

# Documenting your Code with Doxygen

Wyatt Newman

October, 2015

A benefit of open-source code is that developers can take advantage of prior work. For this to be effective, however, the code must be well documented. Certainly, the programmer should embed substantial comments in the body of their code—but there is also a need for the future user to quickly understand how to use the code, hopefully without needing to learn the details of the implementation. For this purpose, one should provide documentation at a level that describes the purpose of the node or library, as well as details and meanings of inputs and outputs. To help the author create effective and attractive documentation, tools such as “Doxygen” have been developed. An example of such documentation is shown below. If one does a search on “ROS Publisher”, one of the options will point to [http://docs.ros.org/jade/api/roscpp/html/classros\\_1\\_1Publisher.html](http://docs.ros.org/jade/api/roscpp/html/classros_1_1Publisher.html), which is shown (in part) below:

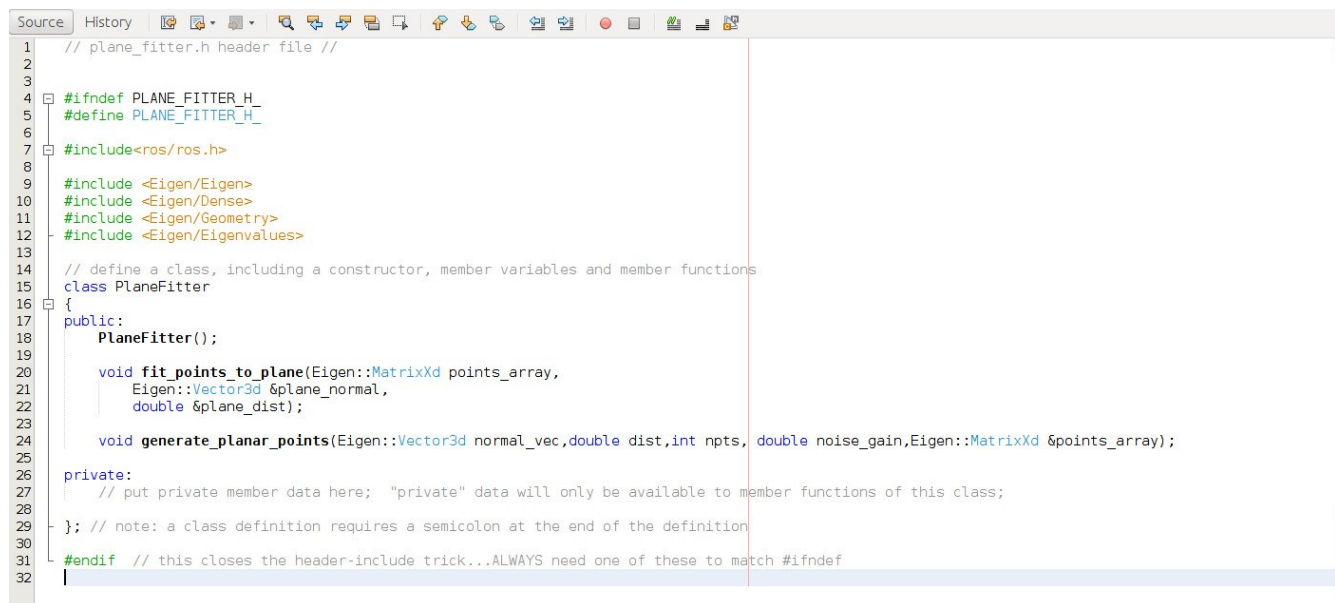
The screenshot shows a web browser displaying the ROS Publisher Class Reference page. The browser's address bar shows the URL `docs.ros.org/jade/api/roscpp/html/classros_1_1Publisher.html`. The page has a navigation bar with tabs for Main Page, Related Pages, Namespaces, Classes (selected), and Files. Below the navigation bar, there are sub-tabs for Class List, Class Hierarchy, and Class Members. The main content area is titled "ros::Publisher Class Reference" and includes a brief description: "Manages an advertisement on a specific topic. More...". It also shows the include directive `#include <publisher.h>` and a link to the list of all members. The "Classes" section lists the `Impl` class. The "Public Member Functions" section lists several functions: `getNumSubscribers() const`, `getTopic() const`, `isLatched() const`, `operator void *() const`, `operator!= (const Publisher &rhs) const`, `operator< (const Publisher &rhs) const`, `operator== (const Publisher &rhs) const`, `publish (const boost::shared_ptr< M > &message) const`, `publish (const M &message) const`, `Publisher()`, `Publisher (const Publisher &rhs)`, `shutdown()`, and `~Publisher()`. The "Private Types" section lists `typedef boost::shared_ptr< Impl > ImplPtr` and `typedef boost::weak_ptr< Impl > ImplWPtr`. The "Private Member Functions" section is partially visible at the bottom.

This display documents the “Application Programming Interface” (API), which helps the potential user

learn how to use this package. Typically, the contributed code will be organized in classes, and these will be documented in terms of member functions and their arguments.

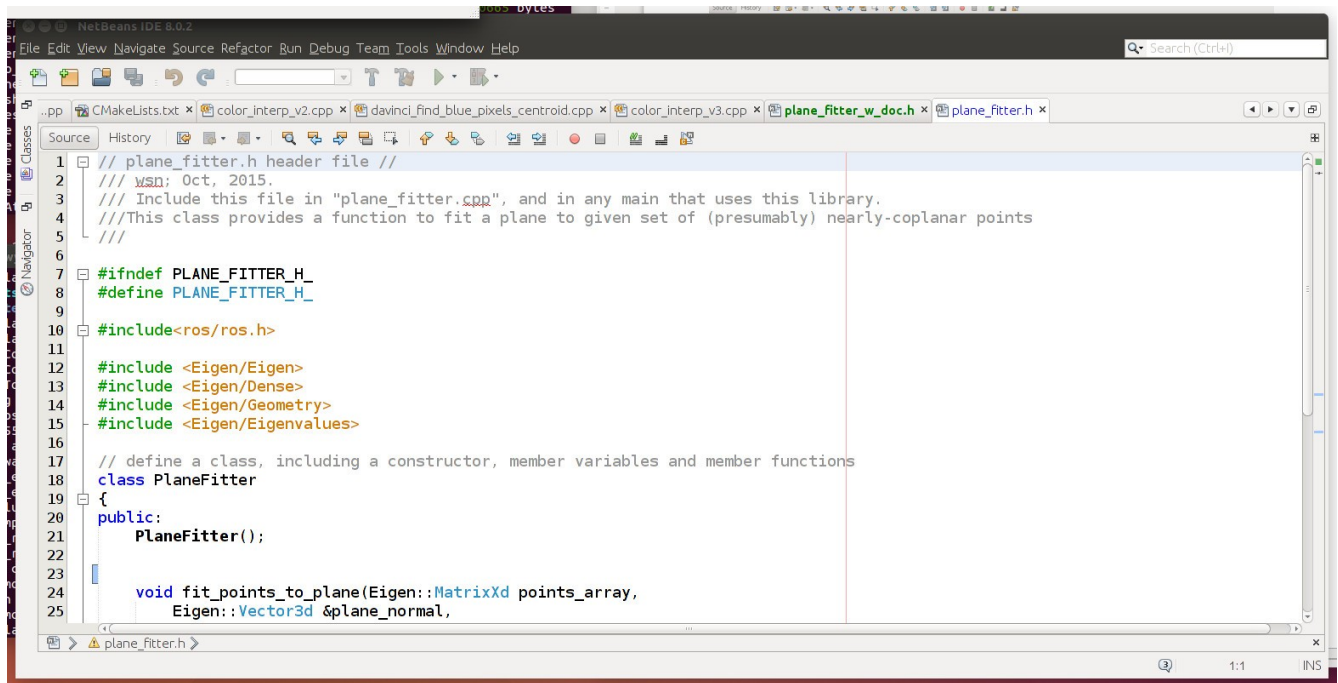
To generate documentation in this style, one can use Doxygen, or the GUI alternative “doxywizard” (which are free applications). Only a very cursory introduction will be presented here. A full manual and details about Doxygen can be found at: [www.doxygen.org](http://www.doxygen.org).

For illustration, the following refers to the header file of our example plane\_fitter library, “plane\_fitter.h.” This header file defines the class PlaneFitter, and it includes a constructor and member functions fit\_points\_to\_plane() and generate\_planar\_points(). The undocumented file is as follows:

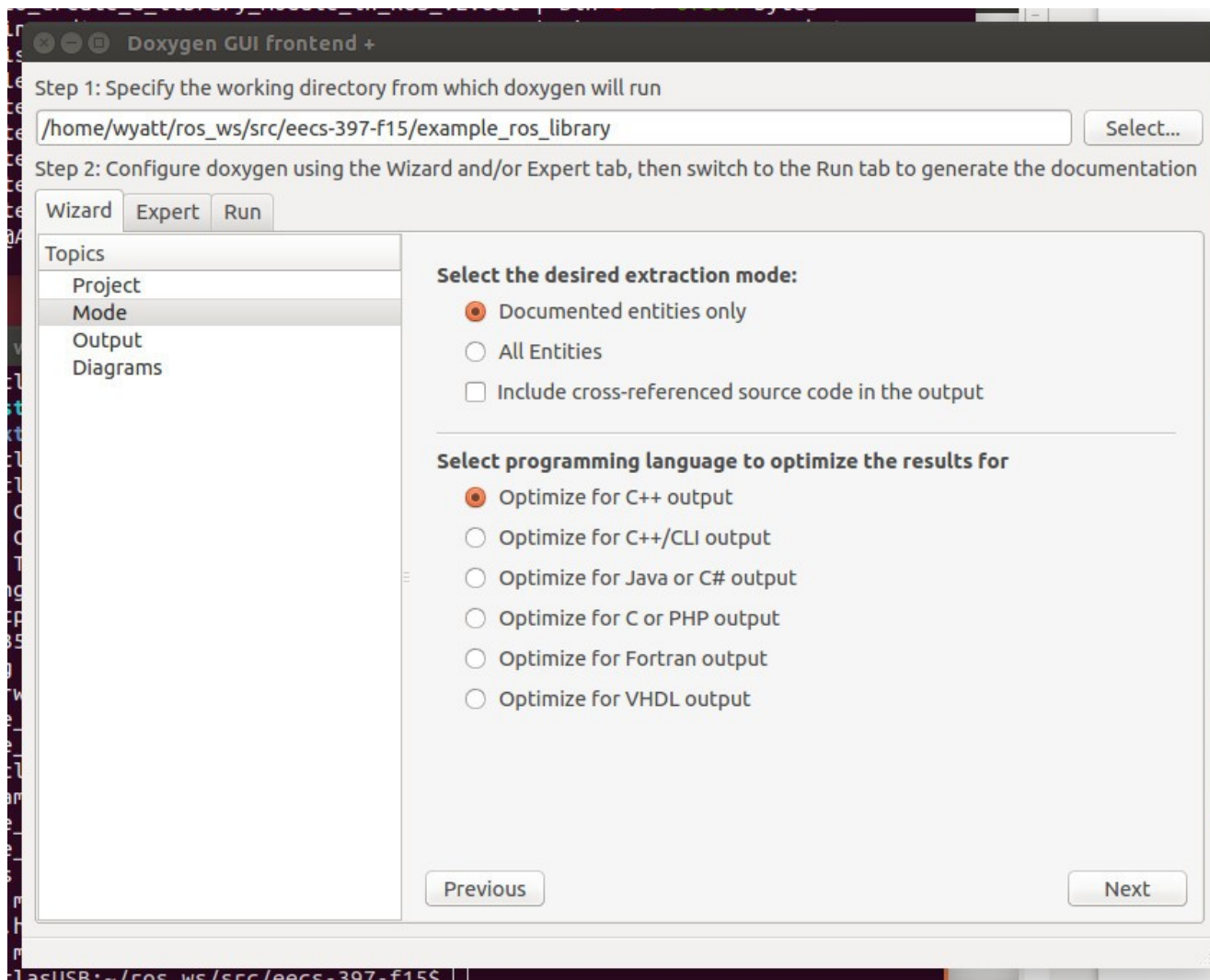


```
1 // plane_fitter.h header file //
2
3
4 #ifndef PLANE_FITTER_H_
5 #define PLANE_FITTER_H_
6
7 #include <ros/ros.h>
8
9 #include <Eigen/Eigen>
10 #include <Eigen/Dense>
11 #include <Eigen/Geometry>
12 #include <Eigen/Eigenvalues>
13
14 // define a class, including a constructor, member variables and member functions
15 class PlaneFitter
16 {
17 public:
18     PlaneFitter();
19
20     void fit_points_to_plane(Eigen::MatrixXd points_array,
21                             Eigen::Vector3d &plane_normal,
22                             double &plane_dist);
23
24     void generate_planar_points(Eigen::Vector3d normal_vec, double dist, int npts, double noise_gain, Eigen::MatrixXd &points_array);
25
26 private:
27     // put private member data here; "private" data will only be available to member functions of this class;
28
29 }; // note: a class definition requires a semicolon at the end of the definition
30
31 #endif // this closes the header-include trick...ALWAYS need one of these to match #ifndef
32
```

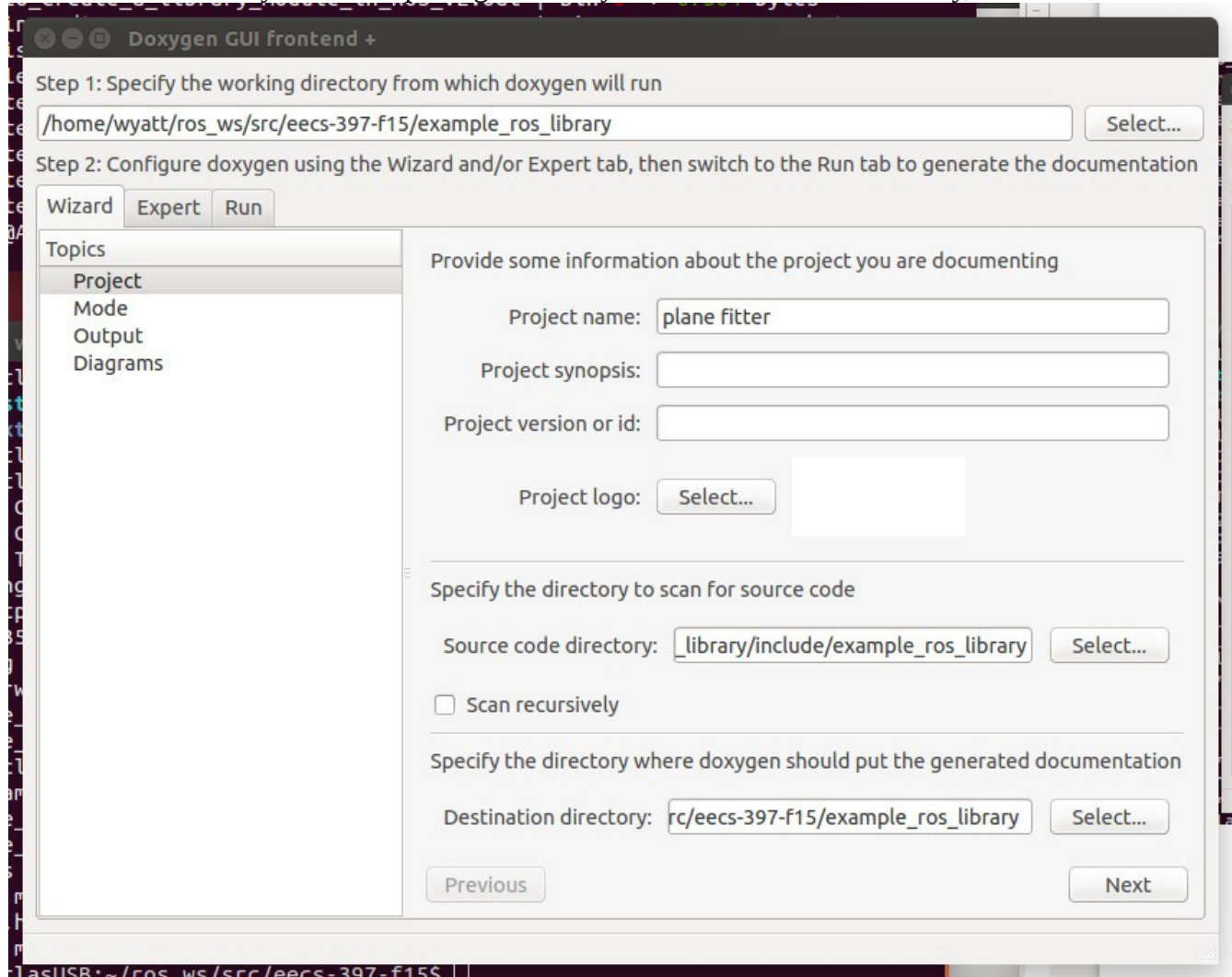
At the top of the file, we will insert our first Doxygen comment. This will begin with three slashes: ///, and there will be at least two lines that begin with ///. The inserted lines are shown in the following:



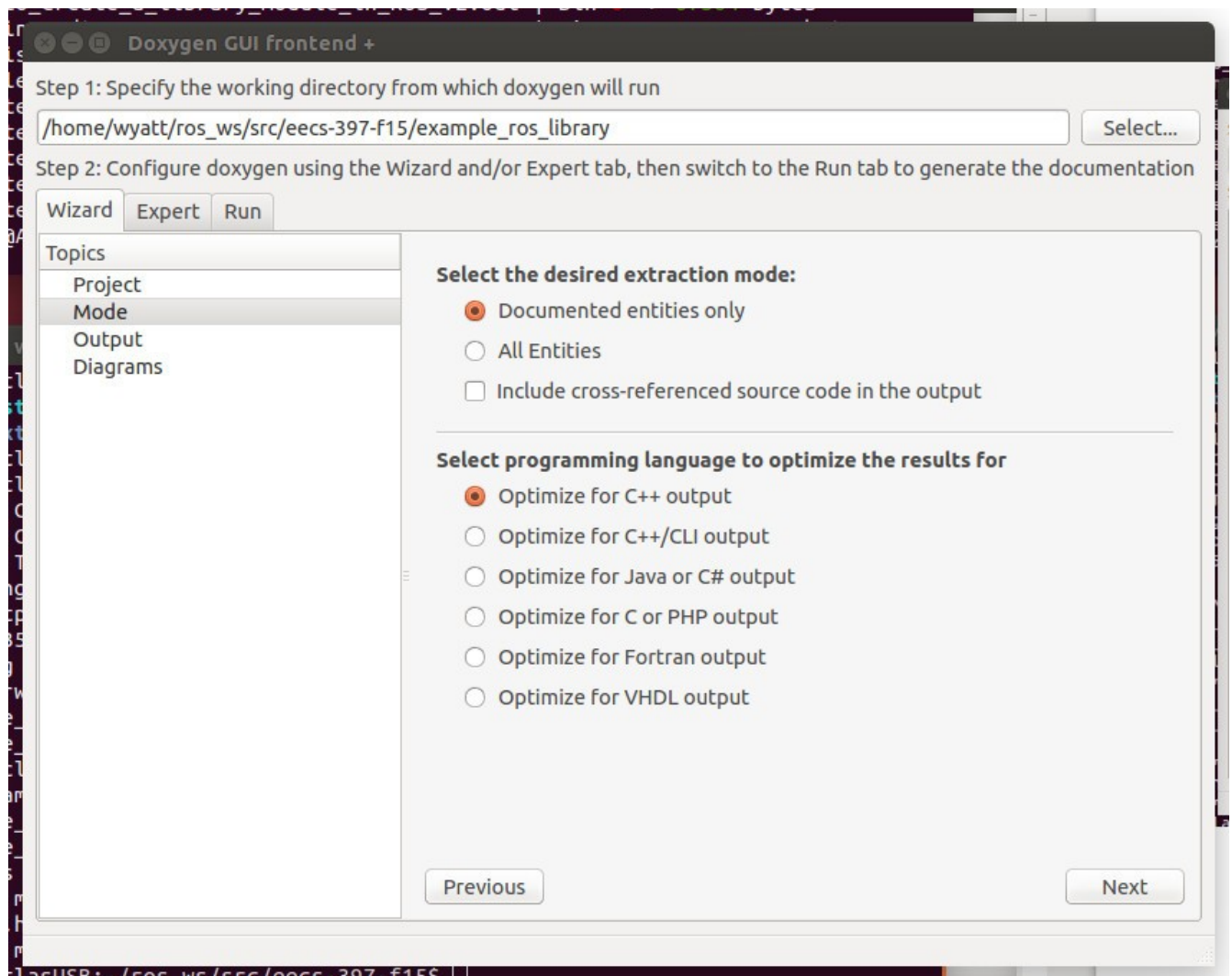
With our first Doxygen comment, we can already begin to generate and preview our formatted documentation. Open a terminal and enter “doxywizard” to start the program doxywizard (assuming you have installed this program; if not, do so). The first screen will appear as below. Navigate to the



package directory of the package to be documented as the doxygen “working directory.” Enter a project name (e.g. the package or library name). Navigate to the “include” subdirectory that contains the class prototype (example\_ros\_library/include/example\_ros\_library, in this case) to specify the “source code directory.” Use the package directory as the “destination” directory, as below:

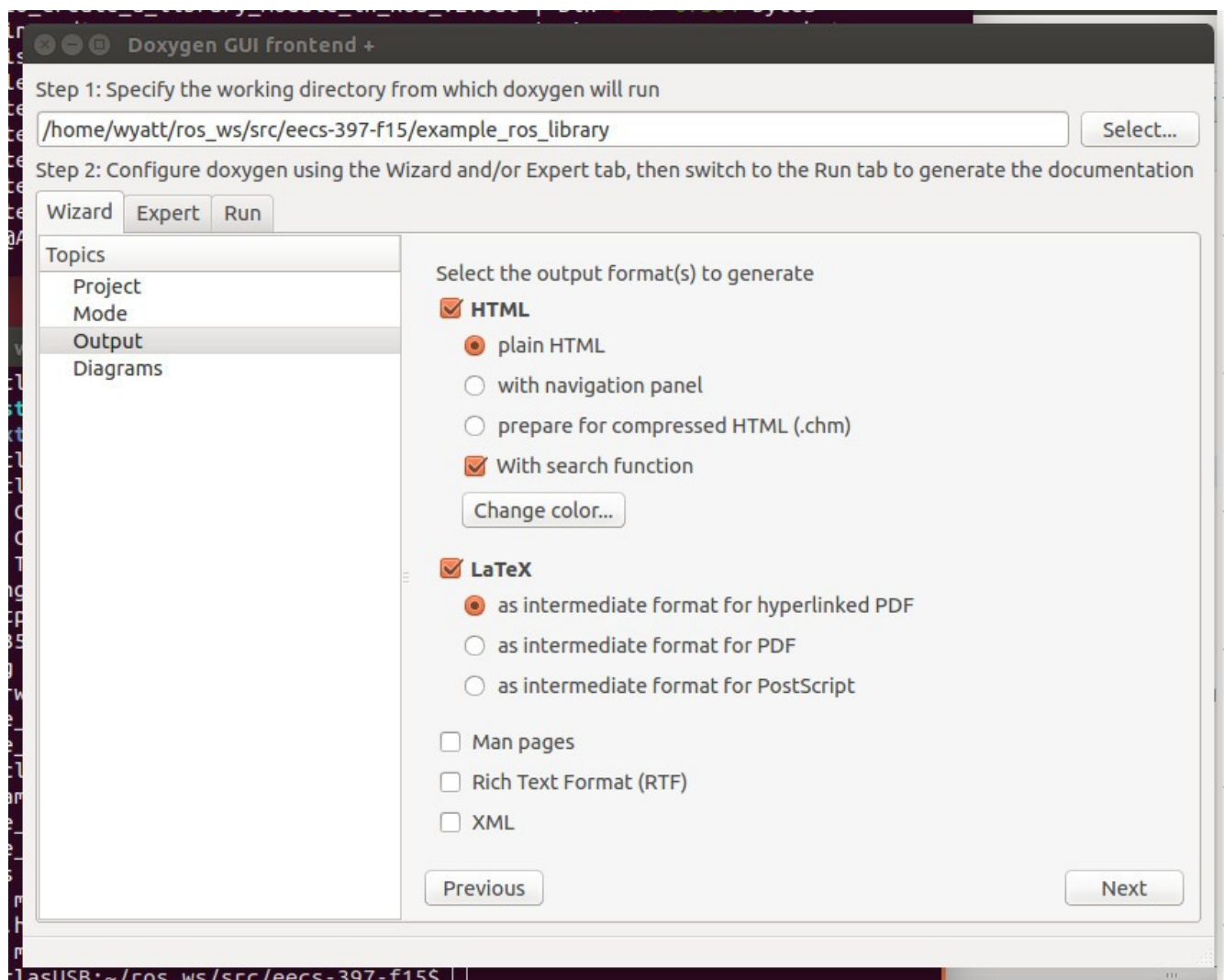


Advance to the next screen with the “Next” button. The next screen appears as follows:

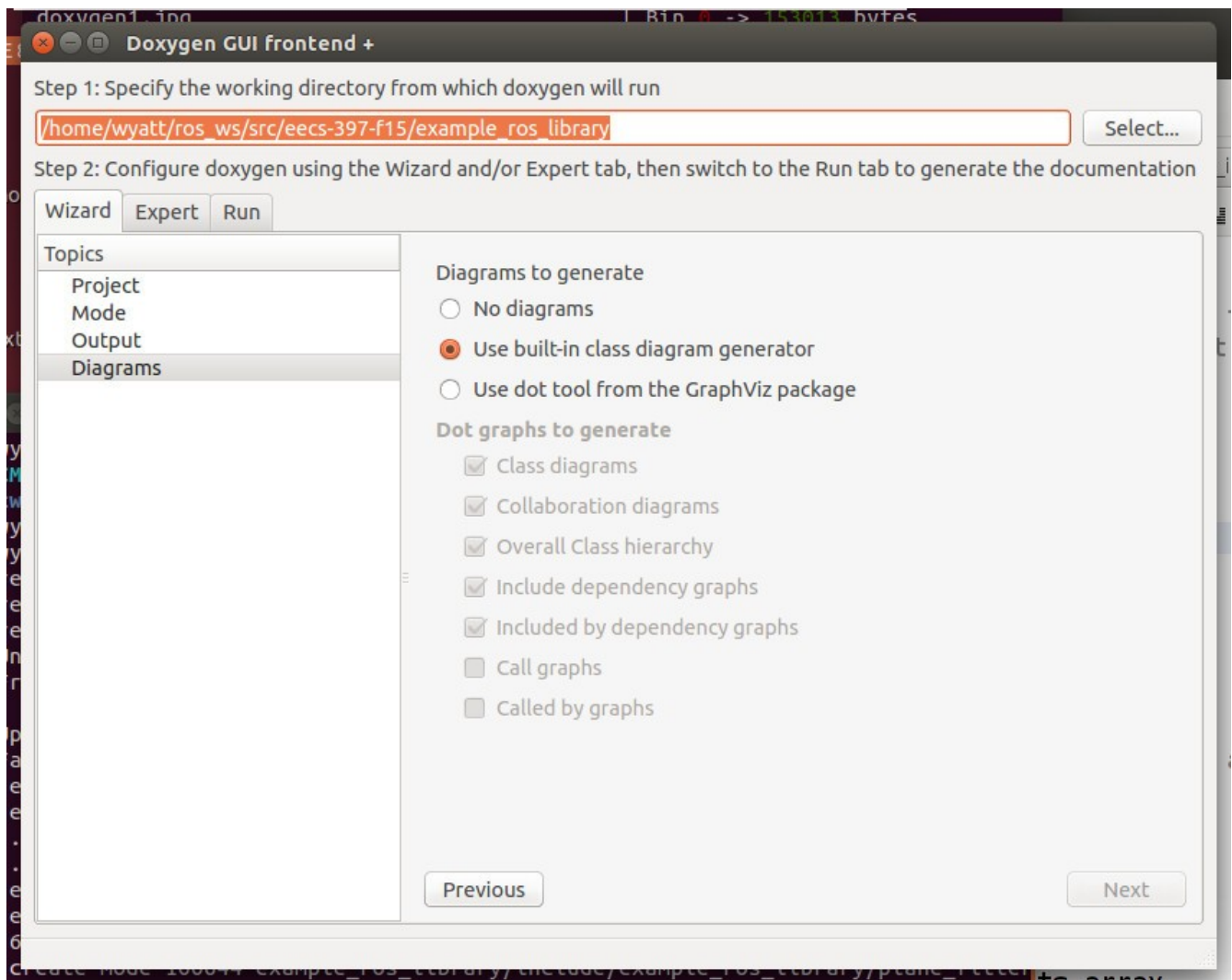


We will accept the defaults of this screen (including “optimize for C++”) and press “Next”. The next screen appears as:

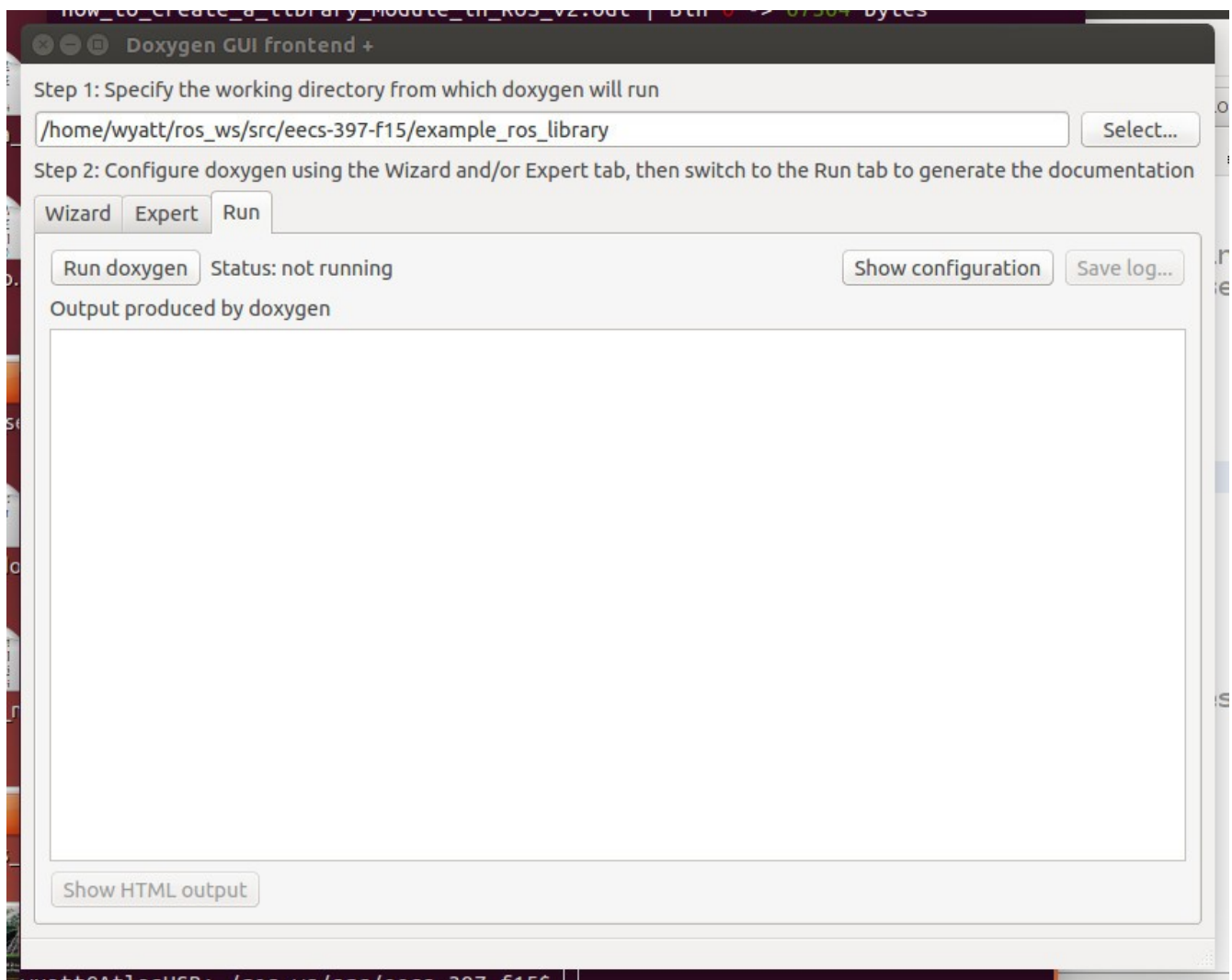




We will accept the defaults here as well, including “HTML” as an output. Click “Next” which brings up:

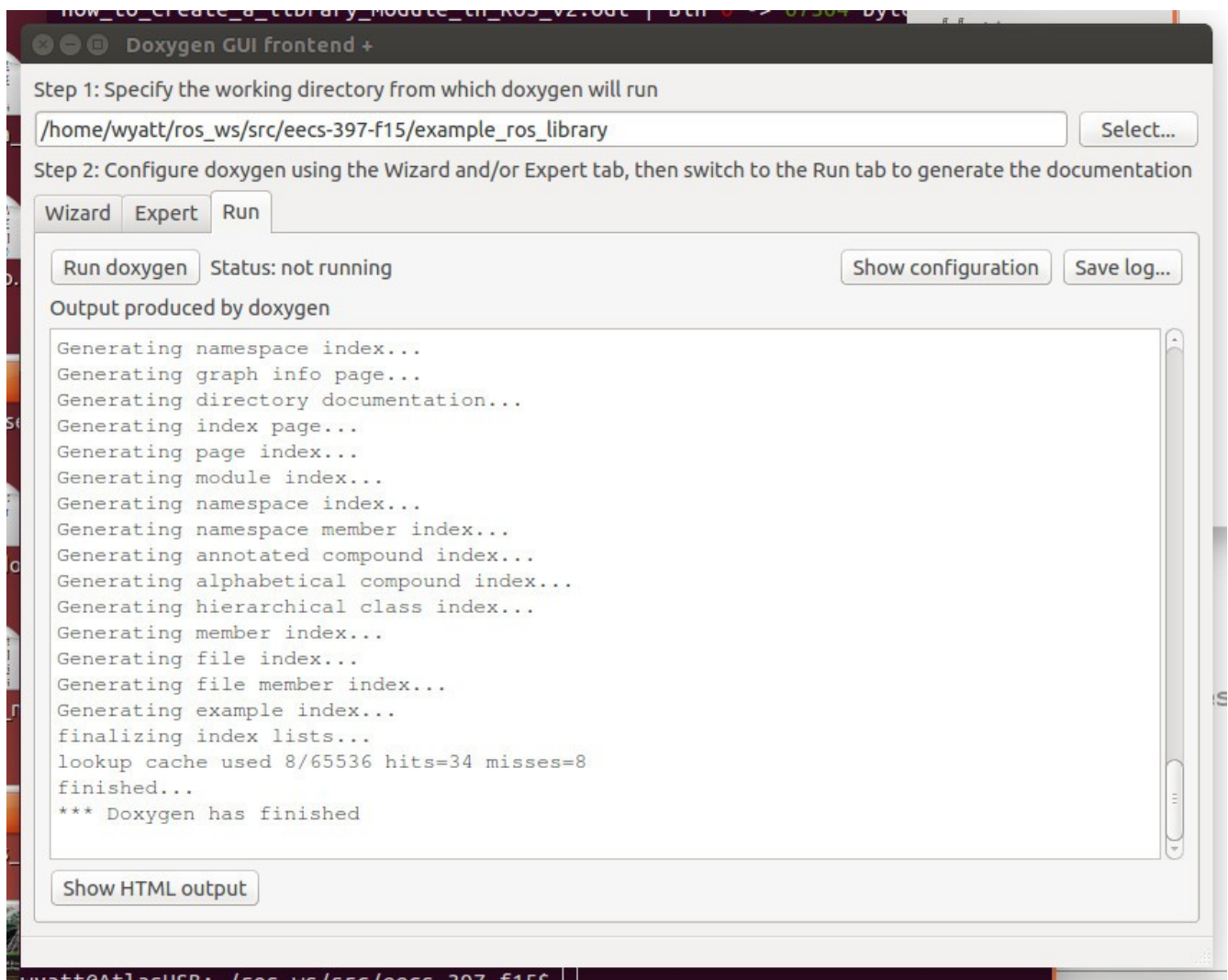


We will accept the defaults here as well. Having configured Doxygen processing, will invoke doxygen by clicking the “Run” tab, which brings up a button option “Run doxygen”, as below:

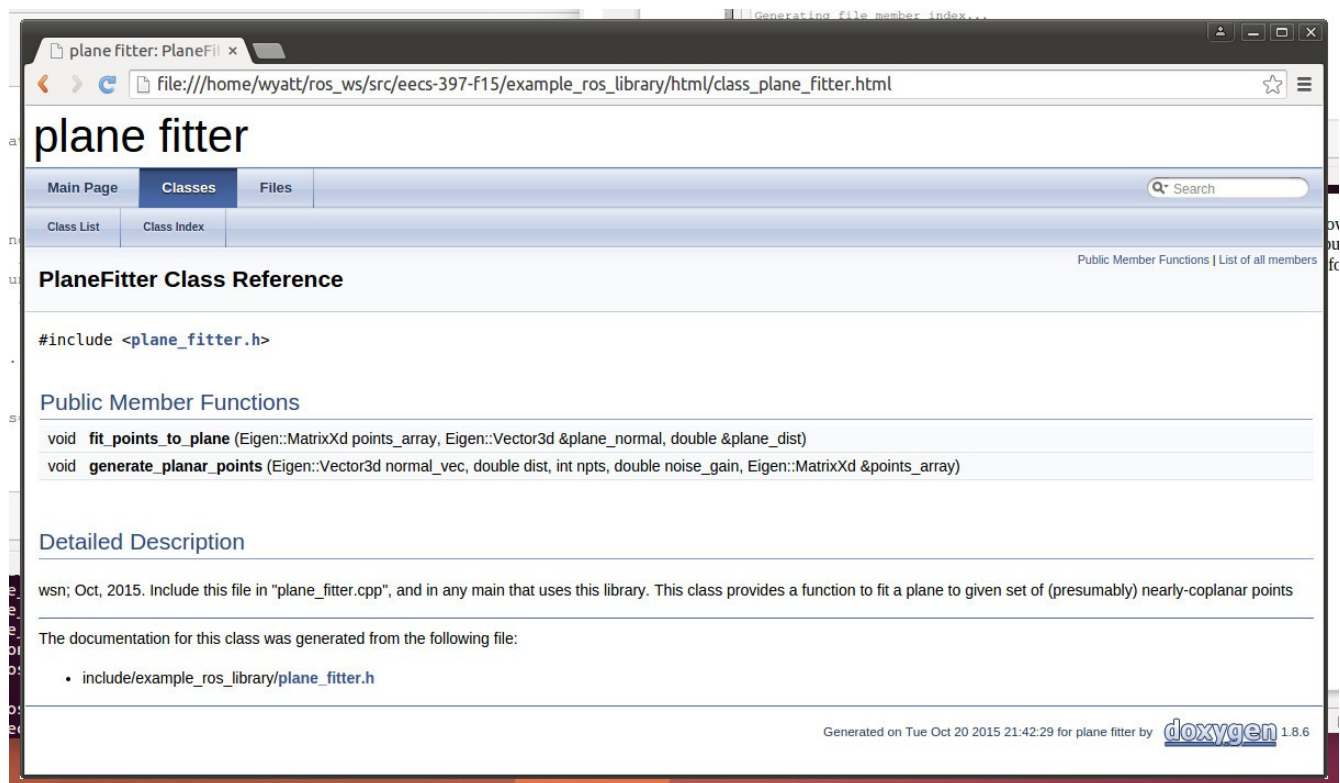


Click the “Run doxygen” button. The window will appear as:





Next, click the button “Show HTML output.” As a result, two new directories will show up within the package: “html” and “latex.” From the doxywizard window, click “Show HTML output”. This will bring up the formatted documentation in a browser. Our example, so far, displays as follows:

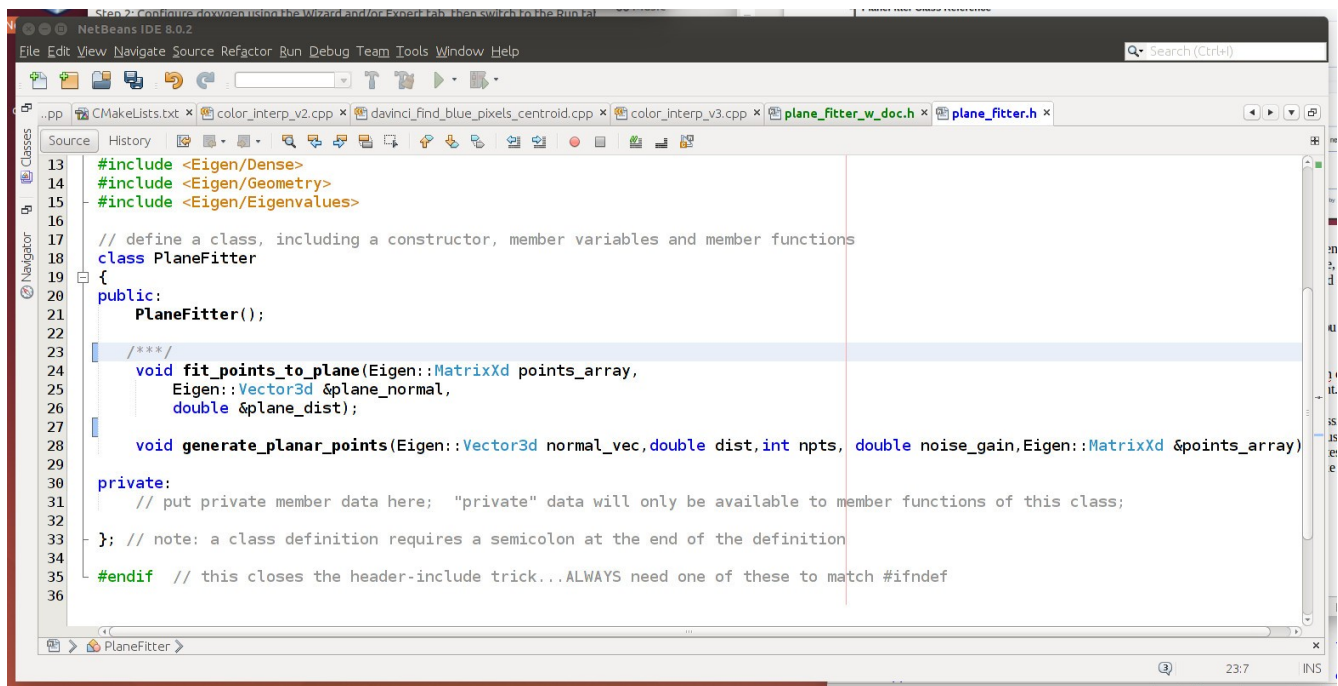


The above view was two clicks away from the “home” level. Clicking on “Classes”, then on the (only) option “PlaneFitter” brings up the above view. In this view, we can see the project name, the class name, our comment, and a list of the public member functions (`fit_points_to_plane()` and `generate_planar_points()`).

At a minimum, we should document the public member functions, since future users would want to know how to invoke this functionality.

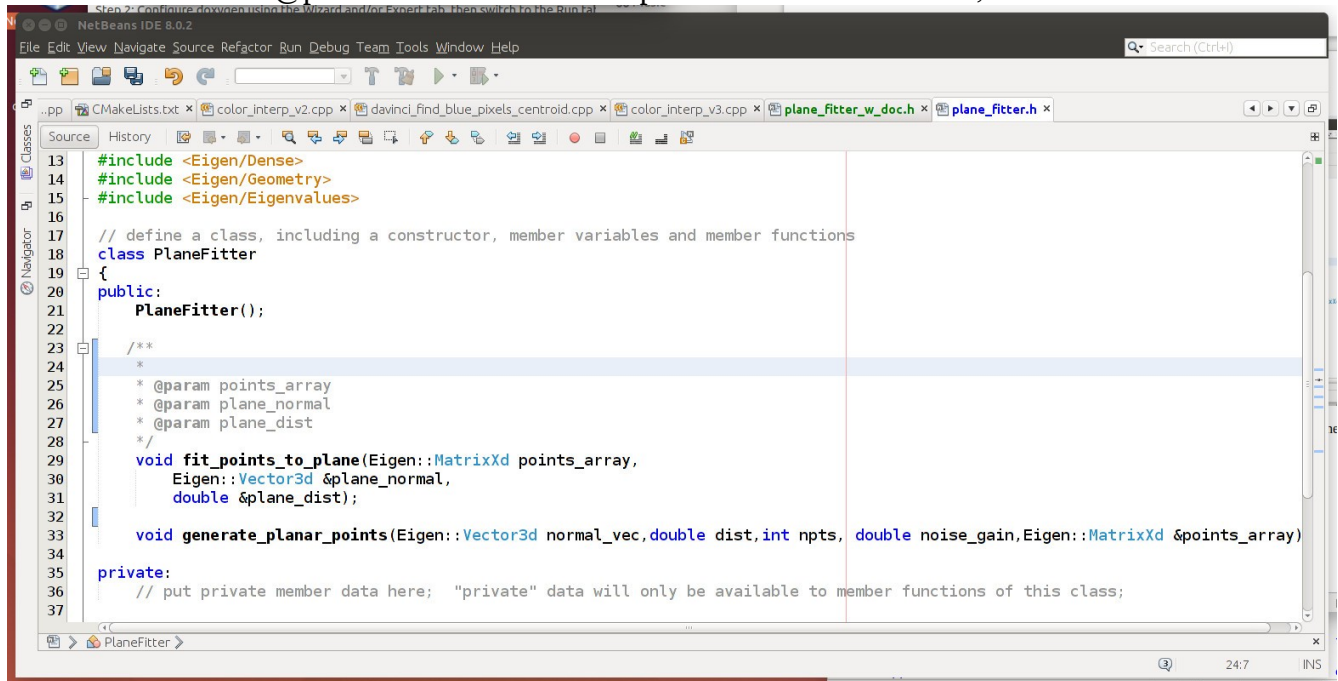
Instead of the “`///`” notation, we will use the alternative “`/**`” notation to start a Doxygen comment. This opening tag must have a corresponding “`*/`” tag at the end of the Doxygen comment.

In a sufficiently “aware” development environment, such as “Netbeans”, there will be assistance for documenting member functions. Going back to the header file, placing the edit cursor just above the member function prototype “`fit_points_to_plane()`”, by entering “`/**`”, the editor completes this comment by automatically adding the terminating tag “`*/`”, and the include file looks like this:



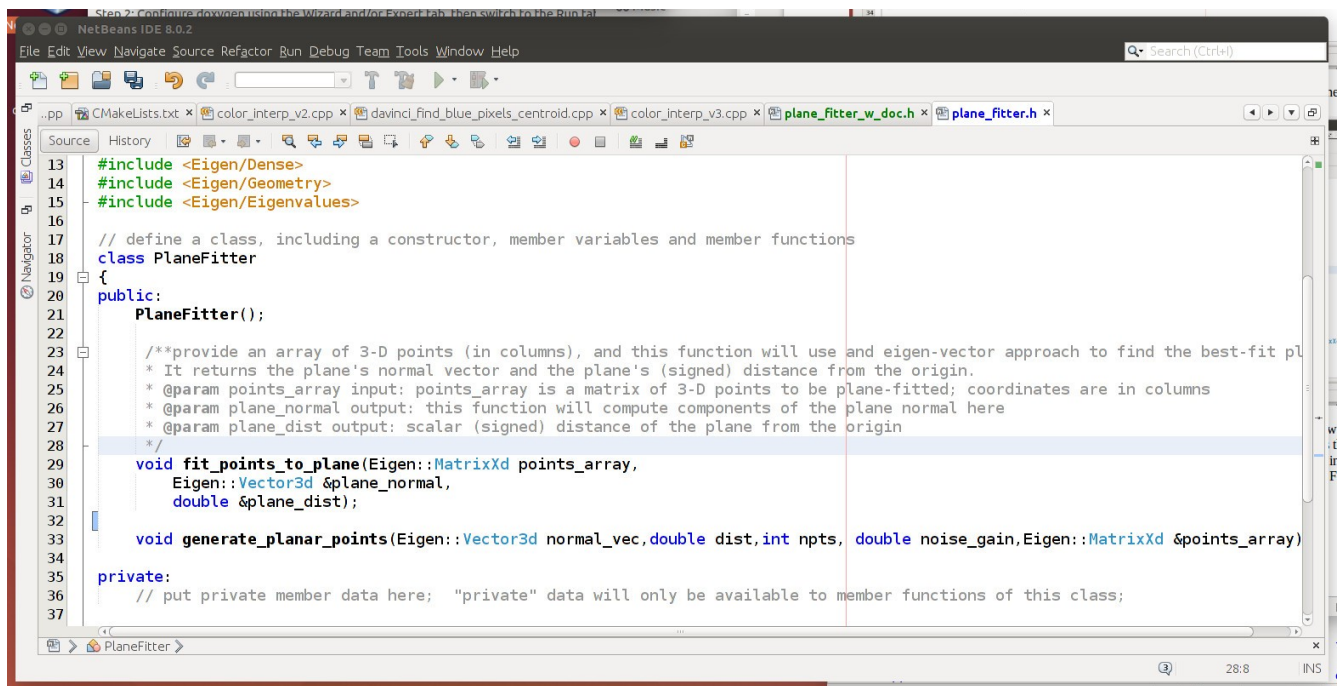
```
13 #include <Eigen/Dense>
14 #include <Eigen/Geometry>
15 #include <Eigen/Eigenvalues>
16
17 // define a class, including a constructor, member variables and member functions
18 class PlaneFitter
19 {
20 public:
21     PlaneFitter();
22
23     /**
24     void fit_points_to_plane(Eigen::MatrixXd points_array,
25                             Eigen::Vector3d &plane_normal,
26                             double &plane_dist);
27
28     void generate_planar_points(Eigen::Vector3d normal_vec, double dist, int npts, double noise_gain, Eigen::MatrixXd &points_array)
29
30 private:
31     // put private member data here; "private" data will only be available to member functions of this class;
32
33 }; // note: a class definition requires a semicolon at the end of the definition
34
35 #endif // this closes the header-include trick...ALWAYS need one of these to match #ifndef
36
```

Now, by merely entering a “return” after “/\*\*”, the editor automatically adds several lines, including lines that start with “@param” and which list the parameters of the function, as follows:



```
13 #include <Eigen/Dense>
14 #include <Eigen/Geometry>
15 #include <Eigen/Eigenvalues>
16
17 // define a class, including a constructor, member variables and member functions
18 class PlaneFitter
19 {
20 public:
21     PlaneFitter();
22
23     /**
24     *
25     * @param points_array
26     * @param plane_normal
27     * @param plane_dist
28     */
29     void fit_points_to_plane(Eigen::MatrixXd points_array,
30                             Eigen::Vector3d &plane_normal,
31                             double &plane_dist);
32
33     void generate_planar_points(Eigen::Vector3d normal_vec, double dist, int npts, double noise_gain, Eigen::MatrixXd &points_array)
34
35 private:
36     // put private member data here; "private" data will only be available to member functions of this class;
37
38
```

We should enter a description of the inputs and outputs here, as well as a description of what this function does. One should clarify here what are the inputs and the outputs. (Arguments that are pointers or reference variables are containers that can be populated with results of invoking the function, thus behaving as outputs that are more sophisticated than a mere return type). For this example, the following comments are added to describe the function and its arguments:



With these additional lines, we can generate and preview the resulting formatted documentation. To do so, from the doxywizard window, click “Run doxygen” then “Show HTML output.” The updated result appears as follows:

plane fitter: PlaneFitter x plane fitter: PlaneFitter x

file:///home/wyatt/ros\_ws/src/eecs-397-f15/example\_ros\_library/html/class\_plane\_fitter.html

# plane fitter

Main Page **Classes** Files

Class List Class Index Class Members

Public Member Functions | List of all members

## PlaneFitter Class Reference

#include <plane\_fitter.h>

### Public Member Functions

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)

### Detailed Description

wsn; Oct, 2015. Include this file in "plane\_fitter.cpp", and in any main that uses this library. This class provides a function to fit a plane to given set of (presumably) nearly-coplanar points

### Member Function Documentation

**void PlaneFitter::fit\_points\_to\_plane** ( Eigen::MatrixXd **points\_array**,  
Eigen::Vector3d & **plane\_normal**,  
double & **plane\_dist**  
)

provide an array of 3-D points (in columns), and this function will use an eigen-vector approach to find the best-fit plane. It returns the plane's normal vector and the plane's (signed) distance from the origin.

**Parameters**

**points\_array** input: points\_array is a matrix of 3-D points to be plane-fitted; coordinates are in columns

**plane\_normal** output: this function will compute components of the plane normal here

**plane\_dist** output: scalar (signed) distance of the plane from the origin

The documentation for this class was generated from the following file:

- include/example\_ros\_library/plane\_fitter.h

Generated on Tue Oct 20 2015 22:14:29 for plane\_fitter by doxygen 1.8.6

The member function description and the input and output parameter explanations are now shown nicely formatted and suitable for inclusion in on-line documentation.

For the second member function, we enter the following comments:



```

10
17 // define a class, including a constructor, member variables and member functions
18 class PlaneFitter
19 {
20 public:
21     PlaneFitter();
22
23     /**provide an array of 3-D points (in columns), and this function will use and eigen-vector approach to find the best-fit plane.
24     * It returns the plane's normal vector and the plane's (signed) distance from the origin.
25     * @param points_array input: points_array is a matrix of 3-D points to be plane-fitted; coordinates are in columns
26     * @param plane_normal output: this function will compute components of the plane normal here
27     * @param plane_dist output: scalar (signed) distance of the plane from the origin
28     */
29     void fit_points_to_plane(Eigen::MatrixX<double> points_array,
30                             Eigen::Vector3d &plane_normal,
31                             double &plane_dist);
32
33     /**
34     * Function to generate a set of random points that are (nearly) co-planar. Useful for testing fit_point_to_plane() fnc.
35     * Provide plane params, number of points desired, and scale of noise to add. Resulting points are returned in points_array
36     * @param normal_vec [in] input: specify plane normal
37     * @param dist input: specify plane's (signed) distance from the origin
38     * @param npts input: specify number of desired points
39     * @param noise_gain input: specify magnitude of noise (multiplier times noise in range [-1,1])
40     * @param points_array output: this array gets populated with the generated points
41     */
42     void generate_planar_points(Eigen::Vector3d normal_vec, double dist, int npts, double noise_gain, Eigen::MatrixX<double> &points_array)
43 private:

```

These additional comments are formatted by again pressing the “Run doxygen” button in the doxywizard window, then clicking “Show HTML output.” The updated output includes the following:

plane fitter: PlaneFitter

plane fitter: PlaneFitter

file:///home/wyatt/ros\_ws/src/eecs-397-f15/example\_ros\_library/html/class\_plane\_fitter.html

provide an array of 3-D points (in columns), and this function will use and eigen-vector approach to find the best-fit plane It returns the plane's normal vector and the plane's (signed) distance from the origin.

**Parameters**

**points\_array** input: points\_array is a matrix of 3-D points to be plane-fitted; coordinates are in columns

**plane\_normal** output: this function will compute components of the plane normal here

**plane\_dist** output: scalar (signed) distance of the plane from the origin

```

void PlaneFitter::generate_planar_points ( Eigen::Vector3d    normal_vec,
                                           double            dist,
                                           int               npts,
                                           double            noise_gain,
                                           Eigen::MatrixX<double> & points_array
                                           )

```

Function to generate a set of random points that are (nearly) co-planar. Useful for testing fit\_point\_to\_plane() fnc. Provide plane params, number of points desired, and scale of noise to add. Resulting points are returned in points\_array

**Parameters**

**normal\_vec** [in] input: specify plane normal

**dist** input: specify plane's (signed) distance from the origin

**npts** input: specify number of desired points

**noise\_gain** input: specify magnitude of noise (multiplier times noise in range [-1,1])

**points\_array** output: this array gets populated with the generated points

The documentation for this class was generated from the following file:

- include/example\_ros\_library/plane\_fitter.h

Generated on Tue Oct 20 2015 22:21:54 for plane fitter by **doxygen** 1.8.6

As shown, the second public function is now also documented, including a brief description of the purpose of the function and explanation of the input and output parameters.

The illustration here only touches the surface of Doxygen capabilities and of how to document your code for clarity of understanding. At a minimum, one should explain why the class exists, explain the purpose/value of each member function, and detail the inputs and outputs. With very little additional effort, normal code comments can result in nicely formatted HTML in a standard, readable style, making your code much more accessible and more likely to have future value.