

## Previous topic

[gnuradio.fft](#)

## Next topic

[gnuradio.noaa](#)

## This Page

[Show Source](#)

## Quick search

 

Enter search terms or a module, class or function name.

## gnuradio.filters

Filter blocks and related functions.

`gnuradio.filter.dc_blocker_cc(int D, bool long_form) → dc_blocker_cc_sptr`

a computationally efficient controllable DC blocker

This block implements a computationally efficient DC blocker that produces a tighter notch filter around DC for a smaller group delay than an equivalent FIR filter or using a single pole IIR filter (though the IIR filter is computationally cheaper).

The block defaults to using a delay line of length 32 and the long form of the filter. Optionally, the delay line length can be changed to alter the width of the DC notch (longer lines will decrease the width).

The long form of the filter produces a nearly flat response outside of the notch but at the cost of a group delay of 2D-2.

The short form of the filter does not have as flat a response in the passband but has a group delay of only D-1 and is cheaper to compute.

The theory behind this block can be found in the paper:

Constructor Specific Documentation:

Make a DC blocker block.

**Parameters:**

- D** – (int) the length of the delay line
- long\_form** – (bool) whether to use long (true, default) or short form

`dc_blocker_cc_sptr.active_thread_priority(dc_blocker_cc_sptr self) → int`
`dc_blocker_cc_sptr.declare_sample_delay(dc_blocker_cc_sptr self, int which, int delay)`  
`declare_sample_delay(dc_blocker_cc_sptr self, unsigned int delay)`
`dc_blocker_cc_sptr.group_delay(dc_blocker_cc_sptr self) → int`
`dc_blocker_cc_sptr.message_subscribers(dc_blocker_cc_sptr self, swig_int_ptr which_port) → swig_int_ptr`
`dc_blocker_cc_sptr.min_noutput_items(dc_blocker_cc_sptr self) → int`
`dc_blocker_cc_sptr.pc_input_buffers_full_avg(dc_blocker_cc_sptr self, int which) → float`  
`pc_input_buffers_full_avg(dc_blocker_cc_sptr self) → pmt_vector_float`
`dc_blocker_cc_sptr.pc_noutput_items_avg(dc_blocker_cc_sptr self) → float`
`dc_blocker_cc_sptr.pc_nproduced_avg(dc_blocker_cc_sptr self) → float`
`dc_blocker_cc_sptr.pc_output_buffers_full_avg(dc_blocker_cc_sptr self, int which) → float`  
`pc_output_buffers_full_avg(dc_blocker_cc_sptr self) → pmt_vector_float`
`dc_blocker_cc_sptr.pc_throughput_avg(dc_blocker_cc_sptr self) → float`
`dc_blocker_cc_sptr.pc_work_time_avg(dc_blocker_cc_sptr self) → float`
`dc_blocker_cc_sptr.pc_work_time_total(dc_blocker_cc_sptr self) → float`
`dc_blocker_cc_sptr.sample_delay(dc_blocker_cc_sptr self, int which) → unsigned int`
`dc_blocker_cc_sptr.set_min_noutput_items(dc_blocker_cc_sptr self, int m)`
`dc_blocker_cc_sptr.set_thread_priority(dc_blocker_cc_sptr self, int priority) → int`
`dc_blocker_cc_sptr.thread_priority(dc_blocker_cc_sptr self) → int`
`gnuradio.filter.dc_blocker_ff(int D, bool long_form=True) → dc_blocker_ff_sptr`

a computationally efficient controllable DC blocker

This block implements a computationally efficient DC blocker that produces a tighter notch filter around DC for a smaller group delay than an equivalent FIR filter or using a single pole IIR filter (though the IIR filter is computationally cheaper).

The block defaults to using a delay line of length 32 and the long form of the filter. Optionally, the delay line length can be changed to alter the width of the DC notch (longer lines will decrease the width).

The long form of the filter produces a nearly flat response outside of the notch but at the cost of a group delay of  $2D-2$ .

The short form of the filter does not have as flat a response in the passband but has a group delay of only  $D-1$  and is cheaper to compute.

The theory behind this block can be found in the paper:

Constructor Specific Documentation:

Make a DC blocker block.

**Parameters:**

- **D** – (int) the length of the delay line
- **long\_form** – (bool) whether to use long (true, default) or short form

```
dc_blocker_ff_sptr.active_thread_priority(dc_blocker_ff_sptr self) → int

dc_blocker_ff_sptr.declare_sample_delay(dc_blocker_ff_sptr self, int which, int delay)
    declare_sample_delay(dc_blocker_ff_sptr self, unsigned int delay)

dc_blocker_ff_sptr.group_delay(dc_blocker_ff_sptr self) → int

dc_blocker_ff_sptr.message_subscribers(dc_blocker_ff_sptr self, swig_int_ptr which_port) →
    swig_int_ptr

dc_blocker_ff_sptr.min_noutput_items(dc_blocker_ff_sptr self) → int

dc_blocker_ff_sptr.pc_input_buffers_full_avg(dc_blocker_ff_sptr self, int which) → float
    pc_input_buffers_full_avg(dc_blocker_ff_sptr self) → pmt_vector_float

dc_blocker_ff_sptr.pc_noutput_items_avg(dc_blocker_ff_sptr self) → float

dc_blocker_ff_sptr.pc_nproduced_avg(dc_blocker_ff_sptr self) → float

dc_blocker_ff_sptr.pc_output_buffers_full_avg(dc_blocker_ff_sptr self, int which) → float
    pc_output_buffers_full_avg(dc_blocker_ff_sptr self) → pmt_vector_float

dc_blocker_ff_sptr.pc_throughput_avg(dc_blocker_ff_sptr self) → float

dc_blocker_ff_sptr.pc_work_time_avg(dc_blocker_ff_sptr self) → float

dc_blocker_ff_sptr.pc_work_time_total(dc_blocker_ff_sptr self) → float

dc_blocker_ff_sptr.sample_delay(dc_blocker_ff_sptr self, int which) → unsigned int

dc_blocker_ff_sptr.set_min_noutput_items(dc_blocker_ff_sptr self, int m)

dc_blocker_ff_sptr.set_thread_priority(dc_blocker_ff_sptr self, int priority) → int

dc_blocker_ff_sptr.thread_priority(dc_blocker_ff_sptr self) → int
```

```
gnuradio.filter.fft_filter_ccc(int decimation, pmt_vector_cfloat taps, int nthreads=1) →
    fft_filter_ccc_sptr
```

Fast FFT filter with gr\_complex input, gr\_complex output and gr\_complex taps.

This block implements a complex decimating filter using the fast convolution method via an FFT. The decimation factor is an integer that is greater than or equal to 1.

The filter takes a set of complex (or real) taps to use in the filtering operation. These taps can be defined as anything that satisfies the user's filtering needs. For standard filters such as lowpass, highpass, bandpass, etc., the filter.firdec and filter.optfir classes provide convenient generating methods.

This filter is implemented by using the FFTW package to perform the required FFTs. An optional argument, nthreads, may be passed to the constructor (or set using the set\_nthreads member function) to split the FFT among N number of threads. This can improve performance on very large FFTs (that is, if the number of taps used is very large) if you have enough threads/cores to support it.

Constructor Specific Documentation:

Build an FFT filter blocks.

**Parameters:**

- **decimation** –  $\geq 1$
- **taps** – complex filter taps
- **nthreads** – number of threads for the FFT to use

```
fft_filter_ccc_sptr.active_thread_priority(fft_filter_ccc_sptr self) → int
```

```

fft_filter_ccc_sptr.declare_sample_delay(fft_filter_ccc_sptr self, int which, int delay)
    declare_sample_delay(fft_filter_ccc_sptr self, unsigned int delay)

fft_filter_ccc_sptr.message_subscribers(fft_filter_ccc_sptr self, swig_int_ptr which_port) →
swig_int_ptr

fft_filter_ccc_sptr.min_noutput_items(fft_filter_ccc_sptr self) → int

fft_filter_ccc_sptr.pc_input_buffers_full_avg(fft_filter_ccc_sptr self, int which) → float
    pc_input_buffers_full_avg(fft_filter_ccc_sptr self) -> pmt_vector_float

fft_filter_ccc_sptr.pc_noutput_items_avg(fft_filter_ccc_sptr self) → float

fft_filter_ccc_sptr.pc_nproduced_avg(fft_filter_ccc_sptr self) → float

fft_filter_ccc_sptr.pc_output_buffers_full_avg(fft_filter_ccc_sptr self, int which) → float
    pc_output_buffers_full_avg(fft_filter_ccc_sptr self) -> pmt_vector_float

fft_filter_ccc_sptr.pc_throughput_avg(fft_filter_ccc_sptr self) → float

fft_filter_ccc_sptr.pc_work_time_avg(fft_filter_ccc_sptr self) → float

fft_filter_ccc_sptr.pc_work_time_total(fft_filter_ccc_sptr self) → float

fft_filter_ccc_sptr.sample_delay(fft_filter_ccc_sptr self, int which) → unsigned int

fft_filter_ccc_sptr.set_min_noutput_items(fft_filter_ccc_sptr self, int m)

fft_filter_ccc_sptr.set_taps(fft_filter_ccc_sptr self, pmt_vector_cfloat taps)

fft_filter_ccc_sptr.set_thread_priority(fft_filter_ccc_sptr self, int priority) → int

fft_filter_ccc_sptr.taps(fft_filter_ccc_sptr self) → pmt_vector_cfloat

fft_filter_ccc_sptr.thread_priority(fft_filter_ccc_sptr self) → int

gnuradio.filter.fft_filter_ccf(int decimation, pmt_vector_float taps, int nthreads=1) →
fft_filter_ccf_sptr

```

Fast FFT filter with gr\_complex input, gr\_complex output and float taps.

This block implements a complex decimating filter using the fast convolution method via an FFT. The decimation factor is an integer that is greater than or equal to 1.

The filter takes a set of complex (or real) taps to use in the filtering operation. These taps can be defined as anything that satisfies the user's filtering needs. For standard filters such as lowpass, highpass, bandpass, etc., the filter.firdec and filter.optfir classes provide convenient generating methods.

This filter is implemented by using the FFTW package to perform the required FFTs. An optional argument, nthreads, may be passed to the constructor (or set using the set\_nthreads member function) to split the FFT among N number of threads. This can improve performance on very large FFTs (that is, if the number of taps used is very large) if you have enough threads/cores to support it.

Constructor Specific Documentation:

Build an FFT filter blocks.

**Parameters:**

- **decimation** –  $\geq 1$
- **taps** – complex filter taps
- **nthreads** – number of threads for the FFT to use

```

fft_filter_ccf_sptr.active_thread_priority(fft_filter_ccf_sptr self) → int

fft_filter_ccf_sptr.declare_sample_delay(fft_filter_ccf_sptr self, int which, int delay)
    declare_sample_delay(fft_filter_ccf_sptr self, unsigned int delay)

fft_filter_ccf_sptr.message_subscribers(fft_filter_ccf_sptr self, swig_int_ptr which_port) →
swig_int_ptr

fft_filter_ccf_sptr.min_noutput_items(fft_filter_ccf_sptr self) → int

fft_filter_ccf_sptr.pc_input_buffers_full_avg(fft_filter_ccf_sptr self, int which) → float
    pc_input_buffers_full_avg(fft_filter_ccf_sptr self) -> pmt_vector_float

fft_filter_ccf_sptr.pc_noutput_items_avg(fft_filter_ccf_sptr self) → float

```

```

fft_filter_ccf_sptr.pc_nproduced_avg(fft_filter_ccf_sptr self) → float

fft_filter_ccf_sptr.pc_output_buffers_full_avg(fft_filter_ccf_sptr self, int which) → float
pc_output_buffers_full_avg(fft_filter_ccf_sptr self) -> pmt_vector_float

fft_filter_ccf_sptr.pc_throughput_avg(fft_filter_ccf_sptr self) → float

fft_filter_ccf_sptr.pc_work_time_avg(fft_filter_ccf_sptr self) → float

fft_filter_ccf_sptr.pc_work_time_total(fft_filter_ccf_sptr self) → float

fft_filter_ccf_sptr.sample_delay(fft_filter_ccf_sptr self, int which) → unsigned int

fft_filter_ccf_sptr.set_min_noutput_items(fft_filter_ccf_sptr self, int m)

fft_filter_ccf_sptr.set_taps(fft_filter_ccf_sptr self, pmt_vector_float taps)

fft_filter_ccf_sptr.set_thread_priority(fft_filter_ccf_sptr self, int priority) → int

fft_filter_ccf_sptr.taps(fft_filter_ccf_sptr self) → pmt_vector_float

fft_filter_ccf_sptr.thread_priority(fft_filter_ccf_sptr self) → int

```

gnuradio.filter.**fft\_filter\_fff**(int decimation, pmt\_vector\_float taps, int nthreads=1) →  
**fft\_filter\_fff\_sptr**

Fast FFT filter with float input, float output and float taps.

This block implements a real-value decimating filter using the fast convolution method via an FFT. The decimation factor is an integer that is greater than or equal to 1.

The filter takes a set of real-valued taps to use in the filtering operation. These taps can be defined as anything that satisfies the user's filtering needs. For standard filters such as lowpass, highpass, bandpass, etc., the filter.firdec and filter.optfir classes provide convenient generating methods.

This filter is implemented by using the FFTW package to perform the required FFTs. An optional argument, nthreads, may be passed to the constructor (or set using the set\_nthreads member function) to split the FFT among N number of threads. This can improve performance on very large FFTs (that is, if the number of taps used is very large) if you have enough threads/cores to support it.

Constructor Specific Documentation:

Build an FFT filter block.

**Parameters:**

- **decimation** – >= 1
- **taps** – float filter taps
- **nthreads** – number of threads for the FFT to use

```

fft_filter_fff_sptr.active_thread_priority(fft_filter_fff_sptr self) → int

fft_filter_fff_sptr.declare_sample_delay(fft_filter_fff_sptr self, int which, int delay)
declare_sample_delay(fft_filter_fff_sptr self, unsigned int delay)

fft_filter_fff_sptr.message_subscribers(fft_filter_fff_sptr self, swig_int_ptr which_port) →
swig_int_ptr

fft_filter_fff_sptr.min_noutput_items(fft_filter_fff_sptr self) → int

fft_filter_fff_sptr.pc_input_buffers_full_avg(fft_filter_fff_sptr self, int which) → float
pc_input_buffers_full_avg(fft_filter_fff_sptr self) -> pmt_vector_float

fft_filter_fff_sptr.pc_noutput_items_avg(fft_filter_fff_sptr self) → float

fft_filter_fff_sptr.pc_nproduced_avg(fft_filter_fff_sptr self) → float

fft_filter_fff_sptr.pc_output_buffers_full_avg(fft_filter_fff_sptr self, int which) → float
pc_output_buffers_full_avg(fft_filter_fff_sptr self) -> pmt_vector_float

fft_filter_fff_sptr.pc_throughput_avg(fft_filter_fff_sptr self) → float

fft_filter_fff_sptr.pc_work_time_avg(fft_filter_fff_sptr self) → float

fft_filter_fff_sptr.pc_work_time_total(fft_filter_fff_sptr self) → float

fft_filter_fff_sptr.sample_delay(fft_filter_fff_sptr self, int which) → unsigned int

fft_filter_fff_sptr.set_min_noutput_items(fft_filter_fff_sptr self, int m)

```

```
fft_filter_fff_sptr.set_taps(fft_filter_fff_sptr self, pmt_vector_float taps)

fft_filter_fff_sptr.set_thread_priority(fft_filter_fff_sptr self, int priority) → int

fft_filter_fff_sptr.taps(fft_filter_fff_sptr self) → pmt_vector_float

fft_filter_fff_sptr.thread_priority(fft_filter_fff_sptr self) → int
```

gnuradio.filter.**filter\_delay\_fc**(pmt\_vector\_float taps) → filter\_delay\_fc\_sptr  
Filter-Delay Combination Block.

The block takes one or two float stream and outputs a complex stream.

If only one float stream is input, the real output is a delayed version of this input and the imaginary output is the filtered output.

If two floats are connected to the input, then the real output is the delayed version of the first input, and the imaginary output is the filtered output.

The delay in the real path accounts for the group delay introduced by the filter in the imaginary path. The filter taps needs to be calculated before initializing this block.

Constructor Specific Documentation:

Build a filter with delay block.

**Parameters:** taps –

```
filter_delay_fc_sptr.active_thread_priority(filter_delay_fc_sptr self) → int

filter_delay_fc_sptr.declare_sample_delay(filter_delay_fc_sptr self, int which, int delay)
    declare_sample_delay(filter_delay_fc_sptr self, unsigned int delay)

filter_delay_fc_sptr.message_subscribers(filter_delay_fc_sptr self, swig_int_ptr which_port) →
    swig_int_ptr

filter_delay_fc_sptr.min_noutput_items(filter_delay_fc_sptr self) → int

filter_delay_fc_sptr.pc_input_buffers_full_avg(filter_delay_fc_sptr self, int which) → float
    pc_input_buffers_full_avg(filter_delay_fc_sptr self) → pmt_vector_float

filter_delay_fc_sptr.pc_noutput_items_avg(filter_delay_fc_sptr self) → float

filter_delay_fc_sptr.pc_nproduced_avg(filter_delay_fc_sptr self) → float

filter_delay_fc_sptr.pc_output_buffers_full_avg(filter_delay_fc_sptr self, int which) → float
    pc_output_buffers_full_avg(filter_delay_fc_sptr self) → pmt_vector_float

filter_delay_fc_sptr.pc_throughput_avg(filter_delay_fc_sptr self) → float

filter_delay_fc_sptr.pc_work_time_avg(filter_delay_fc_sptr self) → float

filter_delay_fc_sptr.pc_work_time_total(filter_delay_fc_sptr self) → float

filter_delay_fc_sptr.sample_delay(filter_delay_fc_sptr self, int which) → unsigned int

filter_delay_fc_sptr.set_min_noutput_items(filter_delay_fc_sptr self, int m)

filter_delay_fc_sptr.set_thread_priority(filter_delay_fc_sptr self, int priority) → int

filter_delay_fc_sptr.thread_priority(filter_delay_fc_sptr self) → int
```

gnuradio.filter.**filterbank\_vcvcf**(std::vector< std::vector< float, std::allocator< float >>, std::allocator< std::vector< float, std::allocator< float >> >> >> const & taps) → filterbank\_vcvcf\_sptr

Filterbank with vector of gr\_complex input, vector of gr\_complex output and float taps.

This block takes in complex vectors and outputs complex vectors of the same size. Vectors of length N, rather than N normal streams are used to reduce overhead.

Constructor Specific Documentation:

Build the filterbank.

**Parameters:** taps – (vector of vector of floats/list of list of floats) Used to populate the filters.

```
filterbank_vcvcf_sptr.active_thread_priority(filterbank_vcvcf_sptr self) → int
```

```

filterbank_vcvcf_sptr.declare_sample_delay(filterbank_vcvcf_sptr self, int which, int delay)
    declare_sample_delay(filterbank_vcvcf_sptr self, unsigned int delay)

filterbank_vcvcf_sptr.message_subscribers(filterbank_vcvcf_sptr self, swig_int_ptr which_port) →
swig_int_ptr

filterbank_vcvcf_sptr.min_noutput_items(filterbank_vcvcf_sptr self) → int

filterbank_vcvcf_sptr.pc_input_buffers_full_avg(filterbank_vcvcf_sptr self, int which) →
float
    pc_input_buffers_full_avg(filterbank_vcvcf_sptr self) -> pmt_vector_float

filterbank_vcvcf_sptr.pc_noutput_items_avg(filterbank_vcvcf_sptr self) → float

filterbank_vcvcf_sptr.pc_nproduced_avg(filterbank_vcvcf_sptr self) → float

filterbank_vcvcf_sptr.pc_output_buffers_full_avg(filterbank_vcvcf_sptr self, int which) →
float
    pc_output_buffers_full_avg(filterbank_vcvcf_sptr self) -> pmt_vector_float

filterbank_vcvcf_sptr.pc_throughput_avg(filterbank_vcvcf_sptr self) → float

filterbank_vcvcf_sptr.pc_work_time_avg(filterbank_vcvcf_sptr self) → float

filterbank_vcvcf_sptr.pc_work_time_total(filterbank_vcvcf_sptr self) → float

filterbank_vcvcf_sptr.print_taps(filterbank_vcvcf_sptr self)
    Print all of the filterbank taps to screen.

filterbank_vcvcf_sptr.sample_delay(filterbank_vcvcf_sptr self, int which) → unsigned int

filterbank_vcvcf_sptr.set_min_noutput_items(filterbank_vcvcf_sptr self, int m)

filterbank_vcvcf_sptr.set_taps(filterbank_vcvcf_sptr self, std::vector< std::vector< float,
std::allocator< float > >, std::allocator< std::vector< float, std::allocator< float > > > > const & taps)
    Resets the filterbank's filter taps with the new prototype filter

filterbank_vcvcf_sptr.set_thread_priority(filterbank_vcvcf_sptr self, int priority) → int

filterbank_vcvcf_sptr.taps(filterbank_vcvcf_sptr self) → std::vector< std::vector< float, std::allocator<
float > >, std::allocator< std::vector< float, std::allocator< float > > > > >
    Return a vector<vector<>> of the filterbank taps

filterbank_vcvcf_sptr.thread_priority(filterbank_vcvcf_sptr self) → int

```

gnuradio.filter.**fir\_filter\_ccc**(int decimation, pmt\_vector\_cfloat taps) → fir\_filter\_ccc\_sptr

FIR filter with gr\_complex input, gr\_complex output, and gr\_complex taps.

The fir\_filter\_XXX blocks create finite impulse response (FIR) filters that perform the convolution in the time domain:

The taps are a C++ vector (or Python list) of values of the type specified by the third letter in the block's suffix. For this block, the value is of type gr\_complex. Taps can be created using the firdes or optfir tools.

These versions of the filter can also act as down-samplers (or decimators) by specifying an integer value for .

Constructor Specific Documentation:

FIR filter with gr\_complex input, gr\_complex output, and gr\_complex taps.

**Parameters:**

- **decimation** – set the integer decimation rate
- **taps** – a vector/list of taps of type gr\_complex

```

fir_filter_ccc_sptr.active_thread_priority(fir_filter_ccc_sptr self) → int

fir_filter_ccc_sptr.declare_sample_delay(fir_filter_ccc_sptr self, int which, int delay)
    declare_sample_delay(fir_filter_ccc_sptr self, unsigned int delay)

fir_filter_ccc_sptr.message_subscribers(fir_filter_ccc_sptr self, swig_int_ptr which_port) →
swig_int_ptr

fir_filter_ccc_sptr.min_noutput_items(fir_filter_ccc_sptr self) → int

fir_filter_ccc_sptr.pc_input_buffers_full_avg(fir_filter_ccc_sptr self, int which) → float

```

```

    pc_input_buffers_full_avg(fir_filter_ccc_sptr self) -> pmt_vector_float

    fir_filter_ccc_sptr.pc_noutput_items_avg(fir_filter_ccc_sptr self) -> float

    fir_filter_ccc_sptr.pc_nproduced_avg(fir_filter_ccc_sptr self) -> float

    fir_filter_ccc_sptr.pc_output_buffers_full_avg(fir_filter_ccc_sptr self, int which) -> float
    pc_output_buffers_full_avg(fir_filter_ccc_sptr self) -> pmt_vector_float

    fir_filter_ccc_sptr.pc_throughput_avg(fir_filter_ccc_sptr self) -> float

    fir_filter_ccc_sptr.pc_work_time_avg(fir_filter_ccc_sptr self) -> float

    fir_filter_ccc_sptr.pc_work_time_total(fir_filter_ccc_sptr self) -> float

    fir_filter_ccc_sptr.sample_delay(fir_filter_ccc_sptr self, int which) -> unsigned int

    fir_filter_ccc_sptr.set_min_noutput_items(fir_filter_ccc_sptr self, int m)

    fir_filter_ccc_sptr.set_taps(fir_filter_ccc_sptr self, pmt_vector_cfloat taps)

    fir_filter_ccc_sptr.set_thread_priority(fir_filter_ccc_sptr self, int priority) -> int

    fir_filter_ccc_sptr.taps(fir_filter_ccc_sptr self) -> pmt_vector_cfloat

    fir_filter_ccc_sptr.thread_priority(fir_filter_ccc_sptr self) -> int

```

gnuradio.filter.**fir\_filter\_ccf**(int decimation, pmt\_vector\_float taps) -> fir\_filter\_ccf\_sptr  
 FIR filter with gr\_complex input, gr\_complex output, and float taps.

The `fir_filter_XXX` blocks create finite impulse response (FIR) filters that perform the convolution in the time domain:

The taps are a C++ vector (or Python list) of values of the type specified by the third letter in the block's suffix. For this block, the value is of type float. Taps can be created using the `firdes` or `optfir` tools.

These versions of the filter can also act as down-samplers (or decimators) by specifying an integer value for `.`

Constructor Specific Documentation:

FIR filter with gr\_complex input, gr\_complex output, and float taps.

**Parameters:**

- **decimation** – set the integer decimation rate
- **taps** – a vector/list of taps of type float

```

    fir_filter_ccf_sptr.active_thread_priority(fir_filter_ccf_sptr self) -> int

    fir_filter_ccf_sptr.declare_sample_delay(fir_filter_ccf_sptr self, int which, int delay)
    declare_sample_delay(fir_filter_ccf_sptr self, unsigned int delay)

    fir_filter_ccf_sptr.message_subscribers(fir_filter_ccf_sptr self, swig_int_ptr which_port) ->
    swig_int_ptr

    fir_filter_ccf_sptr.min_noutput_items(fir_filter_ccf_sptr self) -> int

    fir_filter_ccf_sptr.pc_input_buffers_full_avg(fir_filter_ccf_sptr self, int which) -> float
    pc_input_buffers_full_avg(fir_filter_ccf_sptr self) -> pmt_vector_float

    fir_filter_ccf_sptr.pc_noutput_items_avg(fir_filter_ccf_sptr self) -> float

    fir_filter_ccf_sptr.pc_nproduced_avg(fir_filter_ccf_sptr self) -> float

    fir_filter_ccf_sptr.pc_output_buffers_full_avg(fir_filter_ccf_sptr self, int which) -> float
    pc_output_buffers_full_avg(fir_filter_ccf_sptr self) -> pmt_vector_float

    fir_filter_ccf_sptr.pc_throughput_avg(fir_filter_ccf_sptr self) -> float

    fir_filter_ccf_sptr.pc_work_time_avg(fir_filter_ccf_sptr self) -> float

    fir_filter_ccf_sptr.pc_work_time_total(fir_filter_ccf_sptr self) -> float

    fir_filter_ccf_sptr.sample_delay(fir_filter_ccf_sptr self, int which) -> unsigned int

    fir_filter_ccf_sptr.set_min_noutput_items(fir_filter_ccf_sptr self, int m)

```

```
fir_filter_ccf_sptr.set_taps(fir_filter_ccf_sptr self, pmt_vector_float taps)
```

```
fir_filter_ccf_sptr.set_thread_priority(fir_filter_ccf_sptr self, int priority) → int
```

```
fir_filter_ccf_sptr.taps(fir_filter_ccf_sptr self) → pmt_vector_float
```

```
fir_filter_ccf_sptr.thread_priority(fir_filter_ccf_sptr self) → int
```

```
gnuradio.filter.fir_filter_fcc(int decimation, pmt_vector_cfloat taps) → fir_filter_fcc_sptr
```

FIR filter with float input, gr\_complex output, and gr\_complex taps.

The `fir_filter_XXX` blocks create finite impulse response (FIR) filters that perform the convolution in the time domain:

The `taps` are a C++ vector (or Python list) of values of the type specified by the third letter in the block's suffix. For this block, the value is of type `gr_complex`. Taps can be created using the `firdes` or `optfir` tools.

These versions of the filter can also act as down-samplers (or decimators) by specifying an integer value for `.`

Constructor Specific Documentation:

FIR filter with float input, gr\_complex output, and gr\_complex taps.

**Parameters:**

- **decimation** – set the integer decimation rate
- **taps** – a vector/list of taps of type `gr_complex`

```
fir_filter_fcc_sptr.active_thread_priority(fir_filter_fcc_sptr self) → int
```

```
fir_filter_fcc_sptr.declare_sample_delay(fir_filter_fcc_sptr self, int which, int delay)  
declare_sample_delay(fir_filter_fcc_sptr self, unsigned int delay)
```

```
fir_filter_fcc_sptr.message_subscribers(fir_filter_fcc_sptr self, swig_int_ptr which_port) →  
swig_int_ptr
```

```
fir_filter_fcc_sptr.min_noutput_items(fir_filter_fcc_sptr self) → int
```

```
fir_filter_fcc_sptr.pc_input_buffers_full_avg(fir_filter_fcc_sptr self, int which) → float  
pc_input_buffers_full_avg(fir_filter_fcc_sptr self) → pmt_vector_float
```

```
fir_filter_fcc_sptr.pc_noutput_items_avg(fir_filter_fcc_sptr self) → float
```

```
fir_filter_fcc_sptr.pc_nproduced_avg(fir_filter_fcc_sptr self) → float
```

```
fir_filter_fcc_sptr.pc_output_buffers_full_avg(fir_filter_fcc_sptr self, int which) → float  
pc_output_buffers_full_avg(fir_filter_fcc_sptr self) → pmt_vector_float
```

```
fir_filter_fcc_sptr.pc_throughput_avg(fir_filter_fcc_sptr self) → float
```

```
fir_filter_fcc_sptr.pc_work_time_avg(fir_filter_fcc_sptr self) → float
```

```
fir_filter_fcc_sptr.pc_work_time_total(fir_filter_fcc_sptr self) → float
```

```
fir_filter_fcc_sptr.sample_delay(fir_filter_fcc_sptr self, int which) → unsigned int
```

```
fir_filter_fcc_sptr.set_min_noutput_items(fir_filter_fcc_sptr self, int m)
```

```
fir_filter_fcc_sptr.set_taps(fir_filter_fcc_sptr self, pmt_vector_cfloat taps)
```

```
fir_filter_fcc_sptr.set_thread_priority(fir_filter_fcc_sptr self, int priority) → int
```

```
fir_filter_fcc_sptr.taps(fir_filter_fcc_sptr self) → pmt_vector_cfloat
```

```
fir_filter_fcc_sptr.thread_priority(fir_filter_fcc_sptr self) → int
```

```
gnuradio.filter.fir_filter_fff(int decimation, pmt_vector_float taps) → fir_filter_fff_sptr
```

FIR filter with float input, float output, and float taps.

The `fir_filter_XXX` blocks create finite impulse response (FIR) filters that perform the convolution in the time domain:

The `taps` are a C++ vector (or Python list) of values of the type specified by the third letter in the block's suffix. For this block, the value is of type `float`. Taps can be created using the `firdes` or `optfir` tools.

These versions of the filter can also act as down-samplers (or decimators) by specifying an integer value for `.`

Constructor Specific Documentation:

FIR filter with float input, float output, and float taps.

**Parameters:**

- **decimation** – set the integer decimation rate
- **taps** – a vector/list of taps of type float

```
fir_filter_fff_sptr.active_thread_priority(fir_filter_fff_sptr self) → int

fir_filter_fff_sptr.declare_sample_delay(fir_filter_fff_sptr self, int which, int delay)
    declare_sample_delay(fir_filter_fff_sptr self, unsigned int delay)

fir_filter_fff_sptr.message_subscribers(fir_filter_fff_sptr self, swig_int_ptr which_port) →
    swig_int_ptr

fir_filter_fff_sptr.min_noutput_items(fir_filter_fff_sptr self) → int

fir_filter_fff_sptr.pc_input_buffers_full_avg(fir_filter_fff_sptr self, int which) → float
    pc_input_buffers_full_avg(fir_filter_fff_sptr self) -> pmt_vector_float

fir_filter_fff_sptr.pc_noutput_items_avg(fir_filter_fff_sptr self) → float

fir_filter_fff_sptr.pc_nproduced_avg(fir_filter_fff_sptr self) → float

fir_filter_fff_sptr.pc_output_buffers_full_avg(fir_filter_fff_sptr self, int which) → float
    pc_output_buffers_full_avg(fir_filter_fff_sptr self) -> pmt_vector_float

fir_filter_fff_sptr.pc_throughput_avg(fir_filter_fff_sptr self) → float

fir_filter_fff_sptr.pc_work_time_avg(fir_filter_fff_sptr self) → float

fir_filter_fff_sptr.pc_work_time_total(fir_filter_fff_sptr self) → float

fir_filter_fff_sptr.sample_delay(fir_filter_fff_sptr self, int which) → unsigned int

fir_filter_fff_sptr.set_min_noutput_items(fir_filter_fff_sptr self, int m)

fir_filter_fff_sptr.set_taps(fir_filter_fff_sptr self, pmt_vector_float taps)

fir_filter_fff_sptr.set_thread_priority(fir_filter_fff_sptr self, int priority) → int

fir_filter_fff_sptr.taps(fir_filter_fff_sptr self) → pmt_vector_float

fir_filter_fff_sptr.thread_priority(fir_filter_fff_sptr self) → int
```

```
gnuradio.filter.fir_filter_fsf(int decimation, pmt_vector_float taps) → fir_filter_fsf_sptr
```

FIR filter with float input, short output, and float taps.

The `fir_filter_XXX` blocks create finite impulse response (FIR) filters that perform the convolution in the time domain:

The `taps` are a C++ vector (or Python list) of values of the type specified by the third letter in the block's suffix. For this block, the value is of type float. Taps can be created using the `firdes` or `optfir` tools.

These versions of the filter can also act as down-samplers (or decimators) by specifying an integer value for `.`

Constructor Specific Documentation:

FIR filter with float input, short output, and float taps.

**Parameters:**

- **decimation** – set the integer decimation rate
- **taps** – a vector/list of taps of type float

```
fir_filter_fsf_sptr.active_thread_priority(fir_filter_fsf_sptr self) → int

fir_filter_fsf_sptr.declare_sample_delay(fir_filter_fsf_sptr self, int which, int delay)
    declare_sample_delay(fir_filter_fsf_sptr self, unsigned int delay)

fir_filter_fsf_sptr.message_subscribers(fir_filter_fsf_sptr self, swig_int_ptr which_port) →
    swig_int_ptr

fir_filter_fsf_sptr.min_noutput_items(fir_filter_fsf_sptr self) → int

fir_filter_fsf_sptr.pc_input_buffers_full_avg(fir_filter_fsf_sptr self, int which) → float
    pc_input_buffers_full_avg(fir_filter_fsf_sptr self) -> pmt_vector_float
```

```

fir_filter_fsf_sptr.pc_noutput_items_avg(fir_filter_fsf_sptr self) → float
fir_filter_fsf_sptr.pc_nproduced_avg(fir_filter_fsf_sptr self) → float
fir_filter_fsf_sptr.pc_output_buffers_full_avg(fir_filter_fsf_sptr self, int which) → float
pc_output_buffers_full_avg(fir_filter_fsf_sptr self) -> pmt_vector_float
fir_filter_fsf_sptr.pc_throughput_avg(fir_filter_fsf_sptr self) → float
fir_filter_fsf_sptr.pc_work_time_avg(fir_filter_fsf_sptr self) → float
fir_filter_fsf_sptr.pc_work_time_total(fir_filter_fsf_sptr self) → float
fir_filter_fsf_sptr.sample_delay(fir_filter_fsf_sptr self, int which) → unsigned int
fir_filter_fsf_sptr.set_min_noutput_items(fir_filter_fsf_sptr self, int m)
fir_filter_fsf_sptr.set_taps(fir_filter_fsf_sptr self, pmt_vector_float taps)
fir_filter_fsf_sptr.set_thread_priority(fir_filter_fsf_sptr self, int priority) → int
fir_filter_fsf_sptr.taps(fir_filter_fsf_sptr self) → pmt_vector_float
fir_filter_fsf_sptr.thread_priority(fir_filter_fsf_sptr self) → int

```

gnuradio.filter.**fir\_filter\_scc**(int decimation, pmt\_vector\_cfloat taps) → fir\_filter\_scc\_sptr

FIR filter with short input, gr\_complex output, and gr\_complex taps.

The fir\_filter\_XXX blocks create finite impulse response (FIR) filters that perform the convolution in the time domain:

The taps are a C++ vector (or Python list) of values of the type specified by the third letter in the block's suffix. For this block, the value is of type gr\_complex. Taps can be created using the firdes or optfir tools.

These versions of the filter can also act as down-samplers (or decimators) by specifying an integer value for .

Constructor Specific Documentation:

FIR filter with short input, gr\_complex output, and gr\_complex taps.

**Parameters:**

- **decimation** – set the integer decimation rate
- **taps** – a vector/list of taps of type gr\_complex

```

fir_filter_scc_sptr.active_thread_priority(fir_filter_scc_sptr self) → int
fir_filter_scc_sptr.declare_sample_delay(fir_filter_scc_sptr self, int which, int delay)
declare_sample_delay(fir_filter_scc_sptr self, unsigned int delay)
fir_filter_scc_sptr.message_subscribers(fir_filter_scc_sptr self, swig_int_ptr which_port) →
swig_int_ptr
fir_filter_scc_sptr.min_noutput_items(fir_filter_scc_sptr self) → int
fir_filter_scc_sptr.pc_input_buffers_full_avg(fir_filter_scc_sptr self, int which) → float
pc_input_buffers_full_avg(fir_filter_scc_sptr self) -> pmt_vector_float
fir_filter_scc_sptr.pc_noutput_items_avg(fir_filter_scc_sptr self) → float
fir_filter_scc_sptr.pc_nproduced_avg(fir_filter_scc_sptr self) → float
fir_filter_scc_sptr.pc_output_buffers_full_avg(fir_filter_scc_sptr self, int which) → float
pc_output_buffers_full_avg(fir_filter_scc_sptr self) -> pmt_vector_float
fir_filter_scc_sptr.pc_throughput_avg(fir_filter_scc_sptr self) → float
fir_filter_scc_sptr.pc_work_time_avg(fir_filter_scc_sptr self) → float
fir_filter_scc_sptr.pc_work_time_total(fir_filter_scc_sptr self) → float
fir_filter_scc_sptr.sample_delay(fir_filter_scc_sptr self, int which) → unsigned int
fir_filter_scc_sptr.set_min_noutput_items(fir_filter_scc_sptr self, int m)
fir_filter_scc_sptr.set_taps(fir_filter_scc_sptr self, pmt_vector_cfloat taps)

```

```
fir_filter_scc_sptr.set_thread_priority(fir_filter_scc_sptr self, int priority) → int
```

```
fir_filter_scc_sptr.taps(fir_filter_scc_sptr self) → pmt_vector_cfloat
```

```
fir_filter_scc_sptr.thread_priority(fir_filter_scc_sptr self) → int
```

```
gnuradio.filter.fractional_interpolator_cc(float phase_shift, float interp_ratio) →  
fractional_interpolator_cc_sptr
```

Interpolating MMSE filter with complex input, complex output.

Constructor Specific Documentation:

Build the interpolating MMSE filter (complex input, complex output)

**Parameters:**

- **phase\_shift** – The phase shift of the output signal to the input
- **interp\_ratio** – The interpolation ratio = input\_rate / output\_rate.

```
fractional_interpolator_cc_sptr.active_thread_priority(fractional_interpolator_cc_sptr self) →  
int
```

```
fractional_interpolator_cc_sptr.declare_sample_delay(fractional_interpolator_cc_sptr self, int  
which, int delay)
```

```
declare_sample_delay(fractional_interpolator_cc_sptr self, unsigned int delay)
```

```
fractional_interpolator_cc_sptr.interp_ratio(fractional_interpolator_cc_sptr self) → float
```

```
fractional_interpolator_cc_sptr.message_subscribers(fractional_interpolator_cc_sptr self,  
swig_int_ptr which_port) → swig_int_ptr
```

```
fractional_interpolator_cc_sptr.min_noutput_items(fractional_interpolator_cc_sptr self) → int
```

```
fractional_interpolator_cc_sptr.mu(fractional_interpolator_cc_sptr self) → float
```

```
fractional_interpolator_cc_sptr.pc_input_buffers_full_avg(fractional_interpolator_cc_sptr  
self, int which) → float
```

```
pc_input_buffers_full_avg(fractional_interpolator_cc_sptr self) → pmt_vector_float
```

```
fractional_interpolator_cc_sptr.pc_noutput_items_avg(fractional_interpolator_cc_sptr self) →  
float
```

```
fractional_interpolator_cc_sptr.pc_nproduced_avg(fractional_interpolator_cc_sptr self) → float
```

```
fractional_interpolator_cc_sptr.pc_output_buffers_full_avg(fractional_interpolator_cc_sptr  
self, int which) → float
```

```
pc_output_buffers_full_avg(fractional_interpolator_cc_sptr self) → pmt_vector_float
```

```
fractional_interpolator_cc_sptr.pc_throughput_avg(fractional_interpolator_cc_sptr self) → float
```

```
fractional_interpolator_cc_sptr.pc_work_time_avg(fractional_interpolator_cc_sptr self) → float
```

```
fractional_interpolator_cc_sptr.pc_work_time_total(fractional_interpolator_cc_sptr self) →  
float
```

```
fractional_interpolator_cc_sptr.sample_delay(fractional_interpolator_cc_sptr self, int which) →  
unsigned int
```

```
fractional_interpolator_cc_sptr.set_interp_ratio(fractional_interpolator_cc_sptr self, float  
interp_ratio)
```

```
fractional_interpolator_cc_sptr.set_min_noutput_items(fractional_interpolator_cc_sptr self, int  
m)
```

```
fractional_interpolator_cc_sptr.set_mu(fractional_interpolator_cc_sptr self, float mu)
```

```
fractional_interpolator_cc_sptr.set_thread_priority(fractional_interpolator_cc_sptr self, int  
priority) → int
```

```
fractional_interpolator_cc_sptr.thread_priority(fractional_interpolator_cc_sptr self) → int
```

```
gnuradio.filter.fractional_interpolator_ff(float phase_shift, float interp_ratio) →  
fractional_interpolator_ff_sptr
```

Interpolating MMSE filter with float input, float output.

Constructor Specific Documentation:

Build the interpolating MMSE filter (float input, float output)

**Parameters:**

- **phase\_shift** – The phase shift of the output signal to the input
- **interp\_ratio** – The interpolation ratio = input\_rate / output\_rate.

```
fractional_interpolator_ff_sptr.active_thread_priority(fractional_interpolator_ff_sptr self) → int

fractional_interpolator_ff_sptr.declare_sample_delay(fractional_interpolator_ff_sptr self, int which, int delay)
    declare_sample_delay(fractional_interpolator_ff_sptr self, unsigned int delay)

fractional_interpolator_ff_sptr.interp_ratio(fractional_interpolator_ff_sptr self) → float

fractional_interpolator_ff_sptr.message_subscribers(fractional_interpolator_ff_sptr self, swig_int_ptr which_port) → swig_int_ptr

fractional_interpolator_ff_sptr.min_noutput_items(fractional_interpolator_ff_sptr self) → int

fractional_interpolator_ff_sptr.mu(fractional_interpolator_ff_sptr self) → float

fractional_interpolator_ff_sptr.pc_input_buffers_full_avg(fractional_interpolator_ff_sptr self, int which) → float
    pc_input_buffers_full_avg(fractional_interpolator_ff_sptr self) -> pmt_vector_float

fractional_interpolator_ff_sptr.pc_noutput_items_avg(fractional_interpolator_ff_sptr self) → float

fractional_interpolator_ff_sptr.pc_nproduced_avg(fractional_interpolator_ff_sptr self) → float

fractional_interpolator_ff_sptr.pc_output_buffers_full_avg(fractional_interpolator_ff_sptr self, int which) → float
    pc_output_buffers_full_avg(fractional_interpolator_ff_sptr self) -> pmt_vector_float

fractional_interpolator_ff_sptr.pc_throughput_avg(fractional_interpolator_ff_sptr self) → float

fractional_interpolator_ff_sptr.pc_work_time_avg(fractional_interpolator_ff_sptr self) → float

fractional_interpolator_ff_sptr.pc_work_time_total(fractional_interpolator_ff_sptr self) → float

fractional_interpolator_ff_sptr.sample_delay(fractional_interpolator_ff_sptr self, int which) → unsigned int

fractional_interpolator_ff_sptr.set_interp_ratio(fractional_interpolator_ff_sptr self, float interp_ratio)

fractional_interpolator_ff_sptr.set_min_noutput_items(fractional_interpolator_ff_sptr self, int m)

fractional_interpolator_ff_sptr.set_mu(fractional_interpolator_ff_sptr self, float mu)

fractional_interpolator_ff_sptr.set_thread_priority(fractional_interpolator_ff_sptr self, int priority) → int

fractional_interpolator_ff_sptr.thread_priority(fractional_interpolator_ff_sptr self) → int
```

```
gnuradio.filter.fractional_resampler_cc(float phase_shift, float resamp_ratio) → fractional_resampler_cc_sptr
```

resampling MMSE filter with complex input, complex output

The resampling ratio and mu parameters can be set with a pmt dict message. Keys are pmt symbols with the strings “resamp\_ratio” and “mu” and values are pmt floats.

Constructor Specific Documentation:

Build the resampling MMSE filter (complex input, complex output)

**Parameters:**

- **phase\_shift** – The phase shift of the output signal to the input
- **resamp\_ratio** – The resampling ratio = input\_rate / output\_rate.

```
fractional_resampler_cc_sptr.active_thread_priority(fractional_resampler_cc_sptr self) → int

fractional_resampler_cc_sptr.declare_sample_delay(fractional_resampler_cc_sptr self, int which, int delay)
```

```

        declare_sample_delay(fractional_resampler_cc_sptr self, unsigned int delay)

fractional_resampler_cc_sptr.message_subscribers(fractional_resampler_cc_sptr self,
swig_int_ptr which_port) → swig_int_ptr

fractional_resampler_cc_sptr.min_noutput_items(fractional_resampler_cc_sptr self) → int

fractional_resampler_cc_sptr.mu(fractional_resampler_cc_sptr self) → float

fractional_resampler_cc_sptr.pc_input_buffers_full_avg(fractional_resampler_cc_sptr self, int
which) → float
    pc_input_buffers_full_avg(fractional_resampler_cc_sptr self) -> pmt_vector_float

fractional_resampler_cc_sptr.pc_noutput_items_avg(fractional_resampler_cc_sptr self) → float

fractional_resampler_cc_sptr.pc_nproduced_avg(fractional_resampler_cc_sptr self) → float

fractional_resampler_cc_sptr.pc_output_buffers_full_avg(fractional_resampler_cc_sptr self,
int which) → float
    pc_output_buffers_full_avg(fractional_resampler_cc_sptr self) -> pmt_vector_float

fractional_resampler_cc_sptr.pc_throughput_avg(fractional_resampler_cc_sptr self) → float

fractional_resampler_cc_sptr.pc_work_time_avg(fractional_resampler_cc_sptr self) → float

fractional_resampler_cc_sptr.pc_work_time_total(fractional_resampler_cc_sptr self) → float

fractional_resampler_cc_sptr.resamp_ratio(fractional_resampler_cc_sptr self) → float

fractional_resampler_cc_sptr.sample_delay(fractional_resampler_cc_sptr self, int which) →
unsigned int

fractional_resampler_cc_sptr.set_min_noutput_items(fractional_resampler_cc_sptr self, int m)

fractional_resampler_cc_sptr.set_mu(fractional_resampler_cc_sptr self, float mu)

fractional_resampler_cc_sptr.set_resamp_ratio(fractional_resampler_cc_sptr self, float
resamp_ratio)

fractional_resampler_cc_sptr.set_thread_priority(fractional_resampler_cc_sptr self, int priority)
→ int

fractional_resampler_cc_sptr.thread_priority(fractional_resampler_cc_sptr self) → int

```

gnuradio.filter.**fractional\_resampler\_ff**(float phase\_shift, float resamp\_ratio) →  
fractional\_resampler\_ff\_sptr

Resampling MMSE filter with float input, float output.

The resampling ratio and mu parameters can be set with a pmt dict message. Keys are pmt symbols with the strings “resamp\_ratio” and “mu” and values are pmt floats.

Constructor Specific Documentation:

Build the resampling MMSE filter (float input, float output)

**Parameters:**

- **phase\_shift** – The phase shift of the output signal to the input
- **resamp\_ratio** – The resampling ratio = input\_rate / output\_rate.

```

fractional_resampler_ff_sptr.active_thread_priority(fractional_resampler_ff_sptr self) → int

fractional_resampler_ff_sptr.declare_sample_delay(fractional_resampler_ff_sptr self, int which,
int delay)
    declare_sample_delay(fractional_resampler_ff_sptr self, unsigned int delay)

fractional_resampler_ff_sptr.message_subscribers(fractional_resampler_ff_sptr self, swig_int_ptr
which_port) → swig_int_ptr

fractional_resampler_ff_sptr.min_noutput_items(fractional_resampler_ff_sptr self) → int

fractional_resampler_ff_sptr.mu(fractional_resampler_ff_sptr self) → float

fractional_resampler_ff_sptr.pc_input_buffers_full_avg(fractional_resampler_ff_sptr self, int
which) → float
    pc_input_buffers_full_avg(fractional_resampler_ff_sptr self) -> pmt_vector_float

```

```

fractional_resampler_ff_sptr.pc_noutput_items_avg(fractional_resampler_ff_sptr self) → float
fractional_resampler_ff_sptr.pc_nproduced_avg(fractional_resampler_ff_sptr self) → float
fractional_resampler_ff_sptr.pc_output_buffers_full_avg(fractional_resampler_ff_sptr self, int which) → float
    pc_output_buffers_full_avg(fractional_resampler_ff_sptr self) -> pmt_vector_float
fractional_resampler_ff_sptr.pc_throughput_avg(fractional_resampler_ff_sptr self) → float
fractional_resampler_ff_sptr.pc_work_time_avg(fractional_resampler_ff_sptr self) → float
fractional_resampler_ff_sptr.pc_work_time_total(fractional_resampler_ff_sptr self) → float
fractional_resampler_ff_sptr.resamp_ratio(fractional_resampler_ff_sptr self) → float
fractional_resampler_ff_sptr.sample_delay(fractional_resampler_ff_sptr self, int which) → unsigned int
fractional_resampler_ff_sptr.set_min_noutput_items(fractional_resampler_ff_sptr self, int m)
fractional_resampler_ff_sptr.set_mu(fractional_resampler_ff_sptr self, float mu)
fractional_resampler_ff_sptr.set_resamp_ratio(fractional_resampler_ff_sptr self, float resamp_ratio)
fractional_resampler_ff_sptr.set_thread_priority(fractional_resampler_ff_sptr self, int priority) → int
fractional_resampler_ff_sptr.thread_priority(fractional_resampler_ff_sptr self) → int

```

gnuradio.filter.**freq\_xlating\_fir\_filter\_ccc**(int decimation, pmt\_vector\_cfloat taps, double center\_freq, double sampling\_freq) → freq\_xlating\_fir\_filter\_ccc\_sptr

FIR filter combined with frequency translation with gr\_complex input, gr\_complex output and gr\_complex taps.

This class efficiently combines a frequency translation (typically “down conversion”) with a FIR filter (typically low-pass) and decimation. It is ideally suited for a “channel selection filter” and can be efficiently used to select and decimate a narrow band signal out of wide bandwidth input.

Uses a single input array to produce a single output array. Additional inputs and/or outputs are ignored.

Constructor Specific Documentation:

FIR filter with gr\_complex input, gr\_complex output, and gr\_complex taps that also frequency translates a signal from .

Construct a FIR filter with the given taps and a composite frequency translation that shifts center\_freq down to zero Hz. The frequency translation logically comes before the filtering operation.

**Parameters:**

- **decimation** – set the integer decimation rate
- **taps** – a vector/list of taps of type gr\_complex
- **center\_freq** – Center frequency of signal to down convert from (Hz)
- **sampling\_freq** – Sampling rate of signal (in Hz)

```

freq_xlating_fir_filter_ccc_sptr.active_thread_priority(freq_xlating_fir_filter_ccc_sptr self) → int
freq_xlating_fir_filter_ccc_sptr.center_freq(freq_xlating_fir_filter_ccc_sptr self) → double
freq_xlating_fir_filter_ccc_sptr.declare_sample_delay(freq_xlating_fir_filter_ccc_sptr self, int which, int delay)
    declare_sample_delay(freq_xlating_fir_filter_ccc_sptr self, unsigned int delay)
freq_xlating_fir_filter_ccc_sptr.message_subscribers(freq_xlating_fir_filter_ccc_sptr self, swig_int_ptr which_port) → swig_int_ptr
freq_xlating_fir_filter_ccc_sptr.min_noutput_items(freq_xlating_fir_filter_ccc_sptr self) → int
freq_xlating_fir_filter_ccc_sptr.pc_input_buffers_full_avg(freq_xlating_fir_filter_ccc_sptr self, int which) → float
    pc_input_buffers_full_avg(freq_xlating_fir_filter_ccc_sptr self) -> pmt_vector_float
freq_xlating_fir_filter_ccc_sptr.pc_noutput_items_avg(freq_xlating_fir_filter_ccc_sptr self) → float

```

```

freq_xlating_fir_filter_ccc_sptr.pc_nproduced_avg(freq_xlating_fir_filter_ccc_sptr self) → float

freq_xlating_fir_filter_ccc_sptr.pc_output_buffers_full_avg(freq_xlating_fir_filter_ccc_sptr
self, int which) → float
    pc_output_buffers_full_avg(freq_xlating_fir_filter_ccc_sptr self) -> pmt_vector_float

freq_xlating_fir_filter_ccc_sptr.pc_throughput_avg(freq_xlating_fir_filter_ccc_sptr self) → float

freq_xlating_fir_filter_ccc_sptr.pc_work_time_avg(freq_xlating_fir_filter_ccc_sptr self) → float

freq_xlating_fir_filter_ccc_sptr.pc_work_time_total(freq_xlating_fir_filter_ccc_sptr self) →
float

freq_xlating_fir_filter_ccc_sptr.sample_delay(freq_xlating_fir_filter_ccc_sptr self, int which) →
unsigned int

freq_xlating_fir_filter_ccc_sptr.set_center_freq(freq_xlating_fir_filter_ccc_sptr self, double
center_freq)

freq_xlating_fir_filter_ccc_sptr.set_min_noutput_items(freq_xlating_fir_filter_ccc_sptr self, int
m)

freq_xlating_fir_filter_ccc_sptr.set_taps(freq_xlating_fir_filter_ccc_sptr self, pmt_vector_cfloat
taps)

freq_xlating_fir_filter_ccc_sptr.set_thread_priority(freq_xlating_fir_filter_ccc_sptr self, int
priority) → int

freq_xlating_fir_filter_ccc_sptr.taps(freq_xlating_fir_filter_ccc_sptr self) → pmt_vector_cfloat

freq_xlating_fir_filter_ccc_sptr.thread_priority(freq_xlating_fir_filter_ccc_sptr self) → int

```

gnuradio.filter.**freq\_xlating\_fir\_filter\_ccf**(int decimation, pmt\_vector\_float taps, double
center\_freq, double sampling\_freq) → freq\_xlating\_fir\_filter\_ccf\_sptr

FIR filter combined with frequency translation with gr\_complex input, gr\_complex output and float taps.

This class efficiently combines a frequency translation (typically “down conversion”) with a FIR filter (typically low-pass) and decimation. It is ideally suited for a “channel selection filter” and can be efficiently used to select and decimate a narrow band signal out of wide bandwidth input.

Uses a single input array to produce a single output array. Additional inputs and/or outputs are ignored.

Constructor Specific Documentation:

FIR filter with gr\_complex input, gr\_complex output, and float taps that also frequency translates a signal from .

Construct a FIR filter with the given taps and a composite frequency translation that shifts center\_freq down to zero Hz. The frequency translation logically comes before the filtering operation.

- Parameters:**
- **decimation** – set the integer decimation rate
  - **taps** – a vector/list of taps of type float
  - **center\_freq** – Center frequency of signal to down convert from (Hz)
  - **sampling\_freq** – Sampling rate of signal (in Hz)

```

freq_xlating_fir_filter_ccf_sptr.active_thread_priority(freq_xlating_fir_filter_ccf_sptr self) →
int

```

```

freq_xlating_fir_filter_ccf_sptr.center_freq(freq_xlating_fir_filter_ccf_sptr self) → double

```

```

freq_xlating_fir_filter_ccf_sptr.declare_sample_delay(freq_xlating_fir_filter_ccf_sptr self, int
which, int delay)
    declare_sample_delay(freq_xlating_fir_filter_ccf_sptr self, unsigned int delay)

```

```

freq_xlating_fir_filter_ccf_sptr.message_subscribers(freq_xlating_fir_filter_ccf_sptr self,
swig_int_ptr which_port) → swig_int_ptr

```

```

freq_xlating_fir_filter_ccf_sptr.min_noutput_items(freq_xlating_fir_filter_ccf_sptr self) → int

```

```

freq_xlating_fir_filter_ccf_sptr.pc_input_buffers_full_avg(freq_xlating_fir_filter_ccf_sptr
self, int which) → float
    pc_input_buffers_full_avg(freq_xlating_fir_filter_ccf_sptr self) -> pmt_vector_float

```

```

freq_xlating_fir_filter_ccf_sptr.pc_noutput_items_avg(freq_xlating_fir_filter_ccf_sptr self) →

```

float

freq\_xlating\_fir\_filter\_ccf\_sptr.**pc\_nproduced\_avg**(freq\_xlating\_fir\_filter\_ccf\_sptr self) → float

freq\_xlating\_fir\_filter\_ccf\_sptr.**pc\_output\_buffers\_full\_avg**(freq\_xlating\_fir\_filter\_ccf\_sptr self, int which) → float

pc\_output\_buffers\_full\_avg(freq\_xlating\_fir\_filter\_ccf\_sptr self) -> pmt\_vector\_float

freq\_xlating\_fir\_filter\_ccf\_sptr.**pc\_throughput\_avg**(freq\_xlating\_fir\_filter\_ccf\_sptr self) → float

freq\_xlating\_fir\_filter\_ccf\_sptr.**pc\_work\_time\_avg**(freq\_xlating\_fir\_filter\_ccf\_sptr self) → float

freq\_xlating\_fir\_filter\_ccf\_sptr.**pc\_work\_time\_total**(freq\_xlating\_fir\_filter\_ccf\_sptr self) → float

freq\_xlating\_fir\_filter\_ccf\_sptr.**sample\_delay**(freq\_xlating\_fir\_filter\_ccf\_sptr self, int which) → unsigned int

freq\_xlating\_fir\_filter\_ccf\_sptr.**set\_center\_freq**(freq\_xlating\_fir\_filter\_ccf\_sptr self, double center\_freq)

freq\_xlating\_fir\_filter\_ccf\_sptr.**set\_min\_noutput\_items**(freq\_xlating\_fir\_filter\_ccf\_sptr self, int m)

freq\_xlating\_fir\_filter\_ccf\_sptr.**set\_taps**(freq\_xlating\_fir\_filter\_ccf\_sptr self, pmt\_vector\_float taps)

freq\_xlating\_fir\_filter\_ccf\_sptr.**set\_thread\_priority**(freq\_xlating\_fir\_filter\_ccf\_sptr self, int priority) → int

freq\_xlating\_fir\_filter\_ccf\_sptr.**taps**(freq\_xlating\_fir\_filter\_ccf\_sptr self) → pmt\_vector\_float

freq\_xlating\_fir\_filter\_ccf\_sptr.**thread\_priority**(freq\_xlating\_fir\_filter\_ccf\_sptr self) → int

gnuradio.filter.**freq\_xlating\_fir\_filter\_fcc**(int decimation, pmt\_vector\_cfloat taps, double center\_freq, double sampling\_freq) → freq\_xlating\_fir\_filter\_fcc\_sptr

FIR filter combined with frequency translation with float input, gr\_complex output and gr\_complex taps.

This class efficiently combines a frequency translation (typically “down conversion”) with a FIR filter (typically low-pass) and decimation. It is ideally suited for a “channel selection filter” and can be efficiently used to select and decimate a narrow band signal out of wide bandwidth input.

Uses a single input array to produce a single output array. Additional inputs and/or outputs are ignored.

Constructor Specific Documentation:

FIR filter with float input, gr\_complex output, and gr\_complex taps that also frequency translates a signal from .

Construct a FIR filter with the given taps and a composite frequency translation that shifts center\_freq down to zero Hz. The frequency translation logically comes before the filtering operation.

- Parameters:**
- **decimation** – set the integer decimation rate
  - **taps** – a vector/list of taps of type gr\_complex
  - **center\_freq** – Center frequency of signal to down convert from (Hz)
  - **sampling\_freq** – Sampling rate of signal (in Hz)

freq\_xlating\_fir\_filter\_fcc\_sptr.**active\_thread\_priority**(freq\_xlating\_fir\_filter\_fcc\_sptr self) → int

freq\_xlating\_fir\_filter\_fcc\_sptr.**center\_freq**(freq\_xlating\_fir\_filter\_fcc\_sptr self) → double

freq\_xlating\_fir\_filter\_fcc\_sptr.**declare\_sample\_delay**(freq\_xlating\_fir\_filter\_fcc\_sptr self, int which, int delay)

declare\_sample\_delay(freq\_xlating\_fir\_filter\_fcc\_sptr self, unsigned int delay)

freq\_xlating\_fir\_filter\_fcc\_sptr.**message\_subscribers**(freq\_xlating\_fir\_filter\_fcc\_sptr self, swig\_int\_ptr which\_port) → swig\_int\_ptr

freq\_xlating\_fir\_filter\_fcc\_sptr.**min\_noutput\_items**(freq\_xlating\_fir\_filter\_fcc\_sptr self) → int

freq\_xlating\_fir\_filter\_fcc\_sptr.**pc\_input\_buffers\_full\_avg**(freq\_xlating\_fir\_filter\_fcc\_sptr self, int which) → float

pc\_input\_buffers\_full\_avg(freq\_xlating\_fir\_filter\_fcc\_sptr self) -> pmt\_vector\_float

`freq_xlating_fir_filter_fcc_sptr.pc_noutput_items_avg(freq_xlating_fir_filter_fcc_sptr self) → float`

`freq_xlating_fir_filter_fcc_sptr.pc_nproduced_avg(freq_xlating_fir_filter_fcc_sptr self) → float`

`freq_xlating_fir_filter_fcc_sptr.pc_output_buffers_full_avg(freq_xlating_fir_filter_fcc_sptr self, int which) → float`

`pc_output_buffers_full_avg(freq_xlating_fir_filter_fcc_sptr self) -> pmt_vector_float`

`freq_xlating_fir_filter_fcc_sptr.pc_throughput_avg(freq_xlating_fir_filter_fcc_sptr self) → float`

`freq_xlating_fir_filter_fcc_sptr.pc_work_time_avg(freq_xlating_fir_filter_fcc_sptr self) → float`

`freq_xlating_fir_filter_fcc_sptr.pc_work_time_total(freq_xlating_fir_filter_fcc_sptr self) → float`

`freq_xlating_fir_filter_fcc_sptr.sample_delay(freq_xlating_fir_filter_fcc_sptr self, int which) → unsigned int`

`freq_xlating_fir_filter_fcc_sptr.set_center_freq(freq_xlating_fir_filter_fcc_sptr self, double center_freq)`

`freq_xlating_fir_filter_fcc_sptr.set_min_noutput_items(freq_xlating_fir_filter_fcc_sptr self, int m)`

`freq_xlating_fir_filter_fcc_sptr.set_taps(freq_xlating_fir_filter_fcc_sptr self, pmt_vector_cfloat taps)`

`freq_xlating_fir_filter_fcc_sptr.set_thread_priority(freq_xlating_fir_filter_fcc_sptr self, int priority) → int`

`freq_xlating_fir_filter_fcc_sptr.taps(freq_xlating_fir_filter_fcc_sptr self) → pmt_vector_cfloat`

`freq_xlating_fir_filter_fcc_sptr.thread_priority(freq_xlating_fir_filter_fcc_sptr self) → int`

`gnuradio.filter.freq_xlating_fir_filter_fcf(int decimation, pmt_vector_float taps, double center_freq, double sampling_freq) → freq_xlating_fir_filter_fcf_sptr`

FIR filter combined with frequency translation with float input, gr\_complex output and float taps.

This class efficiently combines a frequency translation (typically “down conversion”) with a FIR filter (typically low-pass) and decimation. It is ideally suited for a “channel selection filter” and can be efficiently used to select and decimate a narrow band signal out of wide bandwidth input.

Uses a single input array to produce a single output array. Additional inputs and/or outputs are ignored.

Constructor Specific Documentation:

FIR filter with float input, gr\_complex output, and float taps that also frequency translates a signal from .

Construct a FIR filter with the given taps and a composite frequency translation that shifts center\_freq down to zero Hz. The frequency translation logically comes before the filtering operation.

- Parameters:**
- **decimation** – set the integer decimation rate
  - **taps** – a vector/list of taps of type float
  - **center\_freq** – Center frequency of signal to down convert from (Hz)
  - **sampling\_freq** – Sampling rate of signal (in Hz)

`freq_xlating_fir_filter_fcf_sptr.active_thread_priority(freq_xlating_fir_filter_fcf_sptr self) → int`

`freq_xlating_fir_filter_fcf_sptr.center_freq(freq_xlating_fir_filter_fcf_sptr self) → double`

`freq_xlating_fir_filter_fcf_sptr.declare_sample_delay(freq_xlating_fir_filter_fcf_sptr self, int which, int delay)`

`declare_sample_delay(freq_xlating_fir_filter_fcf_sptr self, unsigned int delay)`

`freq_xlating_fir_filter_fcf_sptr.message_subscribers(freq_xlating_fir_filter_fcf_sptr self, swig_int_ptr which_port) → swig_int_ptr`

`freq_xlating_fir_filter_fcf_sptr.min_noutput_items(freq_xlating_fir_filter_fcf_sptr self) → int`

`freq_xlating_fir_filter_fcf_sptr.pc_input_buffers_full_avg(freq_xlating_fir_filter_fcf_sptr self, int which) → float`

`pc_input_buffers_full_avg(freq_xlating_fir_filter_fcf_sptr self) -> pmt_vector_float`

```

freq_xlating_fir_filter_fcf_sptr.pc_noutput_items_avg(freq_xlating_fir_filter_fcf_sptr self) → float

freq_xlating_fir_filter_fcf_sptr.pc_nproduced_avg(freq_xlating_fir_filter_fcf_sptr self) → float

freq_xlating_fir_filter_fcf_sptr.pc_output_buffers_full_avg(freq_xlating_fir_filter_fcf_sptr self, int which) → float
    pc_output_buffers_full_avg(freq_xlating_fir_filter_fcf_sptr self) -> pmt_vector_float

freq_xlating_fir_filter_fcf_sptr.pc_throughput_avg(freq_xlating_fir_filter_fcf_sptr self) → float

freq_xlating_fir_filter_fcf_sptr.pc_work_time_avg(freq_xlating_fir_filter_fcf_sptr self) → float

freq_xlating_fir_filter_fcf_sptr.pc_work_time_total(freq_xlating_fir_filter_fcf_sptr self) → float

freq_xlating_fir_filter_fcf_sptr.sample_delay(freq_xlating_fir_filter_fcf_sptr self, int which) → unsigned int

freq_xlating_fir_filter_fcf_sptr.set_center_freq(freq_xlating_fir_filter_fcf_sptr self, double center_freq)

freq_xlating_fir_filter_fcf_sptr.set_min_noutput_items(freq_xlating_fir_filter_fcf_sptr self, int m)

freq_xlating_fir_filter_fcf_sptr.set_taps(freq_xlating_fir_filter_fcf_sptr self, pmt_vector_float taps)

freq_xlating_fir_filter_fcf_sptr.set_thread_priority(freq_xlating_fir_filter_fcf_sptr self, int priority) → int

freq_xlating_fir_filter_fcf_sptr.taps(freq_xlating_fir_filter_fcf_sptr self) → pmt_vector_float

freq_xlating_fir_filter_fcf_sptr.thread_priority(freq_xlating_fir_filter_fcf_sptr self) → int

```

gnuradio.filter.**freq\_xlating\_fir\_filter\_scc**(int decimation, pmt\_vector\_cfloat taps, double center\_freq, double sampling\_freq) → freq\_xlating\_fir\_filter\_scc\_sptr

FIR filter combined with frequency translation with short input, gr\_complex output and gr\_complex taps.

This class efficiently combines a frequency translation (typically “down conversion”) with a FIR filter (typically low-pass) and decimation. It is ideally suited for a “channel selection filter” and can be efficiently used to select and decimate a narrow band signal out of wide bandwidth input.

Uses a single input array to produce a single output array. Additional inputs and/or outputs are ignored.

Constructor Specific Documentation:

FIR filter with short input, gr\_complex output, and gr\_complex taps that also frequency translates a signal from .

Construct a FIR filter with the given taps and a composite frequency translation that shifts center\_freq down to zero Hz. The frequency translation logically comes before the filtering operation.

- Parameters:**
- **decimation** – set the integer decimation rate
  - **taps** – a vector/list of taps of type gr\_complex
  - **center\_freq** – Center frequency of signal to down convert from (Hz)
  - **sampling\_freq** – Sampling rate of signal (in Hz)

```

freq_xlating_fir_filter_scc_sptr.active_thread_priority(freq_xlating_fir_filter_scc_sptr self) → int

freq_xlating_fir_filter_scc_sptr.center_freq(freq_xlating_fir_filter_scc_sptr self) → double

freq_xlating_fir_filter_scc_sptr.declare_sample_delay(freq_xlating_fir_filter_scc_sptr self, int which, int delay)
    declare_sample_delay(freq_xlating_fir_filter_scc_sptr self, unsigned int delay)

freq_xlating_fir_filter_scc_sptr.message_subscribers(freq_xlating_fir_filter_scc_sptr self, swig_int_ptr which_port) → swig_int_ptr

freq_xlating_fir_filter_scc_sptr.min_noutput_items(freq_xlating_fir_filter_scc_sptr self) → int

freq_xlating_fir_filter_scc_sptr.pc_input_buffers_full_avg(freq_xlating_fir_filter_scc_sptr self, int which) → float

```

```

    pc_input_buffers_full_avg(freq_xlating_fir_filter_scc_sptr self) -> pmt_vector_float

freq_xlating_fir_filter_scc_sptr.pc_noutput_items_avg(freq_xlating_fir_filter_scc_sptr self) →
float

freq_xlating_fir_filter_scc_sptr.pc_nproduced_avg(freq_xlating_fir_filter_scc_sptr self) → float

freq_xlating_fir_filter_scc_sptr.pc_output_buffers_full_avg(freq_xlating_fir_filter_scc_sptr
self, int which) → float
    pc_output_buffers_full_avg(freq_xlating_fir_filter_scc_sptr self) -> pmt_vector_float

freq_xlating_fir_filter_scc_sptr.pc_throughput_avg(freq_xlating_fir_filter_scc_sptr self) → float

freq_xlating_fir_filter_scc_sptr.pc_work_time_avg(freq_xlating_fir_filter_scc_sptr self) → float

freq_xlating_fir_filter_scc_sptr.pc_work_time_total(freq_xlating_fir_filter_scc_sptr self) →
float

freq_xlating_fir_filter_scc_sptr.sample_delay(freq_xlating_fir_filter_scc_sptr self, int which) →
unsigned int

freq_xlating_fir_filter_scc_sptr.set_center_freq(freq_xlating_fir_filter_scc_sptr self, double
center_freq)

freq_xlating_fir_filter_scc_sptr.set_min_noutput_items(freq_xlating_fir_filter_scc_sptr self, int
m)

freq_xlating_fir_filter_scc_sptr.set_taps(freq_xlating_fir_filter_scc_sptr self, pmt_vector_cfloat
taps)

freq_xlating_fir_filter_scc_sptr.set_thread_priority(freq_xlating_fir_filter_scc_sptr self, int
priority) → int

freq_xlating_fir_filter_scc_sptr.taps(freq_xlating_fir_filter_scc_sptr self) → pmt_vector_cfloat

freq_xlating_fir_filter_scc_sptr.thread_priority(freq_xlating_fir_filter_scc_sptr self) → int

gnuradio.filter.freq_xlating_fir_filter_scf(int decimation, pmt_vector_float taps, double
center_freq, double sampling_freq) → freq_xlating_fir_filter_scf_sptr

```

FIR filter combined with frequency translation with short input, gr\_complex output and float taps.

This class efficiently combines a frequency translation (typically “down conversion”) with a FIR filter (typically low-pass) and decimation. It is ideally suited for a “channel selection filter” and can be efficiently used to select and decimate a narrow band signal out of wide bandwidth input.

Uses a single input array to produce a single output array. Additional inputs and/or outputs are ignored.

Constructor Specific Documentation:

FIR filter with short input, gr\_complex output, and float taps that also frequency translates a signal from .

Construct a FIR filter with the given taps and a composite frequency translation that shifts center\_freq down to zero Hz. The frequency translation logically comes before the filtering operation.

- Parameters:**
- **decimation** – set the integer decimation rate
  - **taps** – a vector/list of taps of type float
  - **center\_freq** – Center frequency of signal to down convert from (Hz)
  - **sampling\_freq** – Sampling rate of signal (in Hz)

```

freq_xlating_fir_filter_scf_sptr.active_thread_priority(freq_xlating_fir_filter_scf_sptr self) →
int

freq_xlating_fir_filter_scf_sptr.center_freq(freq_xlating_fir_filter_scf_sptr self) → double

freq_xlating_fir_filter_scf_sptr.declare_sample_delay(freq_xlating_fir_filter_scf_sptr self, int
which, int delay)
    declare_sample_delay(freq_xlating_fir_filter_scf_sptr self, unsigned int delay)

freq_xlating_fir_filter_scf_sptr.message_subscribers(freq_xlating_fir_filter_scf_sptr self,
swig_int_ptr which_port) → swig_int_ptr

freq_xlating_fir_filter_scf_sptr.min_noutput_items(freq_xlating_fir_filter_scf_sptr self) → int

freq_xlating_fir_filter_scf_sptr.pc_input_buffers_full_avg(freq_xlating_fir_filter_scf_sptr
self, int which) → float

```

```

    pc_input_buffers_full_avg(freq_xlating_fir_filter_scf_sptr self) -> pmt_vector_float

freq_xlating_fir_filter_scf_sptr.pc_noutput_items_avg(freq_xlating_fir_filter_scf_sptr self) →
float

freq_xlating_fir_filter_scf_sptr.pc_nproduced_avg(freq_xlating_fir_filter_scf_sptr self) → float

freq_xlating_fir_filter_scf_sptr.pc_output_buffers_full_avg(freq_xlating_fir_filter_scf_sptr
self, int which) → float
    pc_output_buffers_full_avg(freq_xlating_fir_filter_scf_sptr self) -> pmt_vector_float

freq_xlating_fir_filter_scf_sptr.pc_throughput_avg(freq_xlating_fir_filter_scf_sptr self) → float

freq_xlating_fir_filter_scf_sptr.pc_work_time_avg(freq_xlating_fir_filter_scf_sptr self) → float

freq_xlating_fir_filter_scf_sptr.pc_work_time_total(freq_xlating_fir_filter_scf_sptr self) →
float

freq_xlating_fir_filter_scf_sptr.sample_delay(freq_xlating_fir_filter_scf_sptr self, int which) →
unsigned int

freq_xlating_fir_filter_scf_sptr.set_center_freq(freq_xlating_fir_filter_scf_sptr self, double
center_freq)

freq_xlating_fir_filter_scf_sptr.set_min_noutput_items(freq_xlating_fir_filter_scf_sptr self, int
m)

freq_xlating_fir_filter_scf_sptr.set_taps(freq_xlating_fir_filter_scf_sptr self, pmt_vector_float
taps)

freq_xlating_fir_filter_scf_sptr.set_thread_priority(freq_xlating_fir_filter_scf_sptr self, int
priority) → int

freq_xlating_fir_filter_scf_sptr.taps(freq_xlating_fir_filter_scf_sptr self) → pmt_vector_float

freq_xlating_fir_filter_scf_sptr.thread_priority(freq_xlating_fir_filter_scf_sptr self) → int

gnuradio.filter.hilbert_fc(unsigned int ntabs, gr::filter::firdes::win_type window, double beta=6.76) →
hilbert_fc_sptr
    Hilbert transformer.

    real output is input appropriately delayed. imaginary output is hilbert filtered (90 degree phase shift)
    version of input.

```

#### Constructor Specific Documentation:

Build a Hilbert transformer filter block.

**Parameters:**

- **ntaps** – The number of taps for the filter.
- **window** – Window type (see firdes::win\_type) to use.
- **beta** – Beta value for a Kaiser window.

```

hilbert_fc_sptr.active_thread_priority(hilbert_fc_sptr self) → int

hilbert_fc_sptr.declare_sample_delay(hilbert_fc_sptr self, int which, int delay)
    declare_sample_delay(hilbert_fc_sptr self, unsigned int delay)

hilbert_fc_sptr.message_subscribers(hilbert_fc_sptr self, swig_int_ptr which_port) →
swig_int_ptr

hilbert_fc_sptr.min_noutput_items(hilbert_fc_sptr self) → int

hilbert_fc_sptr.pc_input_buffers_full_avg(hilbert_fc_sptr self, int which) → float
    pc_input_buffers_full_avg(hilbert_fc_sptr self) -> pmt_vector_float

hilbert_fc_sptr.pc_noutput_items_avg(hilbert_fc_sptr self) → float

hilbert_fc_sptr.pc_nproduced_avg(hilbert_fc_sptr self) → float

hilbert_fc_sptr.pc_output_buffers_full_avg(hilbert_fc_sptr self, int which) → float
    pc_output_buffers_full_avg(hilbert_fc_sptr self) -> pmt_vector_float

hilbert_fc_sptr.pc_throughput_avg(hilbert_fc_sptr self) → float

hilbert_fc_sptr.pc_work_time_avg(hilbert_fc_sptr self) → float

```

`hilbert_fc_sptr.pc_work_time_total(hilbert_fc_sptr self) → float`

`hilbert_fc_sptr.sample_delay(hilbert_fc_sptr self, int which) → unsigned int`

`hilbert_fc_sptr.set_min_noutput_items(hilbert_fc_sptr self, int m)`

`hilbert_fc_sptr.set_thread_priority(hilbert_fc_sptr self, int priority) → int`

`hilbert_fc_sptr.thread_priority(hilbert_fc_sptr self) → int`

`gnuradio.filter.iir_filter_ccc(pmt_vector_cfloat fftaps, pmt_vector_cfloat fbtaps, bool oldstyle=True) → iir_filter_ccc_sptr`

IIR filter with complex input, complex output, and complex taps.

This filter uses the Direct Form I implementation, where contains the feed-forward taps, and the feedback ones.

The old style of the IIR filter uses feedback taps that are negative of what most definitions use (scipy and Matlab among them). This parameter keeps using the old GNU Radio style and is set to TRUE by default. When taps generated from scipy, Matlab, or `gr_filter_design`, use the new style by setting this to FALSE.

The input and output satisfy a difference equation of the form

$$y[n] - \sum_{k=1}^M a_k y[n-k] = \sum_{k=0}^N b_k x[n-k]$$

with the corresponding rational system function

$$H(z) = \frac{\sum_{k=0}^M b_k z^{-k}}{1 - \sum_{k=1}^N a_k z^{-k}}$$

Constructor Specific Documentation:

**Parameters:**

- `fftaps` –
- `fbtaps` –
- `oldstyle` –

`iir_filter_ccc_sptr.active_thread_priority(iir_filter_ccc_sptr self) → int`

`iir_filter_ccc_sptr.declare_sample_delay(iir_filter_ccc_sptr self, int which, int delay)`  
`declare_sample_delay(iir_filter_ccc_sptr self, unsigned int delay)`

`iir_filter_ccc_sptr.message_subscribers(iir_filter_ccc_sptr self, swig_int_ptr which_port) → swig_int_ptr`

`iir_filter_ccc_sptr.min_noutput_items(iir_filter_ccc_sptr self) → int`

`iir_filter_ccc_sptr.pc_input_buffers_full_avg(iir_filter_ccc_sptr self, int which) → float`  
`pc_input_buffers_full_avg(iir_filter_ccc_sptr self) → pmt_vector_float`

`iir_filter_ccc_sptr.pc_noutput_items_avg(iir_filter_ccc_sptr self) → float`

`iir_filter_ccc_sptr.pc_nproduced_avg(iir_filter_ccc_sptr self) → float`

`iir_filter_ccc_sptr.pc_output_buffers_full_avg(iir_filter_ccc_sptr self, int which) → float`  
`pc_output_buffers_full_avg(iir_filter_ccc_sptr self) → pmt_vector_float`

`iir_filter_ccc_sptr.pc_throughput_avg(iir_filter_ccc_sptr self) → float`

`iir_filter_ccc_sptr.pc_work_time_avg(iir_filter_ccc_sptr self) → float`

`iir_filter_ccc_sptr.pc_work_time_total(iir_filter_ccc_sptr self) → float`

`iir_filter_ccc_sptr.sample_delay(iir_filter_ccc_sptr self, int which) → unsigned int`

`iir_filter_ccc_sptr.set_min_noutput_items(iir_filter_ccc_sptr self, int m)`

`iir_filter_ccc_sptr.set_taps(iir_filter_ccc_sptr self, pmt_vector_cfloat fftaps, pmt_vector_cfloat fbtaps)`

`iir_filter_ccc_sptr.set_thread_priority(iir_filter_ccc_sptr self, int priority) → int`

`iir_filter_ccc_sptr.thread_priority(iir_filter_ccc_sptr self) → int`

`gnuradio.filter.iir_filter_ccd(pmt_vector_double fftaps, pmt_vector_double fbtaps, bool oldstyle=True) → iir_filter_ccd_sptr`

IIR filter with complex input, complex output, and double taps.

This filter uses the Direct Form I implementation, where contains the feed-forward taps, and the feedback ones.

The old style of the IIR filter uses feedback taps that are negative of what most definitions use (scipy and Matlab among them). This parameter keeps using the old GNU Radio style and is set to TRUE by default. When taps generated from scipy, Matlab, or gr\_filter\_design, use the new style by setting this to FALSE.

The input and output satisfy a difference equation of the form

$$y[n] - \sum_{k=1}^M a_k y[n-k] = \sum_{k=0}^N b_k x[n-k]$$

with the corresponding rational system function

$$H(z) = \frac{\sum_{k=0}^M b_k z^{-k}}{1 - \sum_{k=1}^N a_k z^{-k}}$$

Constructor Specific Documentation:

**Parameters:**

- **fftaps** –
- **fbtaps** –
- **oldstyle** –

```
iir_filter_ccd_sptr.active_thread_priority(iir_filter_ccd_sptr self) → int  
  
iir_filter_ccd_sptr.declare_sample_delay(iir_filter_ccd_sptr self, int which, int delay)  
    declare_sample_delay(iir_filter_ccd_sptr self, unsigned int delay)  
  
iir_filter_ccd_sptr.message_subscribers(iir_filter_ccd_sptr self, swig_int_ptr which_port) →  
    swig_int_ptr  
  
iir_filter_ccd_sptr.min_noutput_items(iir_filter_ccd_sptr self) → int  
  
iir_filter_ccd_sptr.pc_input_buffers_full_avg(iir_filter_ccd_sptr self, int which) → float  
    pc_input_buffers_full_avg(iir_filter_ccd_sptr self) → pmt_vector_float  
  
iir_filter_ccd_sptr.pc_noutput_items_avg(iir_filter_ccd_sptr self) → float  
  
iir_filter_ccd_sptr.pc_nproduced_avg(iir_filter_ccd_sptr self) → float  
  
iir_filter_ccd_sptr.pc_output_buffers_full_avg(iir_filter_ccd_sptr self, int which) → float  
    pc_output_buffers_full_avg(iir_filter_ccd_sptr self) → pmt_vector_float  
  
iir_filter_ccd_sptr.pc_throughput_avg(iir_filter_ccd_sptr self) → float  
  
iir_filter_ccd_sptr.pc_work_time_avg(iir_filter_ccd_sptr self) → float  
  
iir_filter_ccd_sptr.pc_work_time_total(iir_filter_ccd_sptr self) → float  
  
iir_filter_ccd_sptr.sample_delay(iir_filter_ccd_sptr self, int which) → unsigned int  
  
iir_filter_ccd_sptr.set_min_noutput_items(iir_filter_ccd_sptr self, int m)  
  
iir_filter_ccd_sptr.set_taps(iir_filter_ccd_sptr self, pmt_vector_double fftaps, pmt_vector_double  
    fbtaps)  
  
iir_filter_ccd_sptr.set_thread_priority(iir_filter_ccd_sptr self, int priority) → int  
  
iir_filter_ccd_sptr.thread_priority(iir_filter_ccd_sptr self) → int
```

```
gnuradio.filter.iir_filter_ccf(pmt_vector_float fftaps, pmt_vector_float fbtaps, bool oldstyle=True)  
→ iir_filter_ccf_sptr
```

IIR filter with complex input, complex output, and float taps.

This filter uses the Direct Form I implementation, where contains the feed-forward taps, and the feedback ones.

The old style of the IIR filter uses feedback taps that are negative of what most definitions use (scipy and Matlab among them). This parameter keeps using the old GNU Radio style and is set to TRUE by default. When taps generated from scipy, Matlab, or gr\_filter\_design, use the new style by setting this to FALSE.

The input and output satisfy a difference equation of the form

$$y[n] - \sum_{k=1}^M a_k y[n-k] = \sum_{k=0}^N b_k x[n-k]$$

with the corresponding rational system function

$$H(z) = \frac{\sum_{k=0}^M b_k z^{-k}}{1 - \sum_{k=1}^N a_k z^{-k}}$$

Constructor Specific Documentation:

**Parameters:**

- **fftaps** –
- **fbtaps** –
- **oldstyle** –

```
iir_filter_ccf_sptr.active_thread_priority(iir_filter_ccf_sptr self) → int

iir_filter_ccf_sptr.declare_sample_delay(iir_filter_ccf_sptr self, int which, int delay)
    declare_sample_delay(iir_filter_ccf_sptr self, unsigned int delay)

iir_filter_ccf_sptr.message_subscribers(iir_filter_ccf_sptr self, swig_int_ptr which_port) →
    swig_int_ptr

iir_filter_ccf_sptr.min_noutput_items(iir_filter_ccf_sptr self) → int

iir_filter_ccf_sptr.pc_input_buffers_full_avg(iir_filter_ccf_sptr self, int which) → float
    pc_input_buffers_full_avg(iir_filter_ccf_sptr self) -> pmt_vector_float

iir_filter_ccf_sptr.pc_noutput_items_avg(iir_filter_ccf_sptr self) → float

iir_filter_ccf_sptr.pc_nproduced_avg(iir_filter_ccf_sptr self) → float

iir_filter_ccf_sptr.pc_output_buffers_full_avg(iir_filter_ccf_sptr self, int which) → float
    pc_output_buffers_full_avg(iir_filter_ccf_sptr self) -> pmt_vector_float

iir_filter_ccf_sptr.pc_throughput_avg(iir_filter_ccf_sptr self) → float

iir_filter_ccf_sptr.pc_work_time_avg(iir_filter_ccf_sptr self) → float

iir_filter_ccf_sptr.pc_work_time_total(iir_filter_ccf_sptr self) → float

iir_filter_ccf_sptr.sample_delay(iir_filter_ccf_sptr self, int which) → unsigned int

iir_filter_ccf_sptr.set_min_noutput_items(iir_filter_ccf_sptr self, int m)

iir_filter_ccf_sptr.set_taps(iir_filter_ccf_sptr self, pmt_vector_float fftaps, pmt_vector_float
    fbtaps)

iir_filter_ccf_sptr.set_thread_priority(iir_filter_ccf_sptr self, int priority) → int

iir_filter_ccf_sptr.thread_priority(iir_filter_ccf_sptr self) → int
```

gnuradio.filter.**iir\_filter\_ccz**(pmt\_vector\_cdoube fftaps, pmt\_vector\_cdoube fbtaps, bool  
oldstyle=True) → iir\_filter\_ccz\_sptr

IIR filter with complex input, complex output, and complex (double) taps.

This filter uses the Direct Form I implementation, wherecontains the feed-forward taps, and the feedback ones.

The old style of the IIR filter uses feedback taps that are negative of what most definitions use (scipy and Matlab among them). This parameter keeps using the old GNU Radio style and is set to TRUE by default. When taps generated from scipy, Matlab, or gr\_filter\_design, use the new style by setting this to FALSE.

The input and output satisfy a difference equation of the form

$$y[n] - \sum_{k=1}^M a_k y[n-k] = \sum_{k=0}^N b_k x[n-k]$$

with the corresponding rational system function

$$H(z) = \frac{\sum_{k=0}^M b_k z^{-k}}{1 - \sum_{k=1}^N a_k z^{-k}}$$

Constructor Specific Documentation:

**Parameters:**

- **fftaps** –
- **fbtaps** –
- **oldstyle** –

```
iir_filter_ccz_sptr.active_thread_priority(iir_filter_ccz_sptr self) → int

iir_filter_ccz_sptr.declare_sample_delay(iir_filter_ccz_sptr self, int which, int delay)
    declare_sample_delay(iir_filter_ccz_sptr self, unsigned int delay)
```

`iir_filter_ccz_sptr.message_subscribers(iir_filter_ccz_sptr self, swig_int_ptr which_port) → swig_int_ptr`

`iir_filter_ccz_sptr.min_noutput_items(iir_filter_ccz_sptr self) → int`

`iir_filter_ccz_sptr.pc_input_buffers_full_avg(iir_filter_ccz_sptr self, int which) → float`  
`pc_input_buffers_full_avg(iir_filter_ccz_sptr self) → pmt_vector_float`

`iir_filter_ccz_sptr.pc_noutput_items_avg(iir_filter_ccz_sptr self) → float`

`iir_filter_ccz_sptr.pc_nproduced_avg(iir_filter_ccz_sptr self) → float`

`iir_filter_ccz_sptr.pc_output_buffers_full_avg(iir_filter_ccz_sptr self, int which) → float`  
`pc_output_buffers_full_avg(iir_filter_ccz_sptr self) → pmt_vector_float`

`iir_filter_ccz_sptr.pc_throughput_avg(iir_filter_ccz_sptr self) → float`

`iir_filter_ccz_sptr.pc_work_time_avg(iir_filter_ccz_sptr self) → float`

`iir_filter_ccz_sptr.pc_work_time_total(iir_filter_ccz_sptr self) → float`

`iir_filter_ccz_sptr.sample_delay(iir_filter_ccz_sptr self, int which) → unsigned int`

`iir_filter_ccz_sptr.set_min_noutput_items(iir_filter_ccz_sptr self, int m)`

`iir_filter_ccz_sptr.set_taps(iir_filter_ccz_sptr self, pmt_vector_cdoubles fbtaps, pmt_vector_cdoubles fbtaps)`

`iir_filter_ccz_sptr.set_thread_priority(iir_filter_ccz_sptr self, int priority) → int`

`iir_filter_ccz_sptr.thread_priority(iir_filter_ccz_sptr self) → int`

`gnuradio.filter.iir_filter_ffd(pmt_vector_double fbtaps, pmt_vector_double fbtaps, bool oldstyle=True) → iir_filter_ffd_sptr`

IIR filter with float input, float output and double taps.

This filter uses the Direct Form I implementation, where contains the feed-forward taps, and the feedback ones.

The old style of the IIR filter uses feedback taps that are negative of what most definitions use (scipy and Matlab among them). This parameter keeps using the old GNU Radio style and is set to TRUE by default. When taps generated from scipy, Matlab, or `gr_filter_design`, use the new style by setting this to FALSE.

The input and output satisfy a difference equation of the form

$$y[n] - \sum_{k=1}^M a_k y[n-k] = \sum_{k=0}^N b_k x[n-k]$$

with the corresponding rational system function

$$H(z) = \frac{\sum_{k=0}^M b_k z^{-k}}{1 - \sum_{k=1}^N a_k z^{-k}}$$

Constructor Specific Documentation:

**Parameters:**

- `fbtaps` –
- `fbtaps` –
- `oldstyle` –

`iir_filter_ffd_sptr.active_thread_priority(iir_filter_ffd_sptr self) → int`

`iir_filter_ffd_sptr.declare_sample_delay(iir_filter_ffd_sptr self, int which, int delay)`  
`declare_sample_delay(iir_filter_ffd_sptr self, unsigned int delay)`

`iir_filter_ffd_sptr.message_subscribers(iir_filter_ffd_sptr self, swig_int_ptr which_port) → swig_int_ptr`

`iir_filter_ffd_sptr.min_noutput_items(iir_filter_ffd_sptr self) → int`

`iir_filter_ffd_sptr.pc_input_buffers_full_avg(iir_filter_ffd_sptr self, int which) → float`  
`pc_input_buffers_full_avg(iir_filter_ffd_sptr self) → pmt_vector_float`

`iir_filter_ffd_sptr.pc_noutput_items_avg(iir_filter_ffd_sptr self) → float`

`iir_filter_ffd_sptr.pc_nproduced_avg(iir_filter_ffd_sptr self) → float`

`iir_filter_ffd_sptr.pc_output_buffers_full_avg(iir_filter_ffd_sptr self, int which) → float`

```

        pc_output_buffers_full_avg(iir_filter_ffd_sptr self) -> pmt_vector_float

iir_filter_ffd_sptr.pc_throughput_avg(iir_filter_ffd_sptr self) -> float

iir_filter_ffd_sptr.pc_work_time_avg(iir_filter_ffd_sptr self) -> float

iir_filter_ffd_sptr.pc_work_time_total(iir_filter_ffd_sptr self) -> float

iir_filter_ffd_sptr.sample_delay(iir_filter_ffd_sptr self, int which) -> unsigned int

iir_filter_ffd_sptr.set_min_noutput_items(iir_filter_ffd_sptr self, int m)

iir_filter_ffd_sptr.set_taps(iir_filter_ffd_sptr self, pmt_vector_double fftaps, pmt_vector_double
fbtaps)

iir_filter_ffd_sptr.set_thread_priority(iir_filter_ffd_sptr self, int priority) -> int

iir_filter_ffd_sptr.thread_priority(iir_filter_ffd_sptr self) -> int

gnuradio.filter.interp_fir_filter_ccc(unsigned int interpolation, pmt_vector_cfloat taps) ->
interp_fir_filter_ccc_sptr

```

Interpolating FIR filter with gr\_complex input, gr\_complex output and gr\_complex taps.

The fir\_filter\_XXX blocks create finite impulse response (FIR) filters that perform the convolution in the time domain:

The taps are a C++ vector (or Python list) of values of the type specified by the third letter in the block's suffix. For this block, the value is of type gr\_complex. Taps can be created using the firdes or optfir tools.

These versions of the filter can also act as up-samplers (or interpolators) by specifying an integer value for .

Constructor Specific Documentation:

Interpolating FIR filter with gr\_complex input, gr\_complex output, and gr\_complex taps.

**Parameters:**

- **interpolation** – set the integer interpolation rate
- **taps** – a vector/list of taps of type gr\_complex

```

interp_fir_filter_ccc_sptr.active_thread_priority(interp_fir_filter_ccc_sptr self) -> int

interp_fir_filter_ccc_sptr.declare_sample_delay(interp_fir_filter_ccc_sptr self, int which, int
delay)
    declare_sample_delay(interp_fir_filter_ccc_sptr self, unsigned int delay)

interp_fir_filter_ccc_sptr.message_subscribers(interp_fir_filter_ccc_sptr self, swig_int_ptr
which_port) -> swig_int_ptr

interp_fir_filter_ccc_sptr.min_noutput_items(interp_fir_filter_ccc_sptr self) -> int

interp_fir_filter_ccc_sptr.pc_input_buffers_full_avg(interp_fir_filter_ccc_sptr self, int which)
-> float
    pc_input_buffers_full_avg(interp_fir_filter_ccc_sptr self) -> pmt_vector_float

interp_fir_filter_ccc_sptr.pc_noutput_items_avg(interp_fir_filter_ccc_sptr self) -> float

interp_fir_filter_ccc_sptr.pc_nproduced_avg(interp_fir_filter_ccc_sptr self) -> float

interp_fir_filter_ccc_sptr.pc_output_buffers_full_avg(interp_fir_filter_ccc_sptr self, int
which) -> float
    pc_output_buffers_full_avg(interp_fir_filter_ccc_sptr self) -> pmt_vector_float

interp_fir_filter_ccc_sptr.pc_throughput_avg(interp_fir_filter_ccc_sptr self) -> float

interp_fir_filter_ccc_sptr.pc_work_time_avg(interp_fir_filter_ccc_sptr self) -> float

interp_fir_filter_ccc_sptr.pc_work_time_total(interp_fir_filter_ccc_sptr self) -> float

interp_fir_filter_ccc_sptr.sample_delay(interp_fir_filter_ccc_sptr self, int which) -> unsigned int

interp_fir_filter_ccc_sptr.set_min_noutput_items(interp_fir_filter_ccc_sptr self, int m)

interp_fir_filter_ccc_sptr.set_taps(interp_fir_filter_ccc_sptr self, pmt_vector_cfloat taps)

interp_fir_filter_ccc_sptr.set_thread_priority(interp_fir_filter_ccc_sptr self, int priority) -> int

```

```
interp_fir_filter_ccc_sptr.taps(interp_fir_filter_ccc_sptr self) → pmt_vector_cfloat
```

```
interp_fir_filter_ccc_sptr.thread_priority(interp_fir_filter_ccc_sptr self) → int
```

gnuradio.filter.**interp\_fir\_filter\_ccf**(unsigned int interpolation, pmt\_vector\_float taps) → interp\_fir\_filter\_ccf\_sptr

Interpolating FIR filter with gr\_complex input, gr\_complex output and float taps.

The fir\_filter\_XXX blocks create finite impulse response (FIR) filters that perform the convolution in the time domain:

The taps are a C++ vector (or Python list) of values of the type specified by the third letter in the block's suffix. For this block, the value is of type float. Taps can be created using the firdes or optfir tools.

These versions of the filter can also act as up-samplers (or interpolators) by specifying an integer value for .

Constructor Specific Documentation:

Interpolating FIR filter with gr\_complex input, gr\_complex output, and float taps.

**Parameters:**

- **interpolation** – set the integer interpolation rate
- **taps** – a vector/list of taps of type float

```
interp_fir_filter_ccf_sptr.active_thread_priority(interp_fir_filter_ccf_sptr self) → int
```

```
interp_fir_filter_ccf_sptr.declare_sample_delay(interp_fir_filter_ccf_sptr self, int which, int delay)
```

```
declare_sample_delay(interp_fir_filter_ccf_sptr self, unsigned int delay)
```

```
interp_fir_filter_ccf_sptr.message_subscribers(interp_fir_filter_ccf_sptr self, swig_int_ptr which_port) → swig_int_ptr
```

```
interp_fir_filter_ccf_sptr.min_noutput_items(interp_fir_filter_ccf_sptr self) → int
```

```
interp_fir_filter_ccf_sptr.pc_input_buffers_full_avg(interp_fir_filter_ccf_sptr self, int which) → float
```

```
pc_input_buffers_full_avg(interp_fir_filter_ccf_sptr self) -> pmt_vector_float
```

```
interp_fir_filter_ccf_sptr.pc_noutput_items_avg(interp_fir_filter_ccf_sptr self) → float
```

```
interp_fir_filter_ccf_sptr.pc_nproduced_avg(interp_fir_filter_ccf_sptr self) → float
```

```
interp_fir_filter_ccf_sptr.pc_output_buffers_full_avg(interp_fir_filter_ccf_sptr self, int which) → float
```

```
pc_output_buffers_full_avg(interp_fir_filter_ccf_sptr self) -> pmt_vector_float
```

```
interp_fir_filter_ccf_sptr.pc_throughput_avg(interp_fir_filter_ccf_sptr self) → float
```

```
interp_fir_filter_ccf_sptr.pc_work_time_avg(interp_fir_filter_ccf_sptr self) → float
```

```
interp_fir_filter_ccf_sptr.pc_work_time_total(interp_fir_filter_ccf_sptr self) → float
```

```
interp_fir_filter_ccf_sptr.sample_delay(interp_fir_filter_ccf_sptr self, int which) → unsigned int
```

```
interp_fir_filter_ccf_sptr.set_min_noutput_items(interp_fir_filter_ccf_sptr self, int m)
```

```
interp_fir_filter_ccf_sptr.set_taps(interp_fir_filter_ccf_sptr self, pmt_vector_float taps)
```

```
interp_fir_filter_ccf_sptr.set_thread_priority(interp_fir_filter_ccf_sptr self, int priority) → int
```

```
interp_fir_filter_ccf_sptr.taps(interp_fir_filter_ccf_sptr self) → pmt_vector_float
```

```
interp_fir_filter_ccf_sptr.thread_priority(interp_fir_filter_ccf_sptr self) → int
```

gnuradio.filter.**interp\_fir\_filter\_fcc**(unsigned int interpolation, pmt\_vector\_cfloat taps) → interp\_fir\_filter\_fcc\_sptr

Interpolating FIR filter with float input, gr\_complex output and gr\_complex taps.

The fir\_filter\_XXX blocks create finite impulse response (FIR) filters that perform the convolution in the time domain:

The taps are a C++ vector (or Python list) of values of the type specified by the third letter in the block's suffix. For this block, the value is of type gr\_complex. Taps can be created using the firdes or optfir tools.

These versions of the filter can also act as up-samplers (or interpolators) by specifying an integer value for

#### Constructor Specific Documentation:

Interpolating FIR filter with float input, gr\_complex output, and gr\_complex taps.

- Parameters:**
- **interpolation** – set the integer interpolation rate
  - **taps** – a vector/list of taps of type gr\_complex

```
interp_fir_filter_fcc_sptr.active_thread_priority(interp_fir_filter_fcc_sptr self) → int

interp_fir_filter_fcc_sptr.declare_sample_delay(interp_fir_filter_fcc_sptr self, int which, int delay)
    declare_sample_delay(interp_fir_filter_fcc_sptr self, unsigned int delay)

interp_fir_filter_fcc_sptr.message_subscribers(interp_fir_filter_fcc_sptr self, swig_int_ptr which_port) → swig_int_ptr

interp_fir_filter_fcc_sptr.min_noutput_items(interp_fir_filter_fcc_sptr self) → int

interp_fir_filter_fcc_sptr.pc_input_buffers_full_avg(interp_fir_filter_fcc_sptr self, int which) → float
    pc_input_buffers_full_avg(interp_fir_filter_fcc_sptr self) → pmt_vector_float

interp_fir_filter_fcc_sptr.pc_noutput_items_avg(interp_fir_filter_fcc_sptr self) → float

interp_fir_filter_fcc_sptr.pc_nproduced_avg(interp_fir_filter_fcc_sptr self) → float

interp_fir_filter_fcc_sptr.pc_output_buffers_full_avg(interp_fir_filter_fcc_sptr self, int which) → float
    pc_output_buffers_full_avg(interp_fir_filter_fcc_sptr self) → pmt_vector_float

interp_fir_filter_fcc_sptr.pc_throughput_avg(interp_fir_filter_fcc_sptr self) → float

interp_fir_filter_fcc_sptr.pc_work_time_avg(interp_fir_filter_fcc_sptr self) → float

interp_fir_filter_fcc_sptr.pc_work_time_total(interp_fir_filter_fcc_sptr self) → float

interp_fir_filter_fcc_sptr.sample_delay(interp_fir_filter_fcc_sptr self, int which) → unsigned int

interp_fir_filter_fcc_sptr.set_min_noutput_items(interp_fir_filter_fcc_sptr self, int m)

interp_fir_filter_fcc_sptr.set_taps(interp_fir_filter_fcc_sptr self, pmt_vector_cfloat taps)

interp_fir_filter_fcc_sptr.set_thread_priority(interp_fir_filter_fcc_sptr self, int priority) → int

interp_fir_filter_fcc_sptr.taps(interp_fir_filter_fcc_sptr self) → pmt_vector_cfloat

interp_fir_filter_fcc_sptr.thread_priority(interp_fir_filter_fcc_sptr self) → int

gnuradio.filter.interp_fir_filter_fff(unsigned int interpolation, pmt_vector_float taps) →
interp_fir_filter_fff_sptr
```

Interpolating FIR filter with float input, float output and float taps.

The fir\_filter\_XXX blocks create finite impulse response (FIR) filters that perform the convolution in the time domain:

The taps are a C++ vector (or Python list) of values of the type specified by the third letter in the block's suffix. For this block, the value is of type float. Taps can be created using the firdes or optfir tools.

These versions of the filter can also act as up-samplers (or interpolators) by specifying an integer value for

#### Constructor Specific Documentation:

Interpolating FIR filter with float input, float output, and float taps.

- Parameters:**
- **interpolation** – set the integer interpolation rate
  - **taps** – a vector/list of taps of type float

```
interp_fir_filter_fff_sptr.active_thread_priority(interp_fir_filter_fff_sptr self) → int

interp_fir_filter_fff_sptr.declare_sample_delay(interp_fir_filter_fff_sptr self, int which, int delay)
    declare_sample_delay(interp_fir_filter_fff_sptr self, unsigned int delay)
```

```
interp_fir_filter_fff_sptr.message_subscribers(interp_fir_filter_fff_sptr self, swig_int_ptr which_port) → swig_int_ptr
```

```
interp_fir_filter_fff_sptr.min_noutput_items(interp_fir_filter_fff_sptr self) → int
```

```
interp_fir_filter_fff_sptr.pc_input_buffers_full_avg(interp_fir_filter_fff_sptr self, int which) → float
```

```
pc_input_buffers_full_avg(interp_fir_filter_fff_sptr self) -> pmt_vector_float
```

```
interp_fir_filter_fff_sptr.pc_noutput_items_avg(interp_fir_filter_fff_sptr self) → float
```

```
interp_fir_filter_fff_sptr.pc_nproduced_avg(interp_fir_filter_fff_sptr self) → float
```

```
interp_fir_filter_fff_sptr.pc_output_buffers_full_avg(interp_fir_filter_fff_sptr self, int which) → float
```

```
pc_output_buffers_full_avg(interp_fir_filter_fff_sptr self) -> pmt_vector_float
```

```
interp_fir_filter_fff_sptr.pc_throughput_avg(interp_fir_filter_fff_sptr self) → float
```

```
interp_fir_filter_fff_sptr.pc_work_time_avg(interp_fir_filter_fff_sptr self) → float
```

```
interp_fir_filter_fff_sptr.pc_work_time_total(interp_fir_filter_fff_sptr self) → float
```

```
interp_fir_filter_fff_sptr.sample_delay(interp_fir_filter_fff_sptr self, int which) → unsigned int
```

```
interp_fir_filter_fff_sptr.set_min_noutput_items(interp_fir_filter_fff_sptr self, int m)
```

```
interp_fir_filter_fff_sptr.set_taps(interp_fir_filter_fff_sptr self, pmt_vector_float taps)
```

```
interp_fir_filter_fff_sptr.set_thread_priority(interp_fir_filter_fff_sptr self, int priority) → int
```

```
interp_fir_filter_fff_sptr.taps(interp_fir_filter_fff_sptr self) → pmt_vector_float
```

```
interp_fir_filter_fff_sptr.thread_priority(interp_fir_filter_fff_sptr self) → int
```

```
gnuradio.filter.interp_fir_filter_fsf(unsigned int interpolation, pmt_vector_float taps) → interp_fir_filter_fsf_sptr
```

Interpolating FIR filter with float input, short output and float taps.

The `fir_filter_XXX` blocks create finite impulse response (FIR) filters that perform the convolution in the time domain:

The taps are a C++ vector (or Python list) of values of the type specified by the third letter in the block's suffix. For this block, the value is of type float. Taps can be created using the `firdes` or `optfir` tools.

These versions of the filter can also act as up-samplers (or interpolators) by specifying an integer value for `.`

Constructor Specific Documentation:

Interpolating FIR filter with float input, short output, and float taps.

**Parameters:**

- **interpolation** – set the integer interpolation rate
- **taps** – a vector/list of taps of type float

```
interp_fir_filter_fsf_sptr.active_thread_priority(interp_fir_filter_fsf_sptr self) → int
```

```
interp_fir_filter_fsf_sptr.declare_sample_delay(interp_fir_filter_fsf_sptr self, int which, int delay)
```

```
declare_sample_delay(interp_fir_filter_fsf_sptr self, unsigned int delay)
```

```
interp_fir_filter_fsf_sptr.message_subscribers(interp_fir_filter_fsf_sptr self, swig_int_ptr which_port) → swig_int_ptr
```

```
interp_fir_filter_fsf_sptr.min_noutput_items(interp_fir_filter_fsf_sptr self) → int
```

```
interp_fir_filter_fsf_sptr.pc_input_buffers_full_avg(interp_fir_filter_fsf_sptr self, int which) → float
```

```
pc_input_buffers_full_avg(interp_fir_filter_fsf_sptr self) -> pmt_vector_float
```

```
interp_fir_filter_fsf_sptr.pc_noutput_items_avg(interp_fir_filter_fsf_sptr self) → float
```

```
interp_fir_filter_fsf_sptr.pc_nproduced_avg(interp_fir_filter_fsf_sptr self) → float
```

```
interp_fir_filter_fsf_sptr.pc_output_buffers_full_avg(interp_fir_filter_fsf_sptr self, int which)
```

→ float

pc\_output\_buffers\_full\_avg(interp\_fir\_filter\_fsf\_sptr self) -> pmt\_vector\_float

interp\_fir\_filter\_fsf\_sptr.pc\_throughput\_avg(interp\_fir\_filter\_fsf\_sptr self) → float

interp\_fir\_filter\_fsf\_sptr.pc\_work\_time\_avg(interp\_fir\_filter\_fsf\_sptr self) → float

interp\_fir\_filter\_fsf\_sptr.pc\_work\_time\_total(interp\_fir\_filter\_fsf\_sptr self) → float

interp\_fir\_filter\_fsf\_sptr.sample\_delay(interp\_fir\_filter\_fsf\_sptr self, int which) → unsigned int

interp\_fir\_filter\_fsf\_sptr.set\_min\_noutput\_items(interp\_fir\_filter\_fsf\_sptr self, int m)

interp\_fir\_filter\_fsf\_sptr.set\_taps(interp\_fir\_filter\_fsf\_sptr self, pmt\_vector\_float taps)

interp\_fir\_filter\_fsf\_sptr.set\_thread\_priority(interp\_fir\_filter\_fsf\_sptr self, int priority) → int

interp\_fir\_filter\_fsf\_sptr.taps(interp\_fir\_filter\_fsf\_sptr self) → pmt\_vector\_float

interp\_fir\_filter\_fsf\_sptr.thread\_priority(interp\_fir\_filter\_fsf\_sptr self) → int

gnuradio.filter.interp\_fir\_filter\_scc(unsigned int interpolation, pmt\_vector\_cfloat taps) →  
interp\_fir\_filter\_scc\_sptr

Interpolating FIR filter with short input, gr\_complex output and gr\_complex taps.

The fir\_filter\_XXX blocks create finite impulse response (FIR) filters that perform the convolution in the time domain:

The taps are a C++ vector (or Python list) of values of the type specified by the third letter in the block's suffix. For this block, the value is of type gr\_complex. Taps can be created using the firdes or optfir tools.

These versions of the filter can also act as up-samplers (or interpolators) by specifying an integer value for .

Constructor Specific Documentation:

Interpolating FIR filter with short input, gr\_complex output, and gr\_complex taps.

**Parameters:**

- **interpolation** – set the integer interpolation rate
- **taps** – a vector/list of taps of type gr\_complex

interp\_fir\_filter\_scc\_sptr.active\_thread\_priority(interp\_fir\_filter\_scc\_sptr self) → int

interp\_fir\_filter\_scc\_sptr.declare\_sample\_delay(interp\_fir\_filter\_scc\_sptr self, int which, int delay)

declare\_sample\_delay(interp\_fir\_filter\_scc\_sptr self, unsigned int delay)

interp\_fir\_filter\_scc\_sptr.message\_subscribers(interp\_fir\_filter\_scc\_sptr self, swig\_int\_ptr which\_port) → swig\_int\_ptr

interp\_fir\_filter\_scc\_sptr.min\_noutput\_items(interp\_fir\_filter\_scc\_sptr self) → int

interp\_fir\_filter\_scc\_sptr.pc\_input\_buffers\_full\_avg(interp\_fir\_filter\_scc\_sptr self, int which) → float

pc\_input\_buffers\_full\_avg(interp\_fir\_filter\_scc\_sptr self) -> pmt\_vector\_float

interp\_fir\_filter\_scc\_sptr.pc\_noutput\_items\_avg(interp\_fir\_filter\_scc\_sptr self) → float

interp\_fir\_filter\_scc\_sptr.pc\_nproduced\_avg(interp\_fir\_filter\_scc\_sptr self) → float

interp\_fir\_filter\_scc\_sptr.pc\_output\_buffers\_full\_avg(interp\_fir\_filter\_scc\_sptr self, int which) → float

pc\_output\_buffers\_full\_avg(interp\_fir\_filter\_scc\_sptr self) -> pmt\_vector\_float

interp\_fir\_filter\_scc\_sptr.pc\_throughput\_avg(interp\_fir\_filter\_scc\_sptr self) → float

interp\_fir\_filter\_scc\_sptr.pc\_work\_time\_avg(interp\_fir\_filter\_scc\_sptr self) → float

interp\_fir\_filter\_scc\_sptr.pc\_work\_time\_total(interp\_fir\_filter\_scc\_sptr self) → float

interp\_fir\_filter\_scc\_sptr.sample\_delay(interp\_fir\_filter\_scc\_sptr self, int which) → unsigned int

interp\_fir\_filter\_scc\_sptr.set\_min\_noutput\_items(interp\_fir\_filter\_scc\_sptr self, int m)

interp\_fir\_filter\_scc\_sptr.set\_taps(interp\_fir\_filter\_scc\_sptr self, pmt\_vector\_cfloat taps)

```
interp_fir_filter_scc_sptr.set_thread_priority(interp_fir_filter_scc_sptr self, int priority) → int
```

```
interp_fir_filter_scc_sptr.taps(interp_fir_filter_scc_sptr self) → pmt_vector_cfloat
```

```
interp_fir_filter_scc_sptr.thread_priority(interp_fir_filter_scc_sptr self) → int
```

```
gnuradio.filter.pfb_arb_resampler_ccc(float rate, pmt_vector_cfloat taps, unsigned int filter_size=32) → pfb_arb_resampler_ccc_sptr
```

Polyphase filterbank arbitrary resampler with gr\_complex input, gr\_complex output and gr\_complex taps.

This block takes in a signal stream and calls gr::filter::kernel::pfb\_arb\_resampler\_ccc to perform arbitrary resampling on the stream.

Output sampling rate is \* input rate.

Constructor Specific Documentation:

Build the polyphase filterbank arbitrary resampler.

**Parameters:**

- **rate** – (float) Specifies the resampling rate to use
- **taps** – (vector/list of complex) The prototype filter to populate the filterbank. The taps should be generated at the filter\_size sampling rate.
- **filter\_size** – (unsigned int) The number of filters in the filter bank. This is directly related to quantization noise introduced during the resampling. Defaults to 32 filters.

```
pfb_arb_resampler_ccc_sptr.active_thread_priority(pfb_arb_resampler_ccc_sptr self) → int
```

```
pfb_arb_resampler_ccc_sptr.decimation_rate(pfb_arb_resampler_ccc_sptr self) → unsigned int
```

Gets the decimation rate of the filter.

```
pfb_arb_resampler_ccc_sptr.declare_sample_delay(pfb_arb_resampler_ccc_sptr self, int which, int delay)
```

declare\_sample\_delay(pfb\_arb\_resampler\_ccc\_sptr self, unsigned int delay)

```
pfb_arb_resampler_ccc_sptr.fractional_rate(pfb_arb_resampler_ccc_sptr self) → float
```

Gets the fractional rate of the filter.

```
pfb_arb_resampler_ccc_sptr.group_delay(pfb_arb_resampler_ccc_sptr self) → int
```

Get the group delay of the filter.

```
pfb_arb_resampler_ccc_sptr.interpolation_rate(pfb_arb_resampler_ccc_sptr self) → unsigned int
```

Gets the interpolation rate of the filter.

```
pfb_arb_resampler_ccc_sptr.message_subscribers(pfb_arb_resampler_ccc_sptr self, swig_int_ptr which_port) → swig_int_ptr
```

```
pfb_arb_resampler_ccc_sptr.min_noutput_items(pfb_arb_resampler_ccc_sptr self) → int
```

```
pfb_arb_resampler_ccc_sptr.pc_input_buffers_full_avg(pfb_arb_resampler_ccc_sptr self, int which) → float
```

pc\_input\_buffers\_full\_avg(pfb\_arb\_resampler\_ccc\_sptr self) -> pmt\_vector\_float

```
pfb_arb_resampler_ccc_sptr.pc_noutput_items_avg(pfb_arb_resampler_ccc_sptr self) → float
```

```
pfb_arb_resampler_ccc_sptr.pc_nproduced_avg(pfb_arb_resampler_ccc_sptr self) → float
```

```
pfb_arb_resampler_ccc_sptr.pc_output_buffers_full_avg(pfb_arb_resampler_ccc_sptr self, int which) → float
```

pc\_output\_buffers\_full\_avg(pfb\_arb\_resampler\_ccc\_sptr self) -> pmt\_vector\_float

```
pfb_arb_resampler_ccc_sptr.pc_throughput_avg(pfb_arb_resampler_ccc_sptr self) → float
```

```
pfb_arb_resampler_ccc_sptr.pc_work_time_avg(pfb_arb_resampler_ccc_sptr self) → float
```

```
pfb_arb_resampler_ccc_sptr.pc_work_time_total(pfb_arb_resampler_ccc_sptr self) → float
```

```
pfb_arb_resampler_ccc_sptr.phase(pfb_arb_resampler_ccc_sptr self) → float
```

Gets the current phase of the resampler in radians (2 to 2pi).

```
pfb_arb_resampler_ccc_sptr.phase_offset(pfb_arb_resampler_ccc_sptr self, float freq, float fs) → float
```

Calculates the phase offset expected by a sine wave of frequency and sampling rate (assuming input sine wave has 0 degree phase).

`pfb_arb_resampler_ccc_sptr.print_taps(pfb_arb_resampler_ccc_sptr self)`  
Print all of the filterbank taps to screen.

`pfb_arb_resampler_ccc_sptr.sample_delay(pfb_arb_resampler_ccc_sptr self, int which) → unsigned int`

`pfb_arb_resampler_ccc_sptr.set_min_noutput_items(pfb_arb_resampler_ccc_sptr self, int m)`

`pfb_arb_resampler_ccc_sptr.set_phase(pfb_arb_resampler_ccc_sptr self, float ph)`  
Sets the current phase offset in radians (0 to 2pi).

`pfb_arb_resampler_ccc_sptr.set_rate(pfb_arb_resampler_ccc_sptr self, float rate)`  
Sets the resampling rate of the block.

`pfb_arb_resampler_ccc_sptr.set_taps(pfb_arb_resampler_ccc_sptr self, pmt_vector_cfloat taps)`  
Resets the filterbank's filter taps with the new prototype filter

`pfb_arb_resampler_ccc_sptr.set_thread_priority(pfb_arb_resampler_ccc_sptr self, int priority) → int`

`pfb_arb_resampler_ccc_sptr.taps(pfb_arb_resampler_ccc_sptr self) → gr_vector_vector_complexf`  
Return a vector<vector<>> of the filterbank taps

`pfb_arb_resampler_ccc_sptr.taps_per_filter(pfb_arb_resampler_ccc_sptr self) → unsigned int`  
Gets the number of taps per filter.

`pfb_arb_resampler_ccc_sptr.thread_priority(pfb_arb_resampler_ccc_sptr self) → int`

`gnuradio.filter.pfb_arb_resampler_ccf(float rate, pmt_vector_float taps, unsigned int filter_size=32) → pfb_arb_resampler_ccf_sptr`

Polyphase filterbank arbitrary resampler with gr\_complex input, gr\_complex output and float taps.

This block takes in a signal stream and calls gr::filter::kernel::pfb\_arb\_resampler\_ccf to perform arbitrary resampling on the stream.

Output sampling rate is \* input rate.

Constructor Specific Documentation:

Build the polyphase filterbank arbitrary resampler.

**Parameters:**

- **rate** – (float) Specifies the resampling rate to use
- **taps** – (vector/list of floats) The prototype filter to populate the filterbank. The taps should be generated at the filter\_size sampling rate.
- **filter\_size** – (unsigned int) The number of filters in the filter bank. This is directly related to quantization noise introduced during the resampling. Defaults to 32 filters.

`pfb_arb_resampler_ccf_sptr.active_thread_priority(pfb_arb_resampler_ccf_sptr self) → int`

`pfb_arb_resampler_ccf_sptr.decimation_rate(pfb_arb_resampler_ccf_sptr self) → unsigned int`  
Gets the decimation rate of the filter.

`pfb_arb_resampler_ccf_sptr.declare_sample_delay(pfb_arb_resampler_ccf_sptr self, int which, int delay)`  
declare\_sample\_delay(pfb\_arb\_resampler\_ccf\_sptr self, unsigned int delay)

`pfb_arb_resampler_ccf_sptr.fractional_rate(pfb_arb_resampler_ccf_sptr self) → float`  
Gets the fractional rate of the filter.

`pfb_arb_resampler_ccf_sptr.group_delay(pfb_arb_resampler_ccf_sptr self) → int`  
Get the group delay of the filter.

`pfb_arb_resampler_ccf_sptr.interpolation_rate(pfb_arb_resampler_ccf_sptr self) → unsigned int`  
Gets the interpolation rate of the filter.

`pfb_arb_resampler_ccf_sptr.message_subscribers(pfb_arb_resampler_ccf_sptr self, swig_int_ptr which_port) → swig_int_ptr`

`pfb_arb_resampler_ccf_sptr.min_noutput_items(pfb_arb_resampler_ccf_sptr self) → int`

`pfb_arb_resampler_ccf_sptr.pc_input_buffers_full_avg(pfb_arb_resampler_ccf_sptr self, int which) → float`

pc\_input\_buffers\_full\_avg(pfb\_arb\_resampler\_ccf\_sptr self) -> pmt\_vector\_float

`pfb_arb_resampler_ccf_sptr.pc_noutput_items_avg(pfb_arb_resampler_ccf_sptr self) → float`

`pfb_arb_resampler_ccf_sptr.pc_nproduced_avg(pfb_arb_resampler_ccf_sptr self) → float`

`pfb_arb_resampler_ccf_sptr.pc_output_buffers_full_avg(pfb_arb_resampler_ccf_sptr self, int which) → float`  
`pc_output_buffers_full_avg(pfb_arb_resampler_ccf_sptr self) → pmt_vector_float`

`pfb_arb_resampler_ccf_sptr.pc_throughput_avg(pfb_arb_resampler_ccf_sptr self) → float`

`pfb_arb_resampler_ccf_sptr.pc_work_time_avg(pfb_arb_resampler_ccf_sptr self) → float`

`pfb_arb_resampler_ccf_sptr.pc_work_time_total(pfb_arb_resampler_ccf_sptr self) → float`

`pfb_arb_resampler_ccf_sptr.phase(pfb_arb_resampler_ccf_sptr self) → float`  
 Gets the current phase of the resampler in radians (2 to 2pi).

`pfb_arb_resampler_ccf_sptr.phase_offset(pfb_arb_resampler_ccf_sptr self, float freq, float fs) → float`  
 Calculates the phase offset expected by a sine wave of frequency and sampling rate (assuming input sine wave has 0 degree phase).

`pfb_arb_resampler_ccf_sptr.print_taps(pfb_arb_resampler_ccf_sptr self)`  
 Print all of the filterbank taps to screen.

`pfb_arb_resampler_ccf_sptr.sample_delay(pfb_arb_resampler_ccf_sptr self, int which) → unsigned int`

`pfb_arb_resampler_ccf_sptr.set_min_noutput_items(pfb_arb_resampler_ccf_sptr self, int m)`

`pfb_arb_resampler_ccf_sptr.set_phase(pfb_arb_resampler_ccf_sptr self, float ph)`  
 Sets the current phase offset in radians (0 to 2pi).

`pfb_arb_resampler_ccf_sptr.set_rate(pfb_arb_resampler_ccf_sptr self, float rate)`  
 Sets the resampling rate of the block.

`pfb_arb_resampler_ccf_sptr.set_taps(pfb_arb_resampler_ccf_sptr self, pmt_vector_float taps)`  
 Resets the filterbank's filter taps with the new prototype filter

`pfb_arb_resampler_ccf_sptr.set_thread_priority(pfb_arb_resampler_ccf_sptr self, int priority) → int`

`pfb_arb_resampler_ccf_sptr.taps(pfb_arb_resampler_ccf_sptr self) → std::vector< std::vector< float, std::allocator< float > >, std::allocator< std::vector< float, std::allocator< float > > > >`  
 Return a vector<vector<>> of the filterbank taps

`pfb_arb_resampler_ccf_sptr.taps_per_filter(pfb_arb_resampler_ccf_sptr self) → unsigned int`  
 Gets the number of taps per filter.

`pfb_arb_resampler_ccf_sptr.thread_priority(pfb_arb_resampler_ccf_sptr self) → int`

`gnuradio.filter.pfb_arb_resampler_fff(float rate, pmt_vector_float taps, unsigned int filter_size=32) → pfb_arb_resampler_fff_sptr`

Polyphase filterbank arbitrary resampler with float input, float output and float taps.

This block takes in a signal stream and performs arbitrary resampling. The resampling rate can be any real number . The resampling is done by constructing filters where is the interpolation rate. We then calculate where .

Using and , we can perform rational resampling where is a rational number close to the input rate where we have filters and we cycle through them as a polyphase filterbank with a stride of so that .

To get the arbitrary rate, we want to interpolate between two points. For each value out, we take an output from the current filter , and the next filter and then linearly interpolate between the two based on the real resampling rate we want.

The linear interpolation only provides us with an approximation to the real sampling rate specified. The error is a quantization error between the two filters we used as our interpolation points. To this end, the number of filters , used determines the quantization error; the larger , the smaller the noise. You can design for a specified noise floor by setting the filter size (parameters ). The size defaults to 32 filters, which is about as good as most implementations need.

The trick with designing this filter is in how to specify the taps of the prototype filter. Like the PFB interpolator, the taps are specified using the interpolated filter rate. In this case, that rate is the input

sample rate multiplied by the number of filters in the filterbank, which is also the interpolation rate. All other values should be relative to this rate.

For example, for a 32-filter arbitrary resampler and using the GNU Radio's `firdec` utility to build the filter, we build a low-pass filter with a sampling rate of , a 3-dB bandwidth of and a transition bandwidth of . We can also specify the out-of-band attenuation to use, , and the filter window function (a Blackman-harris window in this case). The first input is the gain of the filter, which we specify here as the interpolation rate ().

The theory behind this block can be found in Chapter 7.5 of the following book:

Constructor Specific Documentation:

Build the polyphase filterbank arbitrary resampler.

- Parameters:**
- **rate** – (float) Specifies the resampling rate to use
  - **taps** – (vector/list of floats) The prototype filter to populate the filterbank. The taps should be generated at the `filter_size` sampling rate.
  - **filter\_size** – (unsigned int) The number of filters in the filter bank. This is directly related to quantization noise introduced during the resampling. Defaults to 32 filters.

`pfb_arb_resampler_fff_sptr.active_thread_priority(pfb_arb_resampler_fff_sptr self) → int`

`pfb_arb_resampler_fff_sptr.decimation_rate(pfb_arb_resampler_fff_sptr self) → unsigned int`  
Gets the decimation rate of the filter.

`pfb_arb_resampler_fff_sptr.declare_sample_delay(pfb_arb_resampler_fff_sptr self, int which, int delay)`  
`declare_sample_delay(pfb_arb_resampler_fff_sptr self, unsigned int delay)`

`pfb_arb_resampler_fff_sptr.fractional_rate(pfb_arb_resampler_fff_sptr self) → float`  
Gets the fractional rate of the filter.

`pfb_arb_resampler_fff_sptr.group_delay(pfb_arb_resampler_fff_sptr self) → int`  
Get the group delay of the filter.

`pfb_arb_resampler_fff_sptr.interpolation_rate(pfb_arb_resampler_fff_sptr self) → unsigned int`  
Gets the interpolation rate of the filter.

`pfb_arb_resampler_fff_sptr.message_subscribers(pfb_arb_resampler_fff_sptr self, swig_int_ptr which_port) → swig_int_ptr`

`pfb_arb_resampler_fff_sptr.min_noutput_items(pfb_arb_resampler_fff_sptr self) → int`

`pfb_arb_resampler_fff_sptr.pc_input_buffers_full_avg(pfb_arb_resampler_fff_sptr self, int which) → float`  
`pc_input_buffers_full_avg(pfb_arb_resampler_fff_sptr self) -> pmt_vector_float`

`pfb_arb_resampler_fff_sptr.pc_noutput_items_avg(pfb_arb_resampler_fff_sptr self) → float`

`pfb_arb_resampler_fff_sptr.pc_nproduced_avg(pfb_arb_resampler_fff_sptr self) → float`

`pfb_arb_resampler_fff_sptr.pc_output_buffers_full_avg(pfb_arb_resampler_fff_sptr self, int which) → float`  
`pc_output_buffers_full_avg(pfb_arb_resampler_fff_sptr self) -> pmt_vector_float`

`pfb_arb_resampler_fff_sptr.pc_throughput_avg(pfb_arb_resampler_fff_sptr self) → float`

`pfb_arb_resampler_fff_sptr.pc_work_time_avg(pfb_arb_resampler_fff_sptr self) → float`

`pfb_arb_resampler_fff_sptr.pc_work_time_total(pfb_arb_resampler_fff_sptr self) → float`

`pfb_arb_resampler_fff_sptr.phase(pfb_arb_resampler_fff_sptr self) → float`  
Gets the current phase of the resampler in radians (2 to 2pi).

`pfb_arb_resampler_fff_sptr.phase_offset(pfb_arb_resampler_fff_sptr self, float freq, float fs) → float`  
Calculates the phase offset expected by a sine wave of frequency and sampling rate (assuming input sine wave has 0 degree phase).

`pfb_arb_resampler_fff_sptr.print_taps(pfb_arb_resampler_fff_sptr self)`  
Print all of the filterbank taps to screen.

`pfb_arb_resampler_fff_sptr.sample_delay(pfb_arb_resampler_fff_sptr self, int which) → unsigned`

int

`pfb_arb_resampler_fff_sptr.set_min_noutput_items(pfb_arb_resampler_fff_sptr self, int m)`

`pfb_arb_resampler_fff_sptr.set_phase(pfb_arb_resampler_fff_sptr self, float ph)`

Sets the current phase offset in radians (0 to 2pi).

`pfb_arb_resampler_fff_sptr.set_rate(pfb_arb_resampler_fff_sptr self, float rate)`

Sets the resampling rate of the block.

`pfb_arb_resampler_fff_sptr.set_taps(pfb_arb_resampler_fff_sptr self, pmt_vector_float taps)`

Resets the filterbank's filter taps with the new prototype filter

`pfb_arb_resampler_fff_sptr.set_thread_priority(pfb_arb_resampler_fff_sptr self, int priority) →`

int

`pfb_arb_resampler_fff_sptr.taps(pfb_arb_resampler_fff_sptr self) → std::vector< std::vector< float, std::allocator< float > >, std::allocator< std::vector< float, std::allocator< float > > > >`

Return a vector<vector<>> of the filterbank taps

`pfb_arb_resampler_fff_sptr.taps_per_filter(pfb_arb_resampler_fff_sptr self) → unsigned int`

Gets the number of taps per filter.

`pfb_arb_resampler_fff_sptr.thread_priority(pfb_arb_resampler_fff_sptr self) → int`

`gnuradio.filter.pfb_channelizer_ccf(unsigned int numchans, pmt_vector_float taps, float oversample_rate) → pfb_channelizer_ccf_sptr`

Polyphase filterbank channelizer with gr\_complex input, gr\_complex output and float taps.

This block takes in complex inputs and channelizes it to channels of equal bandwidth. Each of the resulting channels is decimated to the new rate that is the input sampling rate divided by the number of channels, .

The PFB channelizer code takes the taps generated above and builds a set of filters. The set contains filters and each filter contains  $\text{ceil}(\text{taps.size()}/\text{decim})$  taps. Each tap from the filter prototype is sequentially inserted into the next filter. When all of the input taps are used, the remaining filters in the filterbank are filled out with 0's to make sure each filter has the same number of taps.

Each filter operates using the `gr::blocks::fir_filter_XXX` class of GNU Radio, which takes the input stream at and performs the inner product calculation to where is the number of filter taps. To efficiently handle this in the GNU Radio structure, each filter input must come from its own input stream. So the channelizer must be provided with streams where the input stream has been deinterleaved. This is most easily done using the `gr::blocks::stream_to_streams` block.

The output is then produced as a vector, where index in the vector is the next sample from the *th* channel. This is most easily handled by sending the output to a `gr::blocks::vector_to_streams` block to handle the conversion and passing streams out.

The input and output formatting is done using a `hier_block2` called `pfb_channelizer_ccf`. This can take in a single stream and outputs streams based on the behavior described above.

The filter's taps should be based on the input sampling rate.

For example, using the GNU Radio's `firdes` utility to building filters, we build a low-pass filter with a sampling rate of , a 3-dB bandwidth of and a transition bandwidth of . We can also specify the out-of-band attenuation to use, , and the filter window function (a Blackman-harris window in this case). The first input is the gain of the filter, which we specify here as unity.

The filter output can also be oversampled. The oversampling rate is the ratio of the the actual output sampling rate to the normal output sampling rate. It must be rationally related to the number of channels as  $N/i$  for  $i$  in  $[1, N]$ , which gives an outputsample rate of  $[fs/N, fs]$  where  $fs$  is the input sample rate and  $N$  is the number of channels.

For example, for 6 channels with  $fs = 6000$  Hz, the normal rate is  $6000/6 = 1000$  Hz. Allowable oversampling rates are 6/6, 6/5, 6/4, 6/3, 6/2, and 6/1 where the output sample rate of a 6/1 oversample ratio is 6000 Hz, or 6 times the normal 1000 Hz. A rate of  $6/5 = 1.2$ , so the output rate would be 1200 Hz.

The theory behind this block can be found in Chapter 6 of the following book:

When dealing with oversampling, the above book is still a good reference along with this paper:

Constructor Specific Documentation:

Build the polyphase filterbank decimator. For example, for 6 channels with  $fs = 6000$  Hz, the normal rate is  $6000/6 = 1000$  Hz. Allowable oversampling rates are 6/6, 6/5, 6/4, 6/3, 6/2, and 6/1 where the output sample rate of a 6/1 oversample ratio is 6000 Hz, or 6 times the normal 1000 Hz.

**Parameters:**

- **numchans** – (unsigned integer) Specifies the number of channels
- **taps** – (vector/list of floats) The prototype filter to populate the filterbank.
- **oversample\_rate** – (float) The oversampling rate is the ratio of the the actual output sampling rate to the normal output sampling rate. It must be rationally related to the number of channels as  $N/i$  for  $i$  in  $[1,N]$ , which gives an outputsample rate of  $[fs/N, fs]$  where  $fs$  is the input sample rate and  $N$  is the number of channels.

`pfb_channelizer_ccf_sptr.active_thread_priority(pfb_channelizer_ccf_sptr self) → int`

`pfb_channelizer_ccf_sptr.channel_map(pfb_channelizer_ccf_sptr self) → std::vector<int, std::allocator<int>>`

Gets the current channel map.

`pfb_channelizer_ccf_sptr.declare_sample_delay(pfb_channelizer_ccf_sptr self, int which, int delay)`

`declare_sample_delay(pfb_channelizer_ccf_sptr self, unsigned int delay)`

`pfb_channelizer_ccf_sptr.message_subscribers(pfb_channelizer_ccf_sptr self, swig_int_ptr which_port) → swig_int_ptr`

`pfb_channelizer_ccf_sptr.min_noutput_items(pfb_channelizer_ccf_sptr self) → int`

`pfb_channelizer_ccf_sptr.pc_input_buffers_full_avg(pfb_channelizer_ccf_sptr self, int which) → float`

`pc_input_buffers_full_avg(pfb_channelizer_ccf_sptr self) → pmt_vector_float`

`pfb_channelizer_ccf_sptr.pc_noutput_items_avg(pfb_channelizer_ccf_sptr self) → float`

`pfb_channelizer_ccf_sptr.pc_nproduced_avg(pfb_channelizer_ccf_sptr self) → float`

`pfb_channelizer_ccf_sptr.pc_output_buffers_full_avg(pfb_channelizer_ccf_sptr self, int which) → float`

`pc_output_buffers_full_avg(pfb_channelizer_ccf_sptr self) → pmt_vector_float`

`pfb_channelizer_ccf_sptr.pc_throughput_avg(pfb_channelizer_ccf_sptr self) → float`

`pfb_channelizer_ccf_sptr.pc_work_time_avg(pfb_channelizer_ccf_sptr self) → float`

`pfb_channelizer_ccf_sptr.pc_work_time_total(pfb_channelizer_ccf_sptr self) → float`

`pfb_channelizer_ccf_sptr.print_taps(pfb_channelizer_ccf_sptr self)`

Print all of the filterbank taps to screen.

`pfb_channelizer_ccf_sptr.sample_delay(pfb_channelizer_ccf_sptr self, int which) → unsigned int`

`pfb_channelizer_ccf_sptr.set_channel_map(pfb_channelizer_ccf_sptr self, std::vector<int, std::allocator<int>> const & map)`

Set the channel map. Channels are numbers as:

So output stream 0 comes from channel 0, etc. Setting a new channel map allows the user to specify which channel in frequency he/she wants to go to which output stream.

The map should have the same number of elements as the number of output connections from the block. The minimum value of the map is 0 (for the 0th channel) and the maximum number is  $N-1$  where  $N$  is the number of channels.

We specify  $M$  as the number of output connections made where  $M \leq N$ , so only  $M$  out of  $N$  channels are driven to an output stream. The number of items in the channel map should be at least  $M$  long. If there are more channels specified, any value in the map over  $M-1$  will be ignored. If the size of the map is less than  $M$  the behavior is unknown (we don't wish to check every entry into the work function).

This means that if the channelizer is splitting the signal up into  $N$  channels but only  $M$  channels are specified in the map (where  $M \leq N$ ), then  $M$  output streams must be connected and the map and the channel numbers used must be less than  $N-1$ . Output channel number can be reused, too. By default, the map is  $[0...M-1]$  with  $M = N$ .

`pfb_channelizer_ccf_sptr.set_min_noutput_items(pfb_channelizer_ccf_sptr self, int m)`

`pfb_channelizer_ccf_sptr.set_taps(pfb_channelizer_ccf_sptr self, pmt_vector_float taps)`

Resets the filterbank's filter taps with the new prototype filter

`pfb_channelizer_ccf_sptr.set_thread_priority(pfb_channelizer_ccf_sptr self, int priority) → int`

`pfb_channelizer_ccf_sptr.taps(pfb_channelizer_ccf_sptr self) → std::vector<float, std::allocator<float>>, std::vector<float, std::allocator<float>>>`

Return a vector<vector<>> of the filterbank taps

```
pfb_channelizer_ccf_sptr.thread_priority(pfb_channelizer_ccf_sptr self) → int
```

```
gnuradio.filter.pfb_decimator_ccf(unsigned int decim, pmt_vector_float taps, unsigned int channel,  
bool use_fft_rotator=True, bool use_fft_filters=True) → pfb_decimator_ccf_sptr
```

Polyphase filterbank bandpass decimator with gr\_complex input, gr\_complex output and float taps.

This block takes in a signal stream and performs integer down- sampling (decimation) with a polyphase filterbank. The first input is the integer specifying how much to decimate by. The second input is a vector (Python list) of floating-point taps of the prototype filter. The third input specifies the channel to extract. By default, the zeroth channel is used, which is the baseband channel (first Nyquist zone).

The parameter specifies which channel to use since this class is capable of bandpass decimation. Given a complex input stream at a sampling rate of and a decimation rate of , the input frequency domain is split into channels that represent the Nyquist zones. Using the polyphase filterbank, we can select any one of these channels to decimate.

The output signal will be the basebanded and decimated signal from that channel. This concept is very similar to the PFB channelizer (see gr::filter::pfb\_channelizer\_ccf) where only a single channel is extracted at a time.

The filter's taps should be based on the sampling rate before decimation.

For example, using the GNU Radio's firdec utility to building filters, we build a low-pass filter with a sampling rate of , a 3-dB bandwidth of and a transition bandwidth of . We can also specify the out-of-band attenuation to use, , and the filter window function (a Blackman-harris window in this case). The first input is the gain of the filter, which we specify here as unity.

The PFB decimator code takes the taps generated above and builds a set of filters. The set contains filters and each filter contains ceil(taps.size()/decim) taps. Each tap from the filter prototype is sequentially inserted into the next filter. When all of the input taps are used, the remaining filters in the filterbank are filled out with 0's to make sure each filter has the same number of taps.

The theory behind this block can be found in Chapter 6 of the following book:

Constructor Specific Documentation:

Build the polyphase filterbank decimator.

**Parameters:**

- **decim** – (unsigned integer) Specifies the decimation rate to use
- **taps** – (vector/list of floats) The prototype filter to populate the filterbank.
- **channel** – (unsigned integer) Selects the channel to return [default=0].
- **use\_fft\_rotator** – (bool) Rotate channels using FFT method instead of exp(phi). For larger values of , the FFT method will perform better. Generally, this value of is small (~5), but could be architecture-specific (Default: true).
- **use\_fft\_filters** – (bool) Use FFT filters (fast convolution) instead of FIR filters. FFT filters perform better for larger numbers of taps but is architecture-specific (Default: true).

```
pfb_decimator_ccf_sptr.active_thread_priority(pfb_decimator_ccf_sptr self) → int
```

```
pfb_decimator_ccf_sptr.declare_sample_delay(pfb_decimator_ccf_sptr self, int which, int delay)  
declare_sample_delay(pfb_decimator_ccf_sptr self, unsigned int delay)
```

```
pfb_decimator_ccf_sptr.message_subscribers(pfb_decimator_ccf_sptr self, swig_int_ptr  
which_port) → swig_int_ptr
```

```
pfb_decimator_ccf_sptr.min_noutput_items(pfb_decimator_ccf_sptr self) → int
```

```
pfb_decimator_ccf_sptr.pc_input_buffers_full_avg(pfb_decimator_ccf_sptr self, int which) →  
float
```

```
pc_input_buffers_full_avg(pfb_decimator_ccf_sptr self) -> pmt_vector_float
```

```
pfb_decimator_ccf_sptr.pc_noutput_items_avg(pfb_decimator_ccf_sptr self) → float
```

```
pfb_decimator_ccf_sptr.pc_nproduced_avg(pfb_decimator_ccf_sptr self) → float
```

```
pfb_decimator_ccf_sptr.pc_output_buffers_full_avg(pfb_decimator_ccf_sptr self, int which) →  
float
```

```
pc_output_buffers_full_avg(pfb_decimator_ccf_sptr self) -> pmt_vector_float
```

```
pfb_decimator_ccf_sptr.pc_throughput_avg(pfb_decimator_ccf_sptr self) → float
```

```
pfb_decimator_ccf_sptr.pc_work_time_avg(pfb_decimator_ccf_sptr self) → float
```

```
pfb_decimator_ccf_sptr.pc_work_time_total(pfb_decimator_ccf_sptr self) → float
```

```
pfb_decimator_ccf_sptr.print_taps(pfb_decimator_ccf_sptr self)
```

Print all of the filterbank taps to screen.

```
pfb_decimator_ccf_sptr.sample_delay(pfb_decimator_ccf_sptr self, int which) → unsigned int
```

```
pfb_decimator_ccf_sptr.set_channel(pfb_decimator_ccf_sptr self, unsigned int const channel)
```

```
pfb_decimator_ccf_sptr.set_min_noutput_items(pfb_decimator_ccf_sptr self, int m)
```

```
pfb_decimator_ccf_sptr.set_taps(pfb_decimator_ccf_sptr self, pmt_vector_float taps)
```

Resets the filterbank's filter taps with the new prototype filter

```
pfb_decimator_ccf_sptr.set_thread_priority(pfb_decimator_ccf_sptr self, int priority) → int
```

```
pfb_decimator_ccf_sptr.taps(pfb_decimator_ccf_sptr self) → std::vector< float, std::allocator< float > >, std::vector< float, std::allocator< float > > >
```

Return a vector<vector<>> of the filterbank taps

```
pfb_decimator_ccf_sptr.thread_priority(pfb_decimator_ccf_sptr self) → int
```

```
gnuradio.filter.pfb_interpolator_ccf(unsigned int interp, pmt_vector_float taps) →  
pfb_interpolator_ccf_sptr
```

Polyphase filterbank interpolator with gr\_complex input, gr\_complex output and float taps.

This block takes in a signal stream and performs integer up- sampling (interpolation) with a polyphase filterbank. The first input is the integer specifying how much to interpolate by. The second input is a vector (Python list) of floating-point taps of the prototype filter.

The filter's taps should be based on the interpolation rate specified. That is, the bandwidth specified is relative to the bandwidth after interpolation.

For example, using the GNU Radio's firdec utility to building filters, we build a low-pass filter with a sampling rate of , a 3-dB bandwidth of and a transition bandwidth of . We can also specify the out-of-band attenuation to use, ATT, and the filter window function (a Blackman-harris window in this case). The first input is the gain, which is also specified as the interpolation rate so that the output levels are the same as the input (this creates an overall increase in power).

The PFB interpolator code takes the taps generated above and builds a set of filters. The set contains filters and each filter contains ceil(taps.size()/interp) taps. Each tap from the filter prototype is sequentially inserted into the next filter. When all of the input taps are used, the remaining filters in the filterbank are filled out with 0's to make sure each filter has the same number of taps.

The theory behind this block can be found in Chapter 7.1 of the following book:

Constructor Specific Documentation:

Build the polyphase filterbank interpolator.

**Parameters:**

- **interp** – (unsigned integer) Specifies the interpolation rate to use
- **taps** – (vector/list of floats) The prototype filter to populate the filterbank. The taps should be generated at the interpolated sampling rate.

```
pfb_interpolator_ccf_sptr.active_thread_priority(pfb_interpolator_ccf_sptr self) → int
```

```
pfb_interpolator_ccf_sptr.declare_sample_delay(pfb_interpolator_ccf_sptr self, int which, int delay)
```

declare\_sample\_delay(pfb\_interpolator\_ccf\_sptr self, unsigned int delay)

```
pfb_interpolator_ccf_sptr.message_subscribers(pfb_interpolator_ccf_sptr self, swig_int_ptr which_port) → swig_int_ptr
```

```
pfb_interpolator_ccf_sptr.min_noutput_items(pfb_interpolator_ccf_sptr self) → int
```

```
pfb_interpolator_ccf_sptr.pc_input_buffers_full_avg(pfb_interpolator_ccf_sptr self, int which) → float
```

pc\_input\_buffers\_full\_avg(pfb\_interpolator\_ccf\_sptr self) -> pmt\_vector\_float

```
pfb_interpolator_ccf_sptr.pc_noutput_items_avg(pfb_interpolator_ccf_sptr self) → float
```

```
pfb_interpolator_ccf_sptr.pc_nproduced_avg(pfb_interpolator_ccf_sptr self) → float
```

```
pfb_interpolator_ccf_sptr.pc_output_buffers_full_avg(pfb_interpolator_ccf_sptr self, int which) → float
```

```

pc_output_buffers_full_avg(pfb_interpolator_ccf_sptr self) -> pmt_vector_float

pfb_interpolator_ccf_sptr.pc_throughput_avg(pfb_interpolator_ccf_sptr self) -> float

pfb_interpolator_ccf_sptr.pc_work_time_avg(pfb_interpolator_ccf_sptr self) -> float

pfb_interpolator_ccf_sptr.pc_work_time_total(pfb_interpolator_ccf_sptr self) -> float

pfb_interpolator_ccf_sptr.print_taps(pfb_interpolator_ccf_sptr self)
    Print all of the filterbank taps to screen.

pfb_interpolator_ccf_sptr.sample_delay(pfb_interpolator_ccf_sptr self, int which) -> unsigned int

pfb_interpolator_ccf_sptr.set_min_noutput_items(pfb_interpolator_ccf_sptr self, int m)

pfb_interpolator_ccf_sptr.set_taps(pfb_interpolator_ccf_sptr self, pmt_vector_float taps)
    Resets the filterbank's filter taps with the new prototype filter

pfb_interpolator_ccf_sptr.set_thread_priority(pfb_interpolator_ccf_sptr self, int priority) -> int

pfb_interpolator_ccf_sptr.taps(pfb_interpolator_ccf_sptr self) -> std::vector< float, std::allocator< float > >, std::vector< float, std::allocator< float > > >
    Return a vector<vector<>> of the filterbank taps

pfb_interpolator_ccf_sptr.thread_priority(pfb_interpolator_ccf_sptr self) -> int

```

gnuradio.filter.**pfb\_synthesizer\_ccf**(unsigned int numchans, pmt\_vector\_float taps, bool twox=False) -> pfb\_synthesizer\_ccf\_sptr

Polyphase synthesis filterbank with gr\_complex input, gr\_complex output and float taps.

The PFB synthesis filterbank combines multiple baseband signals into a single channelized signal. Each input stream is, essentially, modulated onto an output channel according to the channel mapping (see set\_channel\_map for details).

Setting this filterbank up means selecting the number of output channels, the prototype filter, and whether to handle channels at 2x the sample rate (this is generally used only for reconstruction filtering).

The number of channels sets the maximum number of channels to use, but not all input streams must be connected. For M total channels, we can connect inputs 0 to N where N < M-1. Because of the way GNU Radio handles stream connections, we must connect the channels consecutively, and so we must use the set\_channel\_map if the desired output channels are not the same as the default mapping. This feature gives us the flexibility to output to any given channel. Generally, we try to not use the channels at the edge of the spectrum to avoid issues with filtering and roll-off of the transmitter or receiver.

When using the 2x sample rate mode, we specify the number of channels that will be used. However, the actual output signal will be twice this number of channels. This is mainly important to know when setting the channel map. For M channels, the channel mapping can specify from 0 to 2M-1 channels to output onto.

For more details about this and the concepts of reconstruction filtering, see:

Constructor Specific Documentation:

Build the polyphase synthesis filterbank.

**Parameters:**

- **numchans** – (unsigned integer) Specifies the number of channels
- **taps** – (vector/list of floats) The prototype filter to populate the filterbank.
- **twox** – (bool) use 2x oversampling or not (default is no)

```

pfb_synthesizer_ccf_sptr.active_thread_priority(pfb_synthesizer_ccf_sptr self) -> int

pfb_synthesizer_ccf_sptr.channel_map(pfb_synthesizer_ccf_sptr self) -> std::vector< int, std::allocator< int > >
    Gets the current channel map.

pfb_synthesizer_ccf_sptr.declare_sample_delay(pfb_synthesizer_ccf_sptr self, int which, int delay)
    declare_sample_delay(pfb_synthesizer_ccf_sptr self, unsigned int delay)

pfb_synthesizer_ccf_sptr.message_subscribers(pfb_synthesizer_ccf_sptr self, swig_int_ptr which_port) -> swig_int_ptr

pfb_synthesizer_ccf_sptr.min_noutput_items(pfb_synthesizer_ccf_sptr self) -> int

pfb_synthesizer_ccf_sptr.pc_input_buffers_full_avg(pfb_synthesizer_ccf_sptr self, int which) -> float

```

```

pc_input_buffers_full_avg(pfb_synthesizer_ccf_sptr self) -> pmt_vector_float

pfb_synthesizer_ccf_sptr.pc_noutput_items_avg(pfb_synthesizer_ccf_sptr self) -> float

pfb_synthesizer_ccf_sptr.pc_nproduced_avg(pfb_synthesizer_ccf_sptr self) -> float

pfb_synthesizer_ccf_sptr.pc_output_buffers_full_avg(pfb_synthesizer_ccf_sptr self, int which)
-> float
    pc_output_buffers_full_avg(pfb_synthesizer_ccf_sptr self) -> pmt_vector_float

pfb_synthesizer_ccf_sptr.pc_throughput_avg(pfb_synthesizer_ccf_sptr self) -> float

pfb_synthesizer_ccf_sptr.pc_work_time_avg(pfb_synthesizer_ccf_sptr self) -> float

pfb_synthesizer_ccf_sptr.pc_work_time_total(pfb_synthesizer_ccf_sptr self) -> float

pfb_synthesizer_ccf_sptr.print_taps(pfb_synthesizer_ccf_sptr self)
    Print all of the filterbank taps to screen.

```

```

pfb_synthesizer_ccf_sptr.sample_delay(pfb_synthesizer_ccf_sptr self, int which) -> unsigned int

pfb_synthesizer_ccf_sptr.set_channel_map(pfb_synthesizer_ccf_sptr self, std::vector< int,
std::allocator< int > > const & map)

```

Set the channel map. Channels are numbers as:

So input stream 0 goes to channel 0, etc. Setting a new channel map allows the user to specify where in frequency he/she wants the input stream to go. This is especially useful to avoid putting signals into the channels on the edge of the spectrum which can either wrap around (in the case of odd number of channels) and be affected by filter rolloff in the transmitter.

The map must be at least the number of streams being sent to the block. Less and the algorithm will not have enough data to properly setup the buffers. Any more channels specified will be ignored.

```

pfb_synthesizer_ccf_sptr.set_min_noutput_items(pfb_synthesizer_ccf_sptr self, int m)

pfb_synthesizer_ccf_sptr.set_taps(pfb_synthesizer_ccf_sptr self, pmt_vector_float taps)
    Resets the filterbank's filter taps with the new prototype filter

pfb_synthesizer_ccf_sptr.set_thread_priority(pfb_synthesizer_ccf_sptr self, int priority) -> int

pfb_synthesizer_ccf_sptr.taps(pfb_synthesizer_ccf_sptr self) -> std::vector< std::vector<
float, std::allocator< float > >, std::allocator< std::vector< float, std::allocator< float > > > >

```

Return a vector<vector<>> of the filterbank taps

```

pfb_synthesizer_ccf_sptr.thread_priority(pfb_synthesizer_ccf_sptr self) -> int

```

```

gnuradio.filter.rational_resampler_base_ccc(unsigned int interpolation, unsigned int decimation,
pmt_vector_cfloat taps) -> rational_resampler_base_ccc_sptr

```

Rational Resampling Polyphase FIR filter with gr\_complex input, gr\_complex output and gr\_complex taps.

Make a rational resampling FIR filter. If the input signal is at rate fs, then the output signal will be at a rate of \* fs / .

The interpolation and decimation rates should be kept as small as possible, and generally should be relatively prime to help reduce complexity in memory and computation.

The set of supplied to this filterbank should be designed around the resampling amount and must avoid aliasing (when interpolation/decimation < 1) and images (when interpolation/decimation > 1).

As with any filter, the behavior of the filter taps (or coefficients) is determined by the highest sampling rate that the filter will ever see. In the case of a resampler that increases the sampling rate, the highest sampling rate observed is since in the filterbank, the number of filter arms is equal to . When the resampler decreases the sampling rate (decimation > interpolation), then the highest rate is the input sample rate of the original signal. We must create a filter based around this value to reduce any aliasing that may occur from out-of-band signals.

Another way to think about how to create the filter taps is that the filter is effectively applied after interpolation and before decimation. And yet another way to think of it is that the taps should be a LPF that is at least as narrow as the narrower of the required anti-image postfilter or anti-alias prefilter.

Constructor Specific Documentation:

Make a rational resampling FIR filter.

- Parameters:**
- **interpolation** – The integer interpolation rate of the filter
  - **decimation** – The integer decimation rate of the filter
  - **taps** – The filter taps to control images and aliases

`rational_resampler_base_ccc_sptr. active_thread_priority(rational_resampler_base_ccc_sptr self) → int`

`rational_resampler_base_ccc_sptr. decimation(rational_resampler_base_ccc_sptr self) → unsigned int`

`rational_resampler_base_ccc_sptr. declare_sample_delay(rational_resampler_base_ccc_sptr self, int which, int delay)`

`declare_sample_delay(rational_resampler_base_ccc_sptr self, unsigned int delay)`

`rational_resampler_base_ccc_sptr. interpolation(rational_resampler_base_ccc_sptr self) → unsigned int`

`rational_resampler_base_ccc_sptr. message_subscribers(rational_resampler_base_ccc_sptr self, swig_int_ptr which_port) → swig_int_ptr`

`rational_resampler_base_ccc_sptr. min_noutput_items(rational_resampler_base_ccc_sptr self) → int`

`rational_resampler_base_ccc_sptr. pc_input_buffers_full_avg(rational_resampler_base_ccc_sptr self, int which) → float`

`pc_input_buffers_full_avg(rational_resampler_base_ccc_sptr self) -> pmt_vector_float`

`rational_resampler_base_ccc_sptr. pc_noutput_items_avg(rational_resampler_base_ccc_sptr self) → float`

`rational_resampler_base_ccc_sptr. pc_nproduced_avg(rational_resampler_base_ccc_sptr self) → float`

`rational_resampler_base_ccc_sptr. pc_output_buffers_full_avg(rational_resampler_base_ccc_sptr self, int which) → float`

`pc_output_buffers_full_avg(rational_resampler_base_ccc_sptr self) -> pmt_vector_float`

`rational_resampler_base_ccc_sptr. pc_throughput_avg(rational_resampler_base_ccc_sptr self) → float`

`rational_resampler_base_ccc_sptr. pc_work_time_avg(rational_resampler_base_ccc_sptr self) → float`

`rational_resampler_base_ccc_sptr. pc_work_time_total(rational_resampler_base_ccc_sptr self) → float`

`rational_resampler_base_ccc_sptr. sample_delay(rational_resampler_base_ccc_sptr self, int which) → unsigned int`

`rational_resampler_base_ccc_sptr. set_min_noutput_items(rational_resampler_base_ccc_sptr self, int m)`

`rational_resampler_base_ccc_sptr. set_taps(rational_resampler_base_ccc_sptr self, pmt_vector_cfloat taps)`

`rational_resampler_base_ccc_sptr. set_thread_priority(rational_resampler_base_ccc_sptr self, int priority) → int`

`rational_resampler_base_ccc_sptr. taps(rational_resampler_base_ccc_sptr self) → pmt_vector_cfloat`

`rational_resampler_base_ccc_sptr. thread_priority(rational_resampler_base_ccc_sptr self) → int`

`gnuradio.filter. rational_resampler_base_ccf(unsigned int interpolation, unsigned int decimation, pmt_vector_float taps) → rational_resampler_base_ccf_sptr`

Rational Resampling Polyphase FIR filter with gr\_complex input, gr\_complex output and float taps.

Make a rational resampling FIR filter. If the input signal is at rate fs, then the output signal will be at a rate of \* fs / .

The interpolation and decimation rates should be kept as small as possible, and generally should be relatively prime to help reduce complexity in memory and computation.

The set of supplied to this filterbank should be designed around the resampling amount and must avoid aliasing (when interpolation/decimation < 1) and images (when interpolation/decimation > 1).

As with any filter, the behavior of the filter taps (or coefficients) is determined by the highest sampling rate that the filter will ever see. In the case of a resampler that increases the sampling rate, the highest sampling rate observed is since in the filterbank, the number of filter arms is equal to . When the resampler decreases the sampling rate (decimation > interpolation), then the highest rate is the input sample rate of the original signal. We must create a filter based around this value to reduce any aliasing that may occur from out-of-band signals.

Another way to think about how to create the filter taps is that the filter is effectively applied after interpolation and before decimation. And yet another way to think of it is that the taps should be a LPF that is at least as narrow as the narrower of the required anti-image postfilter or anti-alias prefilter.

Constructor Specific Documentation:

Make a rational resampling FIR filter.

**Parameters:**

- **interpolation** – The integer interpolation rate of the filter
- **decimation** – The integer decimation rate of the filter
- **taps** – The filter taps to control images and aliases

`rational_resampler_base_ccf_sptr.active_thread_priority(rational_resampler_base_ccf_sptr self) → int`

`rational_resampler_base_ccf_sptr.decimation(rational_resampler_base_ccf_sptr self) → unsigned int`

`rational_resampler_base_ccf_sptr.declare_sample_delay(rational_resampler_base_ccf_sptr self, int which, int delay)`

`declare_sample_delay(rational_resampler_base_ccf_sptr self, unsigned int delay)`

`rational_resampler_base_ccf_sptr.interpolation(rational_resampler_base_ccf_sptr self) → unsigned int`

`rational_resampler_base_ccf_sptr.message_subscribers(rational_resampler_base_ccf_sptr self, swig_int_ptr which_port) → swig_int_ptr`

`rational_resampler_base_ccf_sptr.min_noutput_items(rational_resampler_base_ccf_sptr self) → int`

`rational_resampler_base_ccf_sptr.pc_input_buffers_full_avg(rational_resampler_base_ccf_sptr self, int which) → float`

`pc_input_buffers_full_avg(rational_resampler_base_ccf_sptr self) -> pmt_vector_float`

`rational_resampler_base_ccf_sptr.pc_noutput_items_avg(rational_resampler_base_ccf_sptr self) → float`

`rational_resampler_base_ccf_sptr.pc_nproduced_avg(rational_resampler_base_ccf_sptr self) → float`

`rational_resampler_base_ccf_sptr.pc_output_buffers_full_avg(rational_resampler_base_ccf_sptr self, int which) → float`

`pc_output_buffers_full_avg(rational_resampler_base_ccf_sptr self) -> pmt_vector_float`

`rational_resampler_base_ccf_sptr.pc_throughput_avg(rational_resampler_base_ccf_sptr self) → float`

`rational_resampler_base_ccf_sptr.pc_work_time_avg(rational_resampler_base_ccf_sptr self) → float`

`rational_resampler_base_ccf_sptr.pc_work_time_total(rational_resampler_base_ccf_sptr self) → float`

`rational_resampler_base_ccf_sptr.sample_delay(rational_resampler_base_ccf_sptr self, int which) → unsigned int`

`rational_resampler_base_ccf_sptr.set_min_noutput_items(rational_resampler_base_ccf_sptr self, int m)`

`rational_resampler_base_ccf_sptr.set_taps(rational_resampler_base_ccf_sptr self, pmt_vector_float taps)`

`rational_resampler_base_ccf_sptr.set_thread_priority(rational_resampler_base_ccf_sptr self, int priority) → int`

`rational_resampler_base_ccf_sptr.taps(rational_resampler_base_ccf_sptr self) → pmt_vector_float`

```

rational_resampler_base_ccf_sptr.thread_priority(rational_resampler_base_ccf_sptr self) → int

gnuradio.filter.rational_resampler_base_fcc(unsigned int interpolation, unsigned int decimation,
pmt_vector_cfloat taps) → rational_resampler_base_fcc_sptr

```

Rational Resampling Polyphase FIR filter with float input, gr\_complex output and gr\_complex taps.

Make a rational resampling FIR filter. If the input signal is at rate  $fs$ , then the output signal will be at a rate of  $* fs /$ .

The interpolation and decimation rates should be kept as small as possible, and generally should be relatively prime to help reduce complexity in memory and computation.

The set of supplied to this filterbank should be designed around the resampling amount and must avoid aliasing (when interpolation/decimation < 1) and images (when interpolation/decimation > 1).

As with any filter, the behavior of the filter taps (or coefficients) is determined by the highest sampling rate that the filter will ever see. In the case of a resampler that increases the sampling rate, the highest sampling rate observed is since in the filterbank, the number of filter arms is equal to . When the resampler decreases the sampling rate (decimation > interpolation), then the highest rate is the input sample rate of the original signal. We must create a filter based around this value to reduce any aliasing that may occur from out-of-band signals.

Another way to think about how to create the filter taps is that the filter is effectively applied after interpolation and before decimation. And yet another way to think of it is that the taps should be a LPF that is at least as narrow as the narrower of the required anti-image postfilter or anti-alias prefilter.

Constructor Specific Documentation:

Make a rational resampling FIR filter.

**Parameters:**

- **interpolation** – The integer interpolation rate of the filter
- **decimation** – The integer decimation rate of the filter
- **taps** – The filter taps to control images and aliases

```

rational_resampler_base_fcc_sptr.active_thread_priority(rational_resampler_base_fcc_sptr self) → int

```

```

rational_resampler_base_fcc_sptr.decimation(rational_resampler_base_fcc_sptr self) → unsigned int

```

```

rational_resampler_base_fcc_sptr.declare_sample_delay(rational_resampler_base_fcc_sptr self, int which, int delay)
    declare_sample_delay(rational_resampler_base_fcc_sptr self, unsigned int delay)

```

```

rational_resampler_base_fcc_sptr.interpolation(rational_resampler_base_fcc_sptr self) → unsigned int

```

```

rational_resampler_base_fcc_sptr.message_subscribers(rational_resampler_base_fcc_sptr self, swig_int_ptr which_port) → swig_int_ptr

```

```

rational_resampler_base_fcc_sptr.min_noutput_items(rational_resampler_base_fcc_sptr self) → int

```

```

rational_resampler_base_fcc_sptr.pc_input_buffers_full_avg(rational_resampler_base_fcc_sptr self, int which) → float
    pc_input_buffers_full_avg(rational_resampler_base_fcc_sptr self) -> pmt_vector_float

```

```

rational_resampler_base_fcc_sptr.pc_noutput_items_avg(rational_resampler_base_fcc_sptr self) → float

```

```

rational_resampler_base_fcc_sptr.pc_nproduced_avg(rational_resampler_base_fcc_sptr self) → float

```

```

rational_resampler_base_fcc_sptr.pc_output_buffers_full_avg(rational_resampler_base_fcc_sptr self, int which) → float
    pc_output_buffers_full_avg(rational_resampler_base_fcc_sptr self) -> pmt_vector_float

```

```

rational_resampler_base_fcc_sptr.pc_throughput_avg(rational_resampler_base_fcc_sptr self) → float

```

```

rational_resampler_base_fcc_sptr.pc_work_time_avg(rational_resampler_base_fcc_sptr self) → float

```

```

rational_resampler_base_fcc_sptr.pc_work_time_total(rational_resampler_base_fcc_sptr self) → float

```

```
rational_resampler_base_fcc_sptr.sample_delay(rational_resampler_base_fcc_sptr self, int which) → unsigned int
```

```
rational_resampler_base_fcc_sptr.set_min_noutput_items(rational_resampler_base_fcc_sptr self, int m)
```

```
rational_resampler_base_fcc_sptr.set_taps(rational_resampler_base_fcc_sptr self, pmt_vector_cfloat taps)
```

```
rational_resampler_base_fcc_sptr.set_thread_priority(rational_resampler_base_fcc_sptr self, int priority) → int
```

```
rational_resampler_base_fcc_sptr.taps(rational_resampler_base_fcc_sptr self) → pmt_vector_cfloat
```

```
rational_resampler_base_fcc_sptr.thread_priority(rational_resampler_base_fcc_sptr self) → int
```

```
gnuradio.filter.rational_resampler_base_fff(unsigned int interpolation, unsigned int decimation, pmt_vector_float taps) → rational_resampler_base_fff_sptr
```

Rational Resampling Polyphase FIR filter with float input, float output and float taps.

Make a rational resampling FIR filter. If the input signal is at rate  $f_s$ , then the output signal will be at a rate of  $f_s /$ .

The interpolation and decimation rates should be kept as small as possible, and generally should be relatively prime to help reduce complexity in memory and computation.

The set of supplied to this filterbank should be designed around the resampling amount and must avoid aliasing (when interpolation/decimation  $< 1$ ) and images (when interpolation/decimation  $> 1$ ).

As with any filter, the behavior of the filter taps (or coefficients) is determined by the highest sampling rate that the filter will ever see. In the case of a resampler that increases the sampling rate, the highest sampling rate observed is since in the filterbank, the number of filter arms is equal to . When the resampler decreases the sampling rate (decimation  $>$  interpolation), then the highest rate is the input sample rate of the original signal. We must create a filter based around this value to reduce any aliasing that may occur from out-of-band signals.

Another way to think about how to create the filter taps is that the filter is effectively applied after interpolation and before decimation. And yet another way to think of it is that the taps should be a LPF that is at least as narrow as the narrower of the required anti-image postfilter or anti-alias prefilter.

Constructor Specific Documentation:

Make a rational resampling FIR filter.

**Parameters:**

- **interpolation** – The integer interpolation rate of the filter
- **decimation** – The integer decimation rate of the filter
- **taps** – The filter taps to control images and aliases

```
rational_resampler_base_fff_sptr.active_thread_priority(rational_resampler_base_fff_sptr self) → int
```

```
rational_resampler_base_fff_sptr.decimation(rational_resampler_base_fff_sptr self) → unsigned int
```

```
rational_resampler_base_fff_sptr.declare_sample_delay(rational_resampler_base_fff_sptr self, int which, int delay)
```

```
declare_sample_delay(rational_resampler_base_fff_sptr self, unsigned int delay)
```

```
rational_resampler_base_fff_sptr.interpolation(rational_resampler_base_fff_sptr self) → unsigned int
```

```
rational_resampler_base_fff_sptr.message_subscribers(rational_resampler_base_fff_sptr self, swig_int_ptr which_port) → swig_int_ptr
```

```
rational_resampler_base_fff_sptr.min_noutput_items(rational_resampler_base_fff_sptr self) → int
```

```
rational_resampler_base_fff_sptr.pc_input_buffers_full_avg(rational_resampler_base_fff_sptr self, int which) → float
```

```
pc_input_buffers_full_avg(rational_resampler_base_fff_sptr self) -> pmt_vector_float
```

```
rational_resampler_base_fff_sptr.pc_noutput_items_avg(rational_resampler_base_fff_sptr self) → float
```

```
rational_resampler_base_fff_sptr.pc_nproduced_avg(rational_resampler_base_fff_sptr self) →
```

float

`rational_resampler_base_ffs_ptr.pc_output_buffers_full_avg(rational_resampler_base_ffs_ptr self, int which) → float`

`pc_output_buffers_full_avg(rational_resampler_base_ffs_ptr self) → pmt_vector_float`

`rational_resampler_base_ffs_ptr.pc_throughput_avg(rational_resampler_base_ffs_ptr self) → float`

`rational_resampler_base_ffs_ptr.pc_work_time_avg(rational_resampler_base_ffs_ptr self) → float`

`rational_resampler_base_ffs_ptr.pc_work_time_total(rational_resampler_base_ffs_ptr self) → float`

`rational_resampler_base_ffs_ptr.sample_delay(rational_resampler_base_ffs_ptr self, int which) → unsigned int`

`rational_resampler_base_ffs_ptr.set_min_noutput_items(rational_resampler_base_ffs_ptr self, int m)`

`rational_resampler_base_ffs_ptr.set_taps(rational_resampler_base_ffs_ptr self, pmt_vector_float taps)`

`rational_resampler_base_ffs_ptr.set_thread_priority(rational_resampler_base_ffs_ptr self, int priority) → int`

`rational_resampler_base_ffs_ptr.taps(rational_resampler_base_ffs_ptr self) → pmt_vector_float`

`rational_resampler_base_ffs_ptr.thread_priority(rational_resampler_base_ffs_ptr self) → int`

`gnuradio.filter.rational_resampler_base_fsf(unsigned int interpolation, unsigned int decimation, pmt_vector_float taps) → rational_resampler_base_fsf_ptr`

Rational Resampling Polyphase FIR filter with float input, short output and float taps.

Make a rational resampling FIR filter. If the input signal is at rate  $f_s$ , then the output signal will be at a rate of  $f_s /$ .

The interpolation and decimation rates should be kept as small as possible, and generally should be relatively prime to help reduce complexity in memory and computation.

The set of supplied to this filterbank should be designed around the resampling amount and must avoid aliasing (when interpolation/decimation  $< 1$ ) and images (when interpolation/decimation  $> 1$ ).

As with any filter, the behavior of the filter taps (or coefficients) is determined by the highest sampling rate that the filter will ever see. In the case of a resampler that increases the sampling rate, the highest sampling rate observed is since in the filterbank, the number of filter arms is equal to . When the resampler decreases the sampling rate (decimation  $>$  interpolation), then the highest rate is the input sample rate of the original signal. We must create a filter based around this value to reduce any aliasing that may occur from out-of-band signals.

Another way to think about how to create the filter taps is that the filter is effectively applied after interpolation and before decimation. And yet another way to think of it is that the taps should be a LPF that is at least as narrow as the narrower of the required anti-image postfilter or anti-alias prefilter.

Constructor Specific Documentation:

Make a rational resampling FIR filter.

**Parameters:**

- **interpolation** – The integer interpolation rate of the filter
- **decimation** – The integer decimation rate of the filter
- **taps** – The filter taps to control images and aliases

`rational_resampler_base_fsf_ptr.active_thread_priority(rational_resampler_base_fsf_ptr self) → int`

`rational_resampler_base_fsf_ptr.decimation(rational_resampler_base_fsf_ptr self) → unsigned int`

`rational_resampler_base_fsf_ptr.declare_sample_delay(rational_resampler_base_fsf_ptr self, int which, int delay)`

`declare_sample_delay(rational_resampler_base_fsf_ptr self, unsigned int delay)`

`rational_resampler_base_fsf_ptr.interpolation(rational_resampler_base_fsf_ptr self) → unsigned int`

```

rational_resampler_base_fsf_sptr.message_subscribers(rational_resampler_base_fsf_sptr self,
swig_int_ptr which_port) → swig_int_ptr

rational_resampler_base_fsf_sptr.min_noutput_items(rational_resampler_base_fsf_sptr self) →
int

rational_resampler_base_fsf_sptr.pc_input_buffers_full_avg(rational_resampler_base_fsf_sptr
self, int which) → float
    pc_input_buffers_full_avg(rational_resampler_base_fsf_sptr self) -> pmt_vector_float

rational_resampler_base_fsf_sptr.pc_noutput_items_avg(rational_resampler_base_fsf_sptr self)
→ float

rational_resampler_base_fsf_sptr.pc_nproduced_avg(rational_resampler_base_fsf_sptr self) →
float

rational_resampler_base_fsf_sptr.pc_output_buffers_full_avg(rational_resampler_base_fsf_sptr
self, int which) → float
    pc_output_buffers_full_avg(rational_resampler_base_fsf_sptr self) -> pmt_vector_float

rational_resampler_base_fsf_sptr.pc_throughput_avg(rational_resampler_base_fsf_sptr self) →
float

rational_resampler_base_fsf_sptr.pc_work_time_avg(rational_resampler_base_fsf_sptr self) →
float

rational_resampler_base_fsf_sptr.pc_work_time_total(rational_resampler_base_fsf_sptr self) →
float

rational_resampler_base_fsf_sptr.sample_delay(rational_resampler_base_fsf_sptr self, int which) →
unsigned int

rational_resampler_base_fsf_sptr.set_min_noutput_items(rational_resampler_base_fsf_sptr self,
int m)

rational_resampler_base_fsf_sptr.set_taps(rational_resampler_base_fsf_sptr self, pmt_vector_float
taps)

rational_resampler_base_fsf_sptr.set_thread_priority(rational_resampler_base_fsf_sptr self, int
priority) → int

rational_resampler_base_fsf_sptr.taps(rational_resampler_base_fsf_sptr self) → pmt_vector_float

rational_resampler_base_fsf_sptr.thread_priority(rational_resampler_base_fsf_sptr self) → int

```

`gnuradio.filter.rational_resampler_base_scc(unsigned int interpolation, unsigned int decimation,
pmt_vector_cfloat taps) → rational_resampler_base_scc_sptr`

Rational Resampling Polyphase FIR filter with short input, gr\_complex output and gr\_complex taps.

Make a rational resampling FIR filter. If the input signal is at rate  $fs$ , then the output signal will be at a rate of  $* fs /$ .

The interpolation and decimation rates should be kept as small as possible, and generally should be relatively prime to help reduce complexity in memory and computation.

The set of supplied to this filterbank should be designed around the resampling amount and must avoid aliasing (when interpolation/decimation  $< 1$ ) and images (when interpolation/decimation  $> 1$ ).

As with any filter, the behavior of the filter taps (or coefficients) is determined by the highest sampling rate that the filter will ever see. In the case of a resampler that increases the sampling rate, the highest sampling rate observed is since in the filterbank, the number of filter arms is equal to . When the resampler decreases the sampling rate (decimation  $>$  interpolation), then the highest rate is the input sample rate of the original signal. We must create a filter based around this value to reduce any aliasing that may occur from out-of-band signals.

Another way to think about how to create the filter taps is that the filter is effectively applied after interpolation and before decimation. And yet another way to think of it is that the taps should be a LPF that is at least as narrow as the narrower of the required anti-image postfilter or anti-alias prefilter.

Constructor Specific Documentation:

Make a rational resampling FIR filter.

- Parameters:**
- **interpolation** – The integer interpolation rate of the filter
  - **decimation** – The integer decimation rate of the filter
  - **taps** – The filter taps to control images and aliases

`rational_resampler_base_scc_sptr.active_thread_priority(rational_resampler_base_scc_sptr self) → int`

`rational_resampler_base_scc_sptr.decimation(rational_resampler_base_scc_sptr self) → unsigned int`

`rational_resampler_base_scc_sptr.declare_sample_delay(rational_resampler_base_scc_sptr self, int which, int delay)`

`declare_sample_delay(rational_resampler_base_scc_sptr self, unsigned int delay)`

`rational_resampler_base_scc_sptr.interpolation(rational_resampler_base_scc_sptr self) → unsigned int`

`rational_resampler_base_scc_sptr.message_subscribers(rational_resampler_base_scc_sptr self, swig_int_ptr which_port) → swig_int_ptr`

`rational_resampler_base_scc_sptr.min_noutput_items(rational_resampler_base_scc_sptr self) → int`

`rational_resampler_base_scc_sptr.pc_input_buffers_full_avg(rational_resampler_base_scc_sptr self, int which) → float`

`pc_input_buffers_full_avg(rational_resampler_base_scc_sptr self) → pmt_vector_float`

`rational_resampler_base_scc_sptr.pc_noutput_items_avg(rational_resampler_base_scc_sptr self) → float`

`rational_resampler_base_scc_sptr.pc_nproduced_avg(rational_resampler_base_scc_sptr self) → float`

`rational_resampler_base_scc_sptr.pc_output_buffers_full_avg(rational_resampler_base_scc_sptr self, int which) → float`

`pc_output_buffers_full_avg(rational_resampler_base_scc_sptr self) → pmt_vector_float`

`rational_resampler_base_scc_sptr.pc_throughput_avg(rational_resampler_base_scc_sptr self) → float`

`rational_resampler_base_scc_sptr.pc_work_time_avg(rational_resampler_base_scc_sptr self) → float`

`rational_resampler_base_scc_sptr.pc_work_time_total(rational_resampler_base_scc_sptr self) → float`

`rational_resampler_base_scc_sptr.sample_delay(rational_resampler_base_scc_sptr self, int which) → unsigned int`

`rational_resampler_base_scc_sptr.set_min_noutput_items(rational_resampler_base_scc_sptr self, int m)`

`rational_resampler_base_scc_sptr.set_taps(rational_resampler_base_scc_sptr self, pmt_vector_cfloat taps)`

`rational_resampler_base_scc_sptr.set_thread_priority(rational_resampler_base_scc_sptr self, int priority) → int`

`rational_resampler_base_scc_sptr.taps(rational_resampler_base_scc_sptr self) → pmt_vector_cfloat`

`rational_resampler_base_scc_sptr.thread_priority(rational_resampler_base_scc_sptr self) → int`

`gnuradio.filter.single_pole_iir_filter_cc(double alpha, unsigned int vlen=1) → single_pole_iir_filter_cc_sptr`

single pole IIR filter with complex input, complex output

The input and output satisfy a difference equation of the form

$$y[n] - (1-\alpha) y[n-1] = \alpha x[n]$$

with the corresponding rational system function

$$H(z) = \frac{\alpha}{1 - (1-\alpha) z^{-1}}$$

Note that some texts define the system function with a + in the denominator. If you're using that convention, you'll need to negate the feedback tap.

Constructor Specific Documentation:

**Parameters:**

- **alpha** –
- **vlen** –

`single_pole_iir_filter_cc_sptr.active_thread_priority(single_pole_iir_filter_cc_sptr self) → int`

`single_pole_iir_filter_cc_sptr.declare_sample_delay(single_pole_iir_filter_cc_sptr self, int which, int delay)`

`declare_sample_delay(single_pole_iir_filter_cc_sptr self, unsigned int delay)`

`single_pole_iir_filter_cc_sptr.message_subscribers(single_pole_iir_filter_cc_sptr self, swig_int_ptr which_port) → swig_int_ptr`

`single_pole_iir_filter_cc_sptr.min_noutput_items(single_pole_iir_filter_cc_sptr self) → int`

`single_pole_iir_filter_cc_sptr.pc_input_buffers_full_avg(single_pole_iir_filter_cc_sptr self, int which) → float`

`pc_input_buffers_full_avg(single_pole_iir_filter_cc_sptr self) → pmt_vector_float`

`single_pole_iir_filter_cc_sptr.pc_noutput_items_avg(single_pole_iir_filter_cc_sptr self) → float`

`single_pole_iir_filter_cc_sptr.pc_nproduced_avg(single_pole_iir_filter_cc_sptr self) → float`

`single_pole_iir_filter_cc_sptr.pc_output_buffers_full_avg(single_pole_iir_filter_cc_sptr self, int which) → float`

`pc_output_buffers_full_avg(single_pole_iir_filter_cc_sptr self) → pmt_vector_float`

`single_pole_iir_filter_cc_sptr.pc_throughput_avg(single_pole_iir_filter_cc_sptr self) → float`

`single_pole_iir_filter_cc_sptr.pc_work_time_avg(single_pole_iir_filter_cc_sptr self) → float`

`single_pole_iir_filter_cc_sptr.pc_work_time_total(single_pole_iir_filter_cc_sptr self) → float`

`single_pole_iir_filter_cc_sptr.sample_delay(single_pole_iir_filter_cc_sptr self, int which) → unsigned int`

`single_pole_iir_filter_cc_sptr.set_min_noutput_items(single_pole_iir_filter_cc_sptr self, int m)`

`single_pole_iir_filter_cc_sptr.set_taps(single_pole_iir_filter_cc_sptr self, double alpha)`

`single_pole_iir_filter_cc_sptr.set_thread_priority(single_pole_iir_filter_cc_sptr self, int priority) → int`

`single_pole_iir_filter_cc_sptr.thread_priority(single_pole_iir_filter_cc_sptr self) → int`

`gnuradio.filter.single_pole_iir_filter_ff(double alpha, unsigned int vlen=1) → single_pole_iir_filter_ff_sptr`

single pole IIR filter with float input, float output

The input and output satisfy a difference equation of the form

$$y[n] - (1-\alpha)y[n-1] = \alpha x[n]$$

with the corresponding rational system function

$$H(z) = \frac{\alpha}{1 - (1-\alpha)z^{-1}}$$

Note that some texts define the system function with a + in the denominator. If you're using that convention, you'll need to negate the feedback tap.

Constructor Specific Documentation:

**Parameters:**

- **alpha** –
- **vlen** –

`single_pole_iir_filter_ff_sptr.active_thread_priority(single_pole_iir_filter_ff_sptr self) → int`

`single_pole_iir_filter_ff_sptr.declare_sample_delay(single_pole_iir_filter_ff_sptr self, int which,`

```

int delay)
    declare_sample_delay(single_pole_iir_filter_ff_sptr self, unsigned int delay)

single_pole_iir_filter_ff_sptr.message_subscribers(single_pole_iir_filter_ff_sptr self,
    swig_int_ptr which_port) → swig_int_ptr

single_pole_iir_filter_ff_sptr.min_noutput_items(single_pole_iir_filter_ff_sptr self) → int

single_pole_iir_filter_ff_sptr.pc_input_buffers_full_avg(single_pole_iir_filter_ff_sptr self, int
    which) → float
    pc_input_buffers_full_avg(single_pole_iir_filter_ff_sptr self) -> pmt_vector_float

single_pole_iir_filter_ff_sptr.pc_noutput_items_avg(single_pole_iir_filter_ff_sptr self) → float

single_pole_iir_filter_ff_sptr.pc_nproduced_avg(single_pole_iir_filter_ff_sptr self) → float

single_pole_iir_filter_ff_sptr.pc_output_buffers_full_avg(single_pole_iir_filter_ff_sptr self,
    int which) → float
    pc_output_buffers_full_avg(single_pole_iir_filter_ff_sptr self) -> pmt_vector_float

single_pole_iir_filter_ff_sptr.pc_throughput_avg(single_pole_iir_filter_ff_sptr self) → float

single_pole_iir_filter_ff_sptr.pc_work_time_avg(single_pole_iir_filter_ff_sptr self) → float

single_pole_iir_filter_ff_sptr.pc_work_time_total(single_pole_iir_filter_ff_sptr self) → float

single_pole_iir_filter_ff_sptr.sample_delay(single_pole_iir_filter_ff_sptr self, int which) →
    unsigned int

single_pole_iir_filter_ff_sptr.set_min_noutput_items(single_pole_iir_filter_ff_sptr self, int m)

single_pole_iir_filter_ff_sptr.set_taps(single_pole_iir_filter_ff_sptr self, double alpha)

single_pole_iir_filter_ff_sptr.set_thread_priority(single_pole_iir_filter_ff_sptr self, int priority)
    → int

single_pole_iir_filter_ff_sptr.thread_priority(single_pole_iir_filter_ff_sptr self) → int

```