How to Create a Library Module in ROS

Wyatt Newman
October, 2014

Introduction: So far, we have created independent packages that take advantage of existing libraries. However, as your source code gets lengthier, it is desirable to break it up into smaller modules. If your work may be re-used by future modules, then it is best to create a new "library". This document describes the steps to creating a library. It will refer to the examples in the class repository under:
…/examples/example_library/example_cuttability_ik_lib and
…/examples/example_library/example_cuttability_main

The code in these examples is based on
…/examples/example_ik/src/atlas_cuttability_solns_8dof_v3.cpp. The intent of this code is to compute all viable solutions for holding a tool (a drill motor with a spinning bit) at a specified pose, but allowing for some rotation of the tool about its tool axis, and including q(0) (torso rotation about z-axis) and squatting motions (over a viable range of pelvis from 0.5m to 0.85m above the floor). The original code contained a "main" program, multiple functions defined in the same *.cpp file, and various global variables. As is often the case, this code started out a reasonable length, then grew to become overly cumbersome. Further, the code is "stand-alone" but needs to be integrated within a larger system (including choosing an optimal path among the many options, and executing the resulting plan on the robot). To make the code more manageable and more useful, it is being converted into a library.

In this process, it is preferable to define a "class" and specify its member variables and methods. This gets rid of global variables, and it makes it easier to use by other nodes.

Note: the completed library implementation may be found in the class repository under
...catkin/src/kinematics_stack/cuttability_ik, which creates the library "cuttability_ik."

Making a new library:

To begin, the new library package is created under …/catkin/src/examples/example_library using:

```
hku_create_pkg example_cuttability_ik_lib roscpp hku_msgs
atlas_conversions eigen atlas_msgs geometry_msgs
atlas_hand_kinematics
```

The new package is called "example_cuttability_ik_lib", it will be written in C++, and it depends on various other packages, including team-hku's own "atlas_hand_kinematics" library. We will create a new library in this package.

In addition, to illustrate how to take advantage of the new library, a second package is created, using:

```
hku_create_pkg example_cuttability_main roscpp
example_cuttability_ik_lib
```

This package is intended to use the capabilities in our anticipated new library "example_cuttability_ik_lib."

In our new library package, example_cuttability_ik_lib, there is a subdirectory "include." This is where we will put a header file, cuttability_ik.h, that will be included by packages utilizing this library.

In our header file, cuttability_ik.h, we define a class, cuttability_ik. A common technique (which you should emulate) is to start each header file with:

```
#ifndef CUTTABILITY_IK_H
#define CUTTABILITY_IK_H
```

and end the header file with:

```
#endif
```

The result of this is that, if a future module includes another module that *also* includes this same header file, duplication is avoided. A keyword unique to this header file (in this case, chosen to be "CUTTABILITY_IK_H") is only defined to exist if the compiler pre-processor finds this file. If this keyword is already defined (via some other non-obvious #include), then the inclusions will not be duplicated. If you follow this technique with all header files you create, then you will avoid possible problems with redundant inclusions.

Our header file includes many other headers, which will simplify header-file inclusions in future nodes that utilize our new library.

The heart of the header file prototypes the new class:

```
class cuttability_ik {
private:
   // will use convention: trailing underscore ("_") indicates private member variable/object or method
   TaskVariableXYZRPY *fwd_kin_solver_; // make the fwd_kin solver a class member, so accessible by member fncs
   TaskVariableXYZRPY *fwd_kin_solver_utorso_wrt_pelvis_; // same for utorso solver
   atlas_hand_ik_solver *ik_solver_; //make this a class member, so callback can use it

public:
   bool is_reachable_w_ytool_spin(Eigen::Affine3d a_tool_transform_inv, Eigen::Affine3d a_tool_nom, std::vector<Vectorq6x1> &q_solns);
   bool is_reachable_w_ytool_spin_and_dq0(Eigen::Affine3d a_tool_transform_inv, Eigen::Affine3d a_tool_seed_wrt_pelvis, std::vector<Vectorq7x1> &q_arm_and_utorso_solns);
   bool is_reachable_8dof(Eigen::Affine3d a_tool_transform_inv, Eigen::Affine3d a_tool_seed_wrt_pg, std::vector<Vectorq8x1> &q_arm_and_utorso_and_squat_solns);
   cuttability_ik(); //constructor; optional args?  //const hand_s& hs, const atlas_frame& base_frame, double rot_ang);
};
```

The "implementation" (the actual algorithms within the various methods declared here) should appear in a separate *.cpp file, not the header file.

In the same package (…/catkin/src/examples/example_library), we create a new *.cpp file in the “src” sub-directory. The version in the class code called “cuttability_ik_skeleton.cpp” is a “skeletal” version of our implementation. That is, the member methods are implemented as mere shells, to be filled in with real code later. (This is an intermediate process you might take to convert a bloated C++ program into a class and a library).

In the source code, we include our new header file:
#include "cuttability_ik.h"

In contrast to the original code on which this library is based, we can see that there are no more global variables (although there are still some constants defined in the header). Each of the member functions are defined with the class prefix, e.g.:

bool cuttability_ik::is_reachable_w_ytool_spin();

Since we are defining a class, there must be a class constructor of the same name, appearing as:
```
cuttability_ik::cuttability_ik() { }
```

The constructor is a desirable place to put initializations, making it easy for future users of the library. In the example, two objects are instantiated from the class TaskVariableXYZRPY (which is team-HKU code that uses KDL). These are initialized to compute forward kinematics for the right-hand grasp frame with respect to the pelvis frame, and the second solver computes the utorso frame with respect to the pelvis frame. Additionally, an object of type “atlas_hand_ik_solver” is instantiated, using the Team-HKU “atlas_hand_kinematics” library. All of this initialization work is performed automatically when an object of type “cuttability_ik” is instantiated by a user of the library. (It is also possible to set up callbacks and publishers in the constructor, further simplifying higher-level code).

Each of the methods declared in our header file are defined in this implementation *.cpp file. At present, these are legal, but useful functions. Although our new library does not yet provide value, it should be compilable. This can be tested, first, then each function can be filled in incrementally and tested incrementally.

In the directory: …/catkin/src/examples/example_library, we also have our package.xml and CMakeLists.txt files. The package.xml file references the various libraries on which we will depend. This list was populated by our original hku_create_pkg command, although we can edit this file to add more packages, if we later find we need additional capabilities not already declared. Having anticipated the packages on which we will depend, we will not need to edit the package.xml file.

The CMakeLists.txt file requires some edits. The lines:
```
include_directories(
  include
  ${catkin_INCLUDE_DIRS}
)
```
inform the compiler that we will be using a header file in our “include” directory.

Also, since we will be using some of the C++ extensions, we uncomment the line:

```
SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++0x")
```
(you should be able to do the above by default, in general).

The line:
```
set(CMAKE_BUILD_TYPE Release)
```
will produce “release” code, which has less debugging capability, but runs faster.

The line:
```
cs_add_library(example_cuttability_ik_lib
src/cuttability_ik_skeleton.cpp)
```

tells the compiler that we are creating a new library called “example_cuttability_ik_lib”, based on the source code in the subdirectory /src/ cuttability_ik_skeleton.cpp

This completes the edits to the CMakeLists.txt file. We can now create our new library with:
```
    hku_make -d example_cuttability_ik_lib
```

The new library is now ready to use.

Using the new library: Our second package, `example_cuttability_main,` shows how to make use of the new library. This package was created with:

```
hku_create_pkg example_cuttability_main roscpp
example_cuttability_ik_lib
```

and thus the package.xml makes reference to our new library.

The source code “main_cuttability.cpp” is located within the subdirectory /src. This code includes our new library header:
```
    #include <cuttability_ik.h>
```
The main program instantiates an object called “cuttability” of type “cuttability_ik”, based on our new library:

```
  cuttability_ik cuttability;
```

With this declaration, the constructor in cuttability_ik will perform various initializations, easing use of this object.

Subsequently, the main program is able to invoke any of the public methods within cuttability_ik, as illustrated by the line:

```
  bool result = cuttability.is_reachable_8dof(a_tool_transform_inv,
     a_tool_seed_wrt_pg,q_arm_and_utorso_and_squat_solns);
```

To compile the example main function, the CMakeLists.txt file must be edited. The C++ extensions are enabled by uncommenting:

SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++0x")

and the target file to be compiled is specified with the line:
`cs_add_executable(main_cuttability src/main_cuttability.cpp)`

The client can then be compiled with:
`hku_make -d example_cuttability_main`
The main program can then be run (assuming gazebo has already been started) with:

`rosrun example_cuttability_main main_cuttability`

Upon start-up, you will see output messages from the constructor, as well as messages from constructors of objects instantiated within the cuttability_ik class.