

Tutorial 10: Writing A Signal Processing Block for GNU Radio

– Part I

Dawei Shen*

June 11, 2005

Abstract

This article explains how to write signal processing blocks for GNU Radio. Concretely, we will talk about how to implement a class derived from `gr_block` in C++, the naming conventions and how to use SWIG to generate the interface between Python and C++. The final ‘product’ will be a Python module in *gnuradio* package, allowing us to access the block in a simply way.

1 Overview

In this article, we explain how to write signal processing blocks for GNU Radio. This article is actually an expanded version of the on-line documentation: ‘**How to Write a Signal Processing Block**’, written by *Eric Blossom*. We will cover the same materials and use the same example. However, more comments and details will be added to improve the readability.

In previous tutorials, we have introduced the ‘Python - C++’ two-tier structure of GNU Radio. We also have met a few blocks in the examples, such as `gr_sig_source_f`, `gr_quadrature_demod_cf`, etc. However, we skipped the details about how these blocks are implemented in C++ and how they are glued to Python. In this tutorial, all these secrets will be revealed.

This article will walk through the construction of several simple signal processing blocks, and explain the techniques and idioms used.

2 The view from 30000 feet

From the Python’s point of view, GNU Radio provides a data flow abstraction. The fundamental concepts are signal processing blocks and the connections between them. This abstraction is implemented by the Python `gr.flow_graph` class, as we have discussed in tutorial 6. Each block has a set of input ports and output ports. Each port has an associated data type. The most common port types are float and `gr_complex` (equivalent to `std::complex<float>`).

From the high level point-of-view, infinite streams of data flow through the ports. At the C++ level, streams are dealt with in convenient sized pieces, represented as contiguous arrays of the underlying type.

When we write the block, we need to construct them as shared libraries that may be dynamically loaded into Python using the ‘import’ mechanism. **SWIG**, the Simplified Wrapper and Interface Generator, is used to generate the glue that allows our code to be used from Python. Writing a new signal processing block involves creating 3 files: The `.h` and `.cc` files that define the new block class

*The author is affiliated with Networking Communications and Information Processing (NCIP) Laboratory, Department of Electrical Engineering, University of Notre Dame. Welcome to send me your comments or inquiries. Email: dshen@nd.edu

and the `.i` file that tells SWIG how to generate the glue that binds the class into Python. The new class must derive from `gr_block` or one of its subclasses.

3 The base class of all signal processing blocks: `gr_block`

The C++ class `gr_block` is the base of all signal processing blocks in GNU Radio. The new block class we try to create must derive from `gr_block` or one of its subclasses. So let's study it first by looking at `gr_block.h`. To find the source code of `gr_block.h`, we can open the documentation for GNU Radio, choose the 'File List' tag and search for `gr_block.h`. You can also find it located at `/src/lib/runtime`. Refer to appendix A for the source code.

Let's take a glance at the member variables defined in `gr_block` first:

```
private:
    std::string          d_name;
    gr_io_signature_sptr d_input_signature;
    gr_io_signature_sptr d_output_signature;
    int                 d_output_multiple;
    double              d_relative_rate;    // approx output_rate / input_rate
    gr_block_detail_sptr d_detail;         // implementation details
    long                d_unique_id;      // convenient for debugging
```

`d_name` is a string saving the block's name. `d_unique_id` is a long integer, defined as the 'ID' of the block. This is convenient for debugging purpose.

What are `d_input_signature` and `d_output_signature`? What's the data type `gr_io_signature_sptr`? The story becomes complicated so early. Here we have to explain two issues: the class `gr_io_signature`, and the boost smart pointer. So, let's make some preparations first.

3.1 The class for 'IO signature': `gr_io_signature`

The class `gr_io_signature` is defined in `/src/lib/runtime/gr_io_signature.h`. As indicated by its name, `gr_io_signature` is like an endorsement on the block's input or output flows, telling their basic information. Let's look at part of its definitions in `gr_io_signature.h`:

```
class gr_io_signature {
public:
    ~gr_io_signature ();
    int min_streams () const { return d_min_streams; }
    int max_streams () const { return d_max_streams; }
    size_t sizeof_stream_item (int index) const { return d_sizeof_stream_item; }
private:
    int          d_min_streams;
    int          d_max_streams;
    size_t       d_sizeof_stream_item;
    gr_io_signature (int min_streams, int max_streams, size_t sizeof_stream_item);
    friend gr_io_signature_sptr gr_make_io_signature ( int min_streams,
                                                    int max_streams,
                                                    size_t sizeof_stream_item);
};
```

For a block's input (output), the class `gr_io_signature` defines the minimal (`d_min_streams`) and maximal (`d_max_streams`) number of streams as the lower and upper bound. The 'size' (number of bytes occupied) of an item in the stream is given by `d_sizeof_stream_item`, a member variable with the type of `size_t`.

When we create a block, we need to indicate two 'signatures' for both input and output flows.

3.2 Smart pointers: Boost

What is the data type `gr_io_signature_sptr`? Let's make an extensive investigation on it. `gr_runtime.h` is included in `gr_block.h`. In `gr_runtime.h`, we see `gr_io_signature_sptr` is a type defined as:

```
typedef boost::shared_ptr<gr_io_signature>    gr_io_signature_sptr;
```

Further, `gr_runtime.h` includes `gr_type.h` first. In `gr_type.h`, we include an interesting header file:

```
#include <boost/shared_ptr.hpp>
```

GNU Radio takes the advantage of Boost smart pointers.

3.2.1 What is Boost?

Boost is a collection of C++ libraries, a pre-required package for the installation of GNU Radio. Boost provides powerful extensions to C++ from many aspects, such as algorithm implementation, math/numerics, input/output, iterators, etc. Interested programmers can refer to <http://www.boost.org> for more information.

3.2.2 What is smart pointer?

GNU Radio borrows one cool feature from Boost: the `smart_ptr` library, so called smart pointers. Smart pointers are objects which store pointers to dynamically allocated objects. They behave much like built-in C++ pointers except that they automatically delete the object pointed to at the appropriate time. Smart pointers are particularly useful in the face of exceptions as they ensure proper destruction of dynamically allocated objects. They can also be used to keep track of dynamically allocated objects shared by multiple owners. Conceptually, smart pointers are seen as owning the object pointed to, and thus responsible for deletion of the object when it is no longer needed.

In fact, the smart pointers are defined as class templates. The library `smart_ptr` provides five smart pointer class templates, but in GNU Radio, we only use one of them: `shared_ptr`, defined in `<boost/shared_ptr.hpp>`. `shared_ptr` is used for the case when the pointed object ownership is shared by multiple pointers.

3.2.3 How to use the smart pointers in GNU Radio?

To use the smarter pointer `shared_ptr` in GNU Radio, first we need to include the header file `<boost/shared_ptr.hpp>`. Then we can use it to define a smart pointer as:

```
boost::shared_ptr<T>    pointer_name
```

These smart pointer class templates have a template parameter, `T`, which specifies the type of the object pointed to by the smart pointer. For example, `boost::shared_ptr<gr_io_signature>` declares a smart pointer pointing to an object with the class type of `gr_io_signature`. Ok, thus far we have explained what `gr_io_signature_sptr` means.

Anyway, we can simply treat the Boost smart pointer as a built-in C++ pointer, though it's equipped with some cool features. There is nothing difficult.

Let's go back to the member variables in the class `gr_block`. Now we can understand `d_input_signature` and `d_output_signature` are two smart pointers referring to `gr_io_signature` objects for the input and output flows, which is the basic information of a block. Let's hang up the other member variables for a while and look at two important methods defined in `gr_block`: `forecast()`, and `general_work()`. Before introducing them, it is better to look at the data types defined in GNU Radio first.

3.3 The data types in GNU Radio

The special data types defined in GNU Radio can be found in `gr_types.h` and `gr_complex.h`:

```
typedef std::complex<float>          gr_complex;
typedef std::complex<double>        gr_complexd;

typedef std::vector<int>             gr_vector_int;
typedef std::vector<float>          gr_vector_float;
typedef std::vector<double>         gr_vector_double;
typedef std::vector<void *>         gr_vector_void_star;
typedef std::vector<const void *>   gr_vector_const_void_star;

typedef short                        gr_int16;
typedef int                          gr_int32;
typedef unsigned short               gr_uint16;
typedef unsigned int                 gr_uint32;
```

As we see, the data types with the prefix ‘gr_’ are nothing but new names for the C++ built-in data types. We just represent them using a consistent and convenient way. `vector` and `complex` are both C++ standard libraries, which are quite useful in GNU Radio.

3.4 The core of a block: the method `general_work()`

The method `general_work()` is pure virtual, we definitely need to override that. `general_work` is the method that does the actual signal processing, which is the ‘CPU’ of the block.

```
virtual int general_work ( int          noutput_items,
                          gr_vector_int &ninput_items,
                          gr_vector_const_void_star &input_items,
                          gr_vector_void_star &output_items) = 0;
```

Generally speaking, the method `general_work()` compute the output streams from the input streams. Let’s introduce the four arguments first.

`noutput_items` is the number of output items to write on each output stream. `ninput_items` gives the number of input items available on each input stream. A block may have `x` input streams and `y` output streams. `ninput_items` is an integer ‘vector’ of length `x`, the i^{th} element of which gives the number of available items on the i^{th} input stream. However, for the output flow, why is `noutput_items` simply an integer, not a vector? This is because, due to some technical issues, GNU Radio of the current version only supports a block having the same data rates for all output streams, i.e. the number of output items to write on each output stream is the same for all output streams. The data rates for input streams could be different.

We have introduced the data types `gr_vector_const_void_star` and `gr_vector_void_star` in last subsection. `input_items` is a vector of pointers to the input items, one entry per input stream. `output_items` is a vector of pointers to the output items, one entry per output stream. We actually use these pointers to get the input data and write the computed output data to appropriate streams. Note that the last three arguments `ninput_items`, `input_items` and `output_items` are passed by reference, indicated by the character ‘&’. So they can possibly be modified in the method `general_work()`. The returned value of `general_work()` is the number of items actually written to each output stream, or -1 on EOF. It is OK to return a value less than `noutput_items`.

Finally and significantly, when we override `general_work()` for our own block, we **MUST** call `consume()` or `consume_each()` methods to indicate how many items have been consumed on each input stream.

```
void consume (int which_input, int how_many_items);
```

The method `consume()` tells the scheduler how many items (given by ‘`how_many_items`’) of the i^{th} input stream (given by ‘`which_input`’) have been consumed. If each input stream has consumed the same number of items, we can use `consume_each()` instead.

```
void consume_each (int how_many_items);
```

It tells the scheduler each input stream has consumed ‘`how_many_items`’ items.

The reason why we have to call the `consume()` or `consume_each()` method is we have to tell the scheduler how many items of the input streams have been consumed, so that the scheduler can arrange the upstream buffer and associated pointers accordingly. The stories behind these two methods are overdetailed for us. Just keep in mind that they have been well implemented by GNU Radio and we should remember to call them every time we override the `general_work()` method.

3.5 Brief introduction to other methods and member variables

3.5.1 The method `forecast()`

```
virtual void forecast ( int          noutput_items,
                      gr_vector_int &ninput_items_required);
```

The method `forecast()` is used to estimate the input requirements given an output request.

The first parameter `noutput_items` has been introduced in `general_work()`, which is the number of output items to produce for each output stream. The second parameter `ninput_items_required` is an integer vector, saving the number of input items required on each input stream.

When we override the `forecast()` method, we need to estimate the number of data items required on each input stream, given the request to produce ‘`noutput_items`’ items for each output stream. The estimate doesn’t have to be exact, but should be close. The argument `ninput_items_required` is passed by reference, so that the calculated estimates can be saved into it directly.

We can look at ‘`/src/lib/runtime/gr_block.cc`’ to look at its ‘stub 1:1 implementation’, in which the number of input items required on each input stream is simply set to `noutput_items`. This is valid for many regular blocks, but obviously not appropriate for interpolators, decimators, or blocks with a more complicated relationship between `noutput_items` and the input requirements.

3.5.2 `d_output_multiple` and the method `set_output_multiple()`

Now let’s introduce the fourth member variable defined in `gr_block`: `d_output_multiple`. It is used to constrain the `noutput_items` argument passed to `forecast()` and `general_work()`.

The scheduler will ensure that the `noutput_items` argument passed to `forecast()` and `general_work()` will be an integer multiple of `d_output_multiple`. The default value of `d_output_multiple` is 1.

Here is a critical point worth emphasizing. Suppose we’re going to design a block class, after we override the `general_work()` or `forecast()` methods, who will use or call these methods, and how? Of particular importance, what value will be passed to the argument `noutput_items` and who does that? A simple answer could be: ‘The scheduler will call these methods with appropriate arguments.’ When we implement `general_work()` or `forecast()`, we always assume `noutput_items` and other arguments have been provided conceptually. In fact, we never call these methods and set the arguments explicitly. The ‘scheduler’ will organize everything and call the methods according to the higher level policy and buffer allocation strategy. The tricks behind the scene involve too many details and are beyond the necessity. Don’t worry about them when we design our own blocks:)

We do have some level of control to the argument `noutput_items`. The variable `d_output_multiple` tells the scheduler `noutput_items` must be an integer multiple of `d_output_multiple`.

We can set the value of `d_output_multiple` using the method `set_output_multiple()` and get its value using the method `output_multiple()`.

```
void    gr_block::set_output_multiple (int multiple)
{
```

```

    if (multiple < 1)
        throw std::invalid_argument ("gr_block::set_output_multiple");
    d_output_multiple = multiple;
}
int     output_multiple () const { return d_output_multiple; }

```

3.5.3 d_relative_rate and the method set_relative_rate()

The fifth member variable `d_relative_rate` gives the approximate information on the relative data rate, i.e. the approximate output rate / input rate.

This information provides a hint to the buffer allocator and scheduler, so that they can arrange the buffer allocation and adjust parameters accordingly. `d_relative_rate` is 1.0 by default, which is true for most signal processing blocks. Obviously, the decimators' `d_relative_rates` should be less than 1.0, while the interpolators' `d_relative_rates` is larger than 1.0.

We can set the value of `d_relative_rate` using the method `set_relative_rate()` and get its value using the method `relative_rate()`.

```

void     gr_block::set_relative_rate (double relative_rate)
{
    if (relative_rate < 0.0)
        throw std::invalid_argument ("gr_block::set_relative_rate");
    d_relative_rate = relative_rate;
}
double  relative_rate () const { return d_relative_rate; }

```

OK! At this point, we have introduced the class `gr_block` thoroughly. Note that we skip the the introduction to the member variable `d_detail` and its related methods. They are too involved and really for only internal use. We seldom meet them when we design our own block. It is strongly recommended to read the source code carefully to gain a complete understanding of these stuffs.

4 Naming Conventions

After studying the class `gr_block` and reading several source files, we now should summarize the naming conventions used in GNU Radio. Following the name conventions could assist us in comprehending the code base and gluing C++ and Python together.

In GNU Radio, with the exception of macros and other constant values, all identifiers shall be in lower case with 'words_separated_like_this'. Macros and constant values shall be in `UPPER_CASE`.

4.1 Package prefix

All globally visible names (types, functions, variables, constants, etc.) shall begin with a 'package prefix', followed by an underscore. The bulk of the code in GNU Radio belongs to the 'gr' package, hence names look like `gr_open_file (...)`.

Large coherent bodies of code may use other package prefixes. Here are some frequently seen ones:

```

gr_:    Almost everything in GNU Radio
usrp_:  Universal Software Radio Peripheral (USRP) related packages
qa_:    Quality Assurance, used for our testing code

```

4.2 Class data members (instance variables)

As we have seen, all class data members shall begin with the prefix 'd_'.

The big win is when you're staring at a block of code, it's obvious which of the things being assigned to persist outside of the block. This also keeps you from having to be creative with parameter names for methods and constructors. You just use the same name as the member variables, without the 'd_'.

```

class gr_wonderfulness {
    std::string    d_name;
    double        d_wonderfulness_factor;
public:
    gr_wonderfulness (std::string name, double wonderfulness_factor)
        : d_name (name), d_wonderfulness_factor (wonderfulness_factor)
        ...
};

```

All class static data members shall begin with ‘s_’. We haven’t met it so far.

4.3 File names

Each significant class shall be contained in its own files. For example, the declaration of the class `gr_foo` shall be in `gr_foo.h` and the definition in `gr_foo.cc`.

4.4 Suffixes

By convention, we encode the input and output types of signal processing blocks in their name using suffixes. The suffix is typically one or two characters long. Sources and sinks have single character suffixes. Regular blocks that have both inputs and outputs have two character suffixes. The first character indicates the type of the input streams, while the second indicates the type of the output streams. FIR filter blocks have a three character suffix, indicating the type of the inputs, outputs and taps, respectively.

These are the suffix characters and their interpretations:

```

f - single precision floating point
c - complex<float>
s - short (16-bit integer)
i - integer (32-bit integer)

```

In addition, for those cases where the block deals with streams of vectors, we use the character ‘v’ as the first character of the suffix. An example of this usage is `gr_fft_vcc`. The FFT block takes a vector of complex numbers on its input and produces a vector of complex numbers on its output.

5 Our first block: `howto_square_ff`

Let’s work on our first block `howto_square_ff`. It simply computes the square of the input stream:

$$y[n] = x^2[n] \tag{1}$$

We implement the block class `howto_square_ff` directly derived from `gr_block` and override necessary methods such as `general_work()`. The source code can be found at:

[/gr-howto-write-a-block/src/lib/howto_square_ff.h \(.cc\)](#)

Please also refer to appendix B and C for the code.

Let’s emphasize some of key points in the code.

5.1 The constructor

First let’s take a look at its constructor, where some tricks are involved:

In `howto_square_ff.h`:

```

...
typedef boost::shared_ptr<howto_square_ff> howto_square_ff_sptr;
howto_square_ff_sptr howto_make_square_ff ();
class howto_square_ff : public gr_block
{
private:
    friend howto_square_ff_sptr howto_make_square_ff ();
    howto_square_ff ();
    ...
}
...

```

In `howto_square_ff.cc`:

```

...
howto_square_ff_sptr howto_make_square_ff ()
{
    return howto_square_ff_sptr (new howto_square_ff ());
}
...
static const int MIN_IN = 1;    // minimum number of input streams
static const int MAX_IN = 1;    // maximum number of input streams
static const int MIN_OUT = 1;   // minimum number of output streams
static const int MAX_OUT = 1;   // maximum number of output streams

howto_square_ff::howto_square_ff ()
    : gr_block ("square_ff",
               gr_make_io_signature (MIN_IN, MAX_IN, sizeof (float)),
               gr_make_io_signature (MIN_OUT, MAX_OUT, sizeof (float)))
{
    // nothing else required in this example
}
...

```

We have talked about Boost smart pointers just now. In GNU Radio, we use one of the smart pointers ‘`boost::shared_ptr`’ instead of raw C++ pointers for all access to ‘`gr_`’ blocks (and many other data structures) exclusively. So to avoid accidental use of raw C++ pointers, `howto_square_ff`’s constructor is **private**. Private constructor can’t be called outside the class. So the following creation of a `howto_square_ff` instance is illegal:

```
howto_square_ff* test_block = new howto_square_ff()
```

because the constructor of `howto_square_ff` is not public. By doing this, we avoid the possibility that a block object is pointed by a raw C++ pointer.

Instead, we use the `howto_make_square_ff()` function as the public interface for creating new instances. First, it is declared as the friend function of the class `howto_square_ff`, so that it could access all private member variables and methods defined in `howto_square_ff`.

```
friend howto_square_ff_sptr howto_make_square_ff ();
```

Then in the function `howto_make_square_ff()`, we call the private constructor of `howto_square_ff` to create an instance using the `new` command. However, we cast the data type of the returned pointer from the raw C++ pointer to the smart pointer `howto_square_ff_sptr`:

```

howto_square_ff_sptr howto_make_square_ff ()
{
    return howto_square_ff_sptr (new howto_square_ff ());
}

```

By playing these tricks, we actually assure that all ‘gr_’ blocks are pointed by the smart pointer `boost::shared_ptr` when being created. The function `howto_make_square_ff()` is used as the public interface for creating new instances.

This trick is played for almost every ‘gr_’ block and many other data structures. Let’s see the private constructor of `howto_square_ff`:

```
static const int MIN_IN = 1;    // minimum number of input streams
static const int MAX_IN = 1;    // maximum number of input streams
static const int MIN_OUT = 1;   // minimum number of output streams
static const int MAX_OUT = 1;   // maximum number of output streams

howto_square_ff::howto_square_ff ()
: gr_block ("square_ff",
           gr_make_io_signature (MIN_IN, MAX_IN, sizeof (float)),
           gr_make_io_signature (MIN_OUT, MAX_OUT, sizeof (float))) {}
```

We recall that the constructor of `gr_block` requires the block’s name as the first argument, input and output signatures as the second and third arguments. The 2nd and 3rd arguments are actually smart pointers of the type `gr_io_signature_sptr`, pointing to instances of `gr_io_signature`. Similarly, the constructor of `gr_io_signature` is also private. The friend function `gr_make_io_signature()` is used as the public interface for creating new ‘signatures’ and it returns the smart pointer `gr_io_signature_sptr` pointing to the signatures. As a convention, the `_sptr` suffix indicates the smart pointer `boost::shared_ptr`.

In our block `howto_square_ff`, we need only 1 input stream and 1 output stream. So the minimum and maximum number of streams for both I/O signatures is simply set to 1. The data type of the items in both input and output flows is ‘float’. This is also reflected in the suffix of our block’s name: ‘_ff’.

5.2 Overriding the core method: `general_work()`

```
int howto_square_ff::general_work ( int          noutput_items,
                                   gr_vector_int &ninput_items,
                                   gr_vector_const_void_star &input_items,
                                   gr_vector_void_star &output_items)
{
    const float *in = (const float *) input_items[0];
    float *out = (float *) output_items[0];

    for (int i = 0; i < noutput_items; i++){
        out[i] = in[i] * in[i];
    }

    // Tell runtime system how many input items we consumed on each input stream.
    consume_each (noutput_items);

    // Tell runtime system how many output items we produced.
    return noutput_items;
}
```

This is the ‘real’ signal processing part. Its implementation is quite simple and straightforward. Since our block has only 1 input stream and 1 output stream, the vectors of pointers `input_items` and `output_items` contain only one entry. We compute the items on the output stream one by one:

```
for (int i = 0; i < noutput_items; i++){
    out[i] = in[i] * in[i];
}
```

Don't forget to tell the runtime system how many input items have been consumed on each input stream by calling the `consume()` or `consume_each()` method. In our 1:1 scenario, the number of items consumed on the input stream is simply equal to the items produced on the output stream: `noutput_items`.

Finally, we return the number of items we have produced. Note that in our example, we don't override the `forecast()` method. Because the default 1:1 implementation is reasonable for our block.

OK! We're done! We have got our first own block `howto_square_ff!`

6 conclusion

Now the remaining task is to establish the connection between C++ and Python. Let's take a break and leave it to the next tutorial.

In this article, we analyzed the basic class for all blocks - `gr_block` detailedly. We also introduced the Boost smart pointers and the naming conventions of GNU Radio along the way. This forms the basis of creating a signal processing block. In the next tutorial, we will talk about the hacking skills to glue the Python and C++ together, so that the block we have created in this article can be used from Python in a simple way.

APPENDIX A: The source code: `gr_block.h`

```
#ifndef INCLUDED_GR_BLOCK_H
#define INCLUDED_GR_BLOCK_H

#include <gr_runtime.h>
#include <string>

class gr_block {

public:

    virtual ~gr_block ();

    std::string name () const { return d_name; }
    gr_io_signature_sptr input_signature () const { return d_input_signature; }
    gr_io_signature_sptr output_signature () const { return d_output_signature; }
    long unique_id () const { return d_unique_id; }

    virtual void forecast ( int          noutput_items,
                           gr_vector_int &ninput_items_required);

    virtual int general_work ( int          noutput_items,
                              gr_vector_int &ninput_items,
                              gr_vector_const_void_star &input_items,
                              gr_vector_void_star &output_items) = 0;

    virtual bool check_topology (int ninputs, int noutputs);

    void set_output_multiple (int multiple);
    int output_multiple () const { return d_output_multiple; }

    void consume (int which_input, int how_many_items);

    void consume_each (int how_many_items);

    void set_relative_rate (double relative_rate);
```

```

    double relative_rate () const { return d_relative_rate; }

private:

    std::string          d_name;
    gr_io_signature_sptr d_input_signature;
    gr_io_signature_sptr d_output_signature;
    int                  d_output_multiple;
    double                d_relative_rate;    // approx output_rate / input_rate
    gr_block_detail_sptr d_detail;           // implementation details
    long                  d_unique_id;       // convenient for debugging

protected:

    gr_block (const std::string &name,
              gr_io_signature_sptr input_signature,
              gr_io_signature_sptr output_signature);

    void set_input_signature (gr_io_signature_sptr iosig){
        d_input_signature = iosig;
    }

    void set_output_signature (gr_io_signature_sptr iosig){
        d_output_signature = iosig;
    }

public:
    gr_block_detail_sptr detail () const { return d_detail; }
    void set_detail (gr_block_detail_sptr detail) { d_detail = detail; }
};

    long gr_block_ncurrently_allocated ();

#endif /* INCLUDED_GR_BLOCK_H */

```

APPENDIX B: The source code: howto_square_ff.h

```

#ifndef INCLUDED_HOWTO_SQUARE_FF_H
#define INCLUDED_HOWTO_SQUARE_FF_H

#include <gr_block.h>

class howto_square_ff;

typedef boost::shared_ptr<howto_square_ff> howto_square_ff_sptr;

howto_square_ff_sptr howto_make_square_ff ();

class howto_square_ff : public gr_block
{
private:
    friend howto_square_ff_sptr howto_make_square_ff ();

```

```

    howto_square_ff (); // private constructor

public:
    ~howto_square_ff (); // public destructor
    int general_work ( int                noutput_items,
                      gr_vector_int      &ninput_items,
                      gr_vector_const_void_star &input_items,
                      gr_vector_void_star &output_items);
};

#endif /* INCLUDED_HOWTO_SQUARE_FF_H */

```

APPENDIX C: The source code: howto_square_ff.cc

```

#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include <howto_square_ff.h>
#include <gr_io_signature.h>

howto_square_ff_sptr
howto_make_square_ff ()
{
    return howto_square_ff_sptr (new howto_square_ff ());
}

static const int MIN_IN = 1; // minimum number of input streams
static const int MAX_IN = 1; // maximum number of input streams
static const int MIN_OUT = 1; // minimum number of output streams
static const int MAX_OUT = 1; // maximum number of output streams

howto_square_ff::howto_square_ff ()
: gr_block ("square_ff",
           gr_make_io_signature (MIN_IN, MAX_IN, sizeof (float)),
           gr_make_io_signature (MIN_OUT, MAX_OUT, sizeof (float)))
{
    // nothing else required in this example
}

howto_square_ff::~howto_square_ff ()
{
    // nothing else required in this example
}

int howto_square_ff::general_work (int                noutput_items,
                                   gr_vector_int      &ninput_items,
                                   gr_vector_const_void_star &input_items,
                                   gr_vector_void_star &output_items)
{
    const float *in = (const float *) input_items[0];
    float *out = (float *) output_items[0];
}

```

```
for (int i = 0; i < noutput_items; i++){
    out[i] = in[i] * in[i];
}

consume_each (noutput_items);

// Tell runtime system how many output items we produced.
return noutput_items;
}
```

References

- [1] Eric Blossom, **How to Write a Signal Processing Block**,
<http://www.gnu.org/software/gnuradio/doc/howto-write-a-block.html>
- [2] **Python on-line tutorials**, <http://www.python.org/doc/current/tut/>
- [3] Eric Blossom, **Exploring GNU Radio**,
<http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html>