How to Create a Library Module in ROS

Wyatt Newman
October, 2015

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: …/example_ros_library and …/example_ros_library_usage

The code in these examples is based on …/example_eigen/example_eigen_plane_fit.cpp. The intent of this code is to compute a least-mean-squared-error planar fit to data. In example_eigen_fit.cpp, this was done in a (long) "main()" program. In the present example, we will put the functionality in a class, then make the class a library.

Making a new library:

To begin, the new library package is created under …/example_library using catkin_simple:

```
cwru_create_pkg example_ros_library roscpp std_msgs cmake_modules

The cmake_modules dependency is used to help bring in the Eigen
library (a special case for Eigen; cmake_modules is otherwise not
typically needed).
```

The new package is called "example_ros_library", it will be written in C++. 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:

```
cwru_create_pkg example_ros_library_usage roscpp cmake_modules
std_msgs example_ros_library
```

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

In our new library package, example_ros_library, there is a subdirectory "include." This is where we will put our libraries header file, plane_fitter.h, that will be included by packages utilizing this library. Although it is somewhat cumbersome, it is convention to create a sub-sub-directory for this. Specifically, under "include", create a subdirectory that is the same as the package name, "example_ros_library", and in this subdirectory save the header file "plane_fitter.h." Using this convention can make it easier to find the header files your create, once they are installed in the system.

In our header file, plane_fitter.h, we define a class, PlaneFitter. As introduced earlier, we invoke the compiler instruction:

```
#ifndef PLANE_FITTER_H
#define PLANE_FITTER_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 "PLANE_FITTER_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 also includes other headers, which will simplify header-file inclusions in future nodes that utilize our new library.

The header file prototypes the new class:

```
class PlaneFitter
{
public:
   PlaneFitter();
   void fit_points_to_plane(Eigen::MatrixXd points_array,Eigen::Vector3d &plane_normal, double
&plane_dist);
   void generate_planar_points(Eigen::Vector3d normal_vec,double dist,int npts, double
noise_gain,Eigen::MatrixXd &points_array);

private:
   // put private member data here;  "private" data will only be available to member functions of this
class;

}; // note: a class definition requires a semicolon at the end of the definition
```

This class has a constructor and (only) two member functions—one to generate sample data, and the other to fit a plane to data.

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 (…/example_ros_library), we create a new *.cpp file in the "src" sub-directory: plane_fitter.cpp, which will contain the implementation of our new library. Additionally, a test "main()" is created to illustrate how to use the library. Including an illustrative "main" program can be helpful in reminding yourself—or other users of your library—how to exploit your library.

In the source code (of both cpp files), we include our new header file:
`#include <example_ros_library/plane_fitter.h>`

ROS will looke for the package "example_ros_library", and it will navigate to the expected "include" sub-directory to find the desired header file.

The constructor is a desirable place to put initializations, making it easy for future users of the library. In the present example, there are no intializations. The constructor function thus does nothing.

Each of the methods declared in our header file are defined in this implementation *.cpp file. Drawing from the code example_eigen_plane_fit.cpp within the package example_eigen, the body of this code, both for sample point generation and for plane fitting, is largely copied/pasted into the implementation of our two methods.

Our package.xml is similar to the example_eigen as well. The CmakeLists.txt also includes the references to cmake_modules and to Eigen, as in the example_eigen package.

The notable difference in CmakeLists.txt is the line:
cs_add_library(plane_fitter src/plane_fitter.cpp)

With this statement, we inform the compiler that we wish to create a new library. This library will be called “plane_fitter” and it will reside within the package “example_ros_library.”

In addition, an example “main” program is compiled, using this library, with the lines:

```
cs_add_executable(plane_fitter_main src/plane_fitter_main.cpp)
target_link_libraries(plane_fitter_main plane_fitter ${catkin_LIBRARIES})
```

After running “catkin_make”, 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. Example usage is shown in the main program, “plane_fitter_main.cpp.” This program includes the new library header file:
#include <example_ros_library/plane_fitter.h>

In the body of the code, an object of type “PlaneFitter” is instantiated:
PlaneFitter planeFitter;

Then its member functions are invoked with:
planeFitter.generate_planar_points(normal_vec,dist,npts,noise_gain,points_array);
and:
planeFitter.fit_points_to_plane(points_array,est_normal_vec, est_dist);

The example main may be run with:
rosrun example_ros_library plane_fitter_main.

In addition, it is shown how to use the new library from a new package. A new package,

“example_ros_library_usage” is created, using catkin_simple, with:
  cwru_create_pkg example_ros_library_usage roscpp cmake_modules example_ros_library

The only difference in this case is that our “package.xml” now makes reference to the package that contains our new library (example_ros_library).

Within this package, we edit the CmakeLists.txt to compile our new user program with the lines:
    cs_add_executable(plane_fitter_external_main src/plane_fitter_external_main.cpp)
    target_link_libraries(plane_fitter_external_main plane_fitter ${catkin_LIBRARIES})

The source code “plane_fitter_external_main.cpp” is located within the subdirectory /src. This code is identical to the previous example code, plane_fitter_main.cpp. The name has been changed to avoid ambiguity (and naming conflict) with the original main code.

After compiling with catkin_make, the new main program can be run as:
  rosrun example_ros_library_usage plane_fitter_external_main

This yields the same result as our test function within the library's package.