.2.3	Checking whether the IRE is active (Python thread)	8
3.2.4	Granting execution time to the IRE (Python thread)	8
3.2.5	Cleaning up (Python thread)	8
4	Building the example	9
5	Running the example	9
5.1	Environment	9
5.2	Dependencies	10
5.3	Startup file for pythonEmbeddedIRE	10
5.4	Using pythonEmbeddedIRE	11

1 Introduction

This document provides a quick start to embedding and using the *Interactive Research Environment (IRE)*, provided by `cisstInteractive`, in your C++ programs. The IRE is a graphical user interface that includes an interpreter (shell) for the Python programming language. Although the IRE can be used in standalone mode, it is most interesting when it is embedded in a C++ program and can access C++ objects via the object registry (provided by `cisstCommon`) because it allows the user to dynamically change objects in the C++ program.

The `pythonEmbeddedIRE` example program is used to illustrate the process. The complete source code can be found in the directory:

```
$(SRC)/examples/pythonEmbeddedIRE
```

where `$(SRC)` is the path to the `cisst` source code. The `pythonEmbeddedIRE` program consists of a C++ program that computes and outputs a sine wave of a given amplitude and frequency, using the `SineGenerator` class. This class is wrapped for Python, using `Swig` (www.swig.org). The IRE is embedded in this program and allows the user to dynamically change the amplitude and/or frequency of the generated sine wave. The following sections describe the steps that were necessary to create this example.

2 Sharing C++ Objects with Python

There are three steps that are required to share C++ objects with Python: 1) define a class that can be exported as a shared library and that allows instances (objects) to be stored in the object registry, 2) store the object in the registry, and 3) prepare the class to be wrapped with a Python interface.

2.1 Defining the class

The object registry requires that all objects be derived from `cmnGenericObject`. Therefore, the `SineGenerator` class should be declared as follows:

```
class CISST_EXPORT SineGenerator: public cmnGenericObject {
    CMN_DECLARE_SERVICES(CMN_NO_DYNAMIC_CREATION, 5);
};
```

```

    ...
};

CMN_DECLARE_SERVICES_INSTANTIATION(SineGenerator)

```

The `CISST_EXPORT` macro allows this class to be exported as a shared library (e.g., Windows DLL). The `CMN_DECLARE_SERVICES` (and related) macros register the class in a class registry (different from the object registry) and set up services such as class-specific logging via the `CMN_LOG_CLASS` macro. The `SineGenerator` class also includes public methods to get and set the amplitude and frequency, as well as methods to compute and return the output. See the source code for the complete listing.

2.2 Storing an object in the registry

In this example, the `pythonEmbeddedIRE.cpp` file creates an instance (object) of the `SineGenerator` class and stores it in the object registry (`cmnObjectRegister`). The relevant code to do this is:

```

cout << "*** Creating sine wave generator, amp=5, freq=0 ***" << endl;
SineGenerator wave(5.0, 0.0);

cout << "*** Registering sine wave generator ***" << endl;
cmnObjectRegister::Register("SineGenerator", &wave);

```

Note that the `cmnObjectRegister` class is a Singleton class (i.e., only one instance can be created) and therefore the static method `Register` should be called (this just calls `Instance()->RegisterInstance`). The parameters to this method are a character string to specify the object name (in this case, ‘‘`SineGenerator`’’, but any name can be used) and a pointer to the object.

2.3 Wrapping the class for Python

To wrap a class for Python, it is necessary to create a Swig interface file, such as `SineGeneratorPython.i`. The `SineGenerator` class is fairly straightforward, so a simple interface file suffices:

```

%module SineGeneratorPython
%mutable;

%header %{

```

```

#include "cisstCommon/cisstCommon.i.h"
#include "SineGenerator.h"
%}

%import "cisstCommon/cisstCommon.i"
#include "SineGenerator.h"

```

For further information about wrapping with Swig, consult the Swig documentation at (www.swig.org).

3 Embedding the IRE

Now that the `SineGenerator` class has been prepared, we are ready to embed the IRE in the C++ program. The first step is to include the `ireFramework.h` file, which defines the `ireFramework` Singleton class:

```
#include <cisstInteractive/ireFramework.h>
```

The next step is to decide how to create the IRE thread. The IRE must execute in a separate thread if the user wishes to concurrently execute the C++ program and the IRE. There are two options for creating the IRE thread:

1. Create a new thread in C++ and then call the `ireFramework` function to launch the IRE.
2. Call the `ireFramework` launch function from the current thread and ask Python to create a new thread for the IRE.

The first approach is recommended because the IRE will run in a thread created from C++ and will be managed consistently with other threads in the program (i.e., multithreading will work as expected). The disadvantage, however, is that it is not portable unless an operating system abstraction library, such as `cisstOSAbstraction`, is used.

In contrast, the second approach is completely portable because it uses the Python threading module. The disadvantage, however, is that the C++ threads must proactively grant execution time to the Python-created thread, such as by periodically calling the `ireFramework::JoinIREShell` method, as described in Section 3.2.4.

The `pythonEmbeddedIRE` example program implements both approaches, using the `CISST_OSATHREAD` preprocessor symbol to distinguish between them. The setting of this symbol is controlled by the CMake option `EXAMPLE_IRE_USE_OSATHREAD`, which is defined in the `CMakeLists.txt` file in the `pythonEmbeddedIRE` directory.

The last line above creates a new thread that calls the `Run` method for the IRE object, which is an instance of the `IreLaunch` class. This is equivalent to the call:

```
IRE.Run("from pythonEmbeddedIRE import *");
```

The first parameter to the `Run` method is a startup command that is passed to the Python interpreter. In this case, it imports the `pythonEmbeddedIRE` module, which is presented in Section 5.3. The second parameter to `LaunchIREShell` is set to `false`, which specifies that Python should not create a new thread (i.e., the IRE will run in the current thread). This is the default value for this parameter, and so it could also have been omitted.

Note that the call to `LaunchIREShell` is made within a `try...catch` block because it will throw a `std::runtime_error` exception if the IRE cannot be started (e.g., if the `irepy` module cannot be found).

In this threading scenario, the `LaunchIREShell` method does not return until the IRE is exited, so the `Run` method calls the `FinalizeShell` method to clean up the embedded Python interpreter before exiting and thereby ending the thread.

3.1.2 Waiting for the IRE to initialize (C++ thread)

The IRE framework can take a few seconds to initialize Python and load the `wxPython` package. For programs with real-time tasks, it is advisable to allow this initialization to occur before the real-time tasks are started. The IRE framework includes an `IsStarting` method that returns `true` if the IRE is not active and not finished. This generally means that the IRE is initializing. Note that `IsStarting` does not check whether `LaunchIREShell` was called because the new C++ thread may not yet have started execution. Programs can therefore include a loop such as the following:

```
while (ireFramework::IsStarting())
    osaTime::Sleep(500); // Wait 0.5 seconds
```

Here, the `osaTime::Sleep` method is called to suspend the current thread for the specified amount of time, leaving more processor time for the IRE thread to complete its initialization.

3.1.3 Checking whether the IRE is active (C++ thread)

The `ireFramework::IsActive` method can be used to check whether the IRE is active (i.e., has completed initialization, but has not been exited). Alternatively, one can call `ireFramework::IsFinished` to check whether the IRE has exited.

3.1.4 Granting execution time to the IRE (C++ thread)

Because the IRE thread was created in C++, it is not necessary to explicitly grant execution time to it. IRE initialization can be significantly faster, however, if the C++ program sleeps during this time, as shown above.

3.1.5 Cleaning up (C++ thread)

Before the C++ program exits, it should wait for the IRE thread to terminate; for example, by calling:

```
IreThread.Wait();
```

When the user exits the IRE interface, the `LaunchIREShell` method will return, the `FinalizeShell` method will be called, and the IRE thread will then terminate.

3.2 IRE thread created in Python

This technique is used if `CISST_OSATHREAD` is not defined. The implementation of the `IreLaunch` class is as follows:

```

1 // Launch IRE in Python-created thread
2 class IreLaunch {
3 public:
4     IreLaunch() {}
5     ~IreLaunch() { ireFramework::FinalizeShell();}
6     void Run(char *startup) {
7         try {
8             ireFramework::LaunchIREShell(startup, true);
9         }
10        catch (...) {
11            cout << "***_ERROR:_could_not_launch_IRE_shell_***" << endl;
12        }
13    }
14 };

```

3.2.1 Launching the IRE (Python thread)

The launch process is nearly identical to the first method, except that the second parameter to `LaunchIREShell` is set to `true`. In this threading scenario, the `LaunchIREShell` method returns as soon as Python creates a new thread and begins the IRE initialization, so the current thread can continue with other computations (see, however, Section 3.2.4). As before, the `try...catch` block is recommended to catch exceptions that may be thrown by `LaunchIREShell`.

3.2.2 Waiting for the IRE to initialize (Python thread)

The IRE framework can take a few seconds to initialize Python and load the wxPython package. For programs with real-time tasks, it is advisable to allow this initialization to occur before the real-time tasks are started. The IRE framework includes an `IsStarting` method that returns `true` if the IRE is not active and not finished. Programs can therefore include a loop such as the following:

```
while (ireFramework::IsStarting())
    ireFramework::JoinIREShell(0.001);
```

Here, the `JoinIREShell` method is required to grant execution time to the Python thread (see Section 3.2.4).

3.2.3 Checking whether the IRE is active (Python thread)

The C++ program can check whether the IRE is active by calling `ireFramework::IsActive`. Alternatively, it can call `ireFramework::IsFinished()` to check whether the IRE has exited.

3.2.4 Granting execution time to the IRE (Python thread)

Although the C++ program can continue execution after the `LaunchIREShell` method returns, it must periodically call the `ireFramework::JoinIREShell` method with a non-zero `timeout` parameter. Otherwise, the IRE Python thread will not get any execution time. This is a major disadvantage of using the Python thread.

3.2.5 Cleaning up (Python thread)

Before the C++ program exits, it should clean up the Python interpreter by making the following call:

```
ireFramework::FinalizeShell()
```

It is not necessary to check whether or not the IRE is still active. If it is active, the `FinalizeShell` instance will not return until the IRE is exited. Note that the `IreLaunch` class presented earlier calls `FinalizeShell` from its destructor.

4 Building the example

The build steps are as follows:

1. Run Swig to generate the wrappers for the class(es) for which objects will be shared between Python and C++ (e.g., the example `SineGenerator` class).
2. Compile those wrapped class(es) into a shared library (or libraries), which will later be imported into Python.
3. Compile the example program into an executable. If the example program uses the object registry and embeds the IRE, it must link with the `cisstCommon` and `cisstInteractive` libraries.
4. If the program uses a Python file (besides those generated by Swig), make sure that at least one of the Python source files (`.py` or `.pyc`) is copied to the binary directory. This example uses the `pythonEmbeddedIRE.py` source file, which is described in [Section 5.3](#).

The `pythonEmbeddedIRE` example uses CMake to manage the build process and the preceding four steps are handled in the `CMakeLists.txt` file.

5 Running the example

This section describes the steps that are necessary to run the `pythonEmbeddedIRE` example program. The first two subsections describe the environment and dependencies on external packages. The third subsection presents the startup file, `pythonEmbeddedIRE.py`, which facilitates execution of the example. The final subsection discusses the usage of the `pythonEmbeddedIRE` example program.

5.1 Environment

To execute the software, it is necessary that:

1. The operating system find the shared libraries required by the C++ executable at runtime.
2. The Python interpreter find the modules that it must import.

The recommended solution is to add the necessary paths to the appropriate environment variables, as described in the [CISST Software Build Instructions](#). The alternative (not

recommended) is to copy the shared libraries and Python modules to paths that are already searched for these items.

The first requirement is addressed by adding the path to the shared libraries to the environment variable `PATH` (for Windows), `LD_LIBRARY_PATH` (for Unix/Linux) or `DYLD_LIBRARY_PATH` (for OS X). Note that with Visual C++ under Windows, the `lib` directories (e.g., `libs\lib` and `examples\lib`) in the `cisst` build tree will contain `Debug` and `Release` sub-directories, corresponding to whether the build was performed in debug or release mode, respectively. Therefore, the `PATH` must be set to one of these sub-directories. For all other operating systems, the environment variable should be set to the `lib` directories in the `cisst` build tree.

The second requirement is addressed by defining the `PYTHONPATH` environment variable, which should also be set to the `lib` directories in the `cisst` build tree. For Windows, `PYTHONPATH` should include both the `libs\lib` directory (so that Python can find the `irepy` module) and either the `Debug` or `Release` sub-directory (so that Python can find the wrapped C++ libraries).

For more information, see the [CISST Software Build Instructions](#).

5.2 Dependencies

If the software has been successfully compiled, most dependencies (such as the Python interpreter) have already been resolved. At runtime, the IRE requires the `wxPython` package, version 2.5 or later, which is a Python interface to the popular `wxWidgets` cross-platform GUI development package. The `wxPython` package is already installed on OS X Version 10.4 or later. For other systems, it can be downloaded from www.wxpython.org. Be sure to download the version of `wxPython` that corresponds to the version of Python installed on your system. For more information, see the [CISST Software Build Instructions](#).

5.3 Startup file for pythonEmbeddedIRE

For this example, we created a Python source file, `pythonEmbeddedIRE.py`, to facilitate usage of the program. This file imports all necessary modules, retrieves a reference to the `SineGenerator` object from the object registry (bound to the Python variable `wave`), and prints some helpful information:

```
from cisstCommonPython import *
from SineGeneratorPython import *

wave = cmnObjectRegister.FindObject("SineGenerator")
print "Sine Wave: amplitude = %f, frequency = %f" % \
      (wave.GetAmplitude(), wave.GetFrequency())
```

```
print "Use wave.SetAmplitude(X) to set amplitude to X"  
print "Use wave.SetFrequency(Y) to set frequency to Y"  
print "Use wave.SetFrequency(0) to pause output display"
```

5.4 Using pythonEmbeddedIRE

The pythonEmbeddedIRE program runs in a terminal window. It creates a `SineGenerator` object (with frequency initially set to 0), registers it in the object registry, launches the IRE, and then waits for the IRE to complete its initialization. Once the IRE is initialized, it waits for the frequency to be set to a non-zero value or the IRE to be exited. Note that OS X does not generally allow terminal programs to access the display and therefore pythonEmbeddedIRE employs a workaround where it “borrows” the identity of the `pythonw` program to gain access to the display.

As noted in the instructions displayed by the startup file, the frequency can be changed by specifying a new value to `wave.SetFrequency`. Once a non-zero value is set, the C++ program will display the sine wave parameters (time, amplitude, frequency) and computed output to the terminal window. The amplitude and frequency can be dynamically changed via calls to `wave.SetAmplitude` and `wave.SetFrequency`, respectively. The current settings of these parameters can be obtained by `wave.GetAmplitude` and `wave.GetFrequency`, respectively.

Once the sine wave display has been started, it can be stopped by setting the frequency back to 0 or by exiting the IRE. See the following section for instructions on using the IRE GUI window.

6 IRE User Interface Quick Reference

Once the IRE has been started, a window similar to that shown in Figure 1 will be displayed. This window is divided into the following main areas:

Menu Bar	Contains the following menus (note that on OS X, the menu bar is not attached to the IRE window; it is displayed at the top of the screen, which is the convention for that operating system):
File	Most options (e.g., <i>New</i> , <i>Open</i> , <i>Close</i> , <i>Save</i> , <i>Save As</i>) are used to create, open or save a Python source file. The <i>Exit</i> option is used to exit the IRE. On OS X, <i>Exit</i> is under the <i>Python</i> menu.
Edit	All these options (e.g., <i>Undo</i> , <i>Redo</i> , <i>Cut</i> , <i>Paste</i> , and <i>Select All</i>) can be used when a Python source file is being edited.

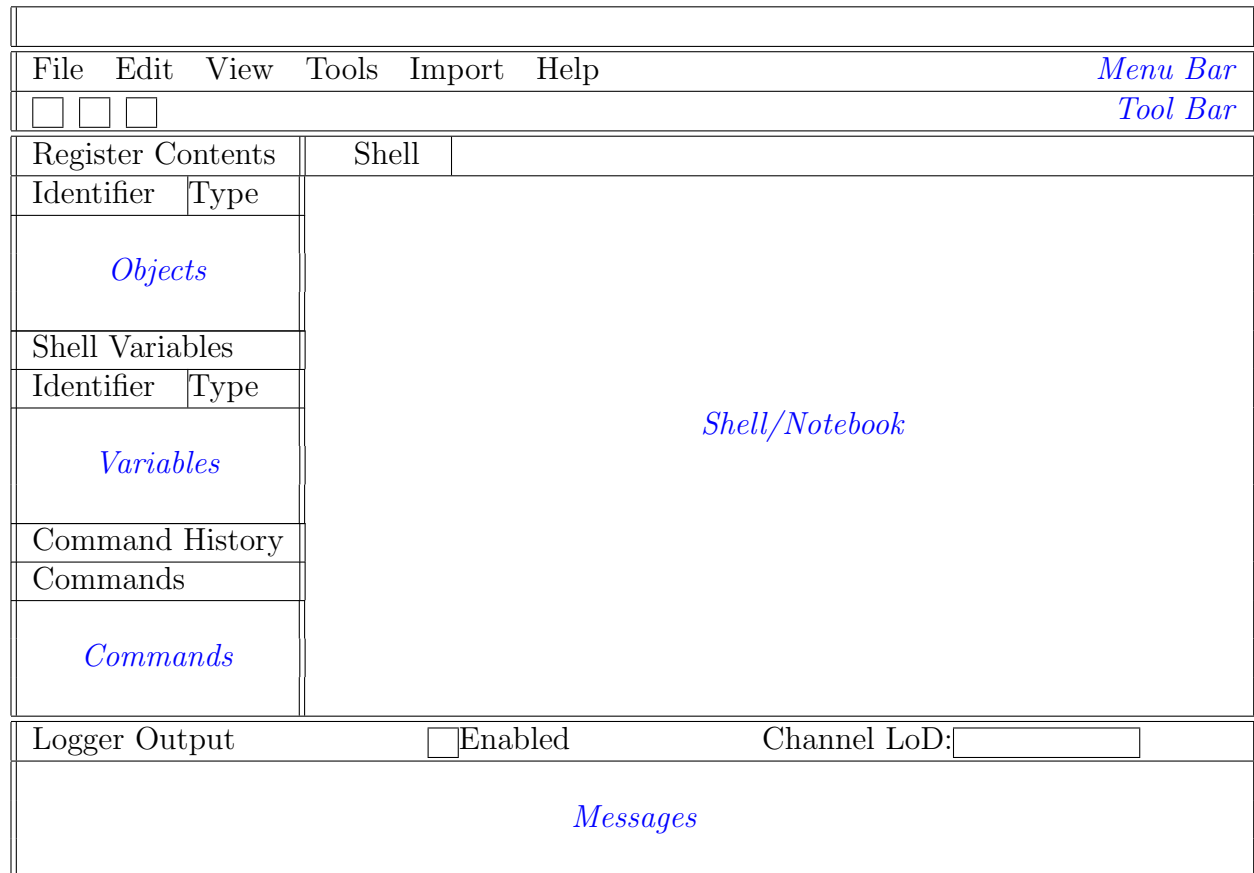


Figure 1: IRE User Interface

View This menu contains options to toggle the display of the *Objects*, *Variables*, *Commands*, and *Messages* windows. Any of these four windows can be hidden from view by unchecking its menu entry. The remaining windows will automatically resize to fill the available space.

Tools This menu contains options to load and save the workspace (i.e., all variables defined in the current shell), run the currently selected Python file in the Shell, and options to manipulate the command history. There is also a menu item to display a *test input box*; this has no useful function.

Import This menu contains options to import the wrapped cisst libraries that are present on the PYTHONPATH. If a library is not present, the *Import* menu option for that library is disabled.

Help This menu contains an *About* option, which displays information about the IRE. On OS X, this option is located under the *Python* menu.

Tool Bar Contains graphical icons for New, Open and Save. These are equivalent to the New, Open and Save options in the File menu.

- Shell/Notebook*** This window initially contains a notebook tab for the *Shell*, which provides the interactive Python interpreter. Additional notebook tabs are created when any Python source files are created or opened and provide a simple editor window. The shell provides pop-up help information, such the documentation string or a list of all class attributes (methods and data), as commands are typed. Commands can be entered directly from the keyboard, dragged in from the Command History window, selected from the shell's built-in command history (navigated using the Ctrl-Up and Ctrl-Down keys), or obtained from any open Python file (e.g., using the *Run in shell* option from the *Tools* menu).
- Objects*** This area displays all objects (identifier and type) that are stored in the C++ object registry. The display can be alphabetically sorted by *Identifier* or *Type* by clicking on the respective column header.
- Variables*** This area displays all variables (identifier and type) that are defined in the global scope of the shell. The display can be alphabetically sorted by *Identifier* or *Type* by clicking on the respective column header. One or more variables can be dragged from this area and dropped into the Shell window to display their current values.
- Commands*** This area displays the command history. It can be sorted in chronological or reverse-chronological order by clicking on the column header. One or more commands can be dragged from this area and dropped into the Shell window for immediate execution. This command history is saved to a file and therefore persists after the IRE is exited. The user can clear the command history or truncate it using options in the *Tools* menu. Note that the truncate menu item also provides the option to archive the deleted commands to a new history file.
- Messages*** If enabled, this area displays all messages received from the cisst C++ logging utilities (`cmnLogger`). Note that the logging utilities do not send messages that have a level of detail (LoD) that is higher than the specified *Channel LoD*.