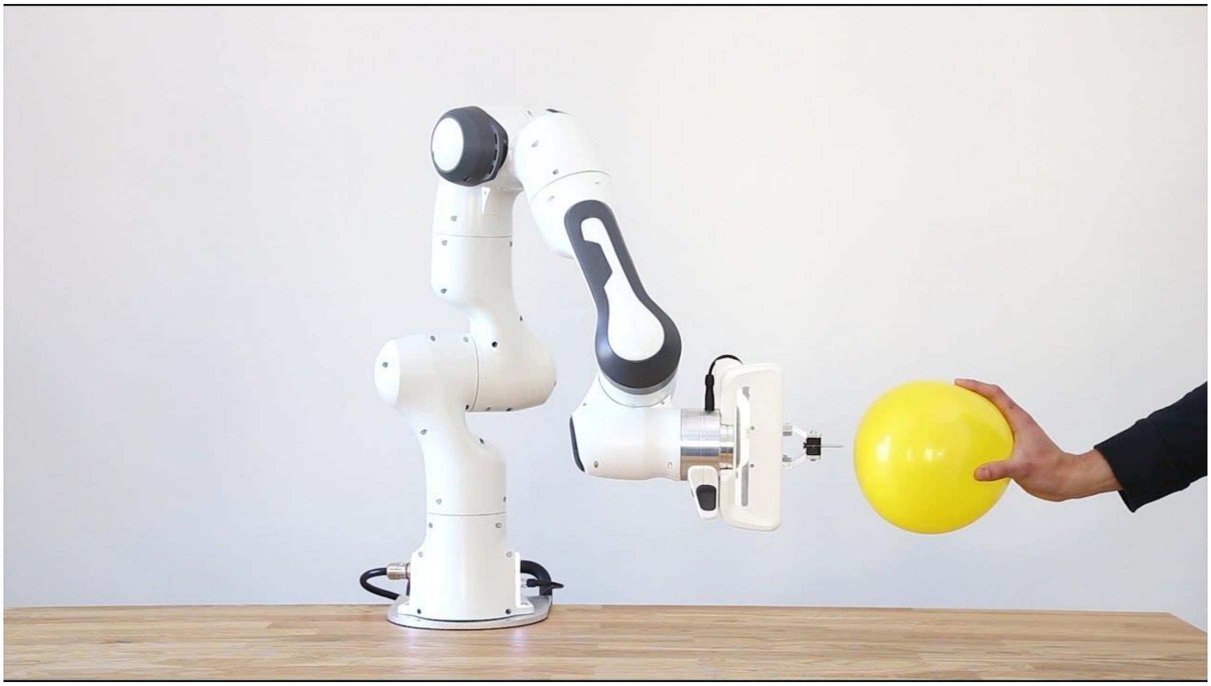
COMP0129: Robotic Sensing, Manipulation and Interaction

Labs

Instructors: *Eddie Edwards, Francisco Vasconcelos*
TAs: *Kefeng Huang, Bowie (Heiyin) Wong*



Lab 1 – Part 2/2

Date: Week 20
Goal: C++ Style Guide (ROS, PCL)

During the second lab, we will first see good practices in writing C++ code and documentation for ROS (http://wiki.ros.org/CppStyleGuide). We will also have a broad introduction to ROS and its functionalities.

ROS C++ Style Guide

Following the documentation here: http://wiki.ros.org/CppStyleGuide (from where the text was taken with some tweaks) and http://wiki.ros.org/DevelopersGuide.

"Coding style is important. A clean, consistent style leads to code that is more readable, debuggable, and maintainable. We strive to write elegant code that will not only perform its desired function today, but will also live on, to be re-used and improved by other developers for many years to come."

1. Naming

The following shortcuts are used in this section to denote naming schemes:

- **CamelCased**: The name starts with a capital letter, and has a capital letter for each new word, with no underscores.
- **camelCased**: Like CamelCase, but with a lower-case first letter
- **under_scored**: The name uses only lower-case letters, with words separated by underscores. (yes, I realize that *under_scored* should be *underscored*, because it's just one word).
- **ALL_CAPITALS**: All capital letters, with words separated by underscores.

2. Packages

- ROS packages are **under_scored**.

3. Topics / Services

- ROS topics and service names are **under_scored**.

4. Files

- All files are **under_scored**.

- **Source** files have the extension **.cpp**.
- **Header** files have the extension **.h**.

- Be **descriptive**, e.g., instead of **laser.cpp**, use **hokuyo_topurg_laser.cpp**.

- If the file primarily **implements** a class, name the file after the class. For example, the class ***ActionServer*** would live in the file ***action_server.h***.

4.1 Libraries

- **Libraries**, being files, are **under_scored**.

- Don't insert an underscore immediately after the **lib** prefix in the library name. For example:

```
lib_my_great_thing ## Bad
libmy_great_thing ## Good
```

5. Classes / Types

- **Class names** (and other type names) are **CamelCased.** For example:

```
class ExampleClass;
```

- **Exception**: if the class name contains a short acronym, the acronym itself should be all capitals, for example:

```
class HokuyoURGLaser;
```

- **Name the class after what it is**. If you can't think of what it is, perhaps you have not thought through the design well enough.

- Compound names of **over three words** are a clue that your design may be unnecessarily confusing.

6. Function / Methods

- In general, function and class method names are **camelCased**, and arguments are **under_scored**, for example:

```
int
exampleMethod (int example_arg);
```

- **Functions** and **methods** usually perform an action, so their name should make clear what they do: checkForErrors () instead of errorCheck (), dumpDataToFile () instead of dataFile ().

- **Classes** are often **nouns**. By making function names verbs and following other naming conventions programs can be read more naturally.

7. Variables

- In general, variable names are **under_scored**.

- Be reasonably **descriptive** and try not to be cryptic. Longer variable names don't take up more space in memory.

- **Integral iterator** variables can be very short, such as **i**, **j**, **k**. Be consistent in how you use iterators (e.g., **i** on the outer loop, **j**on the next inner loop).
- **STL iterator variables** should indicate what they're iterating over, for example:

```
std::list<int> pid_list;
std::list<int>::iterator pid_it;
```

- Alternatively, an **STL iterator** can indicate the type of element that it can point at, for example:

```
std::list<int> pid_list;
std::list<int>::iterator int_it;
```

7.1 Constants

- Constants, wherever they are used, are **ALL_CAPITALS.**

7.2 Member variables

- **Variables that are members of a class** (sometimes called fields) are **under_scored**, with a trailing underscore added. For example:

```
int example_int_;
```

7.3 Global variables

- **Global variables** should almost never be used (see below for more on this). When they are used, global variables are **under_scored** with a leading **g_** added. For example:

```
// I tried everything else, but I really need this global variable
int g_shutdown;
```

8. Namespaces

- Namespace names are **under_scored**.

9. License statements

- Every **source** and **header** file must contain a **license** and **copyright** statement at the beginning of the file.

- In the **ros-pkg** and **wg-ros-pkg** repositories, the **LICENSE** directory contains license templates, commented for inclusion in C/C++ code.

10. Formatting

- Your **editor** should handle most formatting tasks. See EditorHelp for example editor configuration files.

- Indent each block by **2 spaces**. Never insert literal tab characters.

- **Braces**, both open and close, go on their own lines (no "cuddled braces"). For example:

```
if (a < b)
{
  // do stuff
}
else
{
  // do other stuff
}
```

- **Braces** can be **omitted** if the enclosed block is a single-line statement, for example:

```
if (a < b)
  x = 2*a;
```

- Always **include the braces** if the enclosed block is more **complex**, for example:

```
if (a < b)
{
  for (int i=0; i<10; i++)
    printItem (i);
}
```

- Here is a **larger example**: **here**

10.1 Line length

- Maximum line length is **80 characters**.

10.2 #ifndef guards

- **All headers** must be protected against multiple inclusion by #ifndef guards, for example:

```
#ifndef PACKAGE_PATH_FILE_H
#define PACKAGE_PATH_FILE_H
```

```
...
#endif
```

- This guard should begin **immediately after the license statement**, before any code, and should end at the end of the file.

11. Documentation

- Code must be **documented**. Undocumented code, however functional it may be, cannot be maintained.

- We use **doxygen** to auto-document our code. Doxygen parses your code, extracting documentation from specially formatted comment blocks that appear next to functions, variables, classes, etc. Doxygen can also be used to build more narrative, free-form documentation.

- See the **rosdoc page** for examples of inserting **doxygen-style comments** into your code.

- All functions, methods, classes, class variables, enumerations, and constants should be **documented**.

12. Console output

- **Avoid printf** and friends (e.g., cout). Instead, use **rosconsole** for all your outputting needs. It offers **macros** with both printf- and stream-style arguments. Just like printf, rosconsole output goes to screen. Unlike printf, rosconsole output is:
  - color-coded
  - controlled by verbosity level and configuration file
  - published on **/rosout**, and thus viewable by anyone on the network (only when working with roscpp)
  - optionally logged to disk

13. Macros

- **Avoid preprocessor macros** whenever possible. Unlike inline functions and const variables, macros are neither typed nor scoped.

14. Preprocessor directives (#if vs. #ifdef)

- For **conditional compilation** (except for the #ifndef guard explained above), always use #if, not #ifdef.

- Someone might write **code** like:

```
#ifdef DEBUG
        temporary_debugger_break ();
#endif
```

- Someone else might compile the **code** with turned-off debug info like:

```
cc -c lurker.cpp -DDEBUG=0
```

- Always use **#if**, if you have to use the preprocessor. This works fine, and does the right thing, even if DEBUG is not defined at all.

```
#if DEBUG
        temporary_debugger_break ();
#endif
```

15. Output arguments

- **Output arguments** to methods / functions (i.e., variables that the function can modify) are passed **by pointer**, **not by reference**. For example:

```
int
exampleMethod (FooThing input, BarThing* output);
```

- By comparison, when passing output arguments **by reference**, the caller (or subsequent reader of the code) can't tell whether the argument can be modified without reading the prototype of the method.

- See also: Reference Arguments

16. Namespaces

- **Use of namespaces** to scope your code is encouraged. Pick a **descriptive name**, based on the name of the package.

- Never use a **using-directive** in header files. Doing so pollutes the namespace of all code that includes the header.

- It is acceptable to use **using-directives** in a source file. But it is preferred to use **using-declarations**, which pull in only the names you intend to use. For example, instead of this:

```
using namespace std; // Bad, because it imports all names from std::
```

- **Do this**:

```
using std::list;  // I want to refer to std::list as list
using std::vector;  // I want to refer to std::vector as vector
```

- See also: Google:Namespaces

17. Inheritance

- **Inheritance** is the appropriate way to **define and implement a common interface**. The base class defines the interface, and the subclasses implement it.

- **Inheritance** can also be used to **provide common code** from a base class to subclasses. This use of inheritance is **discouraged**. In most cases, the "subclass" could instead contain an instance of the "base class" and achieve the same result with less potential for confusion.

- When **overriding a virtual method in a subclass**, always declare it to be **virtual,** so that the reader knows what's going on.

- See also Google:Inheritance

17.1Multiple inheritance

- **Multiple inheritance is strongly discouraged**, as it can cause intolerable confusion.

- See also Google:Multiple Inheritance

18. Exceptions

- **Exceptions** are the preferred error-reporting mechanism, as opposed to returning integer error codes.

- Always **document** what exceptions can be thrown by your package, on each function / method.

- **Don't** throw exceptions from **destructors**.

- **Don't** throw exceptions from **callbacks** that you don't invoke directly.

- If you choose in your package to use error codes instead of exceptions, use only error codes. **Be consistent.**

18.1 Writing exception-safe code

- When your code can be interrupted by **exceptions**, you must ensure that **resources you hold will be deallocated** when stack variables go out of scope. In particular, mutexes must be released, and heap-allocated memory must be freed.

19. Enumerations

- **Namespaceify** your **enums**, for example:

```
namespace Choices
{
   enum Choice
  {
    Choice1,
    Choice2,
    Choice3
  };
}
typedef Choices::Choice Choice;
```

- This **prevents enums** from polluting the namespace they're inside. Individual items within the enum are referenced by: Choices::Choice1, but the typedef still allows declaration of the Choice enum without the namespace.

20. Globals

- **Globals**, both **variables** and **functions**, are **discouraged**. They pollute the namespace and make code less reusable.

- **Global variables**, in particular, are **strongly discouraged**. They prevent multiple instantiations of a piece of code and make multi-threaded programming a nightmare.

- **Most variables and functions should be declared inside classes**. The remainder should be declared inside namespaces.

- **Exception:** a file may contain a **main ()** function and a handful of small helper functions that are global. But keep in mind that one day those helper function may become useful to someone else.

- See also Google:Static and Global Variables
- See also Google:Nonmember, Static Member, and Global Functions

21. Static class variables

- **Static class variables are discouraged**. They prevent multiple instantiations of a piece of code and make multi-threaded programming a nightmare.

22. Calling exit ()

- Only call **exit ()** at a well-defined exit point for the application.
- Never call **exit ()** in a library.

23. Assertions

- **Use assertions to check preconditions, data structure integrity, and the return value from a memory allocator**. Assertions are better than writing conditional statements that will rarely, if ever, be exercised.
- Don't call **assert ()** directly. Instead use one of these functions, declared in **ros/assert.h** (part of the rosconsole package):

```
/** ROS_ASSERT asserts that the provided expression evaluates to
```

```
  * true.  If it is false, program execution will abort, with an informative
  * statement about which assertion failed, in what file.  Use ROS_ASSERT
  * instead of assert() itself.
  * Example usage:
  */
  ROS_ASSERT (x > y);


/** ROS_ASSERT_MSG(cond, "format string", ...) asserts that the provided
  * condition evaluates to true.
  * If it is false, program execution will abort, with an informative
  * statement about which assertion failed, in what file, and it will print out
  * a printf-style message you define.  Example usage:
  */
  ROS_ASSERT_MSG (x > 0, "Uh oh, x went negative.  Value = %d", x);


/** ROS_ASSERT_CMD (cond, function ())
  * Runs a function if the condition is false. Usage example:
  */
  ROS_ASSERT_CMD (x > 0, handleError (...));


/** ROS_BREAK aborts program execution, with an informative
  * statement about which assertion failed, in what file. Use ROS_BREAK
  * instead of calling assert(0) or ROS_ASSERT(0). You can step over the assert
  * in a debugger.
  * Example usage:
  */
  ROS_BREADK ();
```

- **Do not** do work inside an assertion; **only check logical expressions**. Depending on compilation settings, the assertion may not be executed.

- It is typical to develop software with **assertion-checking enabled**, in order to catch violations. When nearing software completion and when assertions are found to always be true in the face of **extensive testing**, you build with a flag that **removes assertions from compilation, so they take up no space or time**. The following option to **catkin_make** will define the NDEBUG macro for all your ROS packages, and thereby remove assertion checks.

```
catkin_make -DCMAKE_CXX_FLAGS:STRING="-DNDEBUG"
```

- **Note:** cmake will rebuild all your software when you run it with this command, and **will remember the setting** through subsequent catkin_make runs until you delete your build and devel directories and rebuild.

24. Testing

- See gtest.

25. Portability

We're currently support Linux. To that end, it's important to keep the C++ code portable. Here are a few things to watch for:

  - Don't use **uint** as a type. Instead use **unsigned int**.
  - Call **isnan()** from within the **std** namespace, i.e.: **std::isnan()**

26. Deprecation

- To **deprecate an entire header file within a package**, you may include an appropriate warning:

```
#warning mypkg/my_header.h has been deprecated
```

- To **deprecate a function**, add the deprecated attribute:

```
ROS_DEPRECATED int myFunc();
```

- To **deprecate a class**, deprecate its constructor and any static functions:

```
class MyClass
{
public:
  ROS_DEPRECATED MyClass();

  ROS_DEPRECATED static int myStaticFunc();
};
```

PCL C++ Style Guide

The style guide below has been influenced from the PCL guide that we will use later in the course: http://pointclouds.org/documentation/advanced/pcl_style_guide.html and http://pointclouds.org/documentation/tutorials/writing_new_classes.html#writing-new-classes.

Example: a bilateral filter

- **Example:** apply a bilateral filter over intensity data from a given input point cloud and save the results to disk.

```
#include <pcl/point_types.h>
#include <pcl/io/pcd_io.h>
#include <pcl/kdtree/kdtree_flann.h>

typedef pcl::PointXYZI PointT;

//////////////////////////////////////////////////////////////////////////////////////////////////////////
float
G (float x, float sigma)
{
  return std::exp (- (x*x)/(2*sigma*sigma));
}

//////////////////////////////////////////////////////////////////////////////////////////////////////////
int
main (int argc, char *argv[])
{
  std::string incloudfile = argv[1];
  std::string outcloudfile = argv[2];
  float sigma_s = atof (argv[3]);
  float sigma_r = atof (argv[4]);

  // Load cloud
  pcl::PointCloud<PointT>::Ptr cloud (new pcl::PointCloud<PointT>);
  pcl::io::loadPCDFile (incloudfile.c_str (), *cloud);
  int pnumber = (int)cloud->size ();

  // Output Cloud = Input Cloud
  pcl::PointCloud<PointT> outcloud = *cloud;

  // Set up KDTree
  pcl::KdTreeFLANN<PointT>::Ptr tree (new pcl::KdTreeFLANN<PointT>);
  tree->setInputCloud (cloud);

  // Neighbors containers
  std::vector<int> k_indices;
  std::vector<float> k_distances;

  // Main Loop
  for (int point_id = 0; point_id < pnumber; ++point_id)
  {
    float BF = 0;
43        float W = 0;
44
45        tree->radiusSearch (point_id, 2 * sigma_s, k_indices, k_distances);
46
47        // For each neighbor
48        for (std::size_t n_id = 0; n_id < k_indices.size (); ++n_id)
49        {
50           float id = k_indices.at (n_id);
51           float dist = sqrt (k_distances.at (n_id));
52          float intensity_dist = std::abs ((*cloud)[point_id].intensity - (*cloud)[id].intensity);
53
54          float w_a = G (dist, sigma_s);
55          float w_b = G (intensity_dist, sigma_r);
56          float weight = w_a * w_b;
57
58          BF += weight * (*cloud)[id].intensity;
59          W += weight;
60        }
61
62       outcloud[point_id].intensity = BF / W;
63     }
64
65     // Save filtered output
66     pcl::io::savePCDFile (outcloudfile.c_str (), outcloud);

       return (0);
     }
```

- The presented **code snippet** contains:
    - an **I/O component**: lines 21-27 (reading data from disk), and 64 (writing data to disk)
    - an **initialization component**: lines 29-35 (setting up a search method for nearest neighbors using a KdTree)
    - the actual **algorithmic component**: lines 7-11 and 37-61
- Our **goal** here is to convert the algorithm given into a **useful PCL class** so that it can be **reused** elsewhere.

Setting up the structure

- If you're not familiar with the **PCL file structure** already, please go ahead and read the PCL C++ Programming Style Guide to familiarize yourself with the concepts.

- There're two different ways we could **set up the structure**:
  i) set up the code separately, as a **standalone PCL class**, but outside of the PCL code tree;
  or
  ii) set up the files directly in the **PCL code tree**.

- Since our assumption is that the end result will be contributed back to PCL, it's best to **concentrate on the latter**, also because it is a bit more complex (i.e., it involves a few additional steps). You can obviously repeat these steps with the former case as well, with the exception that you don't need the files copied in the PCL tree, nor you need the fancier ***cmake*** **logic**.

- Assuming that we want the new algorithm **to be part of the PCL Filtering library**, we will begin by creating 3 different files under filters:
    - *include/pcl/filters/bilateral.h* - will **contain all definitions**;
    - *include/pcl/filters/impl/bilateral.hpp* - will **contain the templated implementations**;
    - *src/bilateral.cpp* - will **contain the explicit template instantiations** *.
- We also need a **name** for our new class. Let's call it ***BilateralFilter***.

** Some PCL filter algorithms provide two implementations: one for PointCloud<T> types and another one operating on legacy PCLPointCloud2 types. This is no longer required.*

Working on the bilateral.h

- As previously mentioned, the *bilateral.h* header file will contain all the definitions pertinent to the *BilateralFilter* class. Here's a **minimal skeleton**:

```
#ifndef PCL_FILTERS_BILATERAL_H_
#define PCL_FILTERS_BILATERAL_H_

#include <pcl/filters/filter.h>

namespace pcl
{
  template<typename PointT>
  class BilateralFilter : public Filter<PointT>
  {
  };
}

#endif // PCL_FILTERS_BILATERAL_H_
```

Working on the bilateral.hpp

- While we're at it, let's set up two skeleton ***bilateral.hpp*** and ***bilateral.cpp*** files as well.

- First, ***bilateral.hpp***:

```
#ifndef PCL_FILTERS_BILATERAL_IMPL_H_
#define PCL_FILTERS_BILATERAL_IMPL_H_

#include <pcl/filters/bilateral.h>

#endif // PCL_FILTERS_BILATERAL_IMPL_H_
```

- This should be straightforward. **We haven't declared any methods for *BilateralFilter* yet**, therefore there is no implementation.

Working on the bilateral.cpp

- Let’s write ***bilateral.cpp*** too:

```
#include <pcl/filters/bilateral.h>
#include <pcl/filters/impl/bilateral.hpp>
```

- Because we are writing **templated code in PCL** (1.x) where the template parameter is a point type (see Adding your own custom PointT type), we want to explicitly instantiate the most common use cases in *bilateral.cpp*, so that users don’t have to spend extra cycles when compiling code that uses our *BilateralFilter*. To do this, we need to access both the header (*bilateral.h*) and the implementations (*bilateral.hpp*).

Working on the CMakeLists.txt

- Let’s **add all the files** to the PCL Filtering ***CMakeLists.txt*** file, so we can enable the build.

```
# Find "set (srcs", and add a new entry there, e.g.,
set (srcs
    src/conditional_removal.cpp
    # ...
    src/bilateral.cpp)
    )

# Find "set (incs", and add a new entry there, e.g.,
set (incs
    include pcl/${SUBSYS_NAME}/conditional_removal.h
    # ...
    include pcl/${SUBSYS_NAME}/bilateral.h
    )

# Find "set (impl_incs", and add a new entry there, e.g.,
set (impl_incs
    include/pcl/${SUBSYS_NAME}/impl/conditional_removal.hpp
    # ...
    include/pcl/${SUBSYS_NAME}/impl/bilateral.hpp
    )
```

Filling in the class structure

- If you correctly edited all the files above, **recompiling PCL** using the new filter classes in place should work without problems. In this section, we’ll begin **filling in the actual code** in each file. Let’s start with the ***bilateral.cpp*** file, as its content is the shortest.

Working on the bilateral.cpp

- As previously mentioned, we’re going to explicitly instantiate and ***precompile* a number of templated specializations for the *BilateralFilter* class**. While this might lead to an increased compilation time for the PCL Filtering library, it will save users the pain of processing and compiling the templates on their end, when they use the class in code they write. The simplest possible way to do this would be to declare each instance that we want to precompile by hand in the *bilateral.cpp* file as follows:

```
#include <pcl/point_types.h>
#include <pcl/filters/bilateral.h>
#include <pcl/filters/impl/bilateral.hpp>

template class PCL_EXPORTS pcl::BilateralFilter<pcl::PointXYZ>;
template class PCL_EXPORTS pcl::BilateralFilter<pcl::PointXYZI>;
template class PCL_EXPORTS pcl::BilateralFilter<pcl::PointXYZRGB>;
// ...
```

- However, **this becomes cumbersome really fast, as the number of point types PCL supports grows**. Maintaining this list up to date in multiple files in PCL is also painful. Therefore, we are going to use a **special macro** called ***PCL_INSTANTIATE*** and change the above code as follows:

```
#include <pcl/point_types.h>
#include <pcl/impl/instantiate.hpp>
#include <pcl/filters/bilateral.h>
#include <pcl/filters/impl/bilateral.hpp>

PCL_INSTANTIATE(BilateralFilter, PCL_XYZ_POINT_TYPES);
```

- This example, will instantiate a *BilateralFilter* for all XYZ point types defined in the ***point_types.h*** file (see :pcl:`PCL_XYZ_POINT_TYPES<PCL_XYZ_POINT_TYPES>` for more information).

- By looking closer at the code presented in Example: a bilateral filter, we notice constructs such as *(*cloud)[point_id].intensity*. This indicates that our filter expects the presence of an **intensity** field in the point type. Because of this, using **PCL_XYZ_POINT_TYPES** won't work, as not all the types defined there have intensity data present. In fact, it's easy to notice that only two of the types contain intensity, namely::pcl:`PointXYZI<pcl::PointXYZI>` and :pcl:`PointXYZINormal<pcl::PointXYZINormal>`. We therefore replace **PCL_XYZ_POINT_TYPES** and the final *bilateral.cpp* file becomes:

```
#include <pcl/point_types.h>
#include <pcl/impl/instantiate.hpp>
#include <pcl/filters/bilateral.h>
#include <pcl/filters/impl/bilateral.hpp>

PCL_INSTANTIATE(BilateralFilter, (pcl::PointXYZI)(pcl::PointXYZINormal));
```

- Note that at this point we **haven't declared the PCL_INSTANTIATE template** for *BilateralFilter*, nor did we actually implement the pure virtual functions in the abstract class :pcl:`pcl::Filter<pcl::Filter>` so attempting to compile the code will result in errors like:

```
filters/src/bilateral.cpp:6:32: error: expected constructor, destructor, or type conversion before '(' token
```

Working on the bilateral.h

- We begin filling the ***BilateralFilter* class** by first declaring the **constructor**, and its member **variables**. Because the bilateral filtering algorithm has two parameters, we will store these as class members, and implement setters and getters for them, to be compatible with the PCL 1.x API paradigms.

```
...
namespace pcl
{
  template<typename PointT>
  class BilateralFilter : public Filter<PointT>
  {
    public:
      BilateralFilter () : sigma_s_ (0),
                           sigma_r_ (std::numeric_limits<double>::max ())
      {
      }

      void
      setSigmaS (const double sigma_s)
      {
        sigma_s_ = sigma_s;
      }

      double
      getSigmaS () const
      {
        return (sigma_s_);
      }

      void
      setSigmaR (const double sigma_r)
      {
        sigma_r_ = sigma_r;
      }

      double
      getSigmaR () const
      {
        return (sigma_r_);
      }

    private:
      double sigma_s_;
      double sigma_r_;
40   };
41  }
42
43  #endif // PCL_FILTERS_BILATERAL_H_
```

- **Nothing out of the ordinary so far**, except maybe lines 8-9, where we gave some default values to the two parameters. Because our class inherits from :pcl:`pcl::Filter<pcl::Filter>`, and that inherits from:pcl:`pcl::PCLBase<pcl::PCLBase>`, we can make use of the :pcl:`setInputCloud<pcl::PCLBase::setInputCloud>` method to pass the input data to our algorithm (stored as :pcl:`input_<pcl::PCLBase::input_>`). We therefore add an *using* declaration as follows:

```
...
template<typename PointT>
class BilateralFilter : public Filter<PointT>
{
  using Filter<PointT>::input_;
  public:
    BilateralFilter () : sigma_s_ (0),
...
```

- This will make sure that our class **has access to the member variable *input_* without typing the entire construct**. Next, we observe that each class that inherits from :pcl:`pcl::Filter<pcl::Filter>` must inherit a:pcl:`applyFilter<pcl::Filter::applyFilter>` method. We therefore define:

```
1   ...
2     using Filter<PointT>::input_;
3     typedef typename Filter<PointT>::PointCloud PointCloud;
4
5     public:
6       BilateralFilter () : sigma_s_ (0),
7                            sigma_r_ (std::numeric_limits<double>::max ())
8       {
9       }
10
11      void
12      applyFilter (PointCloud &output);
13  ...
```

- The implementation of *applyFilter* will be given in the *bilateral.hpp* file later. Line 3 constructs a typedef so that we can use the type *PointCloud* without typing the entire construct.

- Looking at the original code from section Example: a bilateral filter, we notice that the algorithm consists of applying the same operation to every point in the cloud. To keep the *applyFilter* call clean, we therefore define method called *computePointWeight* whose implementation will contain the corpus defined in between lines 45-58:

```
...
    void
    applyFilter (PointCloud &output);

    double
    computePointWeight (const int pid, const std::vector<int> &indices, const std::vector<float> &distances);
...
```

- In addition, we notice that lines 29-31 and 43 from section Example: a bilateral filter construct a :pcl:`KdTree<pcl::KdTree>` structure for obtaining the nearest neighbors for a given point. **We therefore add**:

```
#include <pcl/kdtree/kdtree.h>
...
    using Filter<PointT>::input_;
    typedef typename Filter<PointT>::PointCloud PointCloud;
    typedef typename pcl::KdTree<PointT>::Ptr KdTreePtr;

  public:
...

    void
    setSearchMethod (const KdTreePtr &tree)
    {
      tree_ = tree;
    }

  private:
...
    KdTreePtr tree_;
...
```

- Finally, we would like to **add the kernel method** (*G (float x, float sigma)*) inline so that we speed up the computation of the filter. Because the method is only useful within the context of the algorithm, we will make it private. The header file becomes:

```
#ifndef PCL_FILTERS_BILATERAL_H_
#define PCL_FILTERS_BILATERAL_H_

#include <pcl/filters/filter.h>
#include <pcl/kdtree/kdtree.h>

namespace pcl
{
  template<typename PointT>
  class BilateralFilter : public Filter<PointT>
  {
    using Filter<PointT>::input_;
    typedef typename Filter<PointT>::PointCloud PointCloud;
    typedef typename pcl::KdTree<PointT>::Ptr KdTreePtr;

    public:
```

```
    BilateralFilter () : sigma_s_ (0),
                  sigma_r_ (std::numeric_limits<double>::max ())
    {
    }


    void
    applyFilter (PointCloud &output);

    double
    computePointWeight (const int pid, const std::vector<int> &indices,
                            const std::vector<float> &distances);

    void
    setSigmaS (const double sigma_s)
    {
      sigma_s_ = sigma_s;
    }

    double
    getSigmaS () const
    {
     return (sigma_s_);
    }

    void
    setSigmaR (const double sigma_r)
    {
     sigma_r_ = sigma_r;
    }

    double
    getSigmaR () const
    {
      return (sigma_r_);
    }

    void
    setSearchMethod (const KdTreePtr &tree)
    {
      tree_ = tree;
    }


  private:

    inline double
    kernel (double x, double sigma)
    {
      return (std::exp (- (x*x)/(2*sigma*sigma)));
    }

    double sigma_s_;
    double sigma_r_;
    KdTreePtr tree_;
  };
}
#endif // PCL_FILTERS_BILATERAL_H_
```

Working on the bilateral.hpp

- There’re two methods that we need to implement here, namely *applyFilter* and *computePointWeight*.

```
template <typename PointT> double
pcl::BilateralFilter<PointT>::computePointWeight (const int pid,
                                                  const std::vector<int> &indices,
                                                  const std::vector<float> &distances)
{
  double BF = 0, W = 0;

  // For each neighbor
  for (std::size_t n_id = 0; n_id < indices.size (); ++n_id)
  {
     double id = indices[n_id];
     double dist = std::sqrt (distances[n_id]);
     double intensity_dist = std::abs ((*input_)[pid].intensity - (*input_)[id].intensity);

     double weight = kernel (dist, sigma_s_) * kernel (intensity_dist, sigma_r_);

     BF += weight * (*input_)[id].intensity;
     W += weight;
  }
  return (BF / W);
}

template <typename PointT> void
pcl::BilateralFilter<PointT>::applyFilter (PointCloud &output)
{
   tree_->setInputCloud (input_);

   std::vector<int> k_indices;
   std::vector<float> k_distances;

  output = *input_;

   for (std::size_t point_id = 0; point_id < input_->size (); ++point_id)
   {
      tree_->radiusSearch (point_id, sigma_s_ * 2, k_indices, k_distances);

      output[point_id].intensity = computePointWeight (point_id, k_indices, k_distances);
   }

}
```

- The ***computePointWeight*** method should be straightforward as it’s *almost identical* to lines 45-58 from section Example: a bilateral filter. We basically pass in a point index that we want to compute the intensity weight for, and a set of neighboring points with distances.

- In ***applyFilter***, we first set the input data in the tree, copy all the input data into the output, and then proceed at computing the new weighted point intensities.
- Looking back at Filling in the class structure, it's now time to declare the ***PCL_INSTANTIATE* entry** for the class:

```
#ifndef PCL_FILTERS_BILATERAL_IMPL_H_
#define PCL_FILTERS_BILATERAL_IMPL_H_

#include <pcl/filters/bilateral.h>

...

#define PCL_INSTANTIATE_BilateralFilter(T) template class PCL_EXPORTS pcl::BilateralFilter<T>;

#endif // PCL_FILTERS_BILATERAL_IMPL_H_
```

- One additional thing that we can do is **error checking** on:
  - whether the two *sigma_s_* and *sigma_r_* parameters have been given;
  - whether the search method object (i.e., *tree_*) has been set.

- For the former, we're going to **check the value of *sigma_s_***, which was set to a default of 0, and has a critical importance for the behavior of the algorithm (it basically defines the size of the support region). Therefore, if at the execution of the code, its value is still 0, we will print an error using the :pcl:`PCL_ERROR<PCL_ERROR>` macro, and return.

- In the case of the **search method**, we can either do the same, or be clever and provide a default option for the user. The best default options are:
  - use an organized search method via :pcl:`pcl::OrganizedNeighbor<pcl::OrganizedNeighbor>` if the point cloud is organized;
  - use a general purpose kdtree via :pcl:`pcl::KdTreeFLANN<pcl::KdTreeFLANN>` if the point cloud is unorganized.

```
#include <pcl/kdtree/kdtree_flann.h>
#include <pcl/kdtree/organized_data.h>

...
template <typename PointT> void
pcl::BilateralFilter<PointT>::applyFilter (PointCloud &output)
{
  if (sigma_s_ == 0)
  {
    PCL_ERROR ("[pcl::BilateralFilter::applyFilter] Need a sigma_s value given before continuing.\n");
    return;
  }
  if (!tree_)
  {
    if (input_->isOrganized ())
```

```
      tree_.reset (new pcl::OrganizedNeighbor<PointT> ());
    else
      tree_.reset (new pcl::KdTreeFLANN<PointT> (false));
  }
  tree_->setInputCloud (input_);
...
```

- The **implementation file header** thus becomes:

```
#ifndef PCL_FILTERS_BILATERAL_IMPL_H_
#define PCL_FILTERS_BILATERAL_IMPL_H_

#include <pcl/filters/bilateral.h>
#include <pcl/kdtree/kdtree_flann.h>
#include <pcl/kdtree/organized_data.h>

template <typename PointT> double
pcl::BilateralFilter<PointT>::computePointWeight (const int pid,
                                                  const std::vector<int> &indices,
                                                  const std::vector<float> &distances)
{
  double BF = 0, W = 0;

  // For each neighbor
  for (std::size_t n_id = 0; n_id < indices.size (); ++n_id)
  {
    double id = indices[n_id];
    double dist = std::sqrt (distances[n_id]);
    double intensity_dist = std::abs ((*input_)[pid].intensity - (*input_)[id].intensity);

    double weight = kernel (dist, sigma_s_) * kernel (intensity_dist, sigma_r_);

    BF += weight * (*input_)[id].intensity;
    W += weight;
  }
  return (BF / W);
}

template <typename PointT> void
pcl::BilateralFilter<PointT>::applyFilter (PointCloud &output)
{
  if (sigma_s_ == 0)
  {
    PCL_ERROR ("[pcl::BilateralFilter::applyFilter] Need a sigma_s value given before continuing.\n");
    return;
  }
  if (!tree_)
  {
    if (input_->isOrganized ())
      tree_.reset (new pcl::OrganizedNeighbor<PointT> ());
    else
      tree_.reset (new pcl::KdTreeFLANN<PointT> (false));
  }
  tree_->setInputCloud (input_);

  std::vector<int> k_indices;
  std::vector<float> k_distances;

  output = *input_;

  for (std::size_t point_id = 0; point_id < input_->size (); ++point_id)
  {
    tree_->radiusSearch (point_id, sigma_s_ * 2, k_indices, k_distances);

    output[point_id].intensity = computePointWeight (point_id, k_indices, k_distances);
  }
}

#define PCL_INSTANTIATE_BilateralFilter(T) template class PCL_EXPORTS pcl::BilateralFilter<T>;

#endif // PCL_FILTERS_BILATERAL_IMPL_H_
```

Taking advantage of other PCL concepts: Point indices

- The standard way of passing point cloud data into PCL algorithms is via :pcl:`setInputCloud<pcl::PCLBase::setInputCloud>` calls. In addition, PCL also defines a way to define a region of interest / *list of point indices* that the algorithm should operate on, rather than the entire cloud, via :pcl:`setIndices<pcl::PCLBase::setIndices>`.

- All classes inheriting from :pcl:`PCLBase<pcl::PCLBase>` exhbit the following behavior: in case no set of indices is given by the user, a fake one is created once and used for the duration of the algorithm. This means that we could easily change the implementation code above to operate on a <*cloud, indices*> tuple, which has the added advantage that if the user does pass a set of indices, only those will be used, and if not, the entire cloud will be used.

- The new *bilateral.hpp* class thus becomes:

```
#include <pcl/kdtree/kdtree_flann.h>
#include <pcl/kdtree/organized_data.h>

...
template <typename PointT> void
pcl::BilateralFilter<PointT>::applyFilter (PointCloud &output)
{
  if (sigma_s_ == 0)
  {
    PCL_ERROR ("[pcl::BilateralFilter::applyFilter] Need a sigma_s value given before continuing.\n");
    return;
  }
  if (!tree_)
  {
    if (input_->isOrganized ())
      tree_.reset (new pcl::OrganizedNeighbor<PointT> ());
    else
      tree_.reset (new pcl::KdTreeFLANN<PointT> (false));
  }
  tree_->setInputCloud (input_);
...
```

- The **implementation file header** thus becomes:

```
#ifndef PCL_FILTERS_BILATERAL_IMPL_H_
#define PCL_FILTERS_BILATERAL_IMPL_H_

#include <pcl/filters/bilateral.h>
#include <pcl/kdtree/kdtree_flann.h>
#include <pcl/kdtree/organized_data.h>

template <typename PointT> double
pcl::BilateralFilter<PointT>::computePointWeight (const int pid,
                                                  const std::vector<int> &indices,
                                                  const std::vector<float> &distances)
{
  double BF = 0, W = 0;

  // For each neighbor
  for (std::size_t n_id = 0; n_id < indices.size (); ++n_id)
  {
    double id = indices[n_id];
    double dist = std::sqrt (distances[n_id]);
    double intensity_dist = std::abs ((*input_)[pid].intensity - (*input_)[id].intensity);

    double weight = kernel (dist, sigma_s_) * kernel (intensity_dist, sigma_r_);

    BF += weight * (*input_)[id].intensity;
    W += weight;
  }
  return (BF / W);
}

template <typename PointT> void
pcl::BilateralFilter<PointT>::applyFilter (PointCloud &output)
{
  if (sigma_s_ == 0)
  {
    PCL_ERROR ("[pcl::BilateralFilter::applyFilter] Need a sigma_s value given before continuing.\n");
    return;
  }
  if (!tree_)
  {
    if (input_->isOrganized ())
      tree_.reset (new pcl::OrganizedNeighbor<PointT> ());
    else
      tree_.reset (new pcl::KdTreeFLANN<PointT> (false));
  }
  tree_->setInputCloud (input_);

  std::vector<int> k_indices;
  std::vector<float> k_distances;

  output = *input_;

  for (std::size_t i = 0; i < indices_->size (); ++i)
  {
    tree_->radiusSearch ((*indices_)[i], sigma_s_ * 2, k_indices, k_distances);

    output[(*indices_)[i]].intensity = computePointWeight ((*indices_)[i], k_indices, k_distances);
  }
}

#define PCL_INSTANTIATE_BilateralFilter(T) template class PCL_EXPORTS pcl::BilateralFilter<T>;

#endif // PCL_FILTERS_BILATERAL_IMPL_H_
```

- To make :pcl:`indices_<pcl::PCLBase::indices_>` work without typing the full construct, we need to add a new line to *bilateral.h* that specifies the class where *indices_* is declared:

```
...
template<typename PointT>
class BilateralFilter : public Filter<PointT>
{
  using Filter<PointT>::input_;
  using Filter<PointT>::indices_;
  public:
    BilateralFilter () : sigma_s_ (0),
...
```

Licenses

- It is advised that each file contains a **license** that describes the author of the code. This is very useful for our users that need to understand what sort of restrictions they are bound to when using the code. PCL is 100% **BSD licensed**, and we insert the corpus of the license as a C++ comment in the file, as follows:

```
/*
 * Software License Agreement (BSD License)
 *
 *  Point Cloud Library (PCL) - www.pointclouds.org
 *  Copyright (c) 2010-2011, Willow Garage, Inc.
 *
 *  All rights reserved.
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above
 *     copyright notice, this list of conditions and the following
 *     disclaimer in the documentation and/or other materials provided
 *     with the distribution.
 *   * Neither the name of Willow Garage, Inc. nor the names of its
 *     contributors may be used to endorse or promote products derived
 *     from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *  POSSIBILITY OF SUCH DAMAGE.
 *
 */
```

- An additional like can be inserted if additional copyright is needed (or the original copyright can be changed):

```
* Copyright (c) XXX, respective authors.
```

Proper naming

- We wrote the tutorial so far by using *silly named* **setters** and **getters** in our example, like *setSigmaS* or *setSigmaR*. In reality, we would like to use a better naming scheme, that actually represents what the parameter is doing. In a final version of the code, we could therefore rename the setters and getters to ***set/getHalfSize*** and ***set/getStdDev*** or something similar.

Code comments

- PCL is trying to maintain a *high standard* with respect to user and API documentation. This sort of **Doxygen** documentation has been stripped from the examples shown above. In reality, we would have had the *bilateral.h* header class look like:

```
/*
 * Software License Agreement (BSD License)
 *
 *  Point Cloud Library (PCL) - www.pointclouds.org
 *  Copyright (c) 2010-2011, Willow Garage, Inc.
 *
 *  All rights reserved.
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above
 *     copyright notice, this list of conditions and the following
 *     disclaimer in the documentation and/or other materials provided
 *     with the distribution.
 *   * Neither the name of Willow Garage, Inc. nor the names of its
 *     contributors may be used to endorse or promote products derived
 *     from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *  POSSIBILITY OF SUCH DAMAGE.
 *
 */

#ifndef PCL_FILTERS_BILATERAL_H_
#define PCL_FILTERS_BILATERAL_H_

#include <pcl/filters/filter.h>
#include <pcl/kdtree/kdtree.h>

namespace pcl
{
  /** \brief A bilateral filter implementation for point cloud data. Uses the intensity data channel.
    * \note For more information please see
    * <b>C. Tomasi and R. Manduchi. Bilateral Filtering for Gray and Color Images.
    * In Proceedings of the IEEE International Conference on Computer Vision,
    * 1998.</b>
    * \author Luca Penasa
    */
  template<typename PointT>
  class BilateralFilter : public Filter<PointT>
  {
    using Filter<PointT>::input_;
    using Filter<PointT>::indices_;
    typedef typename Filter<PointT>::PointCloud PointCloud;
    typedef typename pcl::KdTree<PointT>::Ptr KdTreePtr;

    public:
      /** \brief Constructor.
        * Sets \ref sigma_s_ to 0 and \ref sigma_r_ to MAXDBL
        */
      BilateralFilter () : sigma_s_ (0),
                          sigma_r_ (std::numeric_limits<double>::max ())
      {
      }


      /** \brief Filter the input data and store the results into output
        * \param[out] output the resultant point cloud message
        */
      void
      applyFilter (PointCloud &output);

      /** \brief Compute the intensity average for a single point
        * \param[in] pid the point index to compute the weight for
        * \param[in] indices the set of nearest neighor indices
        * \param[in] distances the set of nearest neighbor distances
        * \return the intensity average at a given point index
        */
      double
      computePointWeight (const int pid, const std::vector<int> &indices, const std::vector<float> &distances);

      /** \brief Set the half size of the Gaussian bilateral filter window.
        * \param[in] sigma_s the half size of the Gaussian bilateral filter window to use
        */
      inline void
      setHalfSize (const double sigma_s)
      {
        sigma_s_ = sigma_s;
      }

      /** \brief Get the half size of the Gaussian bilateral filter window as set by the user. */
      double
      getHalfSize () const
      {
        return (sigma_s_);
      }

      /** \brief Set the standard deviation parameter
       * \param[in] sigma_r the new standard deviation parameter
       */
      void
      setStdDev (const double sigma_r)
      {
        sigma_r_ = sigma_r;
      }

      /** \brief Get the value of the current standard deviation parameter of the bilateral filter. */
      double
      getStdDev () const
      {
        return (sigma_r_);
      }

      /** \brief Provide a pointer to the search object.
        * \param[in] tree a pointer to the spatial search object.
        */
```

```
      void
      setSearchMethod (const KdTreePtr &tree)
      {
        tree_ = tree;
      }

    private:

      /** \brief The bilateral filter Gaussian distance kernel.
        * \param[in] x the spatial distance (distance or intensity)
        * \param[in] sigma standard deviation
        */
      inline double
      kernel (double x, double sigma)
      {
        return (std::exp (- (x*x)/(2*sigma*sigma)));
      }

      /** \brief The half size of the Gaussian bilateral filter window (e.g., spatial extents in Euclidean). */
      double sigma_s_;
      /** \brief The standard deviation of the bilateral filter (e.g., standard deviation in intensity). */
      double sigma_r_;

      /** \brief A pointer to the spatial search object. */
      KdTreePtr tree_;
  };
}

#endif // PCL_FILTERS_BILATERAL_H_
```

- And the *bilateral.hpp* likes:

```
/*
 * Software License Agreement (BSD License)
 *
 *  Point Cloud Library (PCL) - www.pointclouds.org
 *  Copyright (c) 2010-2011, Willow Garage, Inc.
 *
 *  All rights reserved.
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *   * Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   * Redistributions in binary form must reproduce the above
 *     copyright notice, this list of conditions and the following
 *     disclaimer in the documentation and/or other materials provided
 *     with the distribution.
 *   * Neither the name of Willow Garage, Inc. nor the names of its
 *     contributors may be used to endorse or promote products derived
 *     from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 *  COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 *  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 *  BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 *  LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 *  CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 *  LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 *  ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 *  POSSIBILITY OF SUCH DAMAGE.
 *
 */

#ifndef PCL_FILTERS_BILATERAL_IMPL_H_
#define PCL_FILTERS_BILATERAL_IMPL_H_

#include <pcl/filters/bilateral.h>
#include <pcl/kdtree/kdtree_flann.h>
#include <pcl/kdtree/organized_data.h>

//////////////////////////////////////////////////////////////////////////////////////////////
template <typename PointT> double
pcl::BilateralFilter<PointT>::computePointWeight (const int pid,
                                                  const std::vector<int> &indices,
                                                  const std::vector<float> &distances)
{
  double BF = 0, W = 0;

  // For each neighbor
  for (std::size_t n_id = 0; n_id < indices.size (); ++n_id)
  {
    double id = indices[n_id];
    // Compute the difference in intensity
    double intensity_dist = std::abs ((*input_)[pid].intensity - (*input_)[id].intensity);

    // Compute the Gaussian intensity weights both in Euclidean and in intensity space
    double dist = std::sqrt (distances[n_id]);
    double weight = kernel (dist, sigma_s_) * kernel (intensity_dist, sigma_r_);

    // Calculate the bilateral filter response
    BF += weight * (*input_)[id].intensity;
    W += weight;
  }
  return (BF / W);
}

//////////////////////////////////////////////////////////////////////////////////////////////
template <typename PointT> void
pcl::BilateralFilter<PointT>::applyFilter (PointCloud &output)
{
  // Check if sigma_s has been given by the user
  if (sigma_s_ == 0)
  {
    PCL_ERROR ("[pcl::BilateralFilter::applyFilter] Need a sigma_s value given before continuing.\n");
    return;
  }
  // In case a search method has not been given, initialize it using some defaults
  if (!tree_)
  {
    // For organized datasets, use an OrganizedNeighbor
    if (input_->isOrganized ())
      tree_.reset (new pcl::OrganizedNeighbor<PointT> ());
    // For unorganized data, use a FLANN kdtree
    else
      tree_.reset (new pcl::KdTreeFLANN<PointT> (false));
  }
  tree_->setInputCloud (input_);

  std::vector<int> k_indices;
  std::vector<float> k_distances;

  // Copy the input data into the output
  output = *input_;

  // For all the indices given (equal to the entire cloud if none given)
  for (std::size_t i = 0; i < indices_->size (); ++i)
  {
    // Perform a radius search to find the nearest neighbors
    tree_->radiusSearch ((*indices_)[i], sigma_s_ * 2, k_indices, k_distances);

    // Overwrite the intensity value with the computed average
    output[(*indices_)[i]].intensity = computePointWeight ((*indices_)[i], k_indices, k_distances);
  }
}

#define PCL_INSTANTIATE_BilateralFilter(T) template class PCL_EXPORTS pcl::BilateralFilter<T>;

#endif // PCL_FILTERS_BILATERAL_IMPL_H_
```

Testing the new class

- Testing the new class is easy. We'll take the first code snippet example as shown above, strip the algorithm, and make it use the *pcl::BilateralFilter* class instead:

```
#include <pcl/point_types.h>
#include <pcl/io/pcd_io.h>
#include <pcl/kdtree/kdtree_flann.h>
#include <pcl/filters/bilateral.h>

typedef pcl::PointXYZI PointT;

int
main (int argc, char *argv[])
{
  std::string incloudfile = argv[1];
  std::string outcloudfile = argv[2];
  float sigma_s = atof (argv[3]);
  float sigma_r = atof (argv[4]);

  // Load cloud
  pcl::PointCloud<PointT>::Ptr cloud (new pcl::PointCloud<PointT>);
  pcl::io::loadPCDFile (incloudfile.c_str (), *cloud);

  pcl::PointCloud<PointT> outcloud;

  // Set up KDTree
  pcl::KdTreeFLANN<PointT>::Ptr tree (new pcl::KdTreeFLANN<PointT>);

  pcl::BilateralFilter<PointT> bf;
  bf.setInputCloud (cloud);
  bf.setSearchMethod (tree);
  bf.setHalfSize (sigma_s);
  bf.setStdDev (sigma_r);
  bf.filter (outcloud);

  // Save filtered output
  pcl::io::savePCDFile (outcloudfile.c_str (), outcloud);
  return (0);
}
```