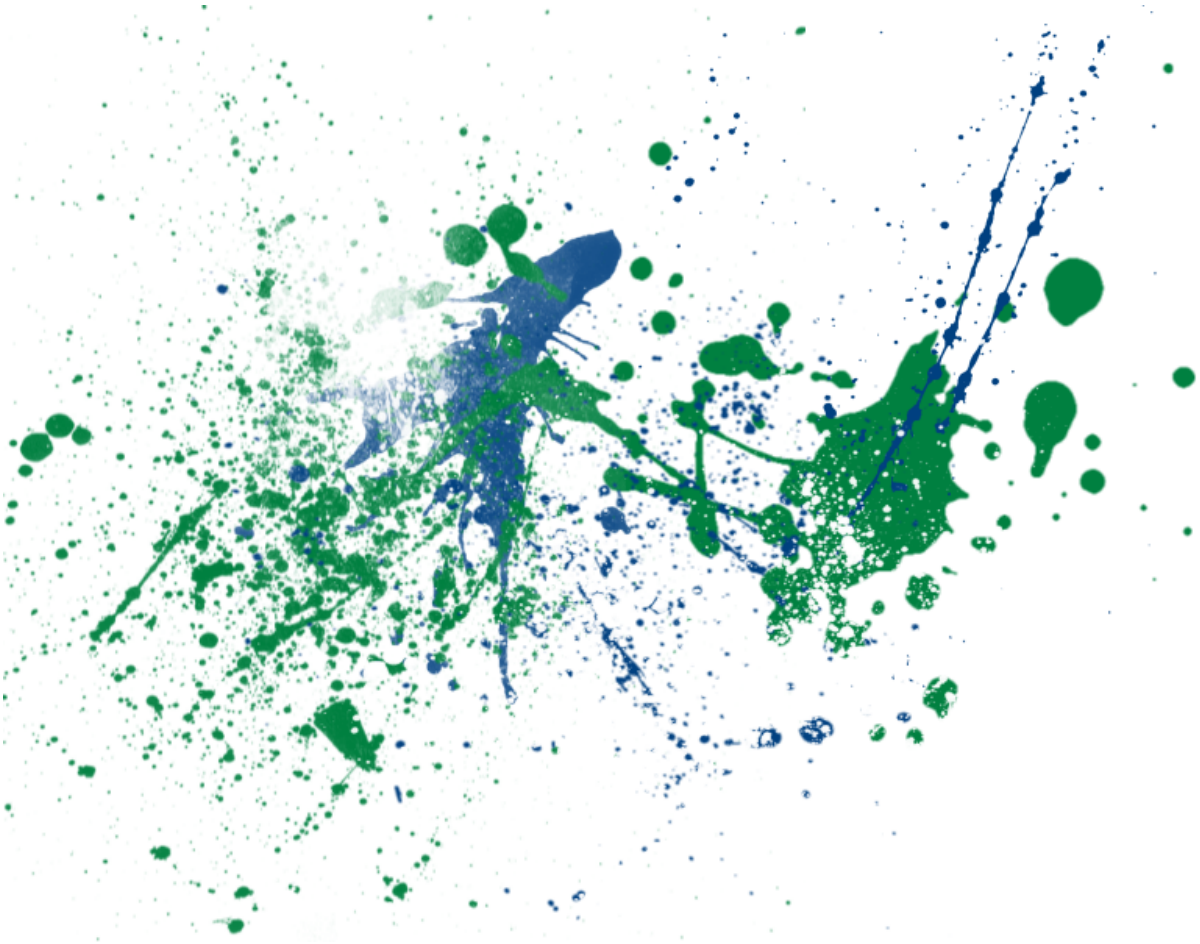


liquid

software-defined radio digital signal processing library

User's Manual for Version 1.0.0

Joseph D. Gaeddert
April 29, 2011
Blacksburg, Virginia
liquid@vt.edu

Contents

Part I
Introduction to *liquid*

The next few sections are designed to give you an understanding of *liquid*'s intended purpose and where it might fit within your project. Included is a quick start guide, example source code, and a brief historical outline.

1 Background and History

liquid is a free and open-source digital signal processing (DSP) library designed specifically for software-defined radios on embedded platforms. The aim is to provide a lightweight DSP library that does not rely on a myriad of external dependencies or proprietary and otherwise cumbersome frameworks. All signal processing elements are designed to be flexible, scalable, and dynamic, including filters, filter design, oscillators, modems, synchronizers, and complex mathematical operations. The source for *liquid* is written entirely in C so that it can be compiled quickly with a low memory footprint and easily deployed on embedded platforms.

liquid was created by J. Gaeddert out of necessity to perform complex digital signal processing algorithms on embedded devices without relying on proprietary and otherwise cumbersome frameworks. This was a critical step in his PhD thesis to adapt DSP algorithms in cognitive dynamic-spectrum radios to optimally manage finite radio resources. The project is not intended to compete with many other well-known and excellent software radio packages freely available (such as *gnuradio* [3] and *OSSIE* [21]) but was created as a lightweight library which can be used to augment these projects' capabilities or be used in embedded platforms were minimizing overhead is critical. You will notice that *liquid* lacks any sort of underlying framework for connecting signal processing "blocks" or "components." The design was chosen because each application requires the signal processing block to be redesigned and recompiled for each application anyway so the notion of a reconfigurable framework is, for the most part, a flawed concept.

In *liquid* there is no model for passing data between structures, no generic interface for data abstraction, no customized/proprietary data types, no framework for handling memory management; this responsibility is left to the designer, and as a consequence the library provides very little computational overhead. This package does *not* provide graphical user interfaces, component models, or debugging tools; *liquid* is simply a collection raw signal processing modules providing flexibility in algorithm development for wireless communications at the physical layer.

2 Quick Start Guide

A full description of installation procedures can be found in Section 25. The library can easily be built from source and is available from several places. The two most typical means of distribution are a compressed archive (a *tarball*) and cloning the source repository. If you are building from a tarball download the compressed archive `liquid-dsp-v.v.v.tar.gz` to your local machine where `v.v.v` denotes the version of the release (e.g. `liquid-dsp-1.0.0.tar.gz`); Unpack the tarball

```
$ tar -xvf liquid-dsp-v.v.v.tar.gz
```

Move into the directory and run the configure script and make the library.

```
$ cd liquid-dsp-v.v.v
$ ./configure
$ make
# make install
```

You may also build the lastest version of the code by cloning the Git repository. The main repository for *liquid* is hosted online by *github* [12] and can be cloned on your local machine via

```
$ git clone git://github.com/jgaeddert/liquid-dsp.git
```

Move into the directory and build as before with the archive, but with the additional bootstrapping step:

```
$ cd liquid-dsp.git
$ ./reconf
$ ./configure
$ make
# make install
```

You might also want to build and run the optional validation program (see Section 26.2) via

```
$ make check
```

and the benchmarking tool (see Section 26.3)

```
$ make bench
```

A comprehensive list of signal processing examples is given in the `examples` directory. You may build all of the example binaries at one time by running

```
$ make examples
```

Sometimes, however, it is useful to build one example individually. This can be accomplished by directly targeting its binary (e.g. "`make examples/cvsd_example`"). The example then can be run at the command line (e.g. "`./examples/cvsd_example`").

3 Data Structures in *liquid*

Most of *liquid*'s signal processing elements are C structures which retain the object's parameters, state, and other useful information. The naming convention is `basename_xxxt_method` where `basename` is the base object name (e.g. `interp`), `xxxt` is the type definition, and `method` is the object method. The type definition describes respective output, internal, and input type. Types are usually `f` to denote standard 32-bit *floating point* precision values and can either be represented as `r` (real) or `c` (complex). For example, a `dotprod` (vector dot product) object with complex input and output types but real internal coefficients operating on 32-bit floating-point precision values is `dotprod_crcf`.

Most objects have at least four standard methods: `create()`, `destroy()`, `print()`, and `execute()`. Certain objects also implement a `recreate()` method which operates similar to that of `realloc()` in C and are used to restructure or reconfigure an object without completely destroying it and creating it again. Typically, the user will create the signal processing object independent of the external (user-defined) data array. The object will manage its own memory until its `destroy()` method is invoked. A few points to note:

1. The object is only used to maintain the state of the signal processing algorithm. For example, a finite impulse response filter (Section 14.4) needs to retain the filter coefficients and a buffer of input samples. Certain algorithms which do not retain information (those which are memoryless) do not use objects. For example, `design_rnyquist_filter()` (Section 14.5.2) calculates the coefficients of a square-root raised-cosine filter, a processing algorithm which does not need to maintain a state after its completion.

2. While the objects do retain internal memory, they typically operate on external user-defined arrays. As such, it is strictly up to the user to manage his/her own memory. Shared pointers are a great way to cause memory leaks, double-free bugs, and severe headaches. The bottom line is to remember that if you created a mess, it is your responsibility to clean it up.

3. Certain objects will allocate memory internally, and consequently will use more memory than others. This memory will only be freed when the appropriate `delete()` method is invoked. Don't forget to clean up your mess!

3.1 Basic Lifecycle

Listed below is an example of the basic lifecycle of a `iirfilt_crcf` object (infinite impulse response filter with complex float inputs/outputs, and real float coefficients). The design parameters of the filter are specified in the *options* section near the top of the file. The `iirfilt_crcf` filter object is then created from the design using the `iirfilt_crcf_create()` method. Input and output data arrays of type `float complex` are allocated and a loop is run which initializes each input sample and computes a filter output using the `iirfilt_crcf_execute()` method. Finally the filter object is destroyed using the `iirfilt_crcf_destroy()` method, freeing all of the object's internally allocated memory.

```
1  // file: doc/listings/lifecycle.example.c
2  #include <liquid/liquid.h>
3
4  int main() {
5      // options
6      unsigned int order=4;   // filter order
7      float fc=0.1f;          // cutoff frequency
8      float f0=0.25f;         // center frequency (bandpass/bandstop)
9      float Ap=1.0f;          // pass-band ripple [dB]
10     float As=40.0f;         // stop-band attenuation [dB]
11     liquid_iirdes_filtertype ftype  = LIQUID_IIRDES_ELLIP;
12     liquid_iirdes_bandtype   btype  = LIQUID_IIRDES_BANDPASS;
13     liquid_iirdes_format     format = LIQUID_IIRDES_SOS;
14
15     // CREATE filter object (and print to stdout)
16     iirfilt_crcf myfilter;
17     myfilter = iirfilt_crcf_create_prototype(ftype,
18                                              btype,
19                                              format,
20                                              order,
21                                              fc, f0,
22                                              Ap, As);
23     iirfilt_crcf_print(myfilter);
24
25     // allocate memory for data arrays
26     unsigned int n=128; // number of samples
27     float complex x[n]; // input samples array
28     float complex y[n]; // output samples array
29
30     // run filter
```

```
31      unsigned int i;
32      for (i=0; i<n; i++) {
33          // initialize input
34          x[i] = randnf() + _Complex_I*randnf();
35
36          // EXECUTE filter (repeat as many times as desired)
37          iirfilt_crcf_execute(myfilter, x[i], &y[i]);
38      }
39
40      // DESTROY filter object
41      iirfilt_crcf_destroy(myfilter);
42  }
```

A more comprehensive example is given in the example file `examples/iirfilt_crcf_example.c`, located under the main *liquid* project directory.

3.2 Why C?

A commonly asked question is "why C and not C++?" The answer is simple: *portability*. The project's aim is to provide a lightweight DSP library for software-defined radio that does not rely on a myriad of dependencies. While C++ is a fine language for many projects (and theoretically runs just as fast as C), it is not as portable to embedded platforms as C and typically has a larger memory footprint. Furthermore, the majority of functions simply perform complex operations on a data sequence and do not require a high-level object-oriented programming interface. The significance of object-oriented programming is the techniques used, not the languages describing it.

While a number of signal processing elements in *liquid* use structures, these are simply to save the internal state of the object. For instance, a `firfilt_crcf` (finite impulse response filter) object is just a structure which contains—among other things—the filter taps (coefficients) and an input buffer. This simplifies the interface to the user; one only needs to "push" elements into the filter's internal buffer and "execute" the dot product when desired. This could also be accomplished with classes, a construct specific to C++ and other high-level object-oriented programming languages; however, for the most part, C++ polymorphic data types and abstract base classes are unnecessary for basic signal processing, and primarily just serve to reduce the code base of a project at the expense of increased compile time and memory overhead. Furthermore, while C++ templates can certainly be useful for library development their benefits are of limited use to signal processing and can be circumvented through the use of pre-processor macros at the gain of increasing the portability of the code. Under the hood, the C++ compiler's pre-processor expands templates and classes before actually compiling the source anyway, so in this sense they are equivalent to the second-order macros used in *liquid*.

The C programming language has a rich history in system programming—specifically targeting embedded applications—and is the basis behind many well-known projects including the linux kernel [15] and the python programming language [25]. Having said this, high-level frameworks and graphical interfaces are much more suited to be written in C++ and will beat an implementation in C any day but lie far outside the scope of this project.

3.3 Data Types

The majority of signal processing for SDR is performed at complex baseband. Complex numbers are handled in *liquid* by defining data type `liquid_float_complex` which is simply a place-holder for the standard C math type `float complex` and C++ type `std::complex<float>`. *There are no custom/proprietary data types in liquid!*[1] Custom data types only promote lack of interoperability between libraries requiring conversion procedures which slow down computation. For those of you who like to dig through the source code might have stumbled upon the `typedef` macros at the beginning of the global header file `include/liquid.h` which creates new complex data types based on the compiler, (e.g. `liquid_complex_float`). While technically this code does define of a new type specification, its purpose is for compatability between compilers and programming language (see Section 3.4 on C++ portability), and is binary compatible with the standard C99 specification. In fact, these data types are only used in the header file and should not be used when programming. For example, the following example program demonstrates the interface in C:

```
// file:    doc/listings/nco.c
// build:   gcc -c -o nco.c.o nco.c
// link:    gcc -lm -lc -lliquid nco.c.o -o nco

#include <stdio.h>
#include <math.h>
#include <liquid/liquid.h>
#include <complex.h>

int main() {
    // create nco object and initialize
    nco_crcf n = nco_crcf_create(LIQUID_NCO);
    nco_crcf_set_phase(n,0.3f);

    // Test native C complex data type
    float complex x;
    nco_crcf_cexpf(n, &x);
    printf("C native complex:   %12.8f + j%12.8f\n", crealf(x), cimagf(x));

    // destroy nco object
    nco_crcf_destroy(n);

    printf("done.\n");
    return 0;
}
```

3.4 Building/Linking with C++

Although *liquid* is written in C, it can be seamlessly compiled and linked with C++ source files. Here is a C++ example comparable to the C program listed in the previous section:

```
// file:    doc/listings/nco.cc
// build:   g++ -c -o nco.cc.o nco.cc
```

[1] The only exception to this are the fixed-point data types, defined in the *liquid-fpm* library which hasn't been released yet, and even these data types are actually standard signed integers.

```
// link:    g++ -lm -lc -lliquid nco.cc.o -o nco

#include <iostream>
#include <math.h>

#include <liquid/liquid.h>

// NOTE: the definition for liquid_float_complex will change
//       depending upon whether the standard C++ <complex>
//       header file is included before or after including
//       <liquid/liquid.h>
#include <complex>

int main() {
    // create nco object and initialize
    nco_crcf n = nco_crcf_create(LIQUID_NCO);
    nco_crcf_set_phase(n,0.3f);

    // Test liquid complex data type
    liquid_float_complex x;
    nco_crcf_cexpf(n, &x);
    std::cout << "liquid complex:     "
              << x.real << " + j" << x.imag << std::endl;

    // Test native c++ complex data type
    std::complex<float> y;
    nco_crcf_cexpf(n, reinterpret_cast<liquid_float_complex*>(&y));
    std::cout << "c++ native complex: "
              << y.real() << " + j" << y.imag() << std::endl;

    // destroy nco object
    nco_crcf_destroy(n);

    std::cout << "done." << std::endl;
    return 0;
}
```

It is important, however, to link the code with a C++ linker rather than a C linker. For example, if the above program (`nco.cc`) is compiled with `g++` it must also be linked with `g++`, viz

```
$ g++ -c -o nco.cc.o nco.cc
$ g++ -lm -lc -lliquid nco.cc.o -o nco
```

3.5 Learning by example

While this document contains numerous examples listed in the text, they are typically condensed to demonstrate only the interface. The `examples/` subdirectory includes more extensive demonstrations and numerous examples for all the signal processing components. Many of these examples write an output file which can be read by *octave* [7] to display the results graphically. For a brief description of each of these examples, see `examples/README`.

Part II
Tutorials

To get you started with using *liquid* signal processing library this manual begins with tutorials rather than diving into the details of each signal processing module.

4 Tutorial: Phase-Locked Loop

This tutorial demonstrates the functionality of a carrier phase-locked loop and introduces the `iirfilt` object. You will need on your local machine:

- the *liquid* DSP libraries built and installed (see Section 25)
- a text editor such as `vim` [31]
- a C compiler such as `gcc` [10]
- a terminal

The problem statement and a brief theoretical description of phase-locked loops is given in the next section. A walk-through of the source code follows.

4.1 Problem Statement

Wireless communications systems up-convert the data signal with a high-frequency carrier before transmitting over the air. This transmitted signal is orthogonal to other signals so long as their bandwidths don't overlap and can be recovered at the receiver by mixing it back down to baseband. Many digital communications systems modulate information in the phase of the carrier requiring the receiver to demodulate the signal coherently in order to recover the original data message. In this regard the receiver must synchronize its carrier oscillator to that of the transmitter. To put it simply, the receiver must lock on to the phase of transmitter's carrier. One of the key advantages to performing signal processing in software is that the radio can operate at complex baseband.

In this simulation, the received signal is simply a complex sinusoid with an unknown initial carrier phase and frequency. The carrer holds no information-bearing symbols and is simply a tone whose frequency and phase represent the residual mismatch between the transmitter and receiver. The received signal x at time step k can be described as

$$x_k = \exp\big\{j(\theta + k\omega)\big\} \tag{1}$$

where $j \triangleq \sqrt{-1}$ and θ and ω represent the unknown initial carrier phase and frequency offsets, respectively. The receiver generates a complex sinusoid with a phase ϕ_k as the phase difference between x_k and y_k and can be computed as

$$y_k = \exp\big\{j\phi_k\big\} \tag{2}$$

The phase error at time step k is expressed as

$$\Delta\phi_k = \arg\big\{x_k y_k^*\big\} \tag{3}$$

where $(^*)$ denotes complex conjugation.[2] The goal of the receiver is to control ϕ_k (the phase of the output signal y at time k) to lock onto the input phase of x, hence the name "phase-locked

[2]Those who are savvy with communications techniques will appreciate that we are dealing in complex baseband and can easily compute the phase error estimate simply as the argument of the product of x_k and y_k. Conventional PLLs which have operated strictly in the real domain multiply only the real components of x_k and y_k for a phase error estimate, assume that the loop filter rejects the high-frequency component, and make the approximation $\Delta\phi \approx \sin(\Delta\phi) = \sin(\phi - \hat{\phi})$ for small phase errors.

loop." If the phase of the output sample y_k is behind that of the input ($\Delta\phi > 0$) then ϕ needs to be advanced appropriately for the next time step. Conversely, if the phase of y_k is ahead of the phase of x_k ($\Delta\phi < 0$) then the receiver need to retard ϕ.

Without going into a great amount of detail, this control is accomplished using a special filter within the loop. This filter, known as a "loop filter," is designed to reject high-frequency noise and is described with the transfer function $H(z)$. Specifically $H(z)$ is a 2^{nd}-order integrating low-pass recursive filter with a natural frequency ω_n, a damping factor ζ, and a loop gain K. The natural frequency is the resonant frequency of $H(z)$ and for all practical purposes is the filter's bandwidth. Increasing ω_n permits the loop to track to the input signal faster (reduces lock time), but also increases the amount of noise passed through the loop. Decreasing ω_n reduces this noise but also increases the loop's acquisition time. The damping factor ζ controls the stability of the filter and is typically set to a value near $1/\sqrt{2} \approx 0.707$. The loop gain K is typically very large (on the order of 1000 or so). For more detailed information on loop filter design the interested reader is referred to Section 19.2.

The estimated phase error $\Delta\phi_k$ is filtered using $H(z)$ resulting in an output phase estimate ϕ_{k+1} which is used for the subsequent output sample y_{k+1} as

$$y_{k+1} = \exp\{j\phi_{k+1}\} \tag{4}$$

Algorithm 1 Phase-locked Loop Control

1: $\boldsymbol{x} \leftarrow \{x_0, x_1, x_2, \ldots\}$ (input array)
2: $\hat{\phi}_0 \leftarrow 0$ (initial output phase)
3: **for** $k = 0, 1, 2, \ldots$ **do**
4: $\quad y_k \leftarrow \exp\{j\hat{\phi}_k\}$ (compute output sample)
5: $\quad \Delta\phi_k \leftarrow \arg\{x_k y_k^*\}$ (phase detector)
6: $\quad \hat{\phi}_{k+1} \leftarrow \text{filter}(\Delta\phi_k)$ (update output phase estimate)
7: **end for**

A summary of the algorithm is given in Algorithm 1. In the next section we will create a simple C program to simulate a phase-locked loop with *liquid*.

4.2 Setting up the Environment

For this tutorial and others, I assume that you are using the GNU compiler collection (`gcc`) for compiling source and linking objects [10], and that you have a familiarity with the C (or C++) programming language. Create a new file `pll.c` and open it with your favorite editor. Include the headers `stdio.h`, `complex.h`, `math.h`, and `liquid/liquid.h` and add the `int main()` definition so that your program looks like this:

```
1  // file: doc/tutorials/pll_init_tutorial.c
2  #include <stdio.h>
3  #include <complex.h>
4  #include <math.h>
5  #include <liquid/liquid.h>
6
7  int main() {
```

```
8       printf("done.\n");
9       return 0;
10  }
```

Compile and link the program using `gcc`:

```
$ gcc -Wall -o pll -lm -lc -lliquid pll.c
```

The flag "`-Wall`" tells the compiler to print all warnings (unused and uninitialized variables, etc.), "`-o pll`" specifies the name of the output program is "`pll`", and "`-lm -lc -lliquid`" tells the linker to link the binary against the math, standard C, and *liquid* DSP libraries, respectively. Notice that the above command invokes both the compiler and the linker collectively. If the compiler did not give any errors, the output executable `pll` is created which can be run as

```
$ ./pll
```

and should simply print "`done.`" to the screen. You are now ready to add functionality to your program.

We will now edit the file to set up the basic simulation but without controlling the phase of the output sinusoid. As such the output won't track to the input resulting in a significant amount of phase error. This simulation will operate one sample at a time and is organized into three sections. First, set up the simulation parameters: the initial phase and frequency offsets (`float`), and number of samples to run (`unsigned int`). Next, initialize the complex input and output variables (`x` and `y`) to zero, as well as the state of the phase error (`phase_error`) and output phase (`phi_hat`) estimates. Finally, set up the computational loop which generates the input and output samples, computes the phase error between them, and then prints the results to the screen. Edit `pll.c` to set up the basic simulation:

```
1   // file: doc/tutorials/pll_basic_tutorial.c
2   #include <stdio.h>
3   #include <complex.h>
4   #include <math.h>
5   #include <liquid/liquid.h>
6
7   int main() {
8       // simulation parameters
9       float phase_offset      = 0.8f;     // initial phase offset
10      float frequency_offset  = 0.01f;    // initial frequency offset
11      unsigned int n          = 40;       // number of iterations
12
13      float complex x   = 0;  // input sample
14      float phase_error = 0;  // phase error estimate
15      float phi_hat     = 0;  // output sample phase
16      float complex y   = 0;  // output sample
17
18      unsigned int i;
19      for (i=0; i<n; i++) {
20          // generate input sample
21          x = cexpf(_Complex_I*(phase_offset + i*frequency_offset));
22
23          // generate output sample
```

```
24          y = cexpf(_Complex_I*phi_hat);
25
26          // compute phase error
27          phase_error = cargf(x*conjf(y));
28
29          // print results to screen
30          printf("%3u : phase = %12.8f, error = %12.8f\n", i, phi_hat, phase_error);
31      }
32
33      printf("done.\n");
34      return 0;
35  }
```

The variables `x` and `y` are of type `float complex` which contains both real and imaginary components of type `float`. The function `cexpf()` computes the complex exponential of its argument which for a purely imaginary input $j\alpha$ is simply $e^{j\alpha} = \cos\alpha + j\sin\alpha$.

Compile and run the program as before. The program should now output something like this:

```
  0 : phase =   0.00000000, error =   0.80000001
  1 : phase =   0.00000000, error =   0.81000000
  2 : phase =   0.00000000, error =   0.81999999
  3 : phase =   0.00000000, error =   0.82999998
  4 : phase =   0.00000000, error =   0.84000003
        ...
 35 : phase =   0.00000000, error =   1.14999998
 36 : phase =   0.00000000, error =   1.15999997
 37 : phase =   0.00000000, error =   1.17000008
 38 : phase =   0.00000000, error =   1.18000007
 39 : phase =   0.00000000, error =   1.19000006
done.
```

Notice that because we aren't controlling the output phase yet the error increases with the input phase. In the next section we will design the loop filter to adjust the output phase to lock onto the input signal given the phase error.

4.3 Designing the Loop Filter

Our program so far has not used any of the *liquid* DSP libraries for computation and has only relied on the standard C libraries for dealing with complex math operations. In this section we will introduce *liquid*'s `iirfilt_rrrf` object to realize a recursive (infinite impulse response) filter with real inputs, coefficients, and outputs. Additionally we will use the function `iirdes_pll_active_lag()` to design the coefficients for the PLL's filter, specifically an "active lag" design. While the explanation in this section is fairly long, relax! We will only need to add about 15 lines of code to our program. If you are eager to edit your program you may skip to Section 4.4.

Digital representations of infinite impulse response (IIR) filters have two sets of coefficients: feedback and feedforward. In the digital domain the transfer function is a ratio of the polynomials in z^{-1} where the feedforward coefficients $\boldsymbol{b} = \{b_0, b_1, b_2, \ldots, b_{N-1}\}$ are in the numerator and the feedback coefficients $\boldsymbol{a} = \{a_0, a_1, a_2, \ldots, a_{M-1}\}$ are in the denominator. Specifically, the transfer

function is

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_{N-1} z^{-(N-1)}}{a_0 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_{M-1} z^{-(M-1)}} \tag{5}$$

This transfer function means that the output of the filter is the linear combination of the N previous filter inputs ($\boldsymbol{x}$) and $M-1$ previous filter outputs ($\boldsymbol{y}$), viz

$$y[k] = \frac{1}{a_0}\Big(b_0 x[k] \quad + \quad b_1 x[k-1] + \cdots + b_{N-1} x[k-N] \tag{6}$$

$$- \quad a_1 y[k-1] - \cdots - a_{M-1} y[k-M]\Big) \tag{7}$$

Typically the number of feedback and feedforward coefficients are equal ($M = N$), and the coefficients themselves are normalized so that $a_0 = 1$.

liquid implements IIR filters with the `iirfilt_xxxt` family of objects where "`xxxt`" denotes the type definition (see Section 3 for details). In our example we will be using the `iirfilt_rrrf` object which indicates that this is an IIR filter with real inputs, outputs, and coefficients with precision of type `float`. The IIR filter objects in *liquid* maintain their state internally, storing the previous inputs and outputs in its internal buffers. Nearly every object in *liquid* (filter or otherwise) has at least four basic methods: `create()`, `print()`, `execute()`, and `destroy()`. For our program we will need to create the filter object by passing to it a vector of each the feedback and feedforward coefficients. The infinite impulse response (IIR) filter we are designing is of order two which means that $\boldsymbol{a}$ and $\boldsymbol{b}$ have three coefficients each.

Generating the loop filter coefficients is fairly straightforward. As stated before, the loop filter has parameters for natural frequency ω_n, damping factor ζ, and loop gain K. Furthermore the filter is 2^{nd}-order which means that it has three coefficients each for $\boldsymbol{a}$ and $\boldsymbol{b}$. *liquid* provides a method for computing such a filter with the `iirdes_pll_active_lag()` function which accepts ω_n, ζ, and K as inputs and generates the coefficients in two output arrays. The coefficients can be computed as follows:

```
float wn = 0.1f;        // pll bandwidth
float zeta = 0.707f;    // pll damping factor
float K = 1000.0f;      // pll loop gain
float b[3];             // feedforward coefficients array
float a[3];             // feedback coefficients array
iirdes_pll_active_lag(wn, zeta, K, b, a);
```

The life cycle of the IIR filter can be summarized as follows

```
iirfilt_rrrf loopfilter = iirfilt_rrrf_create(b,3,a,3);
float sample_in = 0.0f;
float sample_out;
{
    // repeat as necessary
    iirfilt_rrrf_execute(loopfilter, sample_in, &sample_out);
}
iirfilt_rrrf_destroy(loopfilter);
```

noting that the `execute()` method can be repeated as many times as necessary before the object is destroyed.

Using the code snippets above, modify your program to include the loop filter to adjust the output signal's phase. The input to the filter will be the `phase_error` variable, and its output will be `phi_hat`. Don't forget to destroy your filter object once the loop has finished running.

4.4 Final Program

The final program is listed below, and a copy of the source is located in the `doc/tutorials/` subdirectory.

```
// file: doc/tutorials/pll_tutorial.c
#include <stdio.h>
#include <complex.h>
#include <math.h>
#include <liquid/liquid.h>

int main() {
    // simulation parameters
    float phase_offset      = 0.8f;     // initial phase offset
    float frequency_offset  = 0.01f;    // initial frequency offset
    float wn                = 0.10f;    // pll bandwidth
    float zeta              = 0.707f;   // pll damping factor
    float K                 = 1000;     // pll loop gain
    unsigned int n          = 40;       // number of iterations

    // generate IIR loop filter coefficients
    float b[3];     // feedforward coefficients
    float a[3];     // feedback coefficients
    iirdes_pll_active_lag(wn, zeta, K, b, a);

    // create and print the loop filter object
    iirfilt_rrrf loopfilter = iirfilt_rrrf_create(b,3,a,3);
    iirfilt_rrrf_print(loopfilter);

    float complex x   = 0;  // input sample
    float phase_error = 0;  // phase error estimate
    float phi_hat     = 0;  // output sample phase
    float complex y   = 0;  // output sample

    unsigned int i;
    for (i=0; i<n; i++) {
        // generate input sample
        x = cexpf(_Complex_I*(phase_offset + i*frequency_offset));

        // generate output sample
        y = cexpf(_Complex_I*phi_hat);

        // compute phase error
        phase_error = cargf(x*conjf(y));

        // run error through loop filter
        iirfilt_rrrf_execute(loopfilter, phase_error, &phi_hat);
```

```
43
44          // print results to screen
45          printf("%3u : phase = %12.8f, error = %12.8f\n", i, phi_hat, phase_error);
46      }
47
48      // destroy IIR filter object
49      iirfilt_rrrf_destroy(loopfilter);
50
51      printf("done.\n");
52      return 0;
53  }
```

Compile the program as before, creating the executable "`pll`." Running the program should produce an output similar to this:

```
iir filter [normal]:
  b :  0.32277358  0.07999840 -0.24277516
  a :  1.00000000 -1.99995995  0.99996001
  0 : phase =   0.25821885, error =   0.80000001
  1 : phase =   0.75852644, error =   0.55178112
  2 : phase =   1.12857747, error =   0.06147351
  3 : phase =   1.27319980, error =  -0.29857749
  4 : phase =   1.23918116, error =  -0.43319979
        ...
 35 : phase =   1.15999877, error =   0.00000751
 36 : phase =   1.17000139, error =   0.00000122
 37 : phase =   1.18000150, error =  -0.00000131
 38 : phase =   1.19000030, error =  -0.00000140
 39 : phase =   1.19999886, error =  -0.00000024
done.
```

Notice that the phase error at the end of the output is very small. The initial error (at $k = 0$) is 0.8 which is the value of the `phase_offset` parameter ot the beginning of the program. Notice also that the difference in phase of the last several samples (i.e. the difference between the phase at steps `38` and `39`) is approximately 0.1 which is the initial frequency offset that was given in the beginning. Play around with the input parameters, particularly the frequency offset and the phase-locked loop bandwidth. Increasing the PLL bandwidth (`wn`) should reduce the resulting phase error more quickly. The downside of having a PLL with a large bandwidth is that when the input signal has been corrupted by noise then the phase error estimate is also noisy. In this tutorial no noise term was introduced.

5 Tutorial: Forward Error Correction

This tutorial will demonstrate computation at the byte level (raw message data) by introducing the forward error-correction (FEC) coding module. Please note that *liquid* only provides some very basic FEC capabilities including some Hamming block codes and repeat codes. While these codes are very fast and enough to get started, they are not very efficient and add a lot of redunancy without providing a strong level of correcting capabilities. *liquid* will use the convolutional and Reed-Solomon codes described in *libfec* [23] if installed on your machine.

5.1 Problem Statement

Digital communications over a noisy channel can be unreliable, resulting in errors at the receiver. Forward error-correction (FEC) coding adds redundancy to the original data message that allows for some errors to be corrected at the receiver. The error-correction capability of the code is dependent upon many factors, but is usually improved by increasing the amount of redundancy added to the message. The drawback to adding a lot of redundancy is that the communications rate is decreased as the link must be shared among the important data information as well as the redundant bits. The benefit, however, is that the receiver has a better chance of correcting the errors without having to request a retransmission of the message. Volumes of research papers and books have been written about the error-correction capabilities of certain FEC encoder/decoder pairs (codecs) and their performance in a variety of environments. While there is far too much information on the subject to discuss here, it is important to note that *liquid* implements a very small subset of simple FEC codecs, including several Hamming and repeat codes. If the *libfec* [23] library is installed when *liquid* is configured this list extends to convolutional and Reed-Solomon codes.

In this tutorial you will create a simple program that will generate a message, encode it using a simple Hamming(7,4) code, corrupt the encoded message by adding an error, and then try to correct the error with the decoder.

5.2 Setting up the Environment

Create a new file `fec.c` and open it with your favorite editor. Include the headers `stdio.h` and `liquid/liquid.h` and add the `int main()` definition so that your program looks like this:

```
1  // file: doc/tutorials/fec_init_tutorial.c
2  #include <stdio.h>
3  #include <liquid/liquid.h>
4
5  int main() {
6      printf("done.\n");
7      return 0;
8  }
```

Compile and link the program using `gcc`:

```
$ gcc -Wall -o fec -lm -lc -lliquid fec.c
```

The flag "`-Wall`" tells the compiler to print all warnings (unused and uninitialized variables, etc.), "`-o fec`" specifies the name of the output program is "`fec`", and "`-lm -lc -lliquid`" tells the

linker to link the binary against the math, standard C, and *liquid* DSP libraries, respectively. Notice that the above command invokes both the compiler and the linker collectively. If the compiler did not give any errors, the output executable `fec` is created which can be run as

```
$ ./fec
```

and should simply print "`done.`" to the screen. You are now ready to add functionality to your program.

We will now edit the file to set up the basic simulation by generating a message signal and counting errors as a result of channel effects. The error-correction capabilities will be added in the next section. First set up the simulation parameters: for now the only parameter will be the length of the input message, denoted by the variable `n` (`unsigned int`) representing the number of bytes. Initialize `n` to 8 to reflect an original message of 8 bytes. Create another `unsigned int` variable `k` which will represent the length of the encoded message. This length is equal to the original (`n`) with the additional redundancy. For now set `k` equal to `n` as we are not adding FEC coding until the next section. This implies that without any redundancy, the receiver cannot correct for any errors.

Message data in *liquid* are represented as arrays of type `unsigned char`. Allocate space for the original, encoded, and decoded messages as `msg_org[n]`, `msg_enc[k]`, and `msg_dec[n]`, respectively. Initialize the original data message as desired. For example, the elements in `msg_org` can contain `0,1,2,...,n-1`. Copy the contents of `msg_org` to `msg_enc`. This effectively is a placeholder for forward error-correction which will be discussed in the next section. Corrupt one of the bits in `msg_enc` (e.g. `msg_enc[0] ^= 0x01;` will flip the least-significant bit in the first byte of the `msg_enc` array), and copy the results to `msg_dec`. Print the encoded and decoded messages to the screen to verify that they are not equal. Without any error-correction capabilities, the receiver should see a message different than the original because of the corrupted bit. Count the number of bit differences between the original and decoded messages. *liquid* provides a convenient interface for doing this and can be invoked as

```
unsigned int num_bit_errors = count_bit_errors_array(msg_org,
                                                     msg_dec,
                                                     n);
```

Print this number to the screen. Your program should look similar to this:

```
1  // file: doc/tutorials/fec_basic_tutorial.c
2  #include <stdio.h>
3  #include <liquid/liquid.h>
4
5  int main() {
6      // simulation parameters
7      unsigned int n = 8;             // original data length (bytes)
8
9      // compute size of encoded message
10     unsigned int k = n;             // (no encoding yet)
11
12     // create arrays
13     unsigned char msg_org[n];       // original data message
14     unsigned char msg_enc[k];       // encoded/received data message
```

```
15      unsigned char msg_dec[n];       // decoded data message
16
17      unsigned int i;
18      // create message
19      for (i=0; i<n; i++) msg_org[i] = i & 0xff;
20
21      // "encode" message (copy to msg_enc)
22      for (i=0; i<n; i++) msg_enc[i] = msg_org[i];
23
24      // corrupt encoded message (flip bit)
25      msg_enc[0] ^= 0x01;
26
27      // "decode" message (copy to msg_dec)
28      for (i=0; i<n; i++) msg_dec[i] = msg_enc[i];
29
30      printf("original message:  [%3u] ",n);
31      for (i=0; i<n; i++)
32          printf(" %.2X", msg_org[i]);
33      printf("\n");
34
35      printf("decoded message:   [%3u] ",n);
36      for (i=0; i<n; i++)
37          printf(" %.2X", msg_dec[i]);
38      printf("\n");
39
40      // count bit errors
41      unsigned int num_bit_errors = count_bit_errors_array(msg_org, msg_dec, n);
42      printf("number of bit errors received:    %3u / %3u\n", num_bit_errors, n*8);
43
44      return 0;
45  }
```

Compile the program as before, creating the executable "`fec`." Running the program should produce an output similar to this:

```
original message:  [  8]  00 01 02 03 04 05 06 07
decoded message:   [  8]  01 01 02 03 04 05 06 07
number of bit errors received:      1 /  64
```

Notice that the decoded message differs from the original and that the number of received errors is nonzero.

5.3 Creating the Encoder/Decoder

So far our program doesn't use any *liquid* interfaces (except for the function used to count bit errors). The FEC module in *liquid* provides a simple interface for adding forward error-correction capabilities to your project. The `fec` object abstracts from the gritty details behind the bit manipulation (packing/unpacking of bytes, appending tail bits, etc.) of error-correction structures. As an example, convolutional codes observe bits one at a time while Reed-Solomon codes operate on entire blocks of bits. The `fec` object in *liquid* conveniently abstracts from the organization of the codec and takes care of this overhead internally. This allows seamless integration of different codecs with

one simple interface. As with the `iirfilt_rrrf` object in the phase-locked loop tutorial (Section 4) the `fec` object has methods `create()`, `print()`, and `destroy()`. Nearly every object in *liquid* has these methods; however the `fec` object replaces `execute()` with `encode()` and `decode()` as the same object instance can be used for both encoding and decoding. The `fec_create()` method accepts two arguments, although the second one is basically ignored. The first argument is an enumeration of the type of codec that you wish to use.

To begin, create a new `fec` object of type `LIQUID_FEC_HAMMING74` (the second argument can simply be `NULL`) which creates a Hamming(7,4) code:

```
fec q = fec_create(LIQUID_FEC_HAMMING74, NULL);
```

Details of the available codes in *liquid* can be found in Section 12. This codec nominally accepts 4 bits, appends 3 parity bits, and can detect and correct up to one of these seven transmitted bits. The Hamming(7,4) code is not particularly strong for its rate; however it is computationally efficient and has been studied extensively in coding theory. The interface provided by *liquid* conveniently abstracts from the process of managing 8-bit data symbols (bytes), converting to 4-bit input symbols, encoding to 7-bit output symbols, and then re-packing into 8-bit output bytes. This is consistent with *any* forward error-correction code in *liquid*; as the user, you simply see data bytes in and data bytes out. The length of the output sequence can be computed using the method

```
unsigned int k = fec_get_enc_msg_length(LIQUID_FEC_HAMMING74, n);
```

where `n` represents the number of uncoded input bytes and `k` represents the number of encoded output bytes. This value should be used to appropriately allocate enough memory for the encoded message. Encoding the data message is as simple as invoking

```
fec_encode(q, n, msg_org, msg_enc);
```

which uses our newly-created `fec` object `q` to encode `n` input bytes in the array `msg_org` and store the result in the output array `msg_enc`. The interface for decoding is nearly identical:

```
fec_decode(q, n, msg_enc, msg_dec);
```

Notice that the second argument again represents the number of *uncoded* data bytes (`n`). Don't forget to destroy the object once you are finished:

```
fec_destroy(q);
```

5.4 Final Program

The final program is listed below, and a copy of the source is located in the `doc/tutorials/` subdirectory.

```
1  // file: doc/tutorials/fec_tutorial.c
2  #include <stdio.h>
3  #include <liquid/liquid.h>
4
5  int main() {
6      // simulation parameters
7      unsigned int n = 8;                     // original data length (bytes)
8      fec_scheme fs = LIQUID_FEC_HAMMING74;   // error-correcting scheme
```

```
9
10      // compute size of encoded message
11      unsigned int k = fec_get_enc_msg_length(fs,n);
12
13      // create arrays
14      unsigned char msg_org[n];   // original data message
15      unsigned char msg_enc[k];   // encoded/received data message
16      unsigned char msg_dec[n];   // decoded data message
17
18      // CREATE the fec object
19      fec q = fec_create(fs,NULL);
20      fec_print(q);
21
22      unsigned int i;
23      // generate message
24      for (i=0; i<n; i++)
25          msg_org[i] = i & 0xff;
26
27      // encode message
28      fec_encode(q, n, msg_org, msg_enc);
29
30      // corrupt encoded message (flip bit)
31      msg_enc[0] ^= 0x01;
32
33      // decode message
34      fec_decode(q, n, msg_enc, msg_dec);
35
36      // DESTROY the fec object
37      fec_destroy(q);
38
39      printf("original message:  [%3u] ",n);
40      for (i=0; i<n; i++)
41          printf(" %.2X", msg_org[i]);
42      printf("\n");
43
44      printf("decoded message:   [%3u] ",n);
45      for (i=0; i<n; i++)
46          printf(" %.2X", msg_dec[i]);
47      printf("\n");
48
49      // count bit errors
50      unsigned int num_bit_errors = count_bit_errors_array(msg_org, msg_dec, n);
51      printf("number of bit errors received:    %3u / %3u\n", num_bit_errors, n*8);
52
53      printf("done.\n");
54      return 0;
55  }
```

The output should look like this:

```
fec: Hamming(7,4) [rate: 0.571]
original message:  [  8]  00 01 02 03 04 05 06 07
```

```
decoded message:   [  8]  00 01 02 03 04 05 06 07
number of bit errors received:      0 /  64
done.
```

Notice that the decoded message matches that of the original message, even though an error was introduced at the receiver. As discussed above, the Hamming(7,4) code is not particularly strong; if too many bits in the encoded message are corrupted then the decoder will be unable to correct them. Play around with changing the length of the original data message, the encoding scheme, and the number of errors introduced.

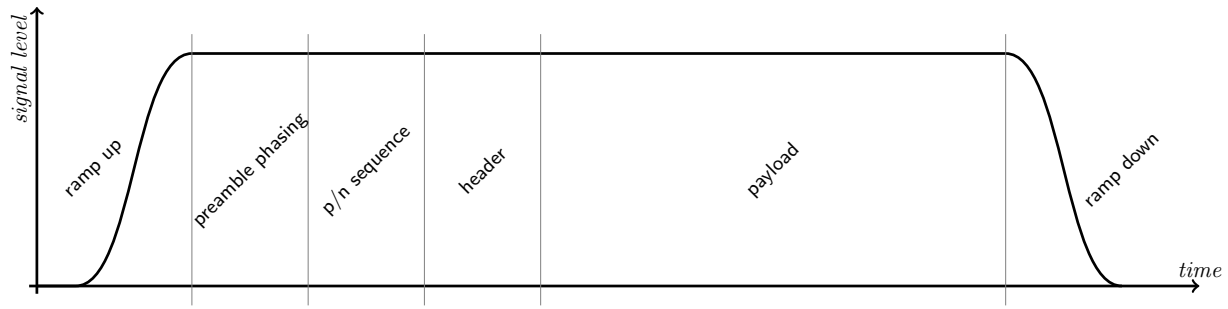
For a more detailed program, see `examples/fec_example.c` in the main *liquid* directory. Section 12 describes *liquid*'s FEC module in detail. Additionally, the `packetizer` object extends the simplicity of the `fec` object by adding a cyclic redundancy check and two layers of forward error-correction and interleaving, all of which can be reconfigured as desired. See `examples/packetizer_example.c` for a detailed example program on how to use the `packetizer` object.

6 Tutorial: Framing

In the previous tutorials we have created only the basic building blocks for wireless communication. This tutorial puts them all together by introducing a very simple framing structure for sending and receiving data over a wireless link. In this context "framing" refers to the encapsulation of data into a modulated time series at complex baseband to be transmitted over a wireless link. Conversely, "packets" refer to packing raw message data bytes with forward error-correction and data validity check redundancy.

6.1 Problem Statement

For this tutorial we will be using the `framegen64` and `framesync64` objects in *liquid*. As you might have guessed `framegen64` is the frame generator object on the transmit side of the link and `framesync64` is the frame synchronizer on the receive side. Together these objects realize a a very simple frame which encapsulates a 12-byte header and 64-byte payload within a frame consisting of 640 symbols at complex baseband. Conveniently the frame generator interpolates these symbols with a matched filter to produce a 1280-sample frame at complex baseband, ready to be up-converted and transmitted over the air. This frame has a nominal spectral efficiency of 0.8 bits/second/Hz (512 bits from 64 payload bytes assembled in 640 symbols).[3] This means that if you transmit with a symbol rate of 10kHz you should expect to see a throughput of 8kbps if all the frames are properly decoded. On the receiving side, raw samples at complex baseband are streamed to an instance of the frame synchronizer which picks out frames and invokes a user-defined callback function. The synchronizer corrects for gain, carrier, and sample timing offsets (channel impairments) in the complex baseband samples with a minimal amount of pre-processing required by the user. To help with synchronization, the frame includes a special preamble which can be seen in the figure below.



After up-conversion (mixing up to a carrier frequency) the frame is transmitted over the link where the receiver mixes the signal back down to complex baseband. The received signal will be attenuated and noisy and typically degrades with distance between the two radios. Also, because receiver's oscillators run independent of the transmitter's, this received signal will have other impairments such as carrier and timing offsets. In our program we will be operating at complex baseband and will add the channel impairments artificially.

The frame synchronizer's purpose is to correct for all of these impairments (within limitations, of course) and attempt to detect the frame and decode its data. The framing preamble assists the synchronizer by introducing special phasing sequences before any information-bearing symbols

[3] For simplicity this computation of spectral efficiency neglects any exceess bandwidth of the pulse-shaping filter.

which aids in correcting for carrier and timing offsets. Without going into great detail, these sequences significantly increase the probability of frame detection and decoding while adding a minimal amount of overhead to the frame; a small price to pay for increased data reliability!

6.2 Setting up the Environment

As with the other tutorials I assume that you are using `gcc` to compile your programs and link to appripriate libraries. Create a new file `framing.c` and include the headers `stdio.h`, `stdlib.h`, `math.h`, `complex.h`, and `liquid/liquid.h`. Add the `int main()` definition so that your program looks like this:

```
1  // file: doc/tutorials/framing_init_tutorial.c
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <math.h>
5  #include <complex.h>
6  #include <liquid/liquid.h>
7
8  int main() {
9      printf("done.\n");
10     return 0;
11 }
```

Compile and link the program using `gcc`:

```
$ gcc -Wall -o framing -lm -lc -lliquid framing.c
```

The flag "`-Wall`" tells the compiler to print all warnings (unused and uninitialized variables, etc.), "`-o framing`" specifies the name of the output program is "`framing`", and "`-lm -lc -lliquid`" tells the linker to link the binary against the math, standard C, and *liquid* DSP libraries, respectively. Notice that the above command invokes both the compiler and the linker collectively. If the compiler did not give any errors, the output executable `framing` is created which can be run as

```
$ ./framing
```

and should simply print "`done.`" to the screen. You are now ready to add functionality to your program.

6.3 Creating the Frame Generator

The particular framing structure we will be using accepts a 12-byte header and a 64-byte payload and assembles them into a frame consisting of 1280 samples. These sizes are fixed and cannot be adjusted for this framing structure.[4] The purpose of the header is to conveniently allow the user a separate control channel to be packaged with the payload. For example, if your application is to send a file using multiple frames, the header can include an identification number to indicate where in the file it should be written. Another application of the header is to include a destination node identifier for use in packet routing for ad hoc networks. Both the header and payload are assembled

[4] Alternatively, the `flexframegen` and `flexframesync` objects implement a dynamic framing structure which has many more options than the `framegen64` and `framesync64` objects. See Section 15 for details.

with a 16-bit cyclic redundancy check (CRC) to validate the integrity of the received data and encoded using the Hamming(12,8) code for error correction. (see Section 12 for more information on error detection and correction capabilities in *liquid*). The encoded header and payload are modulated with QPSK and encapsulated with a BPSK preamble. Finally, the resulting symbols are interpolated using a square-root Nyquist matched filter at a rate of 2 samples per symbol. This entire process is handled internally so that as a user the only thing you will need to do is call one function.

The `framegen64` object can be generated with the `framegen64_create()` method which accepts two arguments: an `unsigned int` and a `float` representing the matched filter's length (in symbols) and excess bandwidth factor, respectively. To begin, create a frame generator having a square-root Nyquist filter with a delay of 3 and an excess bandwidth factor of 0.7 as

```
framegen64 fg = framegen64_create(3, 0.7);
```

As with all structures in *liquid* you will need to invoke the corresponding `destroy()` method when you are finished with the object. Now allocate memory for the header and payload data arrays, remembering that they have lengths 12 and 64, respectively. Raw "message" data are stored as arrays of type `unsigned char` in *liquid*.

```
unsigned char header[12];
unsigned char payload[64];
```

Finally you will need to create a buffer for storing the frame samples. For this framing structure you will need to allocate 1280 samples of type `float complex`, viz

```
float complex y[1280];
```

Initialize the header and payload arrays with whatever values you wish. All that is needed to generate a frame is to invoke the frame generator's `execute()` method:

```
framegen64_execute(fg, header, payload, y);
```

That's it! This completely assembles the frame complete with interpolation and is ready for up-conversion and transmission. To generate another frame simply write whatever data you wish to the header and payload buffers, and invoke the `framegen64_execute()` method again as done above. If you wish, print the first few samples of the generated frame to the screen (you will need to separate the *real* and *imaginary* components of each sample).

```
for (i=0; i<30; i++)
    printf("%3u : %12.8f + j*%12.8f\n", i, crealf(y[i]), cimagf(y[i]));
```

Your program should now look similar to this:

```
1  // file: doc/tutorials/framing_basic_tutorial.c
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <math.h>
5  #include <complex.h>
6  #include <liquid/liquid.h>
7
8  int main() {
```

```
9       // options
10      unsigned int m=3;               // filter length (symbols)
11      float beta=0.7f;                // filter excess bandwidth factor
12
13      // allocate memory for arrays
14      unsigned char header[12];       // data header
15      unsigned char payload[64];      // data payload
16      float complex y[1280];          // frame samples
17
18      // create frame generator
19      framegen64 fg = framegen64_create(m,beta);
20      framegen64_print(fg);
21
22      // initialize header, payload
23      unsigned int i;
24      for (i=0; i<12; i++)
25          header[i] = i;
26      for (i=0; i<64; i++)
27          payload[i] = rand() & 0xff;
28
29      // EXECUTE generator and assemble the frame
30      framegen64_execute(fg, header, payload, y);
31
32      // print a few of the generated frame to the screen
33      for (i=0; i<30; i++)
34          printf("%3u : %12.8f + j*%12.8f\n", i, crealf(y[i]), cimagf(y[i]));
35
36      // DESTROY objects
37      framegen64_destroy(fg);
38
39      printf("done.\n");
40      return 0;
41  }
```

Compile the program as before, creating the executable "`framing`." Running the program should produce an output similar to this:

```
framegen64 [m=3, beta=0.70]:
    ramp/up symbols     :   16
    phasing symbols     :   64
    p/n symbols         :   64
    header symbols      :   84
    payload symbols     :   396
    payload symbols     :   396
    ramp\down symbols   :   16
    total symbols       :   640
  0 :   0.00000000 + j*  0.00000000
  1 :   0.00000000 + j*  0.00000000
  2 :  -0.00011255 + j*  0.00000000
  3 :   0.00014416 + j*  0.00000000
  4 :   0.00040660 + j*  0.00000000
        ...
```

```
 25 :   0.04375378 + j*  0.00000000
 26 :   0.97077769 + j*  0.00000000
 27 :  -0.04032370 + j*  0.00000000
 28 :  -1.09209442 + j*  0.00000000
 29 :   0.03534408 + j*  0.00000000
done.
```

You might notice that the *imaginary* component of the samples in the beginning of the frame are zero. This is because the preamble of the frame is BPSK which has no imaginary component at complex baseband.

6.4 Creating the Frame Synchronizer

As stated earlier the frame synchronizer's purpose is to detect the presence of a frame, correct for the channel impairments, decode the data, and pass it back to the user. In our program we will simply pass to the frame synchronizer the samples we generated in the previous section with the frame generator. Furthermore, the hardware interface might pass the baseband samples to the synchronizer in blocks much smaller than the length of a frame (512 samples, for instance) or even blocks much *larger* than the length of a frame (4096 samples, for instance). How does the synchronizer relay the decoded data back to the program without missing any frames? The answer is through the use of a callback function.

What is a callback function? Put quite simply, a callback function is a function pointer (a designated address in memory) that is invoked during a certain event. For this example the callback function given to the `framesync64` synchronizer object when the object is created and is invoked whenever the synchronizer finds a frame. This happens irrespective of the size of the blocks passed to the synchronizer. If you pass it a block of data samples containing four frames—several thousand samples—then the callback will be invoked four times (assuming that channel impairments haven't corrupted the frame beyond the point of recovery). You can even pass the synchronizer one sample at a time if you wish.

The `framesync64` object can be generated with the `framesync64_create()` method which accepts three pointers as arguments:

```
framesync64 framesync64_create(framesyncprops_s *   _props,
                               framesync64_callback _callback,
                               void *               _userdata);
```

`_props` is a construct that defines the specific properties of the frame synchronizer. This includes loop bandwidths for carrier, timing, and gain recovery, as well as squelch and equalizer control. You may pass the value `NULL` to use the default parameters (recommended for now).

`_callback` is a pointer to your callback function which will be invoked each time a frame is found and decoded.

`_userdata` is a `void` pointer that is passed to the callback function each time it is invoked. This allows you to easily pass data from the callback function. Set to `NULL` if you don't wish to use this.

The `framesync64` object has a callback function which has six arguments and looks like this:

```
int framesync64_callback(unsigned char *  _header,
                         int              _header_valid,
                         unsigned char *  _payload,
                         int              _payload_valid,
                         framesyncstats_s _stats,
                         void *           _userdata);
```

The callback is typically defined to be `static` and is passed to the instance of `framesync64` object when it is created.

`_header` is a pointer to the 12 bytes of decoded header data. This pointer is not static and cannot be used after returning from the callback function. This means that it needs to be copied locally for you to retain the data.

`_header_valid` is simply a flag to indicate if the header passed its cyclic redundancy check ("`0`" means invalid, "`1`" means valid). If the check fails then the header data most likely has been corrupted beyond the point that the internal error-correction code can recover; proceed with caution!

`_payload` is a pointer to the 64 bytes of decoded payload data. Like the header, this pointer is not static and cannot be used after returning from the callback function. Again, this means that it needs to be copied locally for you to retain the data.

`_payload_valid` is simply a flag to indicate if the payload passed its cyclic redundancy check ("`0`" means invalid, "`1`" means valid). As with the header, if this flag is zero then the payload most likely has errors in it. Some applications are error tolerant and so it is possible that the payload data are still useful. Typically, though, the payload should be discarded and a re-transmission request should be issued.

`_stats` is a synchronizer statistics construct that indicates some useful PHY information to the user. We will ignore this information in our program, but it can be quite useful for certain applications. For more information on the `framesyncstats_s` structure, see Section 15.6.

`_userdata` Remember that `void` pointer you passed to the `create()` method? That pointer is passed to the callback and can represent just about anything. Typically it points to another structure and is the method by which the decoded header and payload data are returned to the program outside of the callback.

This can seem a bit overwhelming at first, but relax! The next version of our program will only add about 20 lines of code.

6.5 Putting it All Together

First create your callback function at the beginning of the file, just before the `int main()` definition; you may give it whatever name you like (e.g. `mycallback()`). For now ignore all the function inputs and just print a message to the screen that indicates that the callback has been invoked, and return the integer zero (`0`). This return value for the callback function should always be zero and is reserved for future development. Within your `main()` definition, create an instance of `framesync64` using the `framesync64_create()` method, passing it a `NULL` for the first and third arguments (the

properties and userdata constructs) and the name of your callback function as the second argument. Print the newly created synchronizer object to the screen if you like:

```
framesync64 fs = framesync64_create(NULL,
                                    mycallback,
                                    NULL);
framesync64_print(fs);
```

After your line that generates the frame samples ("`framegen64_execute(fg, header, payload, y);`") invoke the synchronizer's `execute()` method, passing to it the frame synchronizer object you just created (`fs`), the pointer to the array of frame symbols (`y`), and the length of the array (1280):

```
framesync64_execute(fs, y, 1280);
```

Finally, destroy the frame synchronizer object along with the frame generator at the end of the file. That's it! Your program should look something like this:

```
1  // file: doc/tutorials/framing_intermediate_tutorial.c
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <math.h>
5  #include <complex.h>
6  #include <liquid/liquid.h>
7
8  // user-defined static callback function
9  static int mycallback(unsigned char * _header,
10                        int _header_valid,
11                        unsigned char * _payload,
12                        int _payload_valid,
13                        framesyncstats_s _stats,
14                        void * _userdata)
15 {
16     printf("***** callback invoked!\n");
17     return 0;
18 }
19
20 int main() {
21     // options
22     unsigned int m=3;               // filter length (symbols)
23     float beta=0.7f;                // filter excess bandwidth factor
24
25     // allocate memory for arrays
26     unsigned char header[12];       // data header
27     unsigned char payload[64];      // data payload
28     float complex y[1280];          // frame samples
29
30     // create frame generator
31     framegen64 fg = framegen64_create(m,beta);
32     framegen64_print(fg);
33
34     // create frame synchronizer using default properties
```

```
35      framesync64 fs = framesync64_create(NULL,
36                                          mycallback,
37                                          NULL);
38      framesync64_print(fs);
39
40      // initialize header, payload
41      unsigned int i;
42      for (i=0; i<12; i++)
43          header[i] = i;
44      for (i=0; i<64; i++)
45          payload[i] = rand() & 0xff;
46
47      // EXECUTE generator and assemble the frame
48      framegen64_execute(fg, header, payload, y);
49
50      // EXECUTE synchronizer and receive the entire frame at once
51      framesync64_execute(fs, y, 1280);
52
53      // DESTROY objects
54      framegen64_destroy(fg);
55      framesync64_destroy(fs);
56
57      printf("done.\n");
58      return 0;
59  }
```

Compile and run your program as before and verify that your callback function was indeed invoked. Your output should look something like this:

```
framegen64 [m=3, beta=0.70]:
    ramp/up symbols     :   16
    phasing symbols     :   64
    ...
framesync64:
    agc signal min/max  :   -40.0 dB /  30.0dB
    agc b/w open/closed :   1.00e-03 / 1.00e-05
    sym b/w open/closed :   8.00e-02 / 5.00e-02
    pll b/w open/closed :   2.00e-02 / 5.00e-03
    samples/symbol      :   2
    filter length       :   3
    num filters (ppfb)  :   32
    filter excess b/w   :   0.7000
    squelch             :   disabled
    auto-squelch        :   disabled
    squelch threshold   :   -35.00 dB
    ----
    p/n sequence len    :   64
    payload len         :   64 bytes
***** callback invoked!
done.
```

As you can see, the `framesync64` object has a long list of modifiable properties pertaining to synchronization; the default values provide a good initial set for a wide range of channel conditions. Duplicate the line of your code that executes the frame synchronizer. Recompile and run your code again. You should see the "`***** callback invoked!`" printed twice.

Your program has only demonstrated the basic functionality of the frame generator and synchronizer under ideal conditions: no noise, carrier offsets, etc. The next section will add some channel impairments to stress the synchronizer's ability to decode the frame.

6.6 Final Program

In this last section we will add some channel impairments to the frame after it is generated and before it is received. This will simulate non-ideal channel conditions. Specifically we will introduce carrier frequency and phase offsets, channel attenuation, and noise. We will also add a frame counter and pass it through the *userdata* construct in the frame synchronizer's `create()` method to be passed to the callback function when a frame is found. Finally, the program will split the frame into pieces to emulate non-contiguous data partitioning at the receiver.

To begin, add the following parameters to the beginning of your `main()` definition with the other options:

```
unsigned int frame_counter = 0; // userdata passed to callback
float phase_offset=0.3f;        // carrier phase offset
float frequency_offset=0.02f;   // carrier frequency offset
float SNRdB = 10.0f;            // signal-to-noise ratio [dB]
float noise_floor = -40.0f;     // noise floor [dB]
```

The `frame_counter` variable is simply a number we will pass to the callback function to demonstrate the functionality of the userdata construct. Make sure to initialize `frame_counter` to zero. If you completed the tutorial on phase-locked loop design you might recognize the `phase_offset` and `frequency_offset` variables; these will be used in the same way to represent a carrier mismatch between the transmitter and receiver. The channel gain and noise parameters are a bit trickier and are set up by the next two lines. Typically the noise power is a fixed value in a receiver; what changes is the received power based on the transmitter's power and the gain of the channel; however because theory dictates that the performance of a link is governed by the ratio of signal power to noise power, SNR is a more useful than defining signal amplitude and noise variance independently. The `SNRdB` and `noise_floor` parameters fully describe the channel in this regard. The noise standard deviation and channel gain may be derived from these values as follows:

```
float nstd  = powf(10.0f, noise_floor*0.1f);
float gamma = powf(10.0f, (SNRdB+noise_floor)*0.1f);
```

Add to your program (after the `framegen64_execute()` line) a loop that modifies each sample of the generated frame by introducing the channel impairments.

$$y_i \leftarrow \gamma y_i e^{j(\theta+i\omega)} + \sigma n e^{j\pi u}$$

where y_i is the frame sample at index i (`y[i]`), γ is the channel gain defined above (`gamma`), θ is the carrier phase offset (`phase_offset`), ω is the carrier frequency offset (`frequency_offset`), σ is the noise standard deviation defined above (`nstd`), and n and u are random variables to represent noise with normal and uniform distributions, repectively. *liquid* provides the `randnf()` and `randf()` methods to generate real random numbers with Gauss and uniform distributions respectively.

```
    y[i] *= gamma;
    y[i] *= cexpf(_Complex_I*(phase_offset + i*frequency_offset));
    y[i] += nstd * randnf() * cexpf(_Complex_I*M_PI*randf());
```

Check the program listed below if you need help.

Now modify the program to incorporate the frame counter. First modify the piece of code where the frame synchronizer is created: replace the last argument (initially set to `NULL`) with the address of our `frame_counter` variable. For posterity's sake, this address will need to be type cast to `void*` (a void pointer) to prevent the compiler from complaining. In your callback function you will reverse this process: create a new variable of type `unsigned int*` (a pointer to an unsigned integer) and assign it the `_userdata` argument type cast to `unsigned int*`. Now dereference this variable and increment its value. Finally print its value near the end of the `main()` definition to ensure it is being properly incremented. Again, check the program below for assistance.

The last task we will do is push one sample at a time to the frame synchronizer rather than the entire frame block to emulate non-contiguous sample streaming. To do this, simply remove the line that calls `framesync64_execute()` on the entire frame and replace it with a loop that calls the same function but with one sample at a time.

The final program is listed below, and a copy of the source is located in the `doc/tutorials/` subdirectory.

```
1  // file: doc/tutorials/framing_tutorial.c
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <math.h>
5  #include <complex.h>
6  #include <liquid/liquid.h>
7
8  // user-defined static callback function
9  static int mycallback(unsigned char * _header,
10                        int _header_valid,
11                        unsigned char * _payload,
12                        int _payload_valid,
13                        framesyncstats_s _stats,
14                        void * _userdata)
15 {
16     printf("***** callback invoked!\n");
17     printf("  header (%s)\n",  _header_valid  ? "valid" : "INVALID");
18     printf("  payload (%s)\n", _payload_valid ? "valid" : "INVALID");
19
20     // type-cast, de-reference, and increment frame counter
21     unsigned int * counter = (unsigned int *) _userdata;
22     (*counter)++;
23
24     return 0;
25 }
26
27 int main() {
28     // options
29     unsigned int m=3;               // filter length (symbols)
30     float beta=0.7f;                // filter excess bandwidth factor
```

```
31      unsigned int frame_counter = 0; // userdata passed to callback
32      float phase_offset=0.3f;        // carrier phase offset
33      float frequency_offset=0.02f;   // carrier frequency offset
34      float SNRdB = 10.0f;            // signal-to-noise ratio [dB]
35      float noise_floor = -40.0f;     // noise floor [dB]
36
37      // allocate memory for arrays
38      unsigned char header[12];       // data header
39      unsigned char payload[64];      // data payload
40      float complex y[1280];          // frame samples
41
42      // create frame generator
43      framegen64 fg = framegen64_create(m,beta);
44      framegen64_print(fg);
45
46      // create frame synchronizer using default properties
47      framesync64 fs = framesync64_create(NULL,
48                                          mycallback,
49                                          (void*)&frame_counter);
50      framesync64_print(fs);
51
52      // initialize header, payload
53      unsigned int i;
54      for (i=0; i<12; i++)
55          header[i] = i;
56      for (i=0; i<64; i++)
57          payload[i] = rand() & 0xff;
58
59      // EXECUTE generator and assemble the frame
60      framegen64_execute(fg, header, payload, y);
61
62      // add channel impairments (attenuation, carrier offset, noise)
63      float nstd  = powf(10.0f, noise_floor*0.1f);        // noise std. dev.
64      float gamma = powf(10.0f, (SNRdB+noise_floor)*0.1f);// channel gain
65      for (i=0; i<1280; i++) {
66          y[i] *= gamma;
67          y[i] *= cexpf(_Complex_I*(phase_offset + i*frequency_offset));
68          y[i] += nstd * randnf()*cexpf(_Complex_I*M_PI*randf());
69      }
70
71      // EXECUTE synchronizer and receive the frame one sample at a time
72      for (i=0; i<1280; i++)
73          framesync64_execute(fs, &y[i], 1);
74
75      // DESTROY objects
76      framegen64_destroy(fg);
77      framesync64_destroy(fs);
78
79      printf("received %u frames\n", frame_counter);
80      printf("done.\n");
81      return 0;
```

```
82 }
```

Compile and run the program as before. The output of your program should look something like this:

```
framegen64 [m=3, beta=0.70]:
    ramp/up symbols     :   16
    phasing symbols     :   64
    ...
framesync64:
    agc signal min/max  :   -40.0 dB /  30.0dB
    agc b/w open/closed :   1.00e-03 / 1.00e-05
    ...
***** callback invoked!
  header (valid)
  payload (valid)
received 1 frames
done.
```

Play around with the initial options, particularly those pertaining to the channel impairments. Under what circumstances does the synchronizer miss the frame? For example, what is the minimum SNR level that is required to reliably receive a frame? the maximum carrier frequency offset?

The "random" noise generated by the program will be seeded to the same value every time the program is run. A new seed can be initialized on the system's time (e.g. time of day) to help generate new instances of random numbers each time the program is run. To do so, include the `<time.h>` header to the top of your file and add the following line to the beginning of your program's `main()` definition:

```
srand(time(NULL));
```

This will ensure a unique simulation is run each time the program is executed. For a more detailed program, see `examples/framesync64_example.c` in the main *liquid* directory. Section 15 describes *liquid*'s framing module in detail.

While the framing structure described in this section provides a simple interface for transmitting and receiving data over a channel, its functionality is limited and isn't particularly spectrally efficient. *liquid* provides a more robust framing structure which allows the use of any linear modulation scheme, two layers of forward error-correction coding, and a variable preamble and payload length. These properties can be reconfigured for each frame to allow fast adaptation to quickly varying channel conditions. Furthermore, the frame synchronizer on the receiver automatically reconfigures itself for each frame it detects to allow as simple an interface possible. The frame generator and synchronizer objects are denoted `flexframegen` and `flexframesync`, respectively, and are described in Section 15. A detailed example program `examples/flexframesync_example.c` is available in the main *liquid* directory.

Part III
Modules

Source code for *liquid* is organized into *modules* which are, for the most part, self-contained elements. The following sections describe these modules in detail with some basic theory behind their operation, functional interface description, and example code.

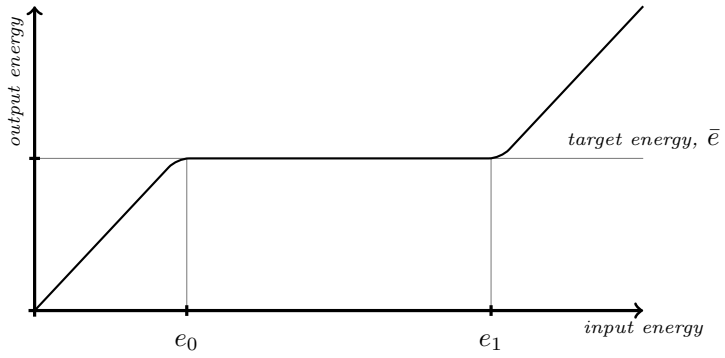


Figure 1: Ideal AGC transfer function of input to output signal energy.

7 agc (automatic gain control)

Normalizing the level of an incoming signal is a critical step in many wireless communications systems and is necessary before further processing can happen in the receiver. This is particularly necessary in digital modulation schemes which encode information in the signal amplitude (e.g. see `MOD_QAM` in Section 18.2). Furthermore, loop filters for tracking carrier and symbol timing are highly sensitive to signal levels and require some degree of amplitude normalization. As such automatic gain control plays a crucial role in SDR. The ideal AGC has a transfer function as in Figure 1. When the input signal level is low, the AGC is disabled and the output is a linear function of the input. When the input level reaches a lower threshold, e_0, the AGC becomes active and the output level is maintained at the target, $\bar{e}$, until the input reaches its upper limit, e_1. The AGC is disabled at this point, and the output level is again a linear function of the input.

liquid implements automatic gain controlling with the `agc_xxxt` family of objects. The goal is to estimate the gain required to force a signal to have a specific target energy, $\bar{e}$. Operating one sample at a time, the `agc` object makes an estimate $\hat{e}$ of the signal energy and updates the internal gain g, applying it to the input to produce an output with the target energy. The gain estimate is updated by way of an open loop filter whose bandwidth determines the update rate of the AGC.

7.1 Theory

Given an input signal $\boldsymbol{x} = \{x_0, x_1, x_2, \ldots, x_{N-1}\}$, its energy is computed as its L_2 norm over the entire sequence, viz

$$E\{\|\boldsymbol{x}\|\} = \left[\sum_{k=0}^{N-1} \|x_k^2\|\right]^{1/2} \tag{8}$$

For received communications signals, however, the goal is to adjust to the gain of the receiver relative to the slowly-varying amplitude of the incoming receiver due to shadowing, path loss, etc. Therefore it is necessary to make an estimate of the instantaneous signal energy, one sample at at time. The short-time average signal energy may be estimated at time k as a partial feedback sum

of the previous estimate

$$\hat{e}_k = \left[\zeta \|x_k\|^2 + (1-\zeta)\hat{e}_{k-1}^2\right]^{1/2} \tag{9}$$

where $\|x_k\|^2$ is the instantaneous signal energy estimate at time k and ζ is the smoothing factor, commensurate of the filter bandwidth.[5] The choice of ζ affects the both the amount of noise allowed in the signal energy estimate, and its speed of adjustment to the dynamics of $\boldsymbol{x}$. *liquid* uses an internal fixed value of $\zeta = 0.1$.

Now that the signal energy has been estimated, all that remains is to adjust the gain of the receiver accordingly. *liquid* implements three types of open-loop gain controls—default, agc, and log—described in the next three sections. Each uses a common loop filter parameter for a given bandwidth, ω, as

$$\alpha \triangleq \sqrt{\omega} \tag{10}$$

If you are unsure which type of AGC to use, we recommend `LIQUID_AGC_DEFAULT` as a good all-around control. The reason for the three types is to allow flexibility in the receiver as each control mechanism has slight performance differences and might require fewer clock cycles, depending upon the architecture of the target platform and the precision of the data type used. Each requires a different complex math operation (division, exponential, logarithm, etc) which can have a varying performance on different processing platforms.

7.2 `LIQUID_AGC_DEFAULT`

The default `agc` type is fairly straightforward. The receiver makes an estimate of the instantaneous input signal energy, given by (9). In order to achieve a target energy $\bar{e}$, the instantaneous ideal gain is therefore the ratio of the target to the estimated energy vectors,

$$\hat{g}_k = \bar{e}/\hat{e}_k \tag{11}$$

Rather than applying the gain directly to the input signal it is first filtered as

$$g_k = \alpha\hat{g}_k + (1-\alpha)g_{k-1} \tag{12}$$

where again $\alpha \triangleq \sqrt{\omega}$ is the smoothing factor of the gain estimate and controls the attack and release time the `agc` object has on an input signal. Notice the similarity the above equation has to (9); because α is typically small, the updated internal gain g_k retains most of its previous gain value g_{k-1} but adds a small portion of its new estimate $\hat{g}_k$.

7.3 `LIQUID_AGC_LOG`

As in the default case, the log `agc` type makes an estimate of the instantaneous signal energy estimate as given by (9) and an estimate of the instantaneous gain as $\hat{g}_k = \bar{e}/\hat{e}_k$. The loop control, however, operates on the logarithm of the gain difference rather than its linear component, and updates its gain estimate accordingly. Therefore the gain error at time $k+1$ can be represented as the log of the ratio of the new instantaneous estimate to the previous gain value, viz

$$\Delta_k = \log\left(\hat{g}_k/g_{k-1}\right) = \log\left(\hat{g}_k\right) - \log\left(g_{k-1}\right) \tag{13}$$

[5] This smoothing factor is necessary to help prevent wild fluctuations in $\hat{e}_k$ which can occur if the input signal's instantaneous amplitude has significant variation, such as in digitally-modulated signals.

The agc then updates its gain estimate according to the magnitude of the error as

$$g_k = g_{k-1} \exp\{\alpha \Delta_k\} \tag{14}$$

As before, α is used to control the sensitivity of the gain update equation, but in this case it is located in the exponent. This gain update gives the AGC an aggressive attack and can suppress large signals very quickly. Notice that when either $\alpha = 0$ or $\Delta_k = 0$ the exponent in (14) becomes 0 and $g_k = g_{k-1}$; therefore the gain is not updated.

7.4 LIQUID_AGC_EXP

As in the default and log cases, the exponential agc type makes an estimate of the instantaneous signal energy as given by (9). The instantaneous output energy is computed as

$$\hat{e}_k = g_{k-1} \hat{e} \tag{15}$$

and the gain error at time k is therefore

$$\Delta_k = \hat{e}_k - \bar{e} \tag{16}$$

The gain estimate is updated proportional to the gain error, viz

$$g_k = g_{k-1} \left(1 - \frac{\alpha \Delta_k}{\text{argmax}\{\bar{e}, \hat{e}_k\}} \right) \tag{17}$$

The multiplier in the above equation on the right is always positive and proportional to the gain error. Notice that when the error is zero ($\hat{e}_k = \bar{e}$) the gain is not updated ($g_k = g_{k-1}$).

7.5 Locking

The agc object permits the gain to be locked when, for example, the header of a frame has been received. This is useful for effectively switching the AGC on and off during short, burt-mode frame transmissions, particularly when the signal has a high-order digital amplitude-modulation (e.g. 64-QAM) and fluctuations in the AGC could potentiallty result in symbol errors. When the agc object is locked, the internal gain control is not updated, and the internal gain at the time of locking is applied directly to the output signal, forcing $g_k = g_{k-1}$. Locking and unlocking is accomplished with the agc_crcf_lock() and agc_crcf_unlock() methods, respectively.

7.6 Squelch

The agc object contains internal squelch control to allow the receiver the ability to disable signal processing when the signal level is too low. In traditional radio design, the squelch circuit suppressed the output of a receiver when the signal strength would fall below a certain level, primarily used to prevent audio static due to noise when no other operators were transmitting. Having said that, the squelch control in *liquid* is actually somewhat of a misnomer as it doesn't actually control the AGC, but rather just monitors the dynamics of the signal level and returns its status to the controlling unit. The squelch control follows six states—enabled, rising edge trigger, signal high, falling edge trigger, signal low, and timeout—as depicted in Figure 2 and Table 1. These states

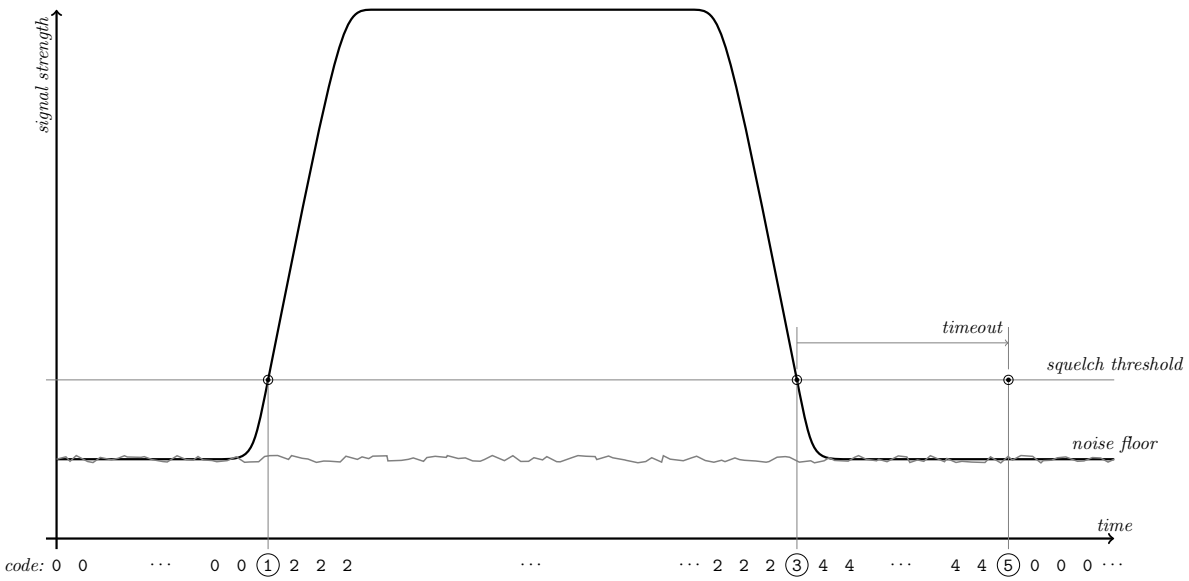


Figure 2: `agc_crcf` squelch

give the user flexibility in programming networks where frames are transmitted in short bursts and the receiver needs to synchronize quickly. The status of the squelch control is retrieved via the `agc_crcf_squelch_get_status()` method.

The typical control cycle for the AGC squelch is depicted in Figure 2. Initially, squelch is enabled (code `0`) as the signal has been low for quite some time. When the beginning of a frame is received, the RSSI increases beyond the squelch threshold (code `1`). All subsequent samples above this threshold return a "signal high" status (code `2`). Once the signal level falls below the threshold, the squelch returns a "falling edge trigger" status (code `3`). All subsequent samples below the threshold until timing out return a "signal low" status (code `4`). When the signal has been low for a sufficient period of time (defined by the user), the squelch will return a "timeout" status (code `5`). All subsequent samples below the threshold will return a "squelch enabled" status.

Table 1: `agc` squelch codes

code	id	description
0	LIQUID_AGC_SQUELCH_ENABLED	squelch enabled
1	LIQUID_AGC_SQUELCH_RISE	rising edge trigger
2	LIQUID_AGC_SQUELCH_SIGNALHI	signal level high
3	LIQUID_AGC_SQUELCH_FALL	falling edge trigger
4	LIQUID_AGC_SQUELCH_SIGNALLO	signal level low, but no timeout
5	LIQUID_AGC_SQUELCH_TIMEOUT	signal level low, timeout

7.6.1 Methodology

The reason for all six states (as opposed to just "squelch on" and "squelch off") are to allow for the AGC to adjust to complex signal dynamics. The default operation for the AGC is to *disable* the squelch. For example if the AGC squelch control is in "signal low" mode (state `4`) and the signal increases above the threshold before timeout, the AGC will move back to the "signal high" mode (state `2`). This is particularly useful for weak signals whose received signal strength is hovering around the squelch threshold; it would be undesireable for the AGC to enable the squelch in the middle of receiving a frame!

7.6.2 auto-squelch

The AGC module also allows for an auto-squelch mechanism which attempts to track the signal threshold to the noise floor of the receiver. This is accomplished by monitoring the signal level when squelch is enabled. The auto-squelch mechanism has a 4dB headroom; if the signal level drops below 4dB beneath the squelch threshold, the threshold will be decremented. This is useful for receiving weak signals slightly above the noise floor, particularly when the exact noise floor is not known or varies slightly over time. Auto-squelch is enabled/disabled using the `agc_crcf_squelch_enable_auto()` and `agc_crcf_squelch_disable_auto()` methods respectively.

7.7 Interface

Listed below is the full interface to the `agc` family of objects. While each method is listed for the `agc_crcf` object, the same functionality applies to the `agc_rrrf` object.

`agc_crcf_create()` creates an `agc` object with default parameters. By default the minimum gain is 10^{-6}, the maximum gain is 10^{6}, the initial gain is 1, and the estimate of the input signal level is 0. Also the AGC type is set to `LIQUID_AGC_DEFAULT`.

`agc_crcf_destroy(q)` destroys the object, freeing all internally-allocated memory.

`agc_crcf_print(q)` prints the `agc` object's internals to `stdout`.

`agc_crcf_reset(q)` resets the state of the `agc` object. This unlocks the AGC and clears the estimate of the input signal level.

`agc_crcf_set_type(q,t)` sets the type of the agc to one of `LIQUID_AGC_DEFAULT`, `LIQUID_AGC_LOG`, or `LIQUID_AGC_EXP`.

`agc_crcf_set_gain_limits(q,gmin,gmax)` sets the minimum and maximum gain values, respectively. This effectively specifies e_0 and e_1 as in Figure 1.

`agc_crcf_lock(q)` prevents the AGC from updating its gain estimate. The internal gain is stored at the time of lock and used for all subsequence occurrances of `_execute()`. This is primarily used when the beginning of a frame has been detected, and perhaps the payload contains amplitude-modulated data which can be corrupted with the AGC agressivly attacking the signal's high dynamics. Also, locking the AGC conserves clock cycles as the gain update is not computed. Typically, the locked AGC consumes about $5\times$ fewer clock cycles than its unlocked state.

`agc_crcf_unlock(q)` unlocks the AGC from a locked state and resumes estimating the input signal level and internal gain.

`agc_crcf_execute(q,x,y)` applies the gain to the input x, storing in the output sample y and updates the AGC's internal tracking loops (of not locked).

`agc_crcf_get_signal_level(q)` returns a linear estimate of the input signal's energy level.

`agc_crcf_get_gain(q)` returns the `agc` object's internal gain.

`agc_crcf_squelch_activate(q)` activates the AGC's squelch module.

`agc_crcf_squelch_deactivate(q)` deactivates the AGC's squelch module.

`agc_crcf_squelch_enable_auto(q)` activates the AGC's automatic squelch module.

`agc_crcf_squelch_disable_auto(q)` deactivates the AGC's automatic squelch module.

`agc_crcf_squelch_set_threshold(q,t)` sets the threshold of the squelch.

`agc_crcf_squelch_set_timeout(q,t)` sets the timeout (number of samples) after the signal level has dropped before enabling the squelch again.

`agc_crcf_squelch_get_status(q)` returns the squelch status code (see Table 1).

Here is a basic example of the `agc` object in *liquid*:

```
1  // file: doc/listings/agc.example.c
2  #include <liquid/liquid.h>
3
4  int main() {
5      agc_rrrf q = agc_rrrf_create();     // create object
6      agc_rrrf_set_target(q,1.0f);        // set target energy level
7      agc_rrrf_set_bandwidth(q,1e-3f);    // set loop filter bandwidth
8
9      float x;                            // input sample
10     float y;                            // output sample
11
12     // ...
13
14     agc_rrrf_execute(q, x, &y);         // repeat as necessary
15
16     agc_rrrf_destroy(q);                // clean it up
17 }
```

A demonstration of the transient response of the `agc_crcf` type can be found in Figure 3 in which an input complex sinusoidal pulse is fed into the AGC. Notice the initial overshoot at the output signal. A few more detailed examples can be found in the `examples` subdirectory.

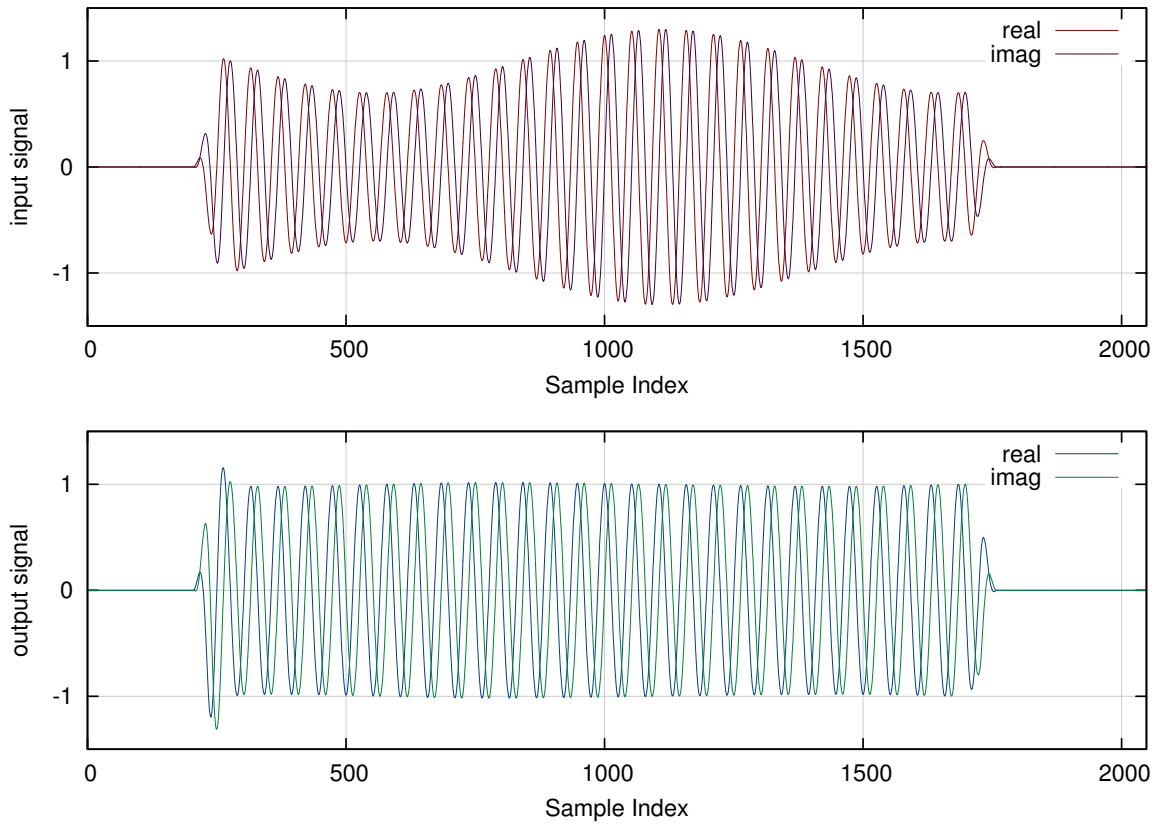


Figure 3: `agc_crcf` transient response

8 audio

The audio module in *liquid* provides several objects and functions for compressing, digitizing, and manipulating audio signals. This is particularly useful for encoding audio data for wireless communications.

8.1 `cvsd` (continuously variable slope delta)

Continuously variable slope delta (CVSD) source encoding is used for data compression of audio signals. CVSD is a lossy compression whose quality is directly related to the sampling frequency and is generally most practical for speech applications. It is a form of delta modulation where Δ (the step size) is changed continuously to minimize slope-overload distortion [24, p. 131]. The output bit stream has a rate equal to that of the sampling frequency. It is considered to be a moderate compromise between quality and complexity.

8.1.1 Theory

The algorithm attempts to dynamically adjust the value of Δ to track to the input signal. As with regular delta modulation algorithms, if the decoded reference signal exceeds the input (the error signal is negative), a binary `0` is sent and Δ is subtracted from the reference, otherwise a binary `1` is sent and Δ is added. However CVSD observes the previous N transmitted bits are stored in a buffer $\hat{\boldsymbol{b}}$; Δ is increased by ζ if they are equal and decreased otherwise. This improves the dynamic range of the encoder over fixed-delta modulation encoders. A summary of the encoding procedure can be found in Algorithm 2.

Algorithm 2 CVSD encoder algorithm

$\boldsymbol{x} \leftarrow \{x_0, x_1, x_2, \ldots\}$ (input audio samples)
$v_0 \leftarrow 0$ (initial output reference)
$\Delta_0 \leftarrow \Delta_{min}$ (initialize step size)
$\hat{\boldsymbol{b}}_0 \leftarrow \{0, 0, \ldots, 0\}$ (initialize N-bit buffer)
for $k = 0, 1, 2, \ldots$ **do**
    $b_k \leftarrow \begin{cases} 0 & v_k > x_k \\ 1 & \text{else} \end{cases}$ (compute output bit)
    $\hat{\boldsymbol{b}}_k \leftarrow \{\hat{b}_1, \hat{b}_2, \ldots, \hat{b}_{N-1}, b_k\}$ (append output bit to end of buffer)
    $m \leftarrow \sum_{i=0}^{N-1} \hat{\boldsymbol{b}}_i$ (compute sum of last N bits)
    $\Delta_k \leftarrow \begin{cases} \Delta_{k-1}\zeta & m = 0, m = N \\ \Delta_{k-1}/\zeta & \text{else} \end{cases}$ (adjust step size)
    $v_{k+1} \leftarrow v_k + (-1)^{1-b_k}\Delta_k$ (adjust reference value)
end for

The decoder reverses this process; by retaining the past N bit inputs in a buffer $\hat{\boldsymbol{b}}$, the value of Δ can be adjusted appropriately. A summary of the decoding procedure can be found in Algorithm 3.

Algorithm 3 CVSD decoder algorithm

$\boldsymbol{b} \leftarrow \{b_0, b_1, b_2, \ldots\}$ (input bit samples)
$v_0 \leftarrow 0$ (initial output reference)
$\Delta_0 \leftarrow \Delta_{min}$ (initialize step size)
$\hat{\boldsymbol{b}}_0 \leftarrow \{0, 0, \ldots, 0\}$ (initialize N-bit buffer)
for $k = 0, 1, 2, \ldots$ **do**
    $\hat{\boldsymbol{b}}_k \leftarrow \{\hat{b}_1, \hat{b}_2, \ldots, \hat{b}_{N-1}, b_k\}$ (append output bit to end of buffer)
    $m \leftarrow \sum_{i=0}^{N-1} \hat{\boldsymbol{b}}_i$ (compute sum of last N bits)
    $\Delta_k \leftarrow \begin{cases} \Delta_{k-1}\zeta & m = 0, m = N \\ \Delta_{k-1}/\zeta & \text{else} \end{cases}$ (adjust step size)
    $v_{k+1} \leftarrow v_k + (-1)^{1-b_k}\Delta_k$ (adjust reference value)
    $y_k \leftarrow v_k$ (set output value)
end for

8.1.2 Pre-/Post-Filtering

To preserve the signal's integrity the encoder applies a pre-filter to emphasize the high-frequency information of the signal before the encoding process. The pre-filter is a simple 2-tap FIR filter defined as

$$H_{pre}(z) = 1 - \alpha z^{-1} \tag{18}$$

where α controls the amount of emphasis applied. Typical values fore pre-emphasis are $0.92 < \alpha < 0.98$; setting $\alpha = 0$ completely disables this emphasis. This process is reversed on the decoder by applying the inverse of $H_{pre}(z)$ as a low-pass de-emphasis filter:

$$H_{pre}^{-1}(z) = \frac{1}{1 - \alpha z^{-1}} \tag{19}$$

Additionally, the decoder adds a DC-blocking filter to reject any residual offset caused by the decoding process. By itself the DC-blocking filter has a transfer function

$$H_0(z) = \frac{1 - z^{-1}}{1 - \beta z^{-1}} \tag{20}$$

where β controls the cut-off frequency of the filter and is typically set very close to 1. The default value for β in *liquid* is 0.99. The full post-emphasis filter is therefore

$$H_{post}(z) = H_{pre}^{-1}(z)H_0(z) = \frac{1 - z^{-1}}{1 - (\alpha + \beta)z^{-1} + \alpha\beta z^{-2}} \tag{21}$$

8.1.3 Interface

The `cvsd` object in *liquid* allows the user to select both ζ as well as N, the number of repeated bits observed before Δ is updated. The combination of these values with the sampling rate yields a speech compression algorithm with moderate quality. Listed below is the full interface to the `cvsd` object:

`cvsd_create(N,zeta,alpha)` creates an `agc` object with parameters N, ζ, and α.

`cvsd_destroy(q)` destroys a `cvsd` object, freeing all internally-allocated memory and objects.

`cvsd_print(q)` prints the `cvsd` object's internal parameters to the standard output.

`cvsd_encode(q,sample)` encodes a single audio sample, returning the encoded bit.

`cvsd_decode(q,bit)` decodes and returns a single audio sample from an input bit.

`cvsd_encode8(q,samples,byte)` encodes a block of 8 samples returning the result in a single byte.

`cvsd_decode8(q,byte,samples)` decodes a block of 8 samples from an encoded byte.

8.1.4 Example

Here is a basic example of the `cvsd` object in *liquid*:

```
1  // file: doc/listings/cvsd.example.c
2  #include <liquid/liquid.h>
3
4  int main() {
5      // options
6      unsigned int nbits=3;       // number of adjacent bits to observe
7      float zeta=1.5f;            // slope adjustment multiplier
8      float alpha = 0.95;         // pre-/post-filter coefficient
9
10     // create cvsd encoder/decoder
11     cvsd q = cvsd_create(nbits, zeta, alpha);
12
13     float x;                    // input sample
14     unsigned char b;            // encoded bit
15     float y;                    // output sample
16
17     // ...
18
19     // repeat as necessary
20     {
21         b = cvsd_encode(q, x);  // encode sample
22
23         y = cvsd_decode(q, b);  // decode sample
24     }
25
26     cvsd_destroy(q);            // destroy cvsd object
27 }
```

A demonstration of the algorithm can be seen in Figure 4 where the encoder attempts to track to an input sinusoid. Notice that the encoder sometimes overshoots the reference signal. This distortion results in degradations, particularly in the upper frequency bands. A more detailed example is given in `examples/cvsd_example.c` under the main *liquid* project directory.

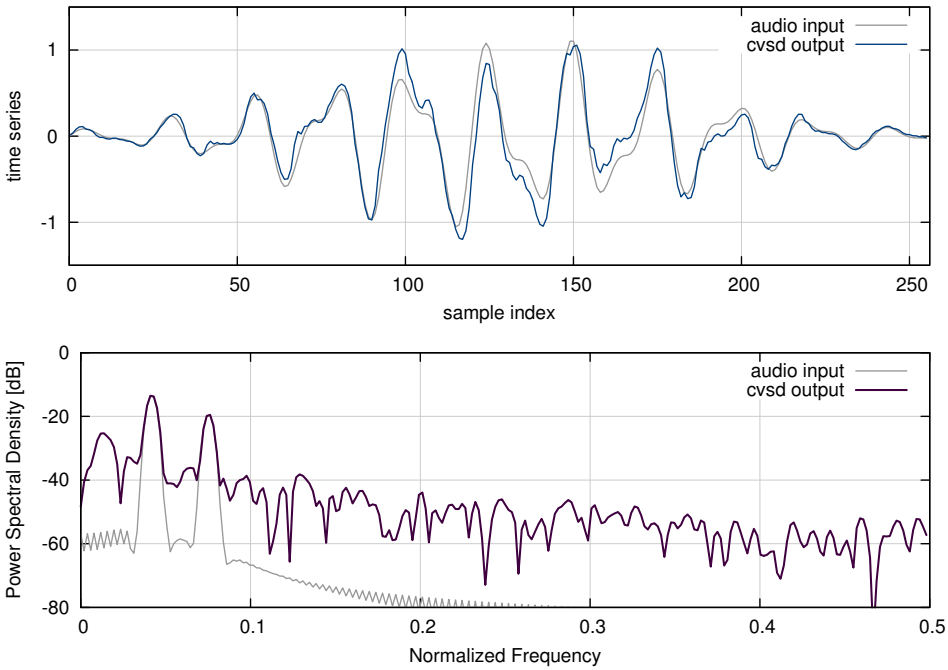


Figure 4: `cvsd` example encoding a windowed sum of sine functions with $\zeta = 1.5$, $N = 2$, and $\alpha = 0.95$.

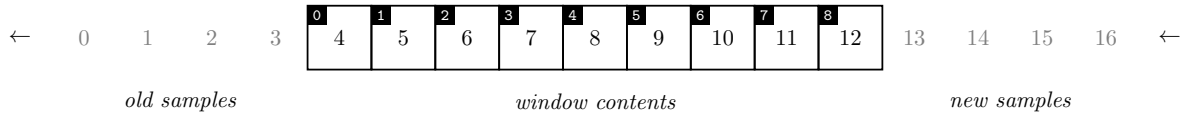
9 buffer

The buffer module includes objects for storing, retrieving, and interfacing with buffered data samples.

9.1 `window` buffer

The `window` object is used to implement a sliding window buffer. It is essentially a first-in, first-out queue but with the constraint that a fixed number of elements is always available, and the ability to read the entire queue at once. This is particularly useful for filtering objects which use time-domain convolution of a fixed length to compute its outputs. Unlike the `gport` object, `window` objects operate on a known data type, e.g. *float* (`windowf`), and *float complex* (`windowcf`).

The buffer has a fixed number of elements which are initially zeros. Values may be pushed into the end of the buffer (into the "right" side) using the `push()` method, or written in blocks via `write()`. In both cases the oldest data samples are removed from the buffer (out of the "left" side). When it is necessary to read the contents of the buffer, the `read()` method returns a pointer to its contents. *liquid* implements this shifting method in the same manner as a ring buffer, and linearizes the data very efficiently, without performing any unnecessary data memory copies. Effectively, the window looks like:



Listed below is the full interface for the `window` family of objects. While each method is listed for `windowcf` (a window with `float complex` elements), the same functionality applies to the `windowf` object.

`windowcf_create(n)` creates a new window with an internal length of n samples.

`windowcf_recreate(q,n)` extends an existing window's size, similar to the standard C library's `realloc()` to n samples. If the size of the new window is larger than the old one, the newest values are retained at the beginning of the buffer and the oldest values are truncated. If the size of the new window is smaller than the old one, the oldest values are truncated.

`windowcf_destroy(q)` destroys the object, freeing all internally-allocated memory.

`windowcf_clear(q)` clears the contents of the buffer by setting all internal values to zero.

`windowcf_index(q,i,*v)` retrieves the i^{th} sample in the window, storing the output value in v. This is equivalent to first invoking `read()` and then indexing on the resulting pointer; however the result is obtained much faster. Therefore invoking `windowcf_index(q,0,*v)` returns the *oldest* value in the window.

`windowcf_read(q,**r)` reads the contents of the window by returning a pointer to the aligned internal memory array. This method guarantees that the elements are linearized. This method should *only* be used for reading; writing values to the buffer has unspecified results.

`windowcf_push(q,v)` shifts a single sample v into the right side of the window, pushing the oldest (left-most) sample out of the end. Unlike stacks, the `windowcf` object has no equivalent "pop" method, as values are retained in memory until they are overwritten.

`windowcf_write(q,*v,n)` writes a block of n samples in the array $\boldsymbol{v}$ to the window. Effectively, it is equivalent to pushing each sample one at a time, but executes much faster.

Here is an example demonstrating the basic functionality of the window object. The comments show the internal state of the window after each function call as if the window were a simple C array.

```
// file: doc/listings/window.example.c
#include <liquid/liquid.h>

int main() {
    // initialize array for writing
    float v[] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};

    // create window with 10 elements
    windowf w = windowf_create(10);
    // window[10] : {0 0 0 0 0 0 0 0 0 0}

    // push 4 elements into the window
    windowf_push(w, 1);
    windowf_push(w, 3);
    windowf_push(w, 6);
    windowf_push(w, 2);
    // window[10] : {0 0 0 0 0 0 1 3 6 2}

    // push 4 elements at a time
    windowf_write(w, v, 4);
    // window[10] : {0 0 1 3 6 2 9 8 7 6}

    // recreate window (truncate to last 6 elements)
    w = windowf_recreate(w,6);
    // window[6] : {6 2 9 8 7 6}

    // recreate window (extend to 12 elements)
    w = windowf_recreate(w,12);
    // window[12] : {0 0 0 0 0 0 6 2 9 8 7 6}

    // read buffer (return pointer to aligned memory)
    float * r;
    windowf_read(w, &r);
    // r[12] : {0 0 0 0 0 0 6 2 9 8 7 6}

    // clean up allocated object
    windowf_destroy(w);
}
```

9.2 wdelay delay buffer

The `wdelay` object in *liquid* implements a an efficient digital delay line with a minimal amount of memory. Specifically, the transfer function is just

$$H_d(z) = z^{-k} \tag{22}$$

where k is the number of samples of delay. The interface for the `wdelay` family of objects is listed below. While the interface is given for `wdelayf` for floating-point precision, equivalent interfaces exist for `float complex` with `wdelaycf`.

`wdelayf_create(k)` creates a new `wdelayf` object with a delay of k samples.

`wdelayf_recreate(q,k)` adjusts the delay size, preserving the internal state of the object.

`wdelayf_destroy(q)` destroys the object, freeing all internally-allocated memory.

`wdelayf_print(q)` prints the object's properties internal state to the standard output.

`wdelayf_clear(q)` clears the contents of the internal buffer by setting all values to zero.

`wdelayf_read(q,y)` reads the sample at the head of the buffer and stores it to the output pointer.

`wdelayf_push(q,x)` pushes a sample into the buffer.

10 dotprod (vector dot product)

This module provides interfaces for computing a vector dot product between two equally-sized vectors. Dot products are commonly used in digital signal processing for communications, particularly in filtering and matrix operations. Given two vectors of equal length $\boldsymbol{x} = [x(0), x(1), \ldots, x(N-1)]^T$ and $\boldsymbol{v} = [v(0), v(1), \ldots, v(N-1)]^T$, the vector dot product between them is computed as

$$\boldsymbol{x} \cdot \boldsymbol{v} = \boldsymbol{x}^T \boldsymbol{v} = \sum_{k=0}^{N-1} x(k) v(k) \tag{23}$$

A number of other *liquid* modules rely on `dotprod`, such as filtering and equalization.

10.1 Specific machine architectures

The vector dot product has a complexity of $\mathcal{O}(N)$ multiply-and-accumulate operations. Because of its prevalence in multimedia applications, a considerable amount of research has been put into computing the vector dot product as efficiently as possible. Software-defined radio is no exception as basic profiling will likely demonstrate that a considerable portion of the processor is spent computing it. Certain machine architectures have specific instructions for computing vector dot products, particularly those which use a single instruction for multiple data (SIMD) such as MMX, SSE, AltiVec, etc.

10.2 Interface

There are effectively two ways to use the `dotprod` module. In the first and most general case, a vector dot product is computed on two input vectors $\boldsymbol{x}$ and $\boldsymbol{v}$ whose values are not known *a priori*. In the second case, a `dotprod` object is created around vector $\boldsymbol{v}$ which does not change (or rarely changes) throughout its life cycle. This is the more convenient method for filtering objects which don't usually have time-dependent coefficients. Listed below is a simple interface example to the `dotprod` module object:

```
1  // file: doc/listings/dotprod_rrrf.example.c
2  #include <liquid/liquid.h>
3
4  int main() {
5      // create input arrays
6      float x[] = { 1.0f,  2.0f,  3.0f,  4.0f,  5.0f};
7      float v[] = { 0.1f, -0.2f,  1.0f, -0.2f,  0.1f};
8      float y;
9
10     // run the basic vector dot product, store in 'y'
11     dotprod_rrrf_run(x,v,5,&y);
12
13     // create dotprod object and execute, store in 'y'
14     dotprod_rrrf q = dotprod_rrrf_create(v,5);
15     dotprod_rrrf_execute(q,x,&y);
16     dotprod_rrrf_destroy(q);
17 }
```

Table 2: `dotprod` object types

precision	*input/output*	*coefficients*	*interface*
float	real	real	`dotprod_rrrf`
float	complex	complex	`dotprod_cccf`
float	complex	real	`dotprod_crcf`

In both cases the `dotprod` can be easily integrated with the `window` object (Section 9.1) for managing input data and alignment. There are three types of dot product objects and are listed in Table 2.

Listed below is a brief description of the `dotprod` object interfaces. While the types are described using the `dotprod_rrrf` object, the same holds true for all other types.

`dotprod_rrrf_run(h,x,n,y)` executes a vector dot product between two vectors $\boldsymbol{h}$ and $\boldsymbol{x}$, each of length n and stores the result in the output y. This is not a structured method and does not require creating a `dotprod` object, however does not take advantage of SIMD instructions if available. Rather than speed, its intent is to provide a simple interface to demonstrate functional correctness.

`dotprod_rrrf_create(v,n)` creates a `dotprod` object with coefficients $\boldsymbol{v}$ of length n.

`dotprod_rrrf_recreate(q,v,n)` recreates a `dotprod` object with a new set of coefficients $\boldsymbol{v}$ with a (possibly) different length n.

`dotprod_rrrf_destroy(q)` destroys a `dotprod` object, freeing all internally-allocated memory.

`dotprod_rrrf_print(q)` prints the object internals to the screen.

`dotprod_rrrf_execute(q,x,y)` executes a dot product with an input vector $\boldsymbol{x}$ and stores the result in y.

11 equalization

This section describes the equalizer module and the functionality of two digital linear adaptive equalizers implemented in *liquid*, LMS and RLS. Their interfaces are nearly identical; however their internal functionality is quite different. Specifically the LMS algorithm is less computationally complex but is slower to converge than the RLS algorithm.

11.1 System Description

Suppose a known transmitted symbol sequence $\boldsymbol{d} = [d(0), d(1), \ldots, d(N-1)]$ which passes through an unknown channel filter $\boldsymbol{h}_n$ of length q. The received symbol at time n is therefore

$$y(n) = \sum_{k=0}^{q-1} h_n(k) d(n-k) + \varphi(n) \tag{24}$$

where $\varphi(n)$ represents white Gauss noise. The adaptive linear equalizer attempts to use a finite impulse response (FIR) filter $\boldsymbol{w}$ of length p to estimate the transmitted symbol, using only the received signal vector $\boldsymbol{y}$ and the known data sequence $\boldsymbol{d}$, viz

$$\hat{d}(n) = \boldsymbol{w}_n^T \boldsymbol{y}_n \tag{25}$$

where $\boldsymbol{y}_n = [y(n), y(n-1), \ldots, y(n-p+1)]^T$. Several methods for estimating $\boldsymbol{w}$ are known in the literature, and typically rely on iteratively adjusting $\boldsymbol{w}$ with each input though a recursion algorithm. This section provides a very brief overview of two prevalent adaptation algorithms; for a more in-depth discussion the interested reader is referred to [24, 13].

11.2 `eqlms` (least mean-squares equalizer)

The least mean-squares (LMS) algorithm adapts the coefficients of the filter estimate using a steepest descent (gradient) of the instantaneous *a priori* error. The filter estimate at time $n+1$ follows the following recursion

$$\boldsymbol{w}_{n+1} = \boldsymbol{w}_n - \mu \boldsymbol{g}_n \tag{26}$$

where μ is the iterative step size, and $\boldsymbol{g}_n$ the normalized gradient vector, estimated from the error signal and the coefficients vector at time n.

11.3 `eqrls` (recursive least-squares equalizer)

The recursive least-squares (RLS) algorithm attempts to minimize the time-average weighted square error of the filter output, viz

$$c(\boldsymbol{w}_n) = \sum_{i=0}^{n} \lambda^{i-n} \left| d(i) - \hat{d}(i) \right|^2 \tag{27}$$

where the forgetting factor $0 < \lambda \leq 1$ which introduces exponential weighting into past data, appropriate for time-varying channels. The solution to minimizing the cost function $c(\boldsymbol{w}_n)$ is achieved by setting its partial derivatives with respect to $\boldsymbol{w}_n$ equal to zero. The solution at time n involves inverting the weighted cross correlation matrix for $\boldsymbol{y}_n$, a computationally complex task.

This step can be circumvented through the use of a recursive algorithm which attempts to estimate the inverse using the *a priori* error from the output of the filter. The update equation is

$$\boldsymbol{w}_{n+1} = \boldsymbol{w}_n + \Delta_n \tag{28}$$

where the correction factor Δ_n depends on $\boldsymbol{y}_n$ and $\boldsymbol{w}_n$, and involves several $p \times p$ matrix multiplications. The RLS algorithm provides a solution which converges much faster than the LMS algorithm, however with a significant increase in computational complexity and memory requirements.

11.4 Interface

The `eqlms` and `eqrls` have nearly identical interfaces so we will leave the discussion to the `eqlms` object here. Like most objects in *liquid*, `eqlms` follows the typical `create()`, `execute()`, `destroy()` lifecycle. Training is accomplished either one sample at a time, or in a batch cycle. If trained one sample at a time, the symbols must be trained in the proper order, otherwise the algorithm won't converge. One can think of the equalizer object in *liquid* as simply a `firfilt` object (finite impulse response filter) which has the additional ability to modify its own internal coefficients based on some error criteria. Listed below is the full interface to the `eqlms` family of objects. While each method is listed for `eqlms_cccf`, the same functionality applies to `eqlms_rrrf` as well as the RLS equalizer objects (`eqrls_cccf` and `eqrls_rrrf`).

`eqlms_cccf_create(*h,n)` creates and returns an equalizer object with n taps, initialized with the input array $\boldsymbol{h}$. If the array value is set to the `NULL` pointer then the internal coefficients are initialized to $\{1, 0, 0, \ldots, 0\}$.

`eqlms_cccf_destroy(q)` destroys the equalizer object, freeing all internally-allocated memory.

`eqlms_cccf_print(q)` prints the internal state of the `eqlms` object.

`eqlms_cccf_set_bw(q,w)` sets the bandwidth of the equalizer to w. For the LMS equalizer this is the learning parameter μ which has a default value of 0.5. For the RLS equalizer the "bandwidth" is the forgetting factor λ which defaults to 0.99.

`eqlms_cccf_reset(q)` clears the internal equalizer buffers and sets the internal coefficients to the default (those specified when `create()` was invoked).

`eqlms_cccf_push(q,x)` pushes a sample x into the internal buffer of the equalizer object.

`eqlms_cccf_execute(q,*y)` generates the output sample y by computing the vector dot product (see Section 10) between the internal filter coefficients and the internal buffer.

`eqlms_cccf_step(q,d,d_hat)` performs a single iteration of equalization with an estimated output $\hat{d}$ for an expected output d. The weights are updated internally defined by (26) for the LMS equalizer and (28) for the RLS equalizer.

`eqlms_cccf_get_weights(q,*w)` returns the internal filter coefficients (weights) at the current state of the equalizer.

Here is a simple example:

```c
// file: doc/listings/eqlms_cccf.example.c
#include <liquid/liquid.h>

int main() {
    // options
    unsigned int n=32;          // number of training symbols
    unsigned int p=10;          // equalizer order
    float mu=0.500f;            // LMS learning rate

    // allocate memory for arrays
    float complex x[n];         // received samples
    float complex d_hat[n];     // output symbols
    float complex d[n];         // traning symbols

    // ...initialize y, d, w...

    // create LMS equalizer and set learning rate
    eqlms_cccf q = eqlms_cccf_create(NULL,p);
    eqlms_cccf_set_bw(q, mu);

    // iterate through equalizer learning
    unsigned int i;
    {
        // push input sample
        eqlms_cccf_push(q, x[i]);

         // compute output sample
        eqlms_cccf_execute(q, &d_hat[i]);

        // update internal weights
        eqlms_cccf_step(q, d[i], d_hat[i]);
    }

    // clean up allocated memory
    eqlms_cccf_destroy(q);
}
```

For more detailed examples, see `examples/eqlms_cccf_example.c` and `examples/eqrls_cccf_example.c`.

11.5 Demonstration

The performance of the `eqlms` and `eqrls` equalizers are compared by generating a channel with an impulse response representing a strong line-of-sight (LoS) component followed by random echoes. Each was trained on 512 iterations of a known QPSK-modulated training sequence with learning rate parameters $\mu = 0.999$ and $\lambda = 0.999$ for the LMS and RLS algorithms, respectively. A small amount of noise was injected after the channel filter to demonstrate the robustness of the algorithms. The results of two simulations are shown in figures 5 and 6; the first demonstrating a 10-tap equalizer applied to the response of a 6-tap channel with an SNR of 40 dB, while the second demonstrates a 18-tap equalizer for a 8-tap channel with an SNR of just 14 dB.

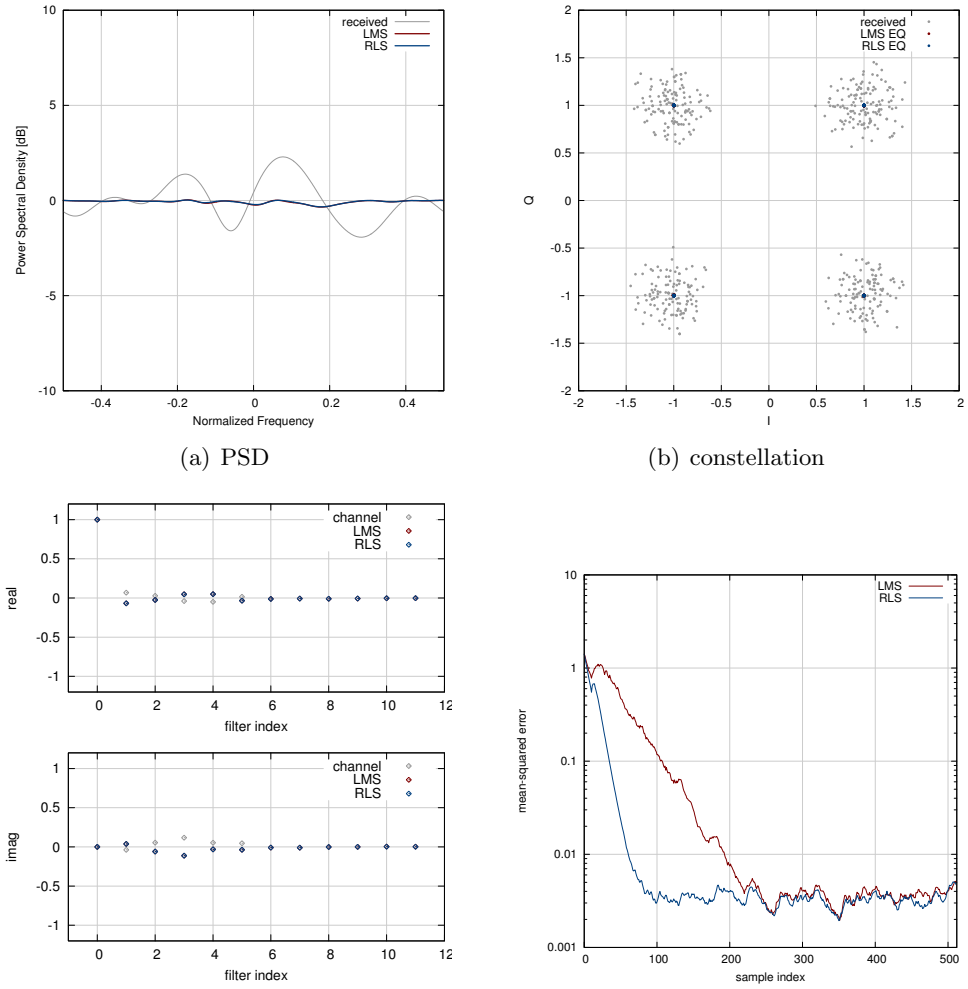
The passband power spectral densities (PSD) of the channel and the equalizer outputs are

depicted in figures 5(a) and 6(a). Notice that the inter-symbol interference of the channel causes its PSD to have a non-flat response. Theoretically, if the inter-symbol interference is completely removed, the response of both the channel and the equalizer will be completely flat (neglecting any noise present). While the PSD of the equalized output is nearly flat in the figures, it is important to realize that these algorithms minimize a cost function defined as the square of the *a priori* filter output error, and do not necessarily force the PSD to zero. The classic zero-forcing equalizer has several drawbacks:

1. the equalizing filter which would give this response is not necessarily realizable; that is, not all channels can be perfectly inverted,

2. forcing the frequency response to zero increases the noise terms of frequencies where the spectra of the channel response is low. In this regard, the zero-forcing equalizer only reduces inter-symbol interference and does not maximize the ratio of signal power to both interference *and* noise power as the LMS and RLS algorithms do.

It is interesting to note that both the LMS and RLS equalizers converge to nearly the same solution. In both scenarios, however, the RLS equalizer has a slightly lower error after training while converging to its error minimum much faster. The RLS equalizer, however, has a much higher computational complexity.



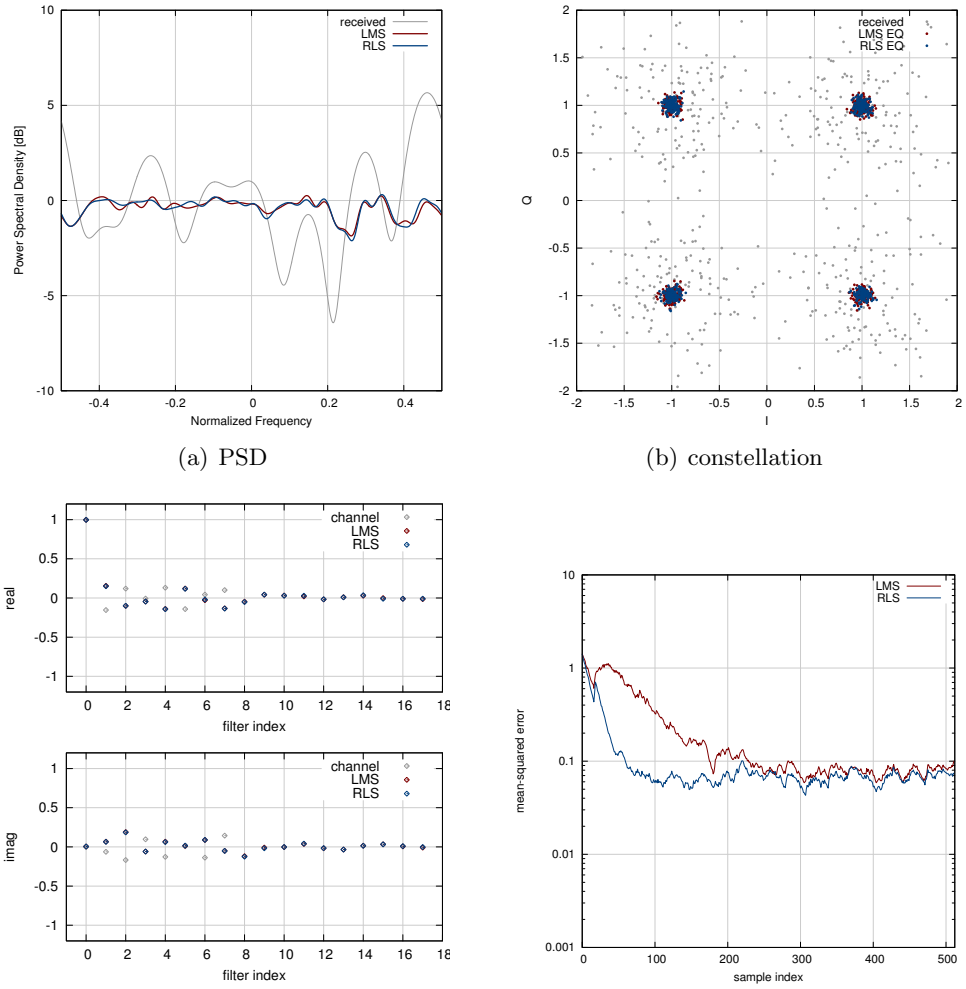
(a) PSD

(b) constellation

(c) taps

(d) mean-squared error

Figure 5: Scenario 1: 10-tap equalizer for a 6-tap channel with 40 dB SNR



(a) PSD (b) constellation

(c) taps (d) mean-squared error

Figure 6: Scenario 2: 18-tap equalizer for a 8-tap channel with 14 dB SNR

12 fec (forward error correction)

The fec module implements a set of forward error-correction codes for ensuring and validating data integrity through a noisy channel. Redundant "parity" bits are added to a data sequence to help correct errors introduced by the channel. The number of correctable errors depends on the number of parity bits of the coding scheme, which in turn affects its rate (efficiency). The `fec` object realizes forward error-correction capabilities in *liquid* while the methods `checksum()` and `crc32()` strictly implement error detection. Certain FEC schemes are only available to *liquid* by installing the external `libfec` library [23], available as a free download. A few low-rate (and fairly low efficiency) codes are availble internally.

12.1 Cyclic Redundancy Check (Error Detection)

A cyclic redundancy check (CRC) is, in essence, a strong algebraic error detection code that computes a key on a block of data using base-2 polynomials. While it is a strong error-detection method, a CRC is not an error-correction code. Here is a simple example:

```
1  // file: doc/listings/crc.example.c
2  #include <liquid/liquid.h>
3
4  int main() {
5      // initialize data array
6      unsigned char data[4] = {0x25, 0x62, 0x3F, 0x52};
7      crc_scheme scheme = LIQUID_CRC_32;
8
9      // compute CRC on original data
10     unsigned char key = crc_generate_key(scheme, data, 4);
11
12     // ... channel ...
13
14     // validate (received) message
15     int valid_data = crc_validate_message(scheme, data, 4, key);
16 }
```

Also available for error detection in *liquid* is a checkssum. A checksum is a simple way to validate data received through un-reliable means (e.g. a noisy channel). A checksum is, in essence, a weak error detection code that simply counts the number of ones in a block of data (modulo 256). The limitation, however, is that multiple bit errors might result in a false positive validation of the corrupted data. The checksum is not a strong an error detection scheme as the cyclic redundany check. Table 3 lists the available codecs and gives a brief description for each. For a detailed example program, see `examples/crc_example.c` in the main *liquid* directory.

12.2 `h74`, `h84`, `h128` (Hamming codes)

Hamming codes are a specific type of block code which use parity bits capable of correcting one bit error in the block. With the addition of an extra parity bit, they are able to detect up to two errors, but are still only able to correct one. *liquid* implements the Hamming(7,4), Hamming(8,4), and Hamming(12,8) codes. The Hamming(8,4) can detect one additional error over the Hamming(7,4) code; however at the time of writing this document the number of detected errors is not passed to

Table 3: Error-detection codecs available in *liquid*

scheme	*size (bits)*	*description*
LIQUID_CRC_UNKNOWN	-	unknown/unsupported scheme
LIQUID_CRC_NONE	0	no error-detection
LIQUID_CRC_CHECKSUM	8	basic checksum
LIQUID_CRC_8	8	8-bit CRC, poly=0x07
LIQUID_CRC_16	16	16-bit CRC, poly=0x8005
LIQUID_CRC_24	24	24-bit CRC, poly=0x5D6DCB
LIQUID_CRC_32	32	32-bit CRC, poly=0x04C11DB7

the user so the Hamming(8,4) code is effectively the same as Hamming(7,4) but with a lower rate. Additionally, *liquid* implements the Hamming(12,8) code which accepts an 8-bit symbol and adds four parity bits, extending it to a 12-bit symbol. This yields a theoretical rate of 2/3, and actually has a performance very similar to that of the Hamming(7,4) code, even with a higher rate.

12.3 `rep3`, `rep5` (simple repeat codes)

The `rep3` code is a simple repeat code which simply repeats the message twice (transmits it three times). The decoder takes a majority vote of the bits received by applying a simple series bit masks. If the original bit is represented as s, then the transmitted bits are sss. Let the received bit sequence be $r_0 r_1 r_2$. The estimated transmitted bit is `0` if the sum of the received bits is less than 2, and `1` otherwise. This is equivalent to

$$\hat{s} = (r_0 \wedge r_1) + (r_0 \wedge r_2) + (r_1 \wedge r_2)$$

where $+$ represents logical *or* and $\wedge$ represents logical *and*. An error is detected if

$$\hat{e} = (r_0 \oplus r_1) + (r_0 \oplus r_2) + (r_1 \oplus r_2)$$

where $\oplus$ represents logical *exclusive or*. In this fashion it is easy to decode several bytes of data at a time because machine architectures have low-level bit-wise manipulation instructions which can compute logical *exclusive or* and *or* very quickly. This is precisely how *liquid* decodes `rep3` data, only in this case, s, r_0, r_1, and r_2 represent a bytes of data rather than bits.

The `rep5` code operates similarly, except that it transmits five copies of the original data sequence, rather than just three. The decoder takes the five received bits $r_0, \ldots, r_4$ and adds (modulo 2) the logical *and* of every combination of three bits, viz

$$\hat{s} = \sum_{i \neq j \neq k} (r_i \wedge r_j \wedge r_k)$$

This roughly doubles the number of clock cycles to decode over `rep3`.

It is well-known that repeat codes do not have strong error-correction capabilities for their rate, are are located far from the Shannon capacity bound [24]. They are exceptionally weak relative to convolutional Viterbi and Reed-Solomon codes. However, their simplicity in implementation and low computational complexity gains them a place in digital communications, particularly in software radios where spectral efficiency goals might be secondary to processing contraints.

12.4 `libfec` (convolutional and Reed-Solomon codes)

liquid takes advantage of convolutional and Reed-Solomon codes defined in `libfec` [23]. These codes have much stronger error-correction capabilities than `rep3`, `rep5`, `h74`, `h84`, and `h128` but are also much more computationally intensive to the host processor. *liquid* uses the rate $1/2(K = 7)$, $1/2(K = 9)$, $1/3(K = 9)$, and $r1/6(K = 15)$ codes defined in `libfec`, but extends the two half-rate codes to punctured codes. These punctured codes (also known as "perforated" codes) are not as strong and cannot correct as many errors, but are more efficient and use less overhead than their half-rate counterparts. The 8-bit Reed-Solomon code is a (255,223) block code, also defined in `libfec`. Nominally, the scheme accepts 223 bytes (8-bit symbols) and adds 32 parity symbols to form a 255-symbol encoded block. `libfec` is an external library that *liquid* will leverage if installed, but will still compile otherwise (see Section 25.1 for details).

12.5 Interface

In designing the `fec` interface, we have tried to keep simplicity and reconfigurability in mind. The various forward error-correction schemes accept bits or symbols formatted in different lengths and have vastly different interfaces. This potentially makes switching from one scheme to another difficult as one needs to restructure the data accordingly. *liquid* takes care of all this formatting under the hood; regardless of the scheme used, the `fec` object accepts a block of uncoded data bytes and encodes them into an output block of coded data bytes.

`fec_create(scheme,*opts)` creates a `fec` object of a specific scheme (see Table 4 for available codecs). Notice that the length of the input message does not need to be specified until `encode()` or `decode()` is invoked. The second argument is intended for future development and should be ignored by passing the `NULL` pointer (see example below).

`fec_recreate(q,scheme,opts)` recreates an existing `fec` object with a different scheme.

`fec_destroy(q)` destroys a `fec` object, freeing all internally-allocated memory arrays.

`fec_encode(q,n,*msg_dec,*msg_enc)` runs the error-correction encoder scheme on an n-byte input data array `msg_dec`, storing the result in the output array `msg_enc`. To obtain the length of the output array necessary, use the `fec_get_enc_msg_length()` method.

`fec_decode(q,n,*msg_enc,*msg_dec)` runs the error-correction decoder on an input array `msg_enc` of k encoded bytes. The resulting best-effort decoded message is written to the n-byte output array `msg_dec`, allocated by the user. Notice that like the `fec_encode()` method, the input length n refers to the *decoded* message length. Depending upon the error-correction capabilities of the scheme, the resulting data might have been corrupted, and therefore it is recommended to use either a checksum or a cyclic redundancy check (section 12.1) to validate data integrity.

`fec_get_enc_msg_length(scheme,n)` returns the length k of the encoded message in bytes for an uncoded input of n bytes using the specified encoding scheme. This method can be called before the `fec` object is created and is useful for allocating initial memory arrays.

Listed below is a simple example demonstrating the basic interface to the `fec` encoder/decoder object:

```c
// file: doc/listings/fec.example.c
#include <liquid/liquid.h>

int main() {
    unsigned int n = 64;                    // decoded message length (bytes)
    fec_scheme fs = LIQUID_FEC_HAMMING74;   // error-correcting scheme

    // compute encoded message length
    unsigned int k = fec_get_enc_msg_length(fs, n);

    // allocate memory for data arrays
    unsigned char msg_org[n];       // original message
    unsigned char msg_enc[k];       // encoded message
    unsigned char msg_rec[k];       // received message
    unsigned char msg_dec[n];       // decoded message

    // create fec objects
    fec encoder = fec_create(fs,NULL);
    fec decoder = fec_create(fs,NULL);

    // repeat as necessary
    {
        // ... initialize message ...

        // encode message
        fec_encode(encoder, n, msg_org, msg_enc);

        // ... push through channel ...

        // decode message
        fec_decode(decoder, n, msg_rec, msg_dec);
    }

    // clean up objects
    fec_destroy(encoder);
    fec_destroy(decoder);

    return 0;
}
```

For a more detailed example demonstrating the full capabilities of the `fec` object, see `examples/fec_example.c`. Table 4 lists the available codecs and gives a brief description for each. All convolutional and Reed-Solomon codes are available only if `libfec` is installed [23].

12.6 Performance

The performance of an error-correction scheme is typically measured in the bit error rate for a anipotally modulated siganl in the presence of additive white Gauss noise (AWGN). Certain applications prefer measuring performance in terms of the *signal* energy while others require *bit* energy,

Table 4: Forward error-correction codecs available in *liquid*

scheme	*asymptotic rate*	*description*
LIQUID_FEC_UNKNOWN	-	unknown/unsupported scheme
LIQUID_FEC_NONE	1	no error-correction
LIQUID_FEC_REP3	1/3	simple repeat code
LIQUID_FEC_REP5	1/5	simple repeat code
LIQUID_FEC_HAMMING74	4/7	Hamming (7,4) block code
LIQUID_FEC_HAMMING84	1/2	Hamming (7,4) with extra parity bit
LIQUID_FEC_HAMMING128	2/3	Hamming (12,8) block code
LIQUID_FEC_CONV_V27	1/2	$K = 7$, $d_{free} = 10$
LIQUID_FEC_CONV_V29	1/2	$K = 9$, $d_{free} = 12$
LIQUID_FEC_CONV_V39	1/3	$K = 9$, $d_{free} = 18$
LIQUID_FEC_CONV_V615	1/6	$K = 15$, $d_{free} <= 57$ (Heller 1968)
LIQUID_FEC_CONV_V27P23	2/3	$K = 7$, $d_{free} = 6$
LIQUID_FEC_CONV_V27P34	3/4	$K = 7$, $d_{free} = 5$
LIQUID_FEC_CONV_V27P45	4/5	$K = 7$, $d_{free} = 4$
LIQUID_FEC_CONV_V27P56	5/6	$K = 7$, $d_{free} = 4$
LIQUID_FEC_CONV_V27P67	6/7	$K = 7$, $d_{free} = 3$
LIQUID_FEC_CONV_V27P78	7/8	$K = 7$, $d_{free} = 3$
LIQUID_FEC_CONV_V29P23	2/3	$K = 9$, $d_{free} = 7$
LIQUID_FEC_CONV_V29P34	3/4	$K = 9$, $d_{free} = 6$
LIQUID_FEC_CONV_V29P45	4/5	$K = 9$, $d_{free} = 5$
LIQUID_FEC_CONV_V29P56	5/6	$K = 9$, $d_{free} = 5$
LIQUID_FEC_CONV_V29P67	6/7	$K = 9$, $d_{free} = 4$
LIQUID_FEC_CONV_V29P78	7/8	$K = 9$, $d_{free} = 4$
LIQUID_FEC_RS_M8	223/255	Reed-Solomon block code, $m = 8$

all relative to the noise variance. The two are related by

$$\frac{E_b}{N_0} = \frac{E_s}{rN_0} \tag{29}$$

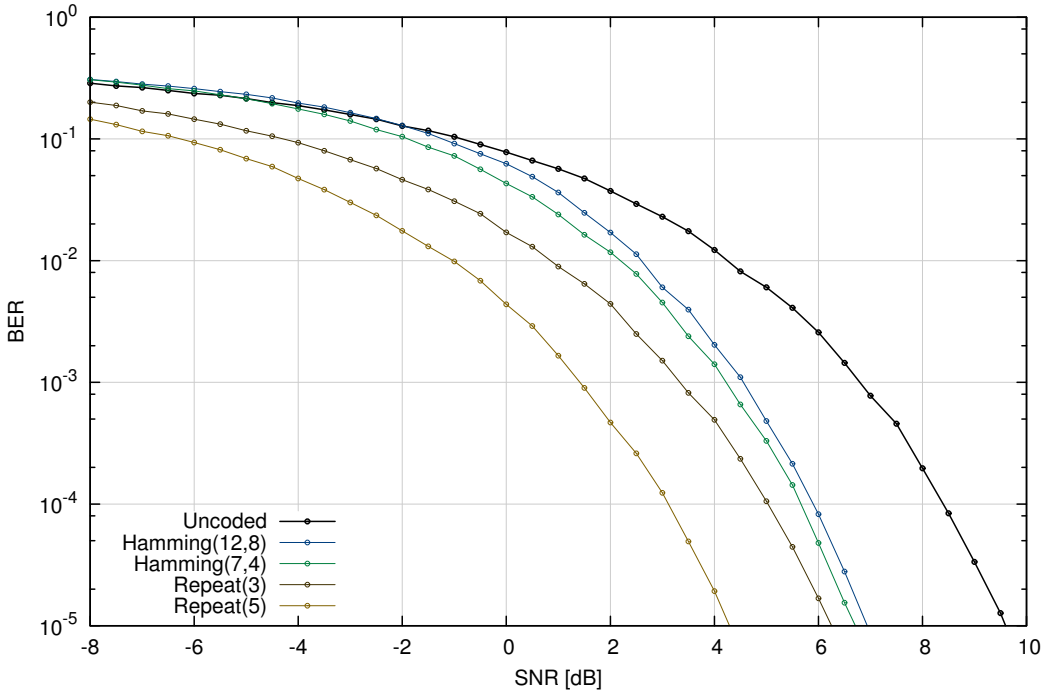
where E_s is the signal enery, E_b is the bit energy, N_0 is the noise energy, and r is the rate of the modulation and coding scheme pair, measured in bits/s/Hz. Figures 7, 8, and 9 demonstrate plot the bit error-rate performance of the forward error-correction schemes available in *liquid* for a BPSK signal in an AWGN channel. Each figure depicts the BER versus both E_s/N_0 (SNR) and E_b/N_0, the latter compensating for the coding rate. The error rates were computed by generating packets of 1024 bits, encoding using the appropriate FEC scheme, modulating the resulting bits with BPSK (see Section 18.2.3), adding noise, demodulating, and decoding. Each point was simulated with a minimum of 80,000 trials and a minimum of 500 bit errors. The raw data can be found in the `doc/data/ber/` subdirectory.

Figure 7 depicts the performance of the available built-in *liquid* FEC codecs, including `LIQUID_FEC_HAMMING74`, `LIQUID_FEC_HAMMING128`, `LIQUID_FEC_REP3`, and `LIQUID_FEC_REP5`. As stated previously, the repeat codes are typically inferior to any other encoding scheme. Notice that in terms of E_b/N_0 none of these schemes performs very well; only the Hamming codes give a slight advantage over uncoded BPSK for high values of E_b/N_0, and the repeat codes actually perform worse.[6] Still, if the goal is not to conserve the energy per bit but improve data reliability through a noisy channel then these simple codecs prove useful.

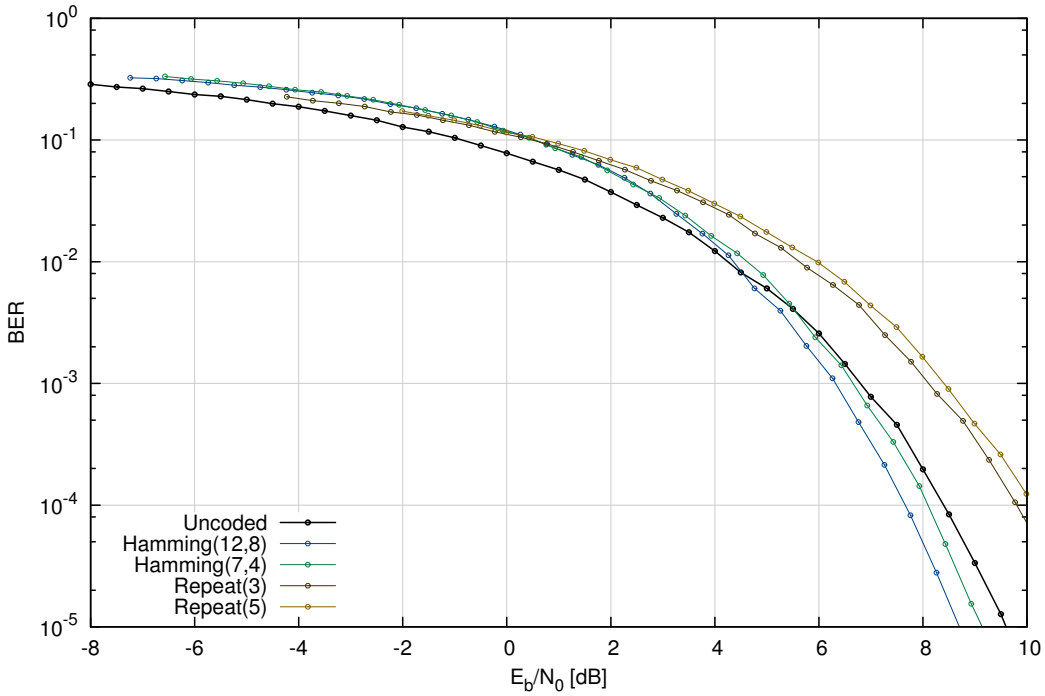
Figure 8 depicts the performance of the convolutional codecs available in *liquid* when the `libfec` library is installed. These include `LIQUID_FEC_CONV_V27`, `LIQUID_FEC_CONV_V29`, `LIQUID_FEC_CONV_V39`, and `LIQUID_FEC_CONV_V615`. Notice that these codecs provide significant error-correction capabilities over the Hamming codes; this is a result of the fact that convolutional encoding effectively spreads the redundancy over a broader range of the original message, correlating the output samples more than the short-length Hamming codes.

Figure 9 depicts the performance of the punctured convolutional codecs ($K = 7$) available in *liquid*, also available when the `libfec` library is installed. These include `LIQUID_FEC_CONV_V27P23`, `LIQUID_FEC_CONV_V27P34`, `LIQUID_FEC_CONV_V27P45`, `LIQUID_FEC_CONV_V27P56`, `LIQUID_FEC_CONV_V27P67`, and `LIQUID_FEC_CONV_V27P78`. Also included is the unpunctured `LIQUID_FEC_CONV_V27` codec, plotted as a reference point. *liquid* also includes punctured convolutional codes for the $K = 9$ encoder; however because the performance is similar to the $K = 7$ codec its performance is omitted for the sake of brevity.

[6] This is a known phenomenon as simply transmitting repetitions of the uncoded message will always perform worse than uncoded in terms of E_b/N_0. This is effectively equivalent to reducing the original transmit power by a factor of r and then transmitting the same message r times. With reference to conserving the amount of energy per uncoded bit, this type of "encoding" is always best when $r = 1$ (no retransmissions).

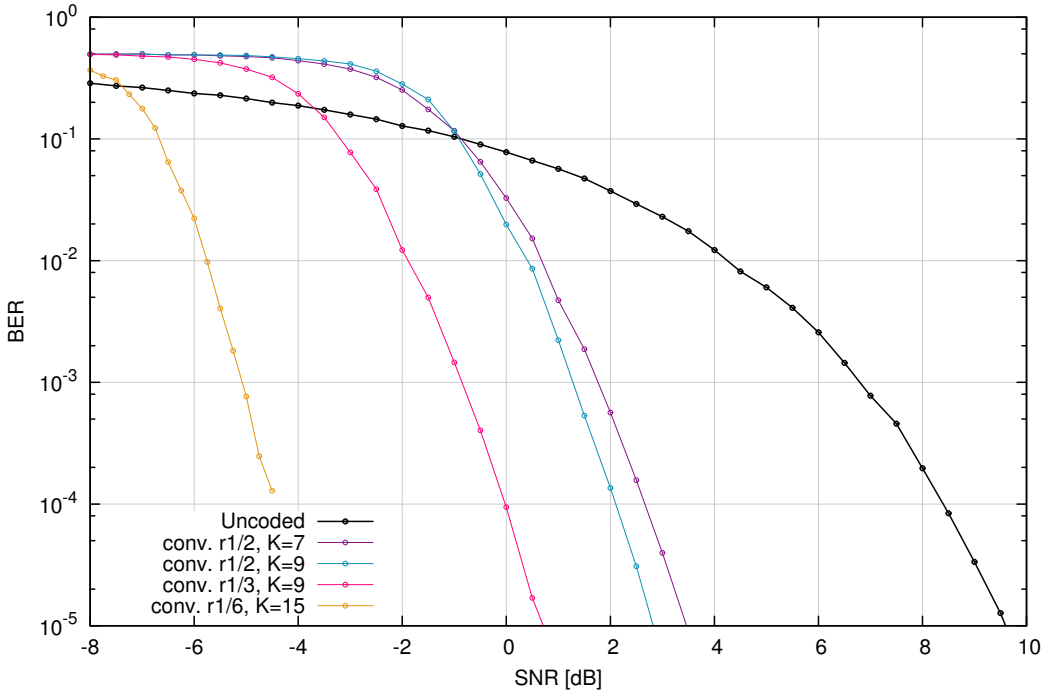


(a) BER vs. E_s/N_0

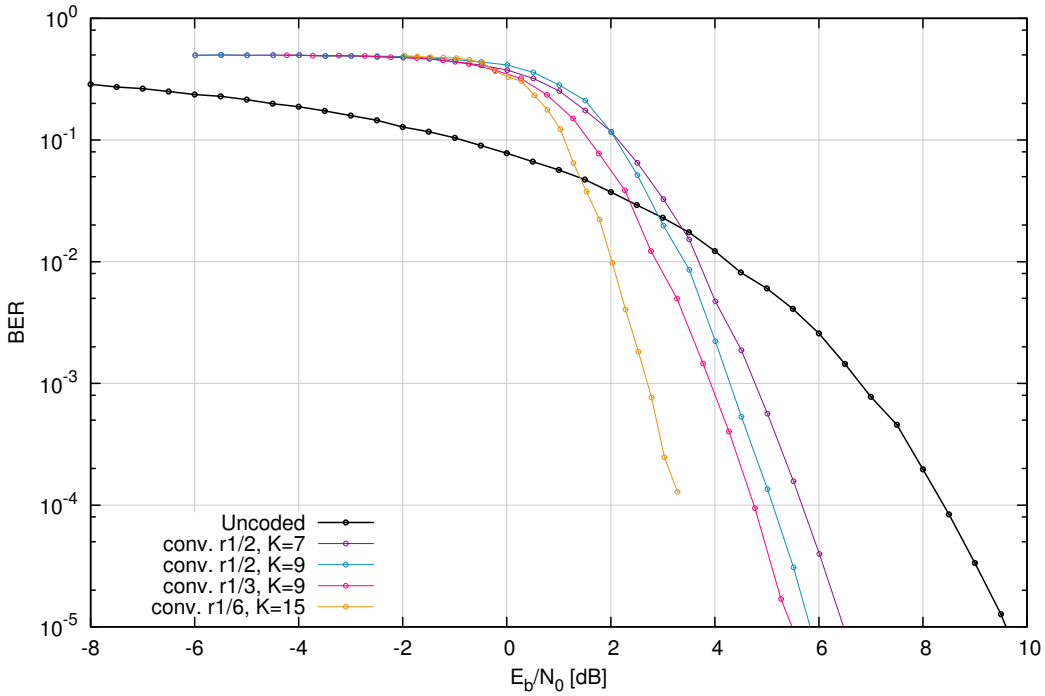


(b) BER vs. E_b/N_0

Figure 7: Forward error-correction codec bit error rates (simulated) for built-in *liquid* codecs using BPSK modulation.

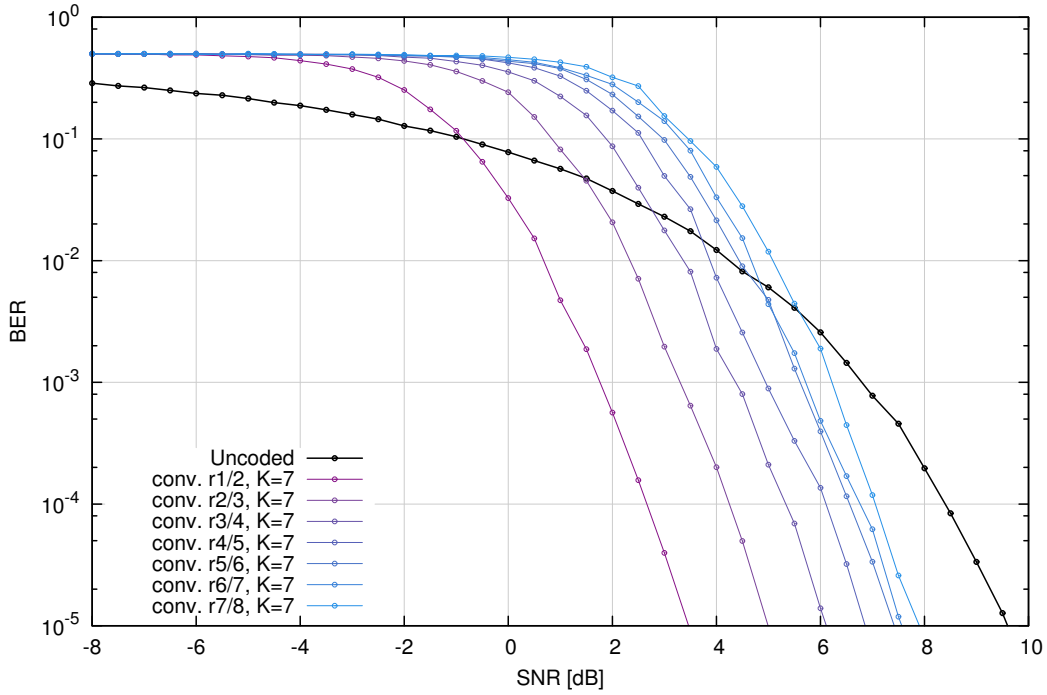


(a) BER vs. E_s/N_0

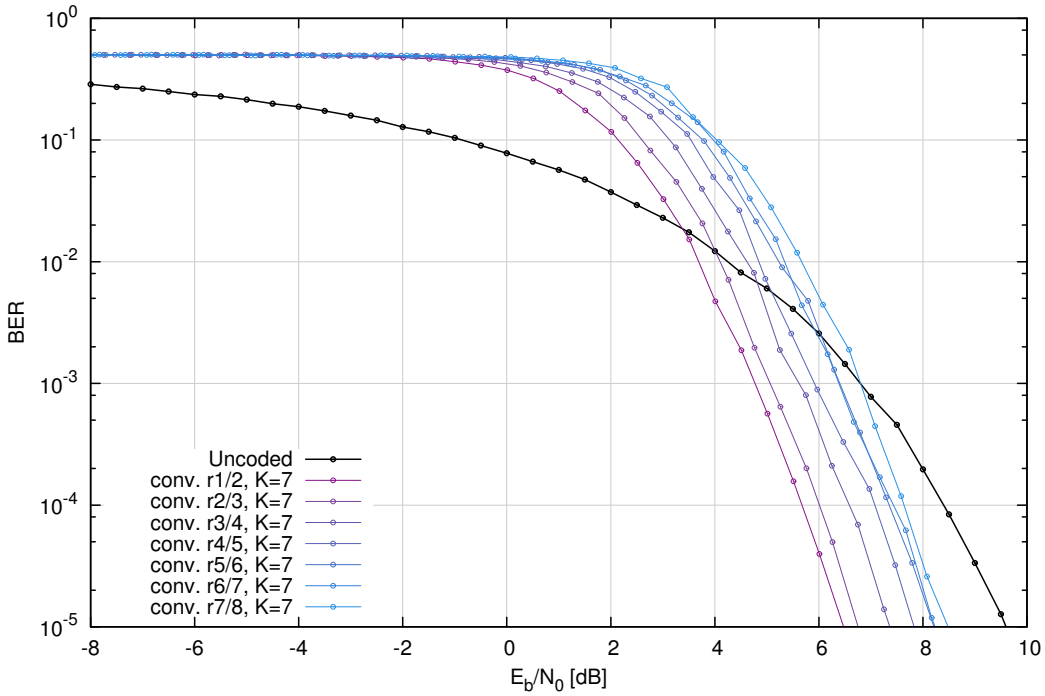


(b) BER vs. E_b/N_0

Figure 8: Forward error-correction codec bit error rates (simulated) for convolutional codes using BPSK modulation.



(a) BER vs. E_s/N_0



(b) BER vs. E_b/N_0

Figure 9: Forward error-correction codec bit error rates (simulated) for punctured convolutional codes using BPSK modulation.

13 fft (fast Fourier transform)

The fft module in *liquid* implements fast discrete Fourier transforms including forward and reverse DFTs as well as real even/odd transforms.

13.1 Complex Transforms

Given a vector of complex time-domain samples $\boldsymbol{x} = [x(0), x(1), \ldots, x(N-1)]^T$ the N-point forward discrete Fourier transform is computed as:

$$X(k) = \sum_{i=0}^{N-1} x(i) e^{-j2\pi ki/N} \tag{30}$$

Similarly, the inverse (reverse) discrete Fourier transform is:

$$x(n) = \sum_{i=0}^{N-1} X(i) e^{j2\pi ni/N} \tag{31}$$

liquid implements only the basic decimation-in-time FFT algorithm for radix-2 transforms and the slow DFT method otherwise. Internal methods requiring FFTs, however, will use the `fftw3` library [16] if available. The presence of `fftw3.h` and `libfftw3` are detected by the configure script at build time. If found, *liquid* will link against `fftw` for better performance (it is, however, the fastest FFT in the west, you know). If `fftw` is unavailable, however, *liquid* will use its own, slower FFT methods for internal processing. This eliminates `libfftw` as an external dependency, but takes advantage of it when available.

An example of the interface for computing complex discrete Fourier transforms is listed below. Notice the stark similarity to `libfftw3`'s interface.

```
1  // file: doc/listings/fft.example.c
2  #include <liquid/liquid.h>
3
4  int main() {
5      // options
6      unsigned int n=16;  // input data size
7      int flags=0;        // fft flags (typically ignored)
8
9      // allocated memory arrays
10     float complex * x = (float complex*) malloc(n * sizeof(float complex));
11     float complex * y = (float complex*) malloc(n * sizeof(float complex));
12
13     // create fft plan
14     fftplan q = fft_create_plan(n, x, y, FFT_FORWARD, flags);
15
16     // ... initialize input ...
17
18     // execute fft (repeat as necessary)
19     fft_execute(q);
20
21     // destroy fft plan and free memory arrays
```

```
22      fft_destroy_plan(q);
23      free(x);
24      free(y);
25  }
```

13.2 Real even/odd DFTs

liquid also implement real even/odd discrete Fourer transforms; however these are not guaranteed to be efficient. A list of the transforms and their descriptions is given below.

13.2.1 `FFT_REDFT00` (DCT-I)

$$X(k) = \frac{1}{2}\Big(x(0) + (-1)^k x(N-1)\Big) + \sum_{n=1}^{N-2} x(n) \cos\left(\frac{\pi}{N-1} nk\right) \tag{32}$$

13.2.2 `FFT_REDFT10` (DCT-II)

$$X(k) = \sum_{n=0}^{N-1} x(n) \cos\left[\frac{\pi}{N}\left(n+0.5\right)k\right] \tag{33}$$

13.2.3 `FFT_REDFT01` (DCT-III)

$$X(k) = \frac{x(0)}{2} + \sum_{n=1}^{N-1} x(n) \cos\left[\frac{\pi}{N} n\left(k+0.5\right)\right] \tag{34}$$

13.2.4 `FFT_REDFT11` (DCT-IV)

$$X(k) = \sum_{n=0}^{N-1} x(n) \cos\left[\frac{\pi}{N}\left(n+0.5\right)\left(k+0.5\right)\right] \tag{35}$$

13.2.5 `FFT_RODFT00` (DST-I)

$$X(k) = \sum_{n=0}^{N-1} x(n) \sin\left[\frac{\pi}{N+1}(n+1)(k+1)\right] \tag{36}$$

13.2.6 `FFT_RODFT10` (DST-II)

$$X(k) = \sum_{n=0}^{N-1} x(n) \sin\left[\frac{\pi}{N}(n+0.5)(k+1)\right] \tag{37}$$

13.2.7 `FFT_RODFT01` (DST-III)

$$X(k) = \frac{(-1)^k}{2} x(N-1) + \sum_{n=0}^{N-2} x(n) \sin\left[\frac{\pi}{N}(n+1)(k+0.5)\right] \tag{38}$$

13.2.8 `FFT_RODFT11` (DST-IV)

$$X(k) = \sum_{n=0}^{N-1} x(n) \sin\left[\frac{\pi}{N}(n+0.5)(k+0.5)\right] \tag{39}$$

14 filter

The filter module is at the core of *liquid*'s digital signal processing functionality. Filter design and implementation is a significant portion of radio engineering, and consumes a considerable portion of the baseband receiver's energy. This section includes interface descriptions for all of the signal processing elements in *liquid* regarding filter design and implemenation. This includes both infinite and finite (recursive and non-recursive) filters, decimators, interpolators, and performance characterization.

14.1 `autocorr` (auto-correlator)

The `autocorr` family of objects implement autocorrelation of signals. The discrete autocorrelation of a signal $\boldsymbol{x}$ is a delay, conjugate multiply, and accumulate operation defined as

$$r_{xx}(n) = \sum_{k=0}^{N-1} x(n-k)x^*(n-k-d) \tag{40}$$

where N is the window length, and d is the overlap delay. An example of the `autocorr` interface is listed below.

```
1  // file: doc/listings/autocorr.example.c
2  #include <liquid/liquid.h>
3
4  int main() {
5      // options
6      unsigned int n = 60;        // autocorr window length
7      unsigned int delay = 20;    // autocorr overlap delay
8
9      // create autocorrelator object
10     autocorr_cccf q = autocorr_cccf_create(n,delay);
11
12     float complex x;            // input sample
13     float complex rxx;          // output auto-correlation
14
15     // compute auto-correlation (repeat as necessary)
16     {
17         autocorr_cccf_push(q, x);
18         autocorr_cccf_execute(q, &rxx);
19     }
20
21     // destroy autocorrelator object
22     autocorr_cccf_destroy(q);
23 }
```

A more detailed example is given in `examples/autocorr_cccf_example.c` in the main *liquid* project directory. Listed below is the full interface to the `autocorr` family of objects. While each method is listed for `autocorr_cccf`, the same functionality applies to `autocorr_rrrf`.

`autocorr_cccf_create(N,d)` creates and returns an `autocorr` object with a window size of N samples and a delay of d samples.

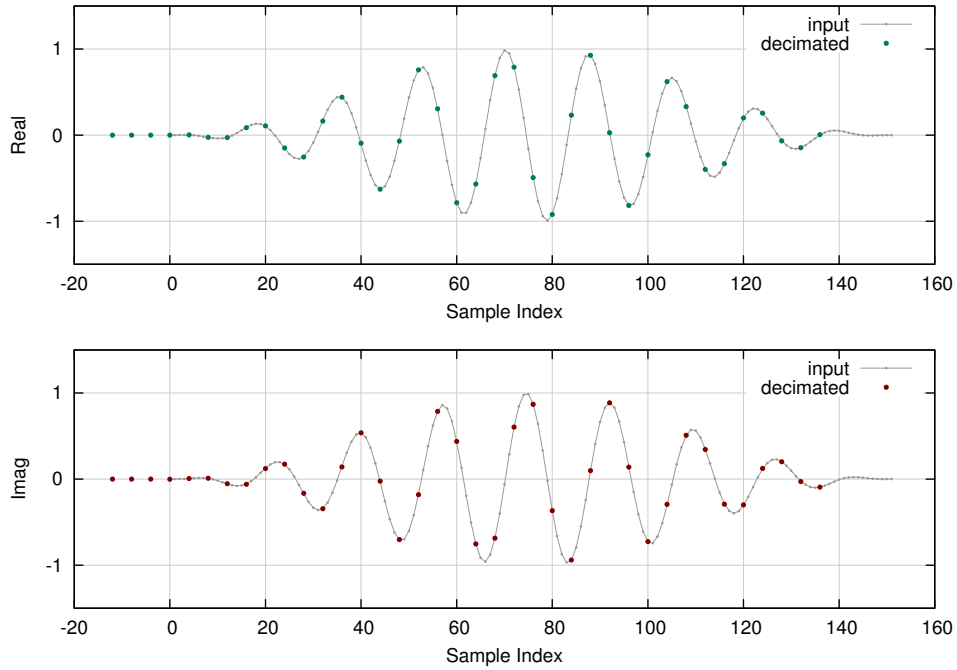


Figure 10: `decim_crcf` (decimator) example with $D = 4$, compensating for filter delay.

`autocorr_cccf_destroy(q)` destroys an `autocorr` object, freeing all internally-allocated memory.

`autocorr_cccf_clear(q)` clears the internal `autocorr` buffers.

`autocorr_cccf_print(q)` prints the internal state of the `autocorr` object.

`autocorr_cccf_push(q,x)` pushes a sample x into the internal buffer of an `autocorr` object.

`autocorr_cccf_execute(q,*rxx)` executes the delay, conjugate multiply, and accumulate operation, storing the result in the output variable r_{xx}.

`autocorr_cccf_get_energy(q)` returns $(1/N) \sum_{k=0}^{N-1} |x(n-k)x^*(n-k-d)|$

14.2 `decim` (decimator)

The `decim` object family implements a basic interpolator with an integer output-to-input resampling ratio D. It is essentially just a `firfilt` object which operates on a block of samples at a time. An example of the decimator can be seen in Figure 10. Listed below is the full interface to the `decim` family of objects. While each method is listed for `decim_crcf`, the same functionality applies to `decim_rrrf` and `decim_cccf`.

`decim_crcf_create(D,*h,N)` creates a `decim` object with a decimation factor D using N filter coefficients $\boldsymbol{h}$.

`decim_crcf_destroy(q)` destroys a `decim` object, freeing all internally-allocated memory.

`decim_crcf_print(q)` prints the parameters of a `decim` object to the standard output.

`decim_crcf_clear(q)` clears the internal buffer of a `decim` object.

`decim_crcf_execute(q,*x,*y,k)` computes the output decimation of the input sequence $\boldsymbol{x}$ (which is D samples in size) at the index k and stores the result in y.

An example of the `decim` interface is listed below.

```
// file: doc/listings/decim.example.c
#include <liquid/liquid.h>

int main() {
    // options
    unsigned int D = 4;         // decimation factor
    unsigned int h_len = 21;    // filter length

    // design filter and create decimator object
    float h[h_len];             // filter coefficients
    decim_crcf q = decim_crcf_create(D,h,h_len);

    // generate input signal and decimate
    float complex x[D];         // input samples
    float complex y;            // output sample

    // run decimator (repeat as necessary)
    {
        decim_crcf_execute(q, x, &y, 0);
    }

    // destroy decimator object
    decim_crcf_destroy(q);
}
```

A more detailed example is given in `examples/decim_crcf_example.c` in the main *liquid* project directory.

14.3 `firfarrow` (finite impulse response Farrow filter)

liquid implements non-recursive Farrow filters using the `firfarrow` family of objects. The Farrow structure is convenient for varying the group delay of a filter. The filter coefficients themselves are not stored explicitly, but are represented as a set of polynomials each with an order Q. The coefficients can be computed dynamically from the polynomial by arbitrarily specifying the fractional sample delay μ. Listed below is the full interface to the `firfarrow` family of objects. While each method is listed for `firfarrow_crcf`, the same functionality applies to `firfarrow_rrrf`.

`firfarrow_crcf_create(N,Q,fc,As)` creates a `firfarrow` object with N coefficients using a polynomial of order Q with a cutoff frequency f_c and as stop-band attenuation of A_s dB.

`firfarrow_crcf_destroy(q)` destroy object, freeing all internally-allocated memory.

`firfarrow_crcf_clear(q)` clear filter internal memory buffer. This does not reset the delay.

`firfarrow_crcf_print(q)` prints the filter's internal state to `stdout`.

`firfarrow_crcf_push(q,x)` push a single sample x into the filter's internal buffer.

`firfarrow_crcf_set_delay(q,mu)` set fractional delay μ of filter.

`firfarrow_crcf_execute(q,*y)` computes the output sample, storing the result in y.

`firfarrow_crcf_get_length(q)` returns length of the filter (number of taps)

`firfarrow_crcf_get_coefficients(q,*h)` returns the internal filter coefficients, storing the result in the output vector $\boldsymbol{h}$.

`firfarrow_crcf_freqresponse(q,fc,*H)` computes the complex response H of the filter at the normalized frequency f_c.

`firfarrow_crcf_groupdelay(q,fc)` returns the group delay of the filter at the normalized frequency f_c.

Listed below is an example of the `firfarrow` object's interface.

```
// file: doc/listings/firfarrow_crcf.example.c
#include <liquid/liquid.h>

int main()
{
    // options
    unsigned int h_len=19;  // filter length
    unsigned int Q=5;       // polynomial order
    float fc=0.45f;         // filter cutoff
    float As=60.0f;         // stop-band attenuation [dB]

    // generate filter object
    firfarrow_crcf q = firfarrow_crcf_create(h_len, Q, fc, As);

    // set fractional sample delay
    firfarrow_crcf_setdelay(q, 0.3f);

    float complex x;    // input sample
    float complex y;    // output sample

    // execute filter (repeat as necessary)
    {
        firfarrow_crcf_push(q, x);      // push input sample
        firfarrow_crcf_execute(q,&y);   // compute output
    }

    // destroy object
    firfarrow_crcf_destroy(q);
}
```

An example of the Farrow filter's group delay can be found in Figure 11.

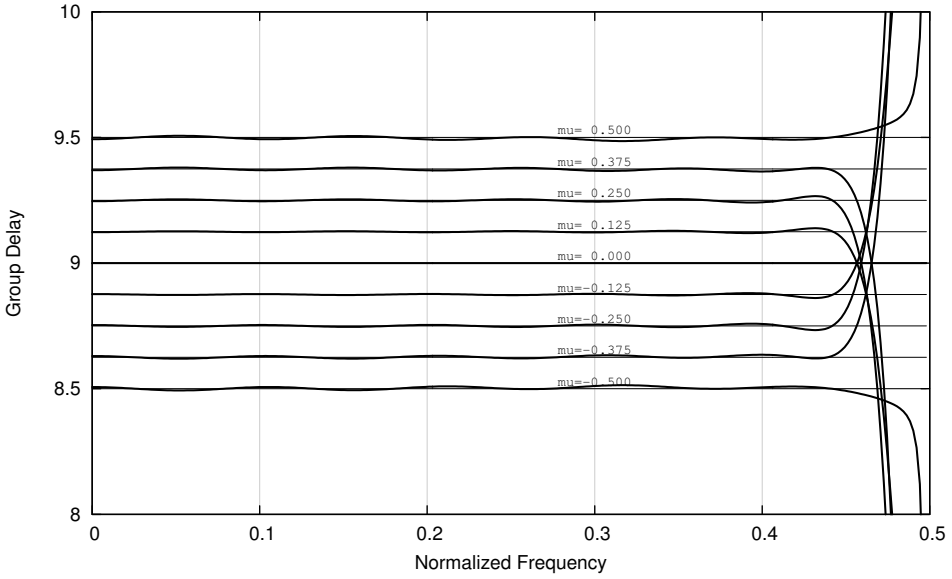


Figure 11: `firfarrow_crcf` (Farrow filter) group delay example with $N = 19$, $Q = 5$, $f_c = 0.45$, and $A_s = 60$ dB.

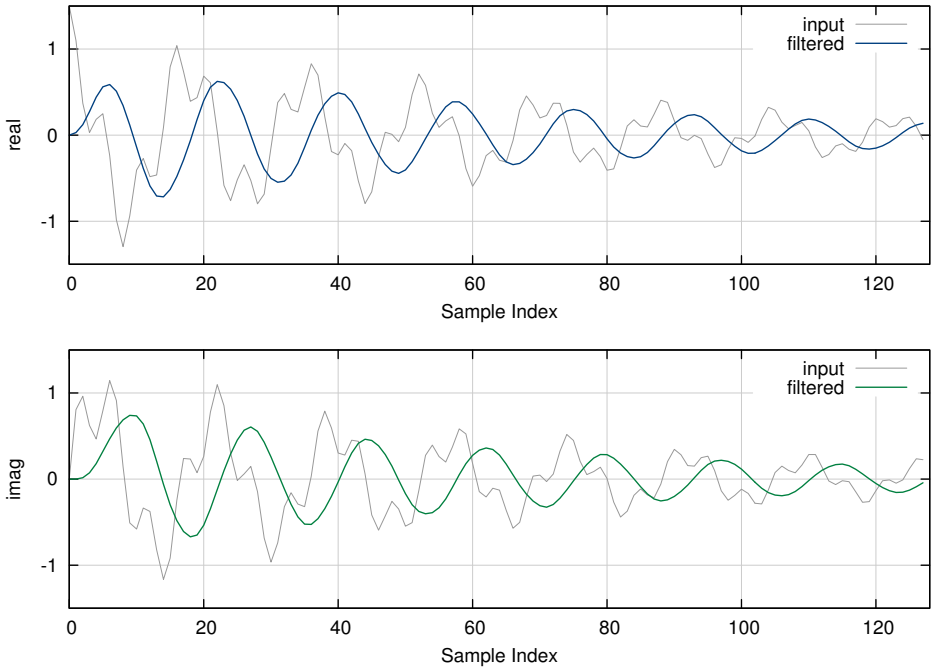


Figure 12: `firfilt_crcf` (finite impulse response filter) demonstration

14.4 `firfilt` (finite impulse response filter)

Finite impulse response (FIR) filters are implemented in *liquid* with the `firfilt` family of objects. FIR filters (also known as *non-recursive filters*) operate on discrete-time samples, computing the output y as the convolution of the input $\boldsymbol{x}$ with the filter coefficients $\boldsymbol{h}$ as

$$y(n) = \sum_{k=0}^{N-1} h(k)x(N-k-1) \tag{41}$$

where $\boldsymbol{h} = [h(0), h(1), \ldots, h(N-1)]$ is the filter impulse response. Notice that the output sample in (41) is simply the vector dot product (see Section 10) of the filter coefficients $\boldsymbol{h}$ with the time-reversed sequence $\boldsymbol{x}$. An example of the `firfilt` can be seen in Figure 12 in which a low-pass filter is applied to a signal to remove a high-frequency component. An example of the `firfilt` interface is listed below.

```
1  // file: doc/listings/firfilt.example.c
2  #include <liquid/liquid.h>
3
4  int main() {
5      // options
6      unsigned int h_len=21;  // filter order
7      float h[h_len];         // filter coefficients
8
9      // ... initialize filter coefficients ...
```

```
10
11      // create filter object
12      firfilt_crcf q = firfilt_crcf_create(h,h_len);
13
14      float complex x;    // input sample
15      float complex y;    // output sample
16
17      // execute filter (repeat as necessary)
18      {
19          firfilt_crcf_push(q, x);    // push input sample
20          firfilt_crcf_execute(q,&y); // compute output
21      }
22
23      // destroy filter object
24      firfilt_crcf_destroy(q);
25  }
```

Listed below is the full interface to the `firfilt` family of objects. While each method is listed for `firfilt_crcf`, the same functionality applies to `firfilt_rrrf` and `firfilt_cccf`.

`firfilt_crcf_create(*h,N)` creates a `firfilt` object with N filter coefficients $\boldsymbol{h}$.

`firfilt_crcf_destroy(q)` destroys a `firfilt` object, freeing all internally-allocated memory.

`firfilt_crcf_print(q)` prints the parameters of a `firfilt` object to the standard output.

`firfilt_crcf_clear(q)` clears the internal buffer of a `firfilt` object.

`firfilt_crcf_push(q,x)` pushes an input sample x into the internal buffer of the filter object.

`firfilt_crcf_execute(q,*y)` generates the output sample y by computing the vector dot product (see Section 10) between the internal filter coefficients and the internal buffer.

14.5 `firdes` (finite impulse response filter design)

This section describes the finite impulse response filter design capabilities in *liquid*. This includes basic low-pass filter design using the windowed-sinc method, square-root Nyquist filters, arbitrary design using the Parks-McClellan algorithm, and some useful miscellaneous functions.

14.5.1 Window prototype

The ideal low-pass filter has a rectangular response in the frequency domain and an infinite $\sin(t)/t$ response in the time domain. Because all time-dependent filters must be causal, this type of filter is unrealizable; furthermore, truncating its response results in poor pass-band ripple stop-band rejection. An improvement over truncation is offered by use of a band-limiting window. Let the finite impulse response of a filter be defined as

$$h(n) = h_i(n)w(n) \tag{42}$$

where $w(n)$ is a time-limited symmetric window and $h_i(n)$ is the impulse response of the ideal filter with a cutoff frequency ω_c, viz.

$$h_i(n) = \frac{\omega_c}{\pi}\left(\frac{\sin \omega_c n}{\omega_c n}\right), \quad \forall n \tag{43}$$

A number of possible windows could be used; the Kaiser window is particularly common due to its systematic ability to trade transition bandwidth for stop-band rejection. The Kaiser window is defined as

$$w(n) = \frac{I_0\left[\pi\alpha\sqrt{1-\left(\frac{n}{N/2}\right)^2}\right]}{I_0\left(\pi\alpha\right)} \quad -N/2 \le n \le N/2,\ \alpha \ge 0 \tag{44}$$

where $I_\nu(z)$ is the modified Bessel function of the first kind of order ν and α is a shape parameter controlling the window decay. $I_\nu(z)$ can be expanded as

$$I_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{k=0}^{\infty} \frac{\left(\frac{1}{4}z^2\right)^k}{k!\Gamma(k+\nu+1)} \tag{45}$$

The sum in (45) converges quickly due to the denominator increasing rapidly, (and in particular for $\nu = 0$ the denominator reduces to $(k!)^2$) and thus only a few terms are necessary for sufficient approximation. The sum (45) converges quickly due to the denominator increasing rapidly, thus only a few terms are necessary for sufficient approximation. For more approximations to $I_0(z)$ and $I_\nu(z)$, see Section 16 in the math module. Kaiser gives an approximation for the value of α to give a particular sidelobe level for the window as [30, (3.2.7)]

$$\alpha = \begin{cases} 0.1102(A_s - 8.7) & A_s > 50 \\ 0.5842(A_s - 21)^{0.4} & 21 < A_s \le 50 \\ 0 & \text{else} \end{cases} \tag{46}$$

where $A_s > 0$ is the stop-band attenuation in decibels. This approximation is provided in *liquid* by the `kaiser_beta_As()` method, and the length of the filter can be approximated with `estimate_req_filter_len()` (see section 14.5.4 for more detail on these methods).

The entire design process is provided in *liquid* with the `firdes_kaiser_window()` method which can be invoked as follows:

```
firdes_kaiser_window(_n, _fc, _As, _mu, *_h)
```

where `_n` is the length of the filter (number of samples), `_fc` is the normalized cutoff frequency ($0 \le f_c \le 0.5$), `_As` is the stop-band attenuation in dB ($A_s > 0$), `_mu` is the fractional sample offset ($-0.5 \le \mu \le 0.5$), and `*_h` is the n-sample output coefficient array. Listed below is an example of the `firdes_kaiser_window` interface.

```
1  // file: doc/listings/firdes_kaiser.example.c
2  #include <liquid/liquid.h>
3
4  int main() {
5      // options
```

Table 5: Square-root Nyquist filter prototypes available in *liquid*

scheme	*description*
`LIQUID_RNYQUIST_ARKAISER`	approximate r-Kaiser
`LIQUID_RNYQUIST_RKAISER`	r-Kaiser
`LIQUID_RNYQUIST_RRC`	square-root raised cosine
`LIQUID_RNYQUIST_hM3`	harris-Moerder type 3 [9]

```
6       float fc=0.15f;         // filter cutoff frequency
7       float ft=0.05f;         // filter transition
8       float As=60.0f;         // stop-band attenuation [dB]
9       float mu=0.0f;          // fractional timing offset
10
11      // estimate required filter length and generate filter
12      unsigned int h_len = estimate_req_filter_len(ft,As);
13      float h[h_len];
14      firdes_kaiser_window(h_len,fc,As,mu,h);
15  }
```

An example of a low-pass filter design using the Kaiser window can be found in Figure 13.

14.5.2 `rnyquist` (square-root Nyquist filter design)

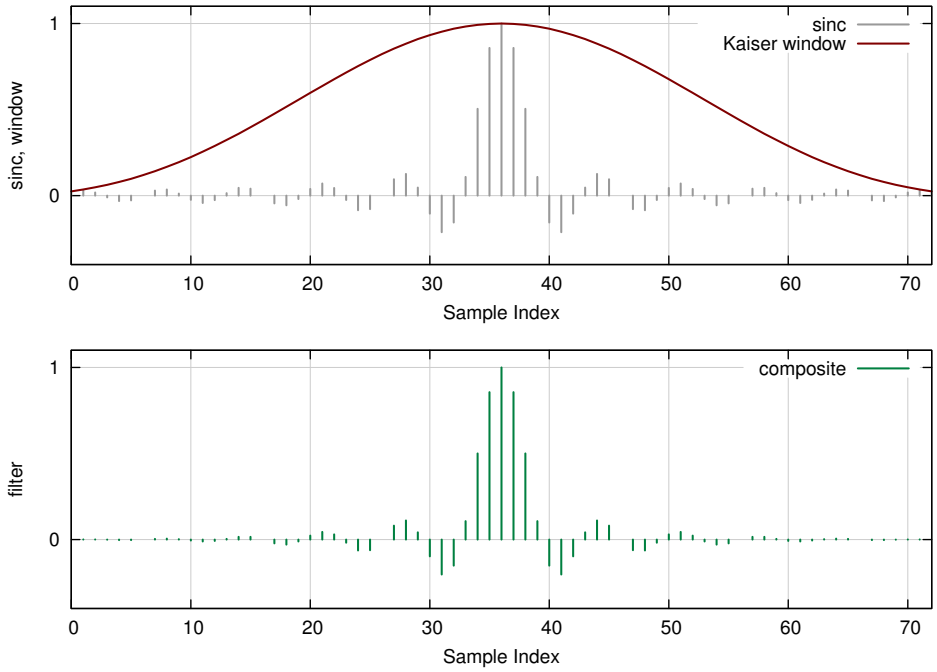
Nyquist's criteria for designing a band-limited filter without inter-symbol interference is for the spectral response of a linear phase filter to be symmetric about its symbol rate. Square-root Nyquist filters are commonly used in digital communcations systems with linear modulation as a pulse shape for matched filtering. Applying a pulse shape to the transmitted symbol sequence limits its occupied spectral bandwidth by smoothing the transitions between symbols. If an identical filter is applied at the receiver then the system is matched resulting in the maximum signal-to-noise ratio and (theoretically) zero inter-symbol interference. While the design of Nyquist filters is trivial and can be accomplished by applying any desired window to a sinc function, designing a square-root Nyquist filter is not as straighforward. *liquid* conveniently provides several square-root Nyquist filter prototypes listed in Table 5. The interface for designing square-root Nyquist filters is simply

```
design_rnyquist_filter(_ftype, _k, _m, _beta, _dt, *h);
```

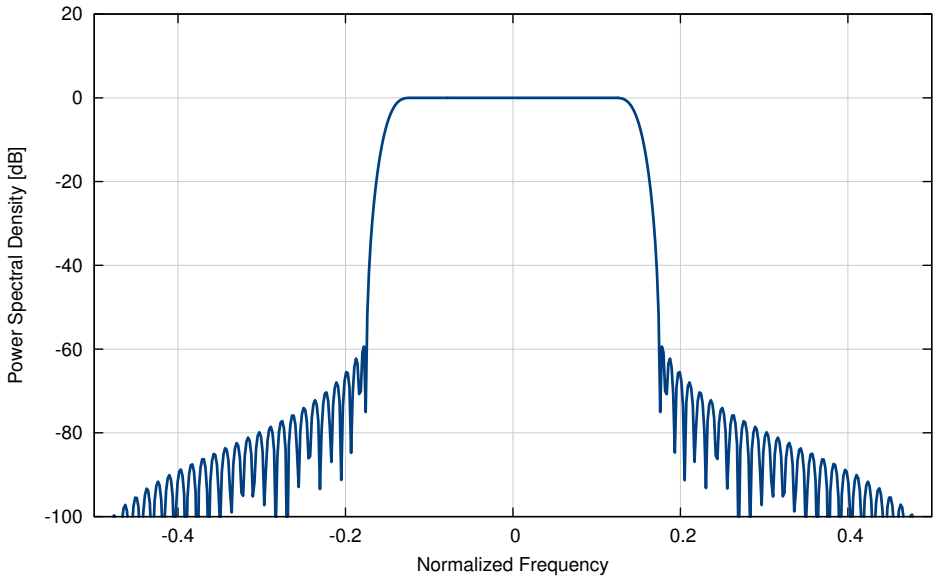
where `_ftype` is one of the filter types in Table 5, k is the number of samples per symbol, m is the filter delay in symbols, β is the excess bandwidth (rolloff) factor, Δt is the fractional sample delay (usually set to zero for typical filter designs), and h is the output coefficients array of length $2km+1$. All square-root Nyquist filters in *liquid* have these four basic properties (k, m, β, Δt) and produce a filter with $N = 2km + 1$ coefficients.

The most common square-root Nyquist filter design in digital communications is the square-root raised-cosine (RRC) filter, likely due to the fact that an expression for its time series can be expressed in closed form. The filter coefficients themselves are derived from the following equation:

$$h\left[z\right] = 4\beta \frac{\cos\left[(1+\beta)\pi z\right] + \sin\left[(1-\beta)\pi z\right]/(4\beta z)}{\pi\sqrt{T}\left[1-16\beta^2 z^2\right]} \tag{47}$$



(a) time



(b) PSD

Figure 13: `firdes_kaiser_window()` demonstration, $f_c = 0.15$, $\Delta f = 0.05$, $A_s = 60$dB

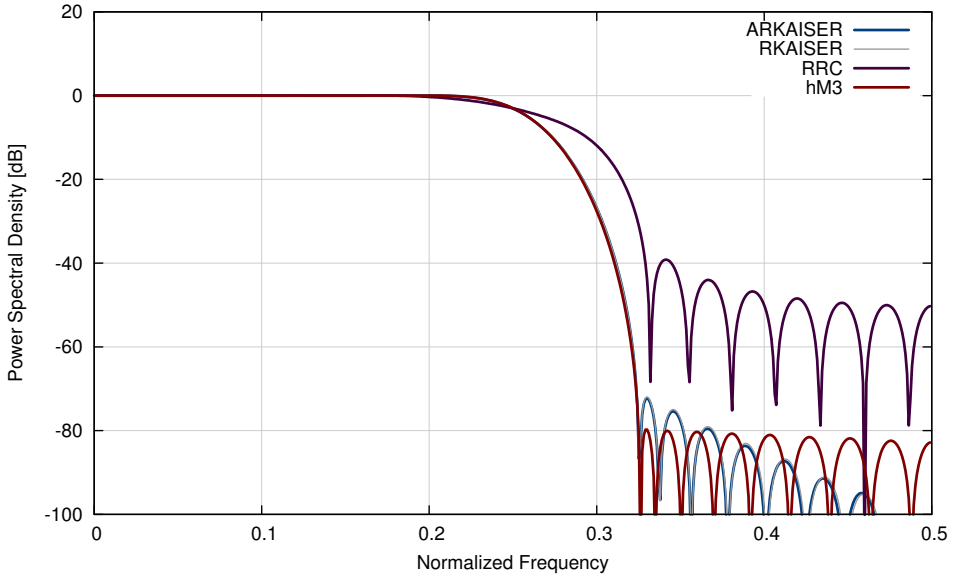


Figure 14: Contrast of the different square-root Nyquist filters available in *liquid* for $k = 2$, $m = 9$, $\beta = 0.3$, and $\Delta t = 0$.

where $z = n/k - m$, and $T = 1$ for most cases. *liquid* compensates for the two cases where $h[n]$ might be undefined in the above equation, i.e.

$$\lim_{z \to 0} h(z) = 1 - \beta + 4\beta/\pi \tag{48}$$

and

$$\lim_{z \to \pm\frac{1}{4\beta}} h(z) = \frac{\beta}{\sqrt{2}} \left[\left(1 + \frac{2}{\pi}\right) \sin\left(\frac{\pi}{4\beta}\right) + \left(1 - \frac{2}{\pi}\right) \cos\left(\frac{\pi}{4\beta}\right) \right] \tag{49}$$

The r-Kaiser and harris-Moerder-3 (hM3) filters cannot be expressed in closed form but rely on iterations over traditional filter design techniques to search for the design parameters which minimize the resulting filter's inter-symbol interference (ISI). Similarly the approximate r-Kaiser filter uses an approximation for the design parameters to eliminate the need for running the search; this comes at the expense of a slight performance degradation.

Figure 14 contrasts the different square-root Nyquist filters available in *liquid*. The square-root raised-cosine filter is inferior to the (approximate) r-Kaiser and harris-Moerder-3 filters in both transition bandwidth as well as side-lobe suppression. In the figure the responses of the r-Kaiser and approximate r-Kaiser filters are indistinguishable.

14.5.3 `firdespm` (Parks-McClellan algorithm)

FIR filter desing using the Parks-McClellan algorithm is implemented in *liquid* with the `firdespm` interface. The Parks-McClellan algorithm uses the Remez exchange algorithm to solve the minimax

problem (minimize the maximum error) for filter design. The interface accepts a description of N_b disjoint and non-overlapping frequency bands with a desired response and relative error weighting for each, and computes the resulting filter coefficients.

```
firdespm_run(_h_len,        // filter length
             _bands*,       // array of frequency bands
             _des*,         // desired response in each band
             _weights*,     // relative weighting for each band
             _num_bands,    // number of bands
             _btype,        // filter type
             _wtype*,       // weighting function for each band
             _h*)
```

`_bands` is a $[N_b \times 2]$ matrix of the band edge descriptions. Each row corresponds to an upper and lower band edge for each region of interest. These regions cannot be overlapping.

`_des` is an array of size N_b with the desired response (linear) for each band.

`_weights` is an array of size N_b with the relative error weighting for each band.

`_num_bands` represents N_b, the number of bands in the design.

`_btype` gives the filter type for the design. This is typically `LIQUID_FIRDESPM_BANDPASS` for the majority of filters.

`_wtype` is an array of length N_b which specifies the weighting function for each band (flat, exponential, or linear).

Listed below is an example of the `firdespm` interface.

```
1  // file: doc/listings/firdespm.example.c
2  #include <liquid/liquid.h>
3
4  int main() {
5      // define filter length, type, number of bands
6      unsigned int n=55;
7      liquid_firdespm_btype btype = LIQUID_FIRDESPM_BANDPASS;
8      unsigned int num_bands = 4;
9
10     // band edge description [size: num_bands x 2]
11     float bands[8]   = {0.0f,   0.1f,   // 1    first pass-band
12                         0.15f,  0.3f,   // 0    first stop-band
13                         0.33f,  0.4f,   // 0.1  second pass-band
14                         0.42f,  0.5f};  // 0    second stop-band
15
16     // desired response [size: num_bands x 1]
17     float des[4]     = {1.0f, 0.0f, 0.1f, 0.0f};
18
19     // relative weights [size: num_bands x 1]
20     float weights[4] = {1.0f, 1.0f, 1.0f, 1.0f};
21
```

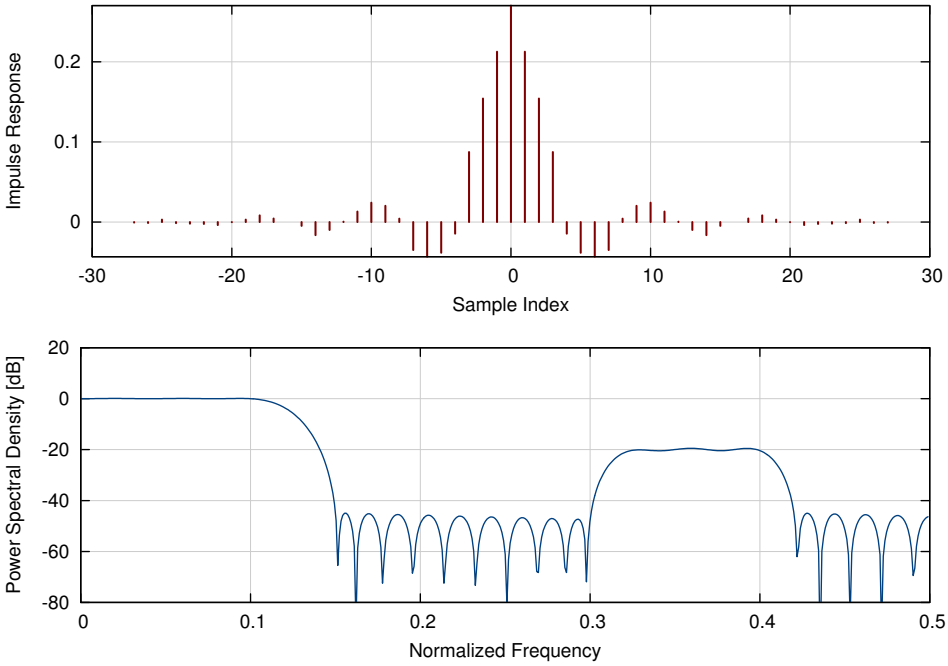


Figure 15: `firdespm` multi-passband filter design demonstration

```
22      // in-band weighting functions [size: num_bands x 1]
23      liquid_firdespm_wtype wtype[4] = {LIQUID_FIRDESPM_FLATWEIGHT,
24                                        LIQUID_FIRDESPM_EXPWEIGHT,
25                                        LIQUID_FIRDESPM_EXPWEIGHT,
26                                        LIQUID_FIRDESPM_EXPWEIGHT};
27
28      // allocate memory for array and design filter
29      float h[n];
30      firdespm_run(n,num_bands,bands,des,weights,wtype,btype,h);
31  }
```

14.5.4 Miscellaneous functions

Here are several miscellaneous functions used in *liquid*'s filter module, useful to filtering and filter design.

`estimate_req_filter_len(df,As)` returns an estimate of the required filter length, given a transition bandwidth Δf and stopband attenuation A_s. The estimate uses Kaiser's formula [30]

$$N \approx \frac{A_s - 7.95}{14.26 \Delta f} \tag{50}$$

`estimate_req_filter_As(df,N)` returns an estimate of the filter's stop-band attenuation A_s given the filter's length N and transition bandwidth Δf. The estimate uses an iterative binary search to find A_s from `extimate_req_filter_As()`.

`estimate_req_filter_df(As,N)` returns an estimate of the filter's transition bandwidth Δf given the filter's length N and stop-band attenuation A_s. The estimate uses an iterative binary search to find Δf from `extimate_req_filter_As()`.

`kaiser_beta_As(As)` returns an estimate of the Kaiser β factor for a particular stop-band attenuation A_s. The estimate uses Kaiser's original formula [30], viz

$$\beta = \begin{cases} 0.1102(A_s - 8.7) & A_s > 50 \\ 0.5842(A_s - 21)^{0.4} & 21 < A_s \leq 50 \\ 0 & \text{else} \end{cases} \tag{51}$$

`fir_group_delay(*h,n,f)` computes the group delay for a finite impulse-response filter with n coefficients $\boldsymbol{h}$ at a frequency f. The group delay τ_g at frequency f for a finite impulse response filter of length N is computed as

$$\tau_g = \Re\left\{ \frac{\sum_{k=0}^{N-1} h(k)e^{j2\pi fk} \cdot k}{\sum_{k=0}^{N-1} h(k)e^{j2\pi fk}} \right\} \tag{52}$$

`iir_group_delay(*b,nb,*a,na,f)` computes the group delay for an infinite impulse-response filter with n_a feed-back coefficients $\boldsymbol{a}$, and n_a feed-forward coefficients $\boldsymbol{b}$ at a frequency f. The group delay τ_g at frequency f for an infinite impulse response filter of order N is computed as

$$\tau_g = \Re\left\{ \frac{\sum_{k=0}^{2(N+1)} c(k)e^{j2\pi fk} \cdot k}{\sum_{k=0}^{2(N+1)} c(k)e^{j2\pi fk}} \right\} - N \tag{53}$$

where $c(n) = \sum_{m=0}^{N-1} a(m)^* b(m-n)$ for $n \in \{0, 1, \ldots, 2(N+1)\}$ which can be described as the flipped convolution of $\boldsymbol{a}$ and $\boldsymbol{b}$.

`iirdes_isstable(*b,*a,n)` checks the stability of an infinite impulse-response filter with n feed-back and feed-forward coefficients $\boldsymbol{a}$ and $\boldsymbol{b}$ respectively. Stability is tested by computing the roots of the denominator (poles) and ensuring that they lie within the unit circle. Notice that the poles in Figures 19–23 all have their poles within the unit circle and are therefore stable (as expected).

`liquid_filter_autocorr(*h,N,n)` computes the auto-correlation of a filter with an array of coefficients $\boldsymbol{h}$ of length N at a specific lag n as

$$r_{hh}(n) = \sum_{k=n}^{N-1} h(k)h^*(k-n) \tag{54}$$

`liquid_filter_isi(*h,k,m,*rms,*max)` computes the inter-symbol interference (both mean-squared error and maximum error) for a filter $\boldsymbol{h}$ with k samples per symbol and delay of m samples. The filter has $2km+1$ coefficients and the resulting RMS and maximum ISI are stored in `rms` and `max`, respectively. This is useful in comparing the performance of root-Nyquist matched filter designs (e.g. root raised-cosine).

`liquid_filter_energy(*h,N,fc,nfft)` computes the relative out-of-band energy E_0 at a cutoff frequency f_c for a finite impulse response filter $\boldsymbol{h}$ with N coefficients. The parameter `nfft` specifies the precision of the computation. The relative out-of-band energy is computed as

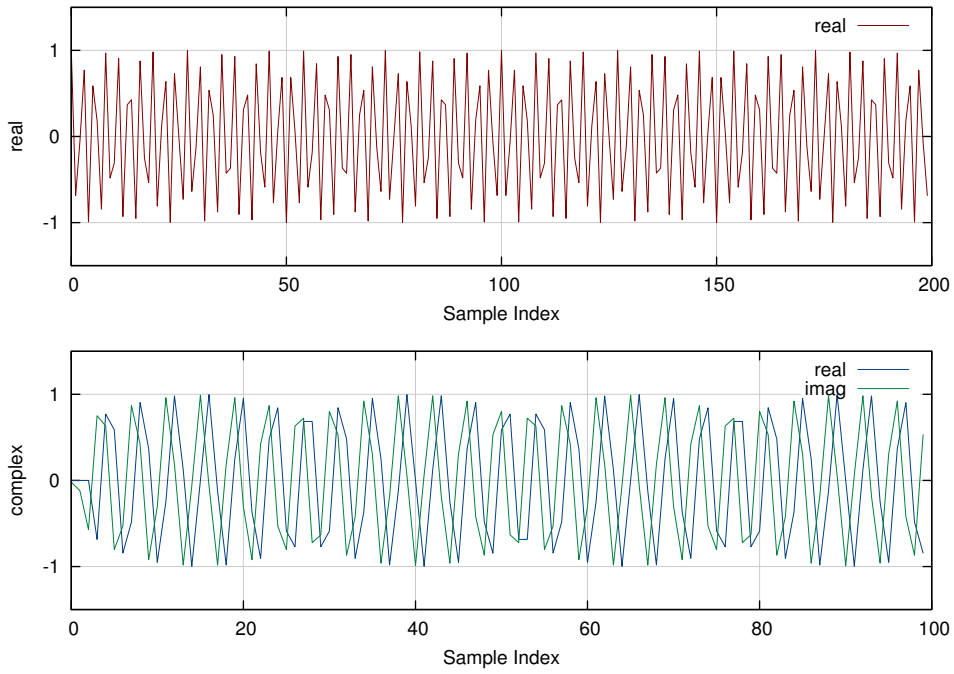
$$E_0 = \frac{\int_{2\pi f_c}^{\infty} H(\omega) d\omega}{\int_0^{\infty} H(\omega) d\omega} \tag{55}$$

14.6 `firhilbf` (finite impulse response Hilbert transform)

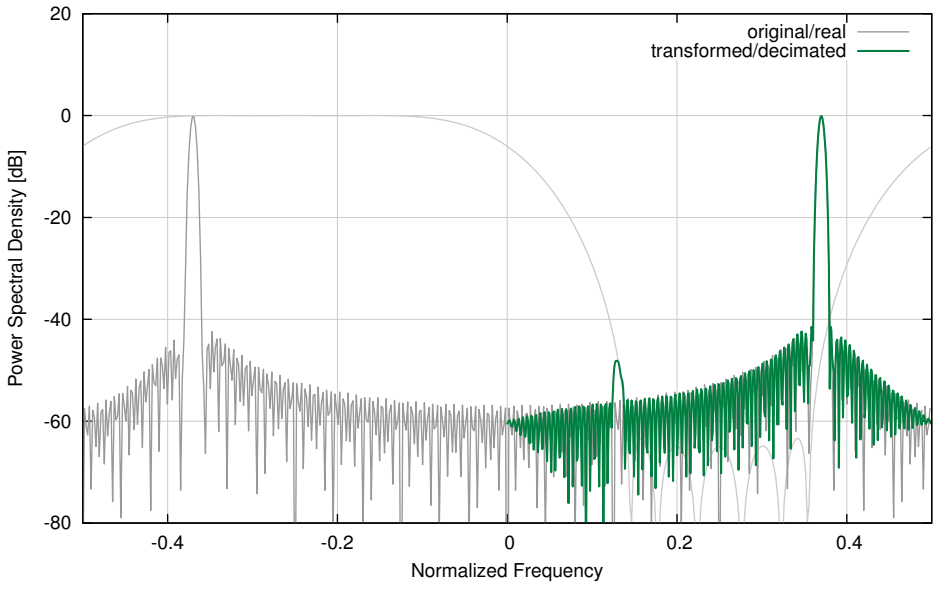
The `firhilbf` object in *liquid* implements a finite impulse response Hilbert transform which converts between real and complex time series. The interpolator takes a complex time series and produces real-valued samples at twice the sample rate. The decimator reverses the process by halving the sample rate of a real-valued time series to a complex-valued one.

Typical trade-offs between filter length, side-lobe suppression, and transition bandwidth apply. The `firhilbf` object uses a half-band filter to implement the transform as efficiently as possible. While any filter length can be accepted, the `firhilbf` object internally forces the length to be of the form $n = 4m + 1$ to reduce the computational load. A halfband filter of this length has $2m$ zeros and $2m + 1$ non-zero coefficients. Of these non-zero coefficients, the center is exactly 1 while the other $2m$ are even symmetric, and therefore only m computations are needed. A graphical example of the Hilbert decimator can be seen in Figure 16 where a real-valued input sinusoid is converted into a complex sinusoid with half the number of samples. An example code listing is given below. Although `firhilbf` is a placeholder for both decimation (real to complex) and interpolation (complex to real), separate objects shoudl be used for each task.

```
1  // file: doc/listings/firhilb.example.c
2  #include <liquid/liquid.h>
3
4  int main() {
5      unsigned int m=5;           // filter semi-length
6      float slsl=60.0f;           // filter sidelobe suppression level
7
8      // create Hilbert transform objects
9      firhilbf q0 = firhilbf_create(m,slsl);
10     firhilbf q1 = firhilbf_create(m,slsl);
11
12     float complex x;    // interpolator input
13     float y[2];         // interpolator output
14     float complex z;    // decimator output
15
16     // ...
17
18     // execute transforms
19     firhilbf_interp_execute(q0, x, y);   // interpolator
20     firhilbf_decim_execute(q1, y, &z);   // decimator
21
22     // clean up allocated memory
23     firhilbf_destroy(q0);
24     firhilbf_destroy(q1);
25 }
```



(a) time



(b) PSD

Figure 16: `firhilbf` (Hilbert transform) decimator demonstration. The small signal at $f = 0.13$ is due to aliasing as a result of imperfect image rejection.

Listed below is the full interface to the `firhilbf` family of objects.

`firhilbf_create(m,As)` creates a `firhilbf` object with a filter semi-length of m samples (equal to the delay) and a stop-band attenuation of A_s dB. The value of m must be at least 2. The internal filter has a length $4m+1$ coefficients and is designed using the `firdes_kaiser_window()` method (see Section 14.5.1 on FIR filter design using windowing functions).

`firhilbf_destroy(q)` destroys the Hilbert transform object, freeing all internally-allocated memory.

`firhilbf_print(q)` prints the internal properties of the object to the standard output.

`firhilbf_clear(q)` clears the internal transform buffers.

`firhilbf_r2c_execute(q,x,*y)` executes the real-to-complex transform as a half-band filter, rejecting the negative frequency band. The input x is a real sample; the output y is complex.

`firhilbf_c2r_execute(q,x,*y)` executes the complex-to-real conversion as $y = \Re\{x\}$.

`firhilbf_decim_execute(q,*x,*y)` executes the transform as a decimator, converting a 2-sample input array $\boldsymbol{x}$ of real values into a single complex output value y.

`firhilbf_interp_execute(q,x,*y)` executes the transform as a decimator, converting a single complex input sample x into a two real-valued samples stored in the output array $\boldsymbol{y}$.

For more detailed examples on Hilbert transforms in *liquid*, refer to the files `examples/firhilb_decim_example.c` and `examples/firhilb_interp_example.c` located within the main *liquid* project directory. *See also:* `resamp2` (Section 14.10), FIR filter design (Section 14.5).

14.7 `interp` (interpolator)

The `interp` object implements a basic interpolator with an integer output-to-input resampling ratio. An example of the `interp` interface is listed below.

```
1  // file: doc/listings/interp.example.c
2  #include <liquid/liquid.h>
3
4  int main() {
5      unsigned int M=4;       // interpolation factor
6      unsigned int h_len;     // interpolation filter length
7
8      // design filter and create interpolator
9      float h[h_len];         // filter coefficients
10     interp_crcf q = interp_crcf_create(M,h,h_len);
11
12     // generate input signal and interpolate
13     float complex x;        // input sample
14     float complex y[M];     // output samples
15
16     // run interpolator (repeat as necessary)
17     {
```

```
18          interp_crcf_execute(q, x, y);
19      }
20
21      // destroy the interpolator object
22      interp_crcf_destroy(q);
23  }
```

Listed below is the full interface to the `interp` family of objects. While each method is listed for `interp_crcf`, the same functionality applies to `interp_rrrf` and `interp_cccf`.

`interp_crcf_create(M,*h,N)` creates an `interp` object with an interpolation factor M using N filter coefficients $\boldsymbol{h}$.

`interp_crcf_create_rnyquist(type,k,m,beta,dt)` creates an `interp` object from a square-root Nyquist filter prototype with k samples per symbol (interpolation factor), m symbols of delay, β excess bandwidth, and a fractional sampling interval Δt. Section 14.5.2 provides a detailed description of the available square-root Nyquist filter prototypes available in *liquid*.

`interp_crcf_destroy(q)` destroys the interpolator, freeing all internally-allocated memory.

`interp_crcf_print(q)` prints the internal properties of the interpolator to the standard output.

`interp_crcf_clear(q)` clears the internal interpolator buffers.

`interp_crcf_execute(q,x,*y)` executes the interpolator for an input x, storing the result in the output array y (which has a length of M samples).

A graphical example of the interpolator can be seen in Figure 17. A detailed example program is given in `examples/interp_crcf_example.c`, located under the main *liquid* project directory.

14.8 `iirfilt` (infinite impulse response filter)

The `iirfilt_crcf` object and family implement the infinite impulse response (IIR) filters. Also known as recursive filters, IIR filters allow a portion of the output to be fed back into the input, thus creating an impulse response which is non-zero for an infinite amount of time. Formally, the output signal $y[n]$ may be written in terms of the input signal $x[n]$ as

$$y[n] = \frac{1}{a_0}\left(\sum_{j=0}^{n_b-1} b_j x[n-j] - \sum_{k=1}^{n_a-1} a_k y[n-k]\right) \tag{56}$$

where $\boldsymbol{b} = [b_0, b_1, \ldots, b_{n_b-1}]^T$ are the feed-forward parameters and $\boldsymbol{a} = [a_0, a_1, \ldots, a_{n_a-1}]^T$ are the feed-back parameters of length n_b and n_a, respectively. The z-transform of the transfer function is therefore

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{j=0}^{n_b-1} b_j z^{-j}}{\sum_{k=0}^{n_a-1} a_k z^{-k}} = \frac{b_0 + b_1 z^{-1} + \cdots + b_{n_b-1} z^{n_b-1}}{a_0 + a_1 z^{-1} + \cdots + a_{n_a-1} z^{n_a-1}} \tag{57}$$

Typically the coefficients in $H(z)$ are normalized such that $a_0 = 1$.

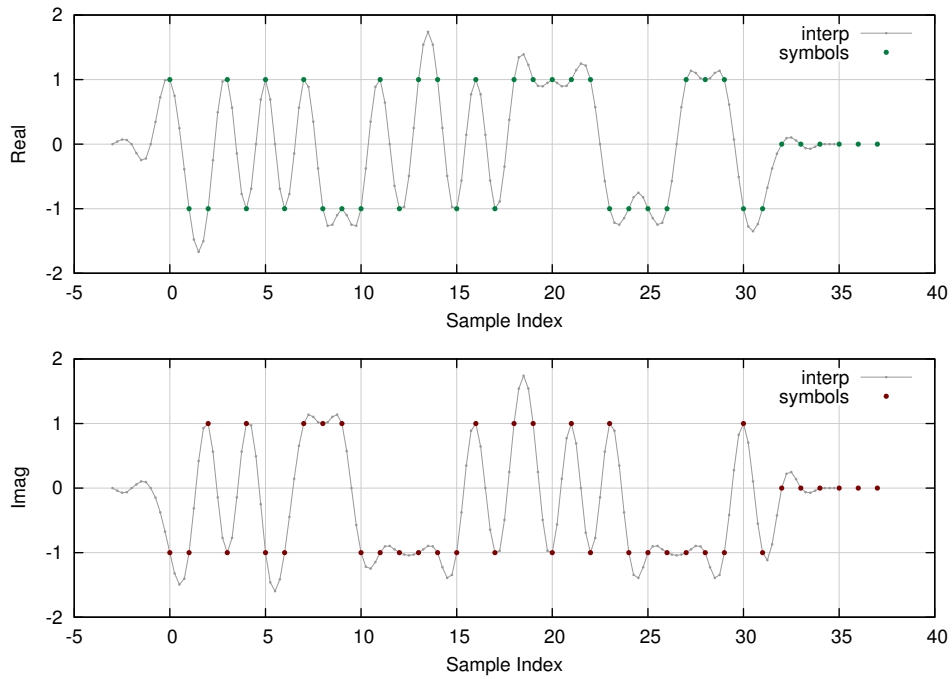


Figure 17: `interp_crcf` (interpolator) example with $M = 4$, compensating for filter delay.

For larger order filters (even as small as $n \approx 8$) the filter can become unstable due to finite machine precision. It is often therefore useful to express $H(z)$ in terms of second-order sections. For a filter of order n, these sections are denoted by the two $(L + r) \times 3$ matrices $\boldsymbol{B}$ and $\boldsymbol{A}$ where $r = n \mod 2$ (0 for odd n, 1 for even n) and $L = (n - r)/2$.

$$H_d(z) = \left[\frac{B_{r,0} + B_{r,1}z^{-1}}{1 + A_{r,1}z^{-1}}\right]^r \prod_{k=1}^{L}\left[\frac{B_{k,0} + B_{k,1}z^{-1} + B_{k,2}z^{-2}}{1 + A_{k,1}z^{-1} + A_{k,2}z^{-2}}\right] \tag{58}$$

Notice that $H(z)$ is now a series of cascaded second-order IIR filters. The 'sos' form is practical when filters are designed from analog prototypes where the poles and zeros are known. *liquid* implements second-order sections efficiently with the internal `iirfiltsos_crcf` family of objects. For a cascaded second-order section IIR filter, use `iirfilt_crcf_create_sos(B,A,n)`. *See also*: `iirdes` (IIR filter design) in Section 14.9.

Listed below is the full interface to the `iirfilt` family of objects. The interface to the `iirfilt` object follows the convention of otherr *liquid* signal processing objects; while each method is listed for `iirfilt_crcf`, the same functionality applies to `iirfilt_rrrf` and `iirfilt_cccf`.

`iirfilt_crcf_create(*b,Nb,*a,Nb)` creates a new `iirfilt` object with N_b feed-forward coefficients $\boldsymbol{b}$ and N_a feed-back coefficients $\boldsymbol{a}$.

`iirfilt_crcf_create_sos(*B,*A,Nsos)` creates a new `iirfilt` object using N_{sos} second-order sections. The $[N_{sos} \times 3]$ feed-forward coefficient matrix is specified by $\boldsymbol{B}$ and the $[N_{sos} \times 3]$ feed-back coefficient matrix is specified by $\boldsymbol{A}$.

`iirfilt_crcf_create_prototype(ftype,btype,format,order,fc,f0,Ap,As)` creates a new IIR filter object using the prototype interface described in Section 14.9.1. This is the simplest method for designing an IIR filter with Butterworth, Chebyshev-I, Chebyshev-II, elliptic/-Cauer, or Bessel coefficients.

`iirfilt_crcf_destroy(q)` destroys an `iirfilt` object, freeing all internally-allocated memory arrays and buffers.

`iirfilt_crcf_print(q)` prints the internals of an `iirfilt` object.

`iirfilt_crcf_clear(q)` clears the filter's internal state.

`iirfilt_crcf_execute(q,x,*y)` executes one iteration of the filter with an input x, storing the result in y, and updating its internal state.

`iirfilt_crcf_get_length(q)` returns the order of the filter.

`iirfilt_crcf_freqresponse(q,fc,*H)` computes the complex response H of the filter at the normalized frequency f_c.

`iirfilt_crcf_groupdelay(q,fc)` returns the group delay of the filter at the normalized frequency f_c.

Listed below is a basic example of the interface. For more detailed and extensive examples, refer to `examples/iirfilt_crcf_example.c` in the main *liquid* project source directory.

```
// file: doc/listings/iirfilt.example.c
#include <liquid/liquid.h>

int main() {
    // options
    unsigned int order=4;   // filter order

    unsigned int n = order+1;
    float b[n], a[n];

    // ... initialize filter coefficients ...

    // create filter object
    iirfilt_crcf q = iirfilt_crcf_create(b,n,a,n);

    float complex x;    // input sample
    float complex y;    // output sample

    // execute filter (repeat as necessary)
    iirfilt_crcf_execute(q,x,&y);

    // destroy filter object
    iirfilt_crcf_destroy(q);
}
```

An example of the `iirfilt` can be seen in Figure 18 in which a low-pass filter is applied to a signal to remove a high-frequency component.

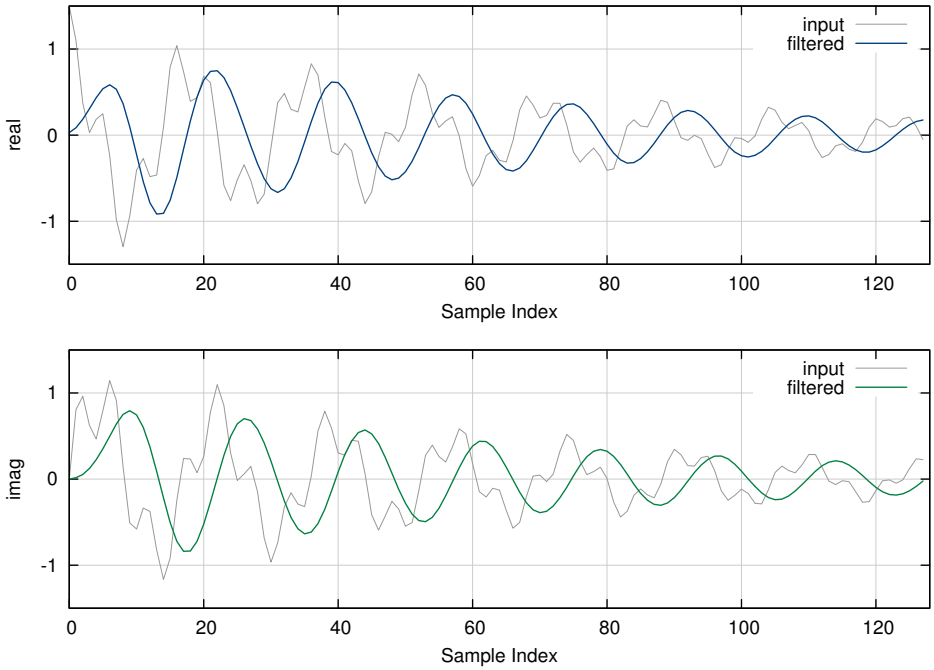


Figure 18: `iirfilt_crcf` (infinite impulse response filter) example.

14.9 `iirdes` (infinite impulse response filter design)

liquid implements infinite impulse respone (IIR) filter design for the four major classes of filters (Butterworth, Chebyshev, elliptic, and Bessel) by first computing their analog low-pass prototypes, performing a bilinear z-transform to convert to the digital domain, then transforming to the appropriate band type (e.g. high pass) if necessary. Externally, the user may abstract the entire process by using the `iirdes()` method. Furthermore, if the end result is to create a filter *object* as opposed to computing the coefficients themselves, the `iirfilt_crcf_create_prototype()` method can be used to generate the object directly (see Section 14.8).

14.9.1 `iirdes()`, the simplified method

The `iirdes()` method designs an IIR filter's coefficients from one of the four major types (Butterworth, Chebyshev, elliptic/Cauer, and Bessel) with as minimal an interface as possible. The user specifies the filter prototype, order, cutoff frequency, and other parameters as well as the resulting filter structure (regular or second-order sections), and the function returns the appropriate filter coefficients that meet that design. Specifically, the interface is

```
iirdes(_ftype, _btype, _format, _n, _fc, _f0, _Ap, _As, *_B, *_A);
```

`_ftype` is the analog filter prototype, e.g. `LIQUID_IIRDES_BUTTER`

`_btype` is the band type, e.g. `LIQUID_IIRDES_BANDPASS`

`_format` is the output format of the coefficients, e.g. `LIQUID_IIRDES_SOS`

`_n` is the filter order

`_fc` is the normalized cutoff frequency of the analog prototype

`_f0` is the normalized center frequency of the analog prototype (only applicable to bandpass and bandstop filter designs, ignored for lowpass and highpass filter designs)

`_Ap` is the passband ripple (only applicable to Chebyshev Type-I and elliptic filter designs, ignored for Butterworth, Chebyshev Type-II, and Bessel designs)

`_As` is the stopband ripple (only applicable to Chebyshev Type-II and elliptic filter designs, ignored for Butterworth, Chebyshev Type-I, and Bessel designs)

`_B`, `_A` are the output feed-forward (numerator) and feed-back (denominator) coefficients, respectively. The format and size of these arrays depends on the value of the `_format` and `_btype` parameters.

14.9.2 internal description

While the user only needs to specify the filter parameters, the internal procedure for computing the coefficients is somewhat complicated. Listed below is the step-by-step process for *liquid*'s IIR filter design procedure.

1. Use `butterf()`, `cheby1f()`, `cheby2f()`, `ellipf()`, `besself()` to design a low-pass analog prototype $H_a(s)$ in terms of complex zeros, poles, and gain. The `azpkf` extension stands for "analog zeros, poles, gain (floating-point)."

   `butter_azpkf()` Butterworth (maximally flat in the passband)

   `cheby1_azpkf()` Chebyshev Type-I (equiripple in the passband)

   `cheby2_azpkf()` Chebyshev Type-II (equiripple in the stopband)

   `ellip_azpkf()` elliptic filter (equiripple in the pass- and stopbands)

   `bessel_azpkf()` Bessel (maximally flat group delay)

2. Compute frequency pre-warping factor, m, to set cutoff frequency (and center frequency if designing a band-pass or band-stop filter) using the `iirdes_freqprewarp()` method.

3. Convert the low-pass analog prototype $H_a(s)$ to its digital equivalent $H_d(z)$ (also in terms of zeros, poles, and gain) using the bilinear z-transform using the `bilinear_zpkf()` method. This maps the analog zeros/poles/gain into digital zeros/poles/gain.

4. Transform the low-pass digital prototype to high-pass, band-pass, or band-stop using the `iirdes_dzpk_lp2bp()` method. For the band-pass and band-stop cases, the number of poles and zeros will need to be doubled.

   LP low-pass filter : $s = m(1 + z^{-1})/(1 - z^{-1})$

   HP high-pass filter : $s = m(1 - z^{-1})/(1 + z^{-1})$

BP band-pass filter : $s = m(1 - c_0 z^{-1} + z^{-2})/(1 - z^{-2})$

BS band-stop filter : $s = m(1 - z^{-2})/(1 - c_0 z^{-1} + z^{-2})$

5. Transform the digital $z/p/k$ form of the filter to one of the two forms:

   TF typical transfer function for digital iir filters of the form $B(z)/A(z)$, `iirdes_dzpk2tff()`

   SOS second-order sections form : $\prod_k B_k(z)/A_k(z)$, `iirdes_dzpk2sosf()`. This is the preferred method.

A simplified example for this procedure is given in `examples/iirdes_example.c`.

14.9.3 Available Filter Types

There are currently five low-pass prototypes available for inifinite impulse response filter design in *liquid*, as described below:

`LIQUID_IIRDES_BUTTER` is a Butterworth filter. This is an all-pole analog design that has a maximally flat magnitude response in the passband. The analog prototype interface is `butter_azpkf()` which computes the n complex roots $p_{a0}, p_{a1}, \ldots, p_{an-1}$ of the n^{th}-order Butterworth polynomial,

$$p_{ak} = \omega_c \exp\left\{ j \frac{(2k + n + 1)\,\pi}{2n} \right\} \tag{59}$$

for $k = 0, 1, \ldots, n-1$. Note that this results in a set of complex conjugate pairs such that $(-1)^n s_0 s_1 \cdots s_{n-1} = 1$. An example of a digital filter response can be found in Figure 19;

`LIQUID_IIRDES_CHEBY1` is a Chebyshev Type-I filter. This design uses Chebyshev polynomials to create a filter with a sharper transition band than the Butterworth design by allowing ripples in the passband. The analog prototype interface is `cheby1_azpkf()` which computes the n complex roots p_{ak} of the n^{th}-order Chebyshev polynomial. An example of a digital filter response can be found in Figure 20;

`LIQUID_IIRDES_CHEBY2` is a Chebyshev Type-II filter. This design is similar to that of Chebyshev Type-I, except that the Chebyshev polynomial is inverted. This inverts the magnitude response of the filter and exhibits an equiripple behavior in the stopband, rather than the passband. The analog prototype interface is `cheby2_azpkf()`. An example of a digital filter response can be found in Figure 21

`LIQUID_IIRDES_ELLIP` is an elliptic (Cauer) filter. This design allows ripples in both the passband and stopbands to create a filter with a very sharp transition band. The design process is somewhat more involved than the Butterworth and Chebyshev prototypes and requires solving the elliptic integral of different moduli. For a more detailed description we refer the interested reader to [20]. The analog prototype interface is `ellip_azpkf()`. An example of a digital filter response can be found in Figure 22;

`LIQUID_IIRDES_BESSEL` is a Bessel filter. This is an all-pole analog design that has a maximally flat group delay response (maximally linear phase response). The solution to the design happens to be the roots to the Bessel polynomials of equal order. Computing the roots to

the polynomial is, again, somewhat complex. For a more detailed description we refer the interested reader to [19]. The analog prototype interface is `bessel_azpkf()`. An example of a digital filter response can be found in Figure 23.

14.9.4 `bilinear_zpkf` (Bilinear z-transform)

The bilinear z-transform converts an analog prototype to its digital counterpart. Given a continuous time analog transfer function in zeros/poles/gain form ("zpk") with n_z zeros and n_p poles

$$H_a(s) = k_a \frac{(s-z_{a0})(s-z_{a1})\cdots(s-z_{an_z-1})}{(s-p_{a0})(s-p_{a1})\cdots(s-p_{an_p-1})} \tag{60}$$

the bilinear z-transform converts $H_a(s)$ into the discretized transfer function $H_d(z)$ by mapping the s-plane onto the z-plane with the approximation

$$s \approx \frac{2}{T}\frac{1-z^{-1}}{1+z^{-1}} \tag{61}$$

This maps $H_a(0) \rightarrow H_d(0)$ and $H_a(\infty) \rightarrow H_d(\omega_s/2)$, however we are free to choose the pre-warping factor which maps the cutoff frequency ω_c.

$$s \rightarrow \omega_c \cot\left(\frac{\pi\omega_c}{\omega_s}\right)\frac{1-z^{-1}}{1+z^{-1}} \tag{62}$$

Substituting this into $H_a(s)$ gives the discrete-time transfer function

$$H(z) = k_a \frac{\left(m\frac{1-z^{-1}}{1+z^{-1}} - z_{a0}\right)\left(m\frac{1-z^{-1}}{1+z^{-1}} - z_{a1}\right)\cdots\left(m\frac{1-z^{-1}}{1+z^{-1}} - z_{an_z-1}\right)}{\left(m\frac{1-z^{-1}}{1+z^{-1}} - p_{a0}\right)\left(m\frac{1-z^{-1}}{1+z^{-1}} - p_{a1}\right)\cdots\left(m\frac{1-z^{-1}}{1+z^{-1}} - p_{an_p-1}\right)} \tag{63}$$

where $m = \omega_c \cot(\pi\omega_c/\omega_s)$ is the frequency pre-warping factor, computed in *liquid* via the method `iirdes_freqprewarp()`. Multiplying both the numerator an denominator by $(1+z^{-1})^{n_p}$ and applying some algebraic manipulation results in the digital filter

$$H_d(s) = k_d \frac{(1-z_{d0}z^{-1})(1-z_{d1}z^{-1})\cdots(1-z_{dn-1}z^{-1})}{(1-p_{d0}z^{-1})(1-p_{d1}z^{-1})\cdots(1-p_{dn-1}z^{-1})} \tag{64}$$

The `bilinear_zpk()` method in *liquid* transforms the the analog zeros (z_{ak}), poles (p_{ak}), and gain (H_0) into their digital equivalents (z_{dk}, p_{dk}, G_0). For a filter with n_z analog zeros z_{ak} the digital zeros z_{dk} are computed as

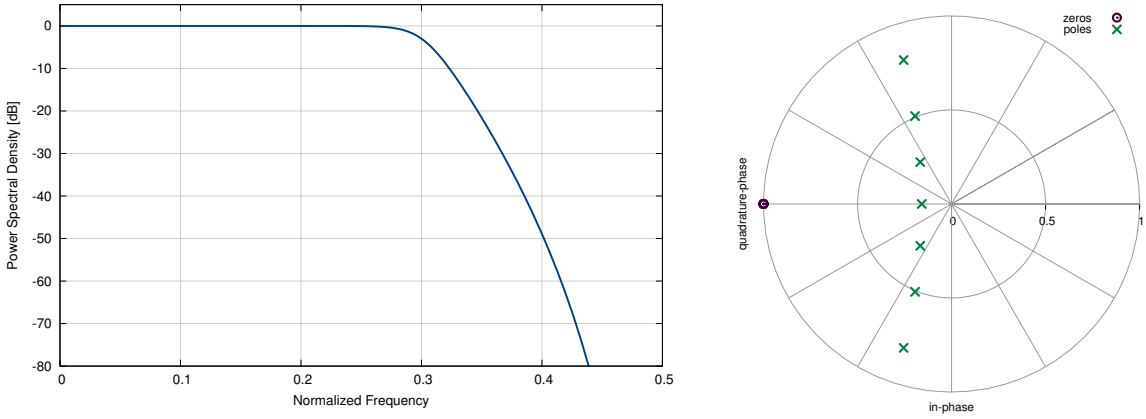
$$z_{dk} = \begin{cases} \frac{1+mz_{ak}}{1-mz_{ak}} & k < n_z \\ -1 & \text{otherwise} \end{cases} \tag{65}$$

where m is the pre-warping factor. For a filter with n_p analog poles p_{ak} the digital poles p_{dk} are computed as

$$p_{dk} = \frac{1+mp_{ak}}{1-mp_{ak}} \tag{66}$$

Keeping in mind that an analog filter's order is defined by its number of poles, the digital gain can be computed as

$$G_0 = H_0 \prod_{k=0}^{n_p-1} \frac{1-p_{dk}}{1-z_{dk}} \tag{67}$$



(a) spectrum (b) zeros, poles

Figure 19: `butterf` (Butterworth filter design)

14.9.5 Filter transformations

The prototype low-pass digital filter can be converted into a high-pass, band-pass, or band-stop filter using a combinatio of the following filter transformations in *liquid*:

`iirdes_dzpk_lp2hp(*_zd,*_pd,_n,*_zdt,*_pdt)` Converts a low-pass digital prototype $H_d(z)$ to a high-pass prototype. This is accomplished by transforming the n zeros and poles (represented by the input arrays `_zd` and `_pd`) into n transformed zeros and poles (represented by the output arrays `_zdt` and `_pdt`).

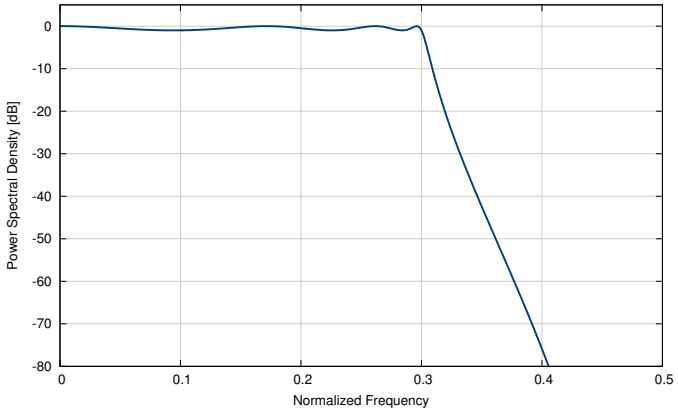
`iirdes_dzpk_lp2bp(*_zd,*_pd,_n,*_zdt,*_pdt)` Converts a low-pass digital prototype $H_d(z)$ to a band-pass prototype. This is accomplished by transforming the n zeros and poles (represented by the input arrays `_zd` and `_pd`) into $2n$ transformed zeros and poles (represented by the output arrays `_zdt` and `_pdt`).

14.9.6 Filter Coefficient Computation

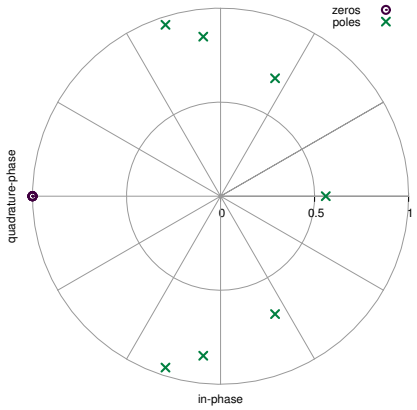
The digital filter defined by (64) can be expanded to fit the familiar IIR transfer function as in (57). This can be accomplished using the `iirdes_dzpk2tff()` method. Alternatively, the filter can be written as a set of cascaded second-order IIR filters:

$$H_d(z) = G_0 \left[\frac{1+z^{-1}}{1-p_0 z^{-1}}\right]^r \prod_{k=1}^{L} \left[G_i \frac{(1-z_i z^{-1})(1-z_i^* z^{-1})}{(1-p_i z^{-1})(1-p_i^* z^{-1})}\right] \tag{68}$$

where $r = 0$ when the filter order is odd, $r = 1$ when the filter order is even, and $L = (n-r)/2$. This can be accomplished using the `iirdes_dzpk2sosf()` method and is preferred over the traditional transfer function design for stability reasons.

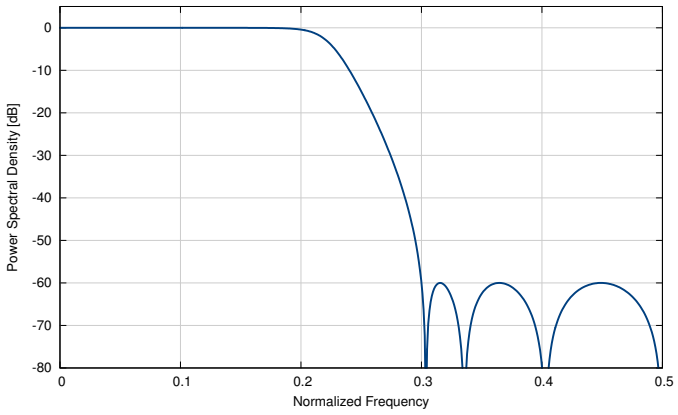


(a) spectrum

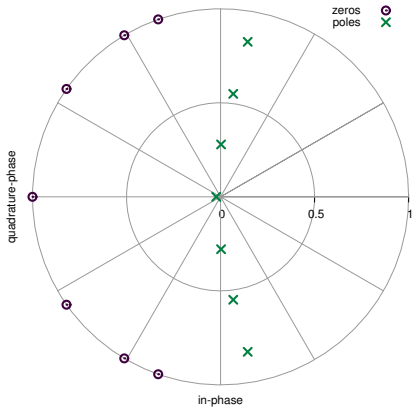


(b) zeros, poles

Figure 20: `cheby1f` (Chebyshev type-I filter design)

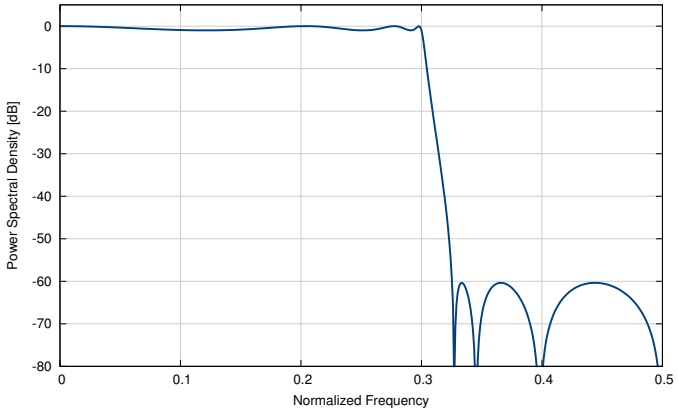


(a) spectrum

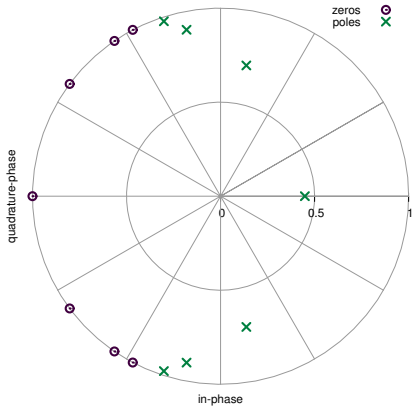


(b) zeros, poles

Figure 21: `cheby2f` (Chebyshev type-II filter design)

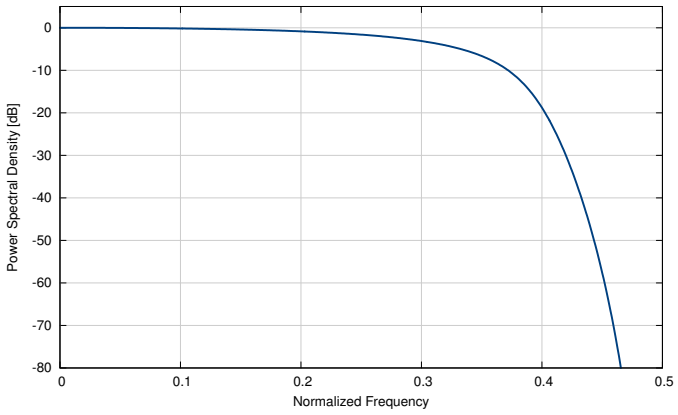


(a) spectrum

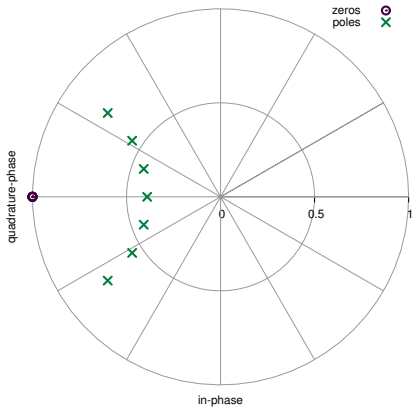


(b) zeros, poles

Figure 22: `ellipf` (Elliptic filter design)



(a) spectrum



(b) zeros, poles

Figure 23: `besself` (Bessel filter design)

14.10 resamp2 (halfband filter/resampler)

resamp2 is a half-band resampler used for efficient interpolation and decimation. The internal filter of the resamp2 object is a Kaiser-windowed sinc (see firdes_kaiser_window, section 14.5) with $f_c = 1/2$. This makes the filter half-band, and puts the half-power (6 dB) cutoff point ω_c at $\pi/2$ (one quarter of the sampling frequency). In fact, any FIR filter design using a windowed sinc function with periodicity $f_c = 1/2$ will generate a Nyquist half-band filter (zero inter-symbol interference). This is because [30, (4.6.3)]

$$h(Mn) = \begin{cases} 1 & n = 0 \\ 0 & \text{otherwise} \end{cases} \tag{69}$$

which holds for $h(n) = w(n)\sin(\pi n/M)/(\pi n)$ since $\sin(\pi n/M) = 0$ for $n =$ any non-zero multiple of M. Additionally, $M = 2$ is the special case of half-band filters. In particular half-band filtering is computationally efficient because half the coefficients of the filter are zero, and the remaining half are symmetric (so long as $w(n)$ is also symmetric). In theory, this means that for a filter length of $4m + 1$ taps, only m computations are necessary [8]. The resamp2 object in *liquid* uses a Kaser window for $w(n)$ for several reasons, but in particular because it is nearly optimum, and it is easy to trade side-lobe attenuation for transition bandwidth. Listed below is the full interface to the resamp2 family of objects. While each method is listed for resamp2_crcf, the same functionality applies to resamp2_rrrf and resamp2_cccf.

`resamp2_crcf_create(m,f0,As)` creates a resamp2 object with a resampling rate 2, a filter semi-length of m samples (equivalent filter length $4m+1$), centered at frequency f_0, and a stop-band suppression of A_s dB.

`resamp2_crcf_recreate(q,m,f0,As)` recreates a resamp2 object with revised parameters.

`resamp2_crcf_destroy(q)` destroys the resampler, freeing all internally-allocated memory.

`resamp2_crcf_print(q)` prints the internal properties of the resampler to the standard output.

`resamp2_crcf_clear(q)` clears the internal resampler buffers.

`resamp2_crcf_filter_execute(q,x,*y0,*y1)` executes the resamp2 object as a half-band filter on an input sample x, storing the low-pass filter output in y_0 and the high-pass filter output in y_1.

`resamp2_crcf_decim_execute(q,*x,*y)` executes the half-band resampler as a decimator for an input array $\boldsymbol{x}$ with two samples, storing the resulting samples in the array y.

`resamp2_crcf_interp_execute(q,x,*y)` executes the half-band resampler as an interpolator for an input sample x, storing the resulting two output samples in the array $\boldsymbol{y}$.

Below is a code example demonstrating the resamp2 interface.

```
1  // file: doc/listings/resamp2_crcf.example.c
2  #include <liquid/liquid.h>
3
4  int main() {
```

```
    // options
    unsigned int m = 7;         // filter semi-length
    float As=-60.0f;            // resampling filter stop-band attenuation

    // create half-band resampler
    resamp2_crcf q = resamp2_crcf_create(m,0.0f,As);

    float complex x;            // complex input
    float complex y[2];         // output buffer

    // ... initialize input ...
    {
        // execute half-band resampler as interpolator
        resamp_crcf_interp_execute(q, x, y);
    }

    // ... repeat as necessary ...

    // clean up allocated objects
    resamp2_crcf_destroy(q);
}
```

For more detailed and extensive examples, refer to `examples/resamp2_crcf_decim_example.c` and `examples/resamp2_crcf_interp_example.c` located in the main *liquid* project source directory.

14.11 resamp (arbitrary resampler)

For arbitrary (e.g. irrational) resampling ratios, the `resamp` object is the ideal solution. It makes no restrictions on the output-to-input resampling ratio (e.g. irrational values are fair game). The arbitrary resampler uses a polyphase filter bank for interpolation between available input sample points.

Because the number of outputs for each input is not fixed, the interface needs some explaining. Over time the true resampling ratio will equal the value specified, however from one input to the next, the number of outputs will change. For example, if the resampling rate is 2, every input will produce exactly two output samples. However, if the resampling rate is $\sqrt{2} \approx 1.4142$, an input sample will usually produce one output, but sometimes two. In the limit (on *average*) however, the ratio of output samples to input samples will be exactly $\sqrt{2}$. The `resamp` object handles this internally by storing the accumulated sampling phase and produces an output for each overflow (i.e. values where the accumulated phase is equal to or exceeds 1).

Below is a code example demonstrating the `resamp` interface. Notice that the `resamp_crcf_execute()` method also returns the number of samples written to the buffer. This number will never exceed $\lceil r \rceil$.

```
// file: doc/listings/resamp_crcf.example.c
#include <liquid/liquid.h>

int main() {
    // options
    unsigned int h_len = 13;    // filter semi-length (filter delay)
```

```
7       float r=0.9f;               // resampling rate (output/input)
8       float bw=0.5f;              // resampling filter bandwidth
9       float slsl=-60.0f;          // resampling filter sidelobe suppression level
10      unsigned int npfb=32;       // number of filters in bank (timing resolution)
11
12      // create resampler
13      resamp_crcf q = resamp_crcf_create(r,h_len,bw,slsl,npfb);
14
15      unsigned int n = (unsigned int)ceilf(r);
16      float complex x;            // complex input
17      float complex y[n];         // output buffer
18      unsigned int num_written;   // number of values written to buffer
19
20      // ... initialize input ...
21
22      // execute resampler, storing result in output buffer
23      resamp_crcf_execute(q, x, y, &num_written);
24
25      // ... repeat as necessary ...
26
27      // clean up allocated objects
28      resamp_crcf_destroy(q);
29  }
```

Figure 24 gives a graphical depiction of the arbitrary resampler, in both the time and frequency domains. The time series has been aligned (shifted by the filter delay and scaled by the resampling rate) to show equivalence. Additionally, the signal's power spectrum has been scaled by r to reflect the change in sampling rate. In the example the input array size is 187 samples; the resampler produced 133 output samples which yields a true resampling rate of $\dot{r} = 133/187 \approx 0.71123$ which is close to the target rate of $r = 1/\sqrt{2} \approx 0.70711$.

It is important to understand how filter design impacts the performance of the resampler. The `resamp` object interpolates between available sample points to minimize aliasing effects on the output signal. This is apparent in the power spectral density plot in figure 24 which shows very little aliasing on the output signal. Aliasing can be reduced by increasing the filter length at the cost of additional computational complexity; additionally the number of filters in the bank can be increased to improve timing resolution between samples. For synchronization of digital receivers, it is always good practice to preceed the resampler with an anti-aliasing filter to remove out-of-band interference.

Listed below is the full interface to the `resamp` family of objects. While each method is listed for `resamp_crcf`, the same functionality applies to `resamp_rrrf` and `resamp_cccf`.

`resamp_crcf_create(r,m,fc,As,N)` creates a `resamp` object with a resampling rate r, a nominal filter delay of m samples, a cutoff frequency of f_c, a stop-band suppression of A_s dB, using a polyphase filterbank with N filters.

`resamp_crcf_destroy(q)` destroys the resampler, freeing all internally-allocated memory.

`resamp_crcf_print(q)` prints the internal properties of the resampler to the standard output.

`resamp_crcf_reset(q)` clears the internal resampler buffers.

`resamp_crcf_setrate(q,r)` sets the resampling rate to r.

`resamp_crcf_execute(q,x,*y,*nw)` executes the resampler for an input sample x, storing the resulting samples in the output array y specifying the number of samples written as n_w. The output buffer y needs to be at least $\lceil r \rceil$.

See also: `resamp2`, `firpfb`, `symsync`, `examples/resamp_crcf_example.c`

14.12 symsync (symbol synchronizer)

The `symsync` object is a multi-rate symbol timing synchronizer useful for locking a received digital signal to the receiver's clock. It is effectively the same as the `resamp` object, but includes an internal control mechanism for tracking to timing phase and frequency offsets. The filter structure is a polyphase representation of a Nyquist matched filter. The instantaneous timing error is computed from the maximum likelihood timing error detector [17] which relies on the derivative to the matched filter impulse response. *liquid* internally computes polyphase filter banks for both the matched and derivative-matched filters. If the output of the matched filter at sample k is $y(k)$ and the output of the derivative matched filter is $\dot{y}(k)$ then the instantaneous timing estimate is

$$e_\tau(k) = \tanh\Big(y(k)\dot{y}(k)\Big) \tag{70}$$

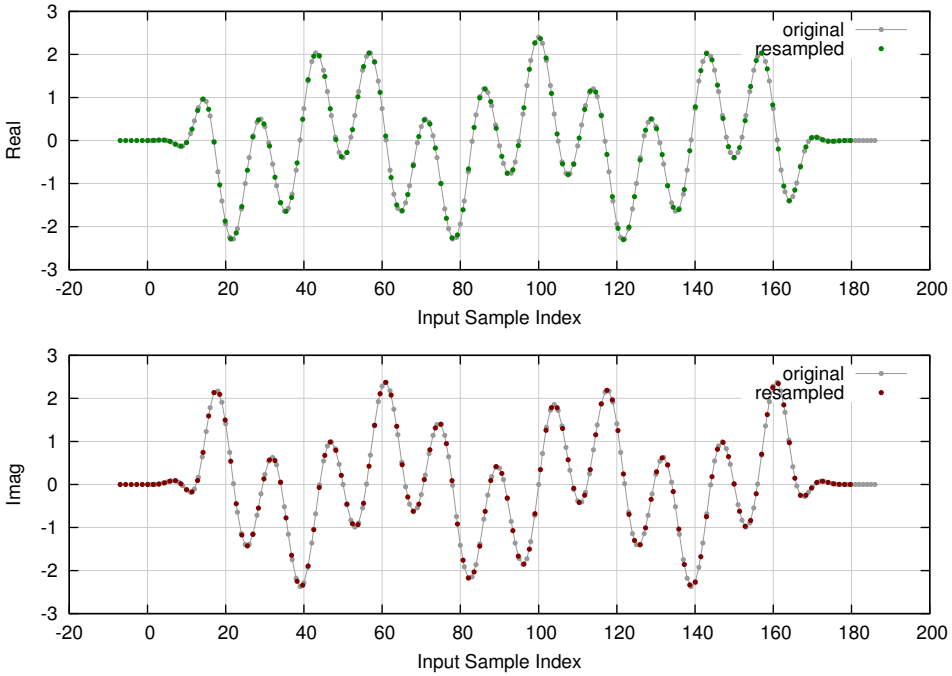
This timing error estimate has significant improvements over heuristic-based estimates such as the popular Mueller and Müller timing recovery scheme [18]. Applying a simple first-order recursive loop filter yields the averaged timing estimate

$$\Delta\tau(k) = \beta e_\tau(k) + \alpha\Delta\tau(k-1) \tag{71}$$

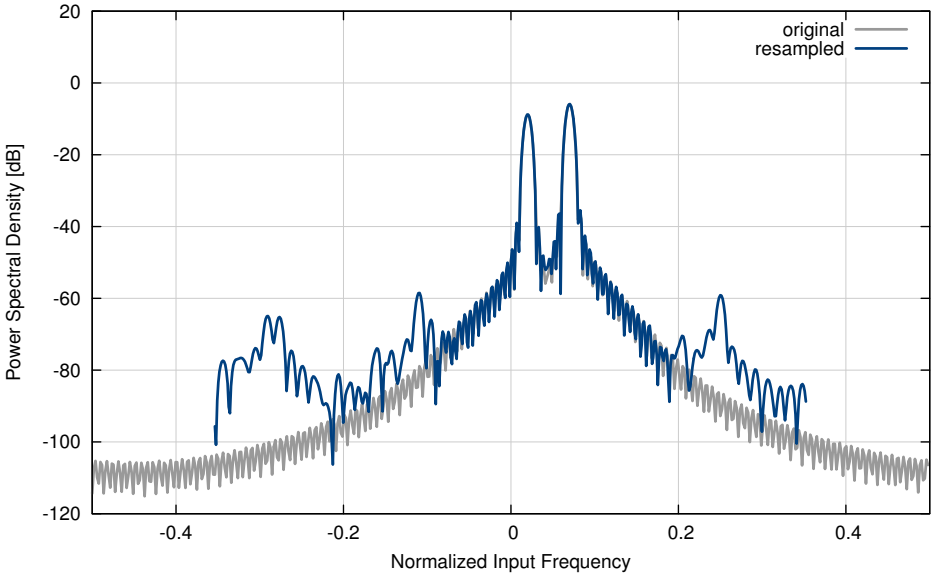
where $\alpha = 1 - \omega_\tau$ and $\beta = 0.22\omega_\tau$ are the loop filter coefficients for a given filter bandwidth ω_τ. While these coefficients are certainly not optimized, it is important to understand the difficulty in computing loop filter coefficients when a delay is introduced into a control loop. This delay is the result of the matched filter itself and can cause instability with traditional phase-locked loop filter designs. Internally the `symsync` object uses the principles of the `resamp` object (arbitrary resampler, see Section 14.11) for resampling the signal—actually decimating to one sample per symbol. Its internal control loop dynamically adjusts the rate r such that the timing phase of the receiver is aligned with the incoming signal's symbol timing.

Below is a code example demonstrating the `symsync` interface. Notice that the `symsync_crcf_execute()` method also returns the number of symbols written to the output buffer.

```
1  // file: doc/listings/symsync_crcf.example.c
2  #include <liquid/liquid.h>
3
4  int main() {
5      // options
6      unsigned int k=2;           // samples/symbol
7      unsigned int m=3;           // filter delay (symbols)
8      float beta=0.3f;            // filter excess bandwidth factor
9      unsigned int Npfb=32;       // number of polyphase filters in bank
10     liquid_rnyquist_type ftype = LIQUID_RNYQUIST_RRC;
```



(a) time



(b) PSD

Figure 24: `resamp_crcf` (arbitrary resampler) demonstration, $r = 1/\sqrt{2} \approx 0.7071$

```
11
12      // create symbol synchronizer
13      symsync_crcf q = symsync_crcf_create_rnyquist(ftype,k,m,beta,Npfb);
14
15      float complex * x;          // complex input
16      float complex * y;          // output buffer
17      unsigned int nx;            // number of input samples
18      unsigned int num_written;   // number of values written to buffer
19
20      // ... initialize input, output ...
21
22      // execute symbol synchronizer, storing result in output buffer
23      symsync_crcf_execute(q, x, nx, y, &num_written);
24
25      // ... repeat as necessary ...
26
27      // clean up allocated objects
28      symsync_crcf_destroy(q);
29  }
```

Listed below is the full interface to the `symsync` family of objects. While each method is listed for `symsync_crcf`, the same functionality applies to `symsync_rrrf` and `symsync_cccf`.

`symsync_crcf_create(k,N,*h,h_len)` creates a `symsync` object from a prototype filter h of length h_{len} and having k samples per symbol. The internal object restructures the input filter into a polyphase prototype having N filters.

`symsync_crcf_create_rnyquist(ftype,k,m,beta,N)` creates a `symsync` object from a square-root Nyquist protoype of type `ftype` (see Section 14.5.2 for a description of available square-root Nyquist filters in *liquid*). The generated filter has k samples per symbol, a nominal delay of m symbols, and an excess bandwidth factor of β. The internal polyphase filter bank has N filters.

`symsync_crcf_destroy(q)` destroys the symbol synchronizer, freeing all internally-allocated memory.

`symsync_crcf_print(q)` prints the internal properties of the symbol syncrhonizer object to the standard output.

`symsync_crcf_clear(q)` resets the symbol synchronizer, clearing the internal buffers and filter state.

`symsync_crcf_set_lf_bw(q,w)` sets the internal bandwidth of the loop filter to ω.

`symsync_crcf_lock(q)` locks the symbol synchronizer such that it will still decimate the incoming signal but will not update its internal state.

`symsync_crcf_unlock(q)` unlocks the symbol synchronizer, resuming its ability to track to the input signal.

`symsync_crcf_execute(q,*x,nx,*y,*ny)` executes the resampler for an input array $\boldsymbol{x}$ with n_x samples, storing the resulting samples in the output array y specifying the number of samples written as n_y.

`symsync_crcf_get_tau(q)` returns the current timing estimate (fractional sampling interval) of the object.

For a more detailed example, refer to `examples/symsync_crcf_example.c` located under the main *liquid* project source directory.

15 framing

The framing module contains objects and methods for packaging data into manageable frames and packets. For convention, *liquid* refers to a "packet" as a group of binary data bytes (often with forward error-correction applied) that need to be communicated over a wireless link. By contrast, a "frame" is a representation of the data once it has been properly partitioned, encapsulated, and modulated before transmitting over the air. Included in this module are the `packetizer`, `frame64`, and `flexframe` structures which greatly simplify over-the-air digital communication of raw data.

15.1 `interleaver`

This section describes the functionality of the *liquid* `interleaver` object. In wireless communications systems, bit errors are often grouped together as a result of multi-path fading, demodulator symbol errors, and synchronizer instability. Interleavers serve to distribute grouped bit errors evenly throughout a block of data which aids certain forward error-correction (FEC) codes in their decoding process (see section 12 on error-correcting codes). On the transmit side of the wireless link, the interleaver re-orders the bits after FEC encoding and before modulation. On the receiving side, the de-interleaver re-shuffles the bits to their original position before attempting to run the FEC decoder. The bit-shuffling order must be known at both the transmitter and receiver.

The `interleaver` object operates by permuting indices on the input data sequence. The indices are computed during the `interleaver_create()` method and stored internally. At each iteration data bytes are re-shuffled using the permutation array. Depending upon the properties of the array, multiple iterations should not result in observing the original data sequence. Shown below is a simple example where 8 symbols $(0, \ldots, 7)$ are re-ordered using a random permutation. The data at iteration 0 are the original data which are permuted twice.

```
forward
permutation     iter[0]     iter[1]     iter[2]
0 -> 6          0           6           1
1 -> 4          1           4           3
2 -> 7          2           7           5
3 -> 0          3           0           6
4 -> 3          4           3           0
5 -> 2          5           2           7
6 -> 1          6           1           4
7 -> 5          7           5           2
```

Reversing the process is as simple as computing the reverse permutation from the input; this is equivalent to reversing the arrows in the forward permutation (e.g. the $2 \rightarrow 7$ forward permutation becomes the $7 \rightarrow 2$ reverse permutation).

```
reverse
permutation     iter[2]     iter[1]     iter[0]
0 -> 3          1           6           0
1 -> 6          3           4           1
2 -> 5          5           7           2
3 -> 4          6           0           3
```

```
4 -> 1          0          3          4
5 -> 7          7          2          5
6 -> 0          4          1          6
7 -> 2          2          5          7
```

Notice that permuting indices only re-orders the bytes of data and does nothing to shuffle the bits within the byte. It is beneficial to FEC decoders to separate the bit errors as much as possible. Therefore, in addition to index permutation, *liquid* also applies masks to the data while permuting.

Two options are available in *liquid* for shuffling bits: `LIQUID_INTERLEAVER_BLOCK` and `LIQUID_INTERLEAVER_SEQ` These two methods are described here.

15.1.1 `LIQUID_INTERLEAVER_BLOCK` (block interleaving)

The block interleaver observes the block data as a matrix: samples are read in by rows and out by columns.

15.1.2 `LIQUID_INTERLEAVER_SEQUENCE` (m-sequence interleaving)

This type of interleaving uses a special linear feedback shift register called an m-sequence in order to compute its permutations. The m-sequence has the special property that each symbol in the register is both unique and pseudo-random.

See also `msequence` (section 22).

15.1.3 Interface

The `interleaver` object operates like most objects in *liquid* with typical `create()`, `destroy()`, and `execute()` methods.

`interleaver_create(n,type)` creates an interleaver object accepting n bytes, with either a `LIQUID_INTERLEAVER_` or `LIQUID_INTERLEAVER_SEQUENCE` type, and defaulting to 2 iterations.

`interleaver_destroy(q)` destroys the interleaver object, freeing all internally-allocated memory arrays.

`interleaver_set_num_iterations(q,k)` sets the number of iterations of the interleaver. Increasing the number of iterations helps improve bit dispersion, but can also increase execution time. The default number of iterations at the time of creation is 2 (see Figure 25).

`interleaver_encode(q,*msg_dec,*msg_enc)` runs the forward interleaver, reading data from the first array argument and writing the result to the second array argument. The array pointers can reference the same block of memory, if necessary.

`interleaver_decode(q,*msg_enc,*msg_dec)` runs the reverse interleaver, reading data from the first array argument and writing the result to the second array argument. Like the `encode()` method, the array pointers can reference the same block of memory.

This listing gives a basic demonstration to the interface to the `interleaver` object:

```c
// file: doc/listings/interleaver.example.c
#include <liquid/liquid.h>

int main() {
    // options
    unsigned int n=9; // message length (bytes)

    // create the interleaver
    interleaver q = interleaver_create(n, LIQUID_INTERLEAVER_SEQUENCE);

    // create arrays
    unsigned char msg_org[n];   // original message data
    unsigned char msg_int[n];   // interleaved data
    unsigned char msg_rec[n];   // de-interleaved, recovered data

    // ...initialize msg_org...

    // interleave/de-interleave the data
    interleaver_encode(q, msg_org, msg_int);
    interleaver_decode(q, msg_int, msg_rec);

    // destroy the interleaver object
    interleaver_destroy(q);
}
```

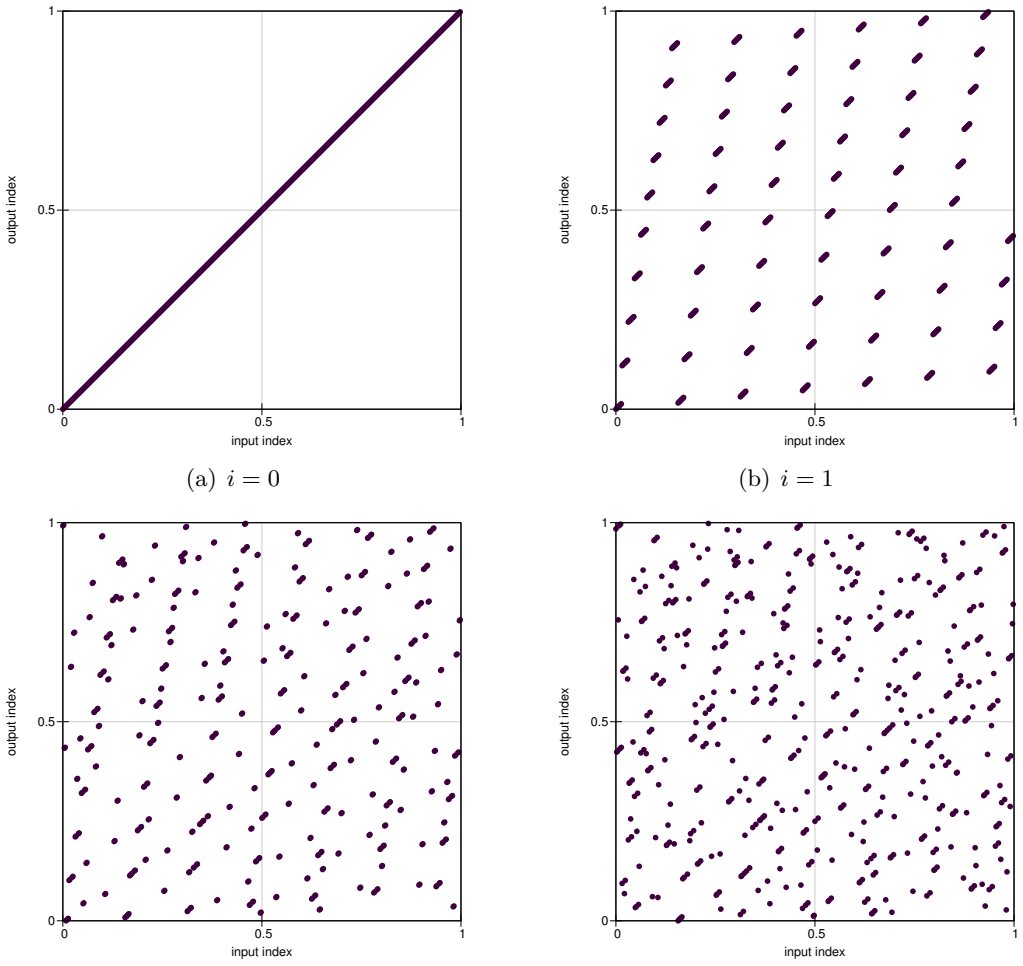
A visualization of the interleaver can be seen in Figure 25 where the input index is plotted against the output index for varying number of interations. Notice that with zero iterations, the output and input are identical (no interleaving). With one iteration only the bytes are interleaved, and so the output is grouped into 8-bit blocks. Further iterations, however, result in sufficiently dispersed bits, and patterns between input and output indices become less evident. The `packetizer` object (Section 15.2) uses the `interleaver` object in conjunction to forward error-correction coding (Section 12) to provide a simple interface for generating protected data packets. A full example can be found in `examples/interleaver_example.c`.

15.2 `packetizer` (multi-level error-correction)

The *liquid* packetizer is a structure for abstracting multi-level forward error-correction from the user. The packetizer accepts a buffer of uncoded data bytes and adds a cyclic redundancy check (CRC) before applying two levels of forward error-correction and bit-level interleaving. The user may choose any two supported FEC schemes (including none) and the packetizer object will handle buffering and data management internally, providing a truly abstract interface. The same is true for the packet decoder which accepts an array of possibly corrupt data and attempts to recover the original message using the FEC schemes provided. The packet decoder returns the validity of the resulting CRC as well as its best effort of decoding the message.

The packetizer also allows for re-structuring if the user wishes to change error-correction schemes or data lengths. This is accomplished with the `packetier_recreate()` method. Listed below is the full interface to the `packetizer` object.

`packetizer_create(n,crc,fec0,fec1)` creates and returns a `packetizer` object which accepts



(a) $i = 0$ (b) $i = 1$

(c) $i = 2$ (d) $i = 3$

Figure 25: `interleaver` (block) demonstration of a 64-byte (512-bit) array with increasing number of iterations (interleaving depth)

n uncoded input bytes and uses the specified CRC and bi-level FEC schemes.

`packetizer_recreate(q,n,crc,fec0,fec1)` re-creates an existing `packetizer` object with new parameters.

`packetizer_destroy(q)` destroys an `packetizer` object, freeing all internally-allocated memory.

`packetizer_print(q)` prints the internal state of the `packetizer` object to the standard output.

`packetizer_get_dec_msg_len(q)` returns the specified decoded message length n in bytes.

`packetizer_get_enc_msg_len(q)` returns the fully-encoded message length k in bytes.

`packetizer_encode(q,*msg,*pkt)` encodes the n-byte input message storing the result in the k-byte encoded output message.

`packetizer_encode(q,*pkt,*msg)` decodes the k-byte encoded input message storing the result in the k-byte output. The function returns a `1` if the internal CRC passed and a `0` if it failed. If no CRC was specified (e.g. `LIQUID_CRC_NONE`) then a `1` is always returned.

Here is a minimal example demonstrating the packetizer's most basic functionality:

```
1  // file: doc/listings/packetizer.example.c
2  #include <liquid/liquid.h>
3
4  int main() {
5      // set up the options
6      unsigned int n=16;                      // uncoded data length
7      crc_scheme crc  = LIQUID_CRC_32;        // validity check
8      fec_scheme fec0 = LIQUID_FEC_HAMMING74; // inner code
9      fec_scheme fec1 = LIQUID_FEC_REP3;      // outer code
10
11     // compute resulting packet length
12     unsigned int k = packetizer_compute_enc_msg_len(n,crc,fec0,fec1);
13
14     // set up the arrays
15     unsigned char msg[n];       // original message
16     unsigned char packet[k];    // encoded message
17     unsigned char msg_dec[n];   // decoded message
18     int crc_pass;               // decoder validity check
19
20     // create the packetizer object
21     packetizer p = packetizer_create(n,crc,fec0,fec1);
22
23     // initialize msg here
24     unsigned int i;
25     for (i=0; i<n; i++) msg[i] = i & 0xff;
26
27     // encode the packet
28     packetizer_encode(p,msg,packet);
29
30     // decode the packet, returning validity
```

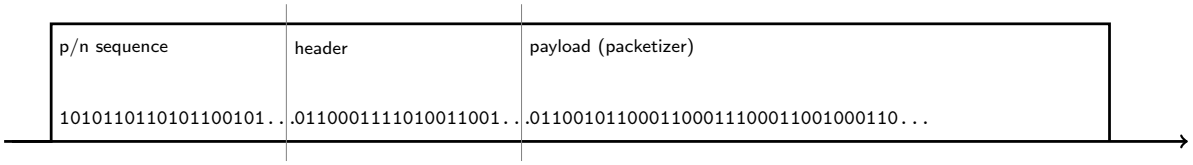


Figure 26: Structure used for the `bpacketgen` and `bpacketsync` objects.

```
31      crc_pass = packetizer_decode(p,packet,msg_dec);
32
33      // destroy the packetizer object
34      packetizer_destroy(p);
35  }
```

See also: fec module, `examples/packetizer_example.c`

15.3 `bpacket` (binary packet generator/synchronizer)

The `bpacketgen` and `bpacketsync` objects realize a pair of binary packet generator and synchronizer objects useful for streaming data applications. The `bpacketgen` object generates packets by encapsulating data using a `packetizer` object but adds a special bit sequence and header to the beginning of the packet. The bit sequence at the beginning of the packet allows the synchronizer to find it using a binary cross-correlator; the header includes information about how the packet is encoded, including the two levels of forward error-correction coding used, the validity check (e.g. cyclic redundancy check), and the length of the payload. The full packet is assembled according to Figure 26.

At the receiver the `bpacketsync` object correlates against the bit sequence looking for the beginning of the packet. It is important to realize that the receiver does not need to be byte-aligned as the packet synchronizer takes care of this internally. Once a packet has been found the packet synchronizer decodes the header to determine how the payload is to be decoded. The payload is decoded and the resulting data is passed to a callback function. The synchronizer compensates for the situation where all the bits are flipped (e.g. coherent BPSK with a phase offset of π radians). Because the packet's header includes information about how to decode the payload the synchronizer automatically reconfigures itself to the packet parameters without any additional specification by the user. This allows great flexibility adapting encoding parameters to dynamic channel environments.

15.3.1 `bpacketgen` interface

The functionality of the `bpacket` structure is split into two objects: the `bpacketgen` object generates the packets and runs on the transmit side of the link while the `bpacketsync` object synchronizes and decodes the packets and runs on the receive side of the link. Listed below is the full interface to the `bpacketgen` object.

`bpacketgen_create(m,n,crc,fec0,fec1)` creates and returns a `bpacketgen` object which accepts n uncoded input bytes and uses the specified CRC and bi-level FEC schemes. The first

parameter (m) is reserved for future development and is currently ignored.

`bpacketgen_recreate(q,m,n,crc,fec0,fec1)` re-creates an existing `bpacketgen` object with new parameters.

`bpacketgen_destroy(q)` destroys an `bpacketgen` object, freeing all internally-allocated memory.

`bpacketgen_print(q)` prints the internal state of the `bpacketgen` object to the standard output.

`bpacketgen_get_packet_len(q)` returns the length in bytes of the fully-encoded packet.

`bpacketgen_encode(q,*msg,*pkt)` encodes the n-byte input message `msg`, storing the result in the encoded output packet `pkt`.

15.3.2 `bpacketsync` interface

As stated before, the `bpacketsync` runs on the receiver to synchronize to and decode the incoming packets. Listed below is the full interface to the `bpacketsync` object.

`bpacketsync_create(m,callback,*userdata)` creates and returns a `bpacketsync` object which invokes a user-defined callback function, passing to it a user-defined object pointer. The first parameter (m) is reserved for future development and is currently ignored.

`bpacketsync_destroy(q)` destroys an `bpacketsync` object, freeing all internally-allocated memory.

`bpacketsync_print(q)` prints the internal state of the `bpacketsync` object to the standard output.

`bpacketsync_reset(q)` resets the internal state of the object.

`bpacketsync_execute(q,*bytes,n)` runs the synchronizer on n bytes of received data.

`bpacketsync_execute_byte(q,byte)` runs the synchronizer on a single byte of received data.

`bpacketsync_execute_sym(q,sym,bps)` runs the synchronizer on a symbol with `bps` bits of information.

`bpacketsync_execute_bit(q,bit)` runs the synchronizer on a single bit.

The `bpacketsync` object has a callback function which has four arguments and looks like this:

```
int bpacketsync_callback(unsigned char *  _payload,
                         int              _payload_valid,
                         unsigned int     _payload_len,
                         void *           _userdata);
```

The callback is typically defined to be `static` and is passed to the instance of `bpacketsync` object when it is created.

`_payload` is a pointer to the decoded bytes of payload data. This pointer is not static and cannot be used after returning from the callback function. This means that it needs to be copied locally for you to retain the data.

_payload_valid is simply a flag to indicate if the payload passed its cyclic redundancy check ("0" means invalid, "1" means valid). If this flag is zero then the payload most likely has errors in it. Some applications are error tolerant and so it is possible that the payload data are still useful. Typically, though, the payload should be discarded and a re-transmission request should be issued.

_payload_len indicates the number of bytes in the _payload argument.

_userdata is a pointer that given to the bpacketsync object when it was created. This pointer is passed to the callback and can represent just about anything. Typically it points to another structure and is the method by which the decoded header and payload data are returned to the program outside of the callback.

15.3.3 Code example

Listed below is a basic example of of the interface to the bpacketgen and bpacketsync objects. For a detailed example program, see examples/bpacketsync_example.c under the main *liquid* project directory.

```
1  // file: doc/listings/bpacket.example.c
2  #include <liquid/liquid.h>
3
4  int callback(unsigned char * _payload,
5               int _payload_valid,
6               unsigned int _payload_len,
7               void * _userdata)
8  {
9      printf("callback invoked\n");
10     return 0;
11 }
12
13 int main() {
14     // options
15     unsigned int n=64;                       // original data message length
16     crc_scheme check = LIQUID_CRC_32;        // data integrity check
17     fec_scheme fec0 = LIQUID_FEC_HAMMING128; // inner code
18     fec_scheme fec1 = LIQUID_FEC_NONE;       // outer code
19
20     // create packet generator and compute packet length
21     bpacketgen pg = bpacketgen_create(0, n, check, fec0, fec1);
22     unsigned int k = bpacketgen_get_packet_len(pg);
23
24     // initialize arrays
25     unsigned char msg_org[n];   // original message
26     unsigned char msg_enc[k];   // encoded message
27     unsigned char msg_dec[n];   // decoded message
28
29     // create packet synchronizer
30     bpacketsync ps = bpacketsync_create(0, callback, NULL);
31
```

```
32      // initialize original data message
33      unsigned int i;
34      for (i=0; i<n; i++) msg_org[i] = rand() % 256;
35
36      // encode packet
37      bpacketgen_encode(pg, msg_org, msg_enc);
38
39      // ... channel ...
40
41      // push packet through synchronizer
42      bpacketsync_execute(ps, msg_enc, k);
43
44      // clean up allocated objects
45      bpacketgen_destroy(pg);
46      bpacketsync_destroy(ps);
47  }
```

15.4 frame64, flexframe (basic framing structures)

liquid comes packaged with two basic framing structures: `frame64` and `flexframe` which can be used with little modification to transmit data over a wireless link. The interface for both of these objects is intended to be as simple as possible while allowing control over some of the parameters of the system. On the transmitter side, the appropriate frame generator object is created, configured, and executed. The receiver side uses an appropriate frame synchronizer object which simply picks packets of a stream of samples, invoking a callback function for each packet it finds. The simplicity of the receiver is that the frame synchronizer object automatically reconfigures itself for packets of different size, modulation scheme, and other parameters.

15.4.1 frame64 description

The `framegen64` and `framesync64` objects implement a basic framing structure for communicating packetized data over the air. The `framegen64` object accepts a 12-byte header and 64-byte payload and assemble a 1280sample frame. Internally, the frame generator encodes the header and payload each with a Hamming(12,8) block code, 16-bit cyclic redundancy check, and modulates the result with a QPSK modem. The header and payload are encapsulated with special phasing sequences, and finally the resulting symbols are interpolated using a half-rate root-raised cosine filter (see section 14.5.2).

The true spectral efficiency of the frame is exactly 4/5; 64 bytes of data (512 bits) encoded into 640 symbols. The `frame64` structure has the advantage of simplicity but lacks the ability for true flexibility.

15.4.2 flexframe description

The `flexframegen` and `flexframesync` objects are similar to their `frame[gen|sync]64` counterparts, however extend functionality to include a number of options in structuring the frame.

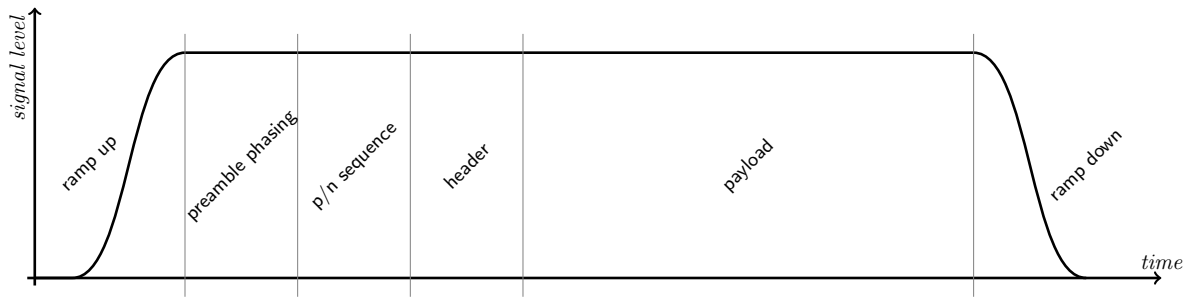


Figure 27: Framing structure used for the `frame64` and `flexframe` objects.

15.4.3 Framing Structures

While the specifics of the `frame64` and `flexframe` structures are different, both frames consist of six basic parts:

ramp/up gracefully increases the output signal level to avoid "key clicking" and reduce spectral side-lobes in the transmitted signal. Furthermore, it allows the receiver's automatic gain control unit to lock on to the incoming signal, preventing sharp transitions in its output.

preamble phasing is a BPSK pattern which flips phase for each transmitted symbol (`+1,-1,+1,-1,...`). This sequence serves several purposes but primarily to help the receiver's symbol synchronization circuit lock onto the proper timing phase. [This works] because the phasing pattern maximizes the number of symbol transitions [reword].

p/n sequence is an m-sequence (see section 22) exhibiting good auto- and cross-correlation properties. This sequence aligns the frame synchronizers to the remainder of the frame, telling them when to start receiving and decoding the frame header, as well as if the phase of the received signal needs to be reversed. At this point, the receiver's AGC, carrier PLL, and timing PLL should all have locked. The p/n sequence is of length 64 for both the `frame64` and `flexframe` structures (63-bit m-sequence with additional padded bit).

header is a fixed-length data sequence which contains a small amount of information about the rest of the frame. The headers for the `frame64` and `flexframe` structures are vastly different and are described independently.

payload is the meat of the frame, containing the raw data to be transferred across the link. For the `frame64` structure, the payload is fixed at 64 bytes (hence its moniker), encoded using the Hamming(12,8) code (section 12), and modulated using QPSK. The `flexframe` structure has a variable length payload and can be modulated using whatever schemes the user desires, however forward error-correction is executed externally. In both cases the synchronizer object invokes the callback upon receiving the payload.

ramp/down gracefully decreases the output signal level as per ramp/up.

A graphical depiction of the framing signal level can be seen in figure 27. The relative lengths of each section are not necessarily to scale, particularly as the `flexframe` structure allows many

of these sections to be variable in length. NOTE: while the `flexframegen` and `flexframesync` objects are intended to be used in conjunction with one another, the output of `flexframegen` requires matched-filtering interpolation before the `flexframesync` object can recover the data.

15.4.4 The Decoding Process

Both the `frame64` and `flexframe` objects operate very similarly in their decoding processes. On the receiver, frames are pulled from a stream of input samples which can exhibit channel impairments such as noise, sample timing offset, and carrier frequency and phase offsets. The receiver corrects for these impairments as best it can using various other signal processing elements in *liquid* and attempts to decode the frame. If at any time a frame is decoded (even if improperly), its appropriate user-defined callback function is invoked. When seeking a frame the synchronizer initially sets its internal loop bandwidths high for acquisition, including those for the automatic gain control, symbol timing recovery, and carrier frequency/phase recovery. This is known as *acquisition* mode, and is typical for packet-based communications systems. Once the p/n sequence has been found, the receiver assumes it has a sufficient lock on the channel impairments and reduces its control loop bandwidths significantly, moving to *tracking* mode.

15.5 `framesyncprops_s` (frame synchronizer properties)

Governing the behavior any frame synchronizer in *liquid* is the `framesyncprops_s` object. In general the frame synchronizer open the bandwidths of their control loops until a certain sequence has been detected; this helps reduce acquisition time of the received signal. After the frame has been detected the control loop bandwidths are reduced to improve stability and reduce the possiblity of losing a lock on the signal. Listed below is a description of the `framesyncprops_` object members.

`agc_bw0/agc_bw1` are the respective open/closed automatic gain control bandwidths. The default values are 10^{-3} and 10^{-5}, respectively.

`agc_gmin/agc_gmax` are the respective maximum/minimum automatic gain control gain values. The default values are 10^{-3} and 10^{4}, respectively.

`sym_bw0/sym_bw1` are the respective open/closed symbol synchronizer bandwidths. The default values are 0.08 and 0.05, respectively.

`pll_bw0/pll_bw1` are the respective open/closed carrier phase-locked loop bandwidths. The default values are 0.02 and 0.005, respectively.

`k` represents the matched filter's samples per symbol; however this parameter is reserved for future development. At present this number should be equal to 2 and should not be changed.

`npfb` represents the number of filters in the internal symbol timing recovery object's polyphase filter bank (see Section 14.12); however this parameter is reserved for future development and should not be changed. The default value is 32.

`m` represents the matched filter's symbol delay; however this parameter is reserved for future development and should not be changed. the default value is 3.

`beta` represents the matched filter's excess bandwidth factor; however this parameter is reserved for future development and should not be changed. the default value is 0.7.

`squelch_enabled` is a flag that specifies if the automatic gain control's squelch is enabled (see Section 7.6). Enabling the squelch (setting `squelch_enabled` equal to `1`) will ignore received signals below the `squelch_threshold` value (see below) to help prevent the reciever's control loops from drifting. Enabling the squelch is usually desirable; however care must be taken to properly set the threshold—ideally about 4 dB above the noise floor—so as not to miss frames with a weak signal. By default the squelch is *disabled*.

`autosquelch_enabled` is a flag that specifies if the automatic gain control's *auto-squelch* is enabled (see Section 7.6.2). In brief, the auto-squelch attempts to track the signal's power to automatically squelch signals 4 dB above the noise floor. By default the auto-squelch is disabled.

`squelch_threshold` is the squelch threshold value in dB (see Section 7.6). The default value is -35.0, but the ideal value is about 4 dB above the noise floor.

`eq_len` specifies the length of the internal equalizer (see Section 11). By default the length is set to zero which disables equalization of the receiver.

`eqrls_lambda` is the recursive least-squares equalizer forgetting factor λ (see Section 11.3). The default value is $\lambda = 0.999$.

15.6 `framesyncstats_s` (frame synchronizer statistics)

When the synchronizer finds a frame and invokes the user-defined callback function, a special structure is passed to the callback that includes some useful information about the frame. This information is contained within the `framesyncstats_s` structure. While useful, the information contained within the structure is not necessary for decoding and can be ignored by the user. Listed below is a description of the `framesyncstats_` object members.

`SNR` is an estimate of the received signal-to-noise ratio in dB. This estimate is in fact the error vector magnitude of the demodulated header. Depending upon the amount of distortion in the signal (due to timing mismatch, inter-symbol interference, etc.) this estimate will not be particularly accurate. In fact a more accurate estimate of the frame SNR is by using the `rssi` property (see below) and subtracting the noise floor.

`rssi` is an estimate of the received signal strength in dB. This is derived from the synchronizer's internal automatic gain control object (see Section 7).

`framesyms` a pointer to an array of the frame symbols (e.g. QPSK) at complex baseband before demodulation. This is useful for plotting purposes. This pointer is not static and cannot be used after returning from the callback function. This means that it needs to be copied locally for you to retain the data.

`num_framesyms` the length of the `framesyms` pointer array.

`mod_scheme` the modulation scheme of the frame (see Section 18).

`mod_bps` the modulation depth (bits per symbol) of the modulation scheme used in the frame.

`check` the error-detection scheme (e.g. cyclic redundancy check) used in the payload of the frame (see Section 12).

`fec0` the inner forward error-correction code used in the payload (see Section 12).

`fec1` the outer forward error-correction code used in the payload (see Section 12).

A simple way to display the information in an instance of `framesyncstats_s` is to use the `framesyncstats_print()` method.

Table 6: Summary of Transcendental Math Interfaces

function	*interface*
$\ln\Gamma(z)$	`liquid_lngammaf(z)`
$\Gamma(z)$	`liquid_gammaf(z)`
$\ln\gamma(z,\alpha)$	`liquid_lnlowergammaf(z,alpha)`
$\gamma(z,\alpha)$	`liquid_lowergammaf(z,alpha)`
$\ln\Gamma(z,\alpha)$	`liquid_lnuppergammaf(z,alpha)`
$\Gamma(z,\alpha)$	`liquid_uppergammaf(z,alpha)`
$n!$	`liquid_factorialf(n)`
$\ln I_\nu(z)$	`liquid_lnbesselif(nu,z)`
$I_\nu(z)$	`liquid_besselif(nu,z)`
$I_0(z)$	`liquid_besseli0f(z)`
$J_\nu(z)$	`liquid_besseljf(nu,z)`
$J_0(z)$	`liquid_bessellj0f(z)`
$Q(z)$	`liquid_Qf(z)`
$Q_M(\alpha,\beta)$	`liquid_MarcumQf(M,alpha,beta)`
$Q_1(\alpha,\beta)$	`liquid_MarcumQ1f(alpha,beta)`
$\text{sinc}(z)$	`liquid_sincf(z)`
$\lceil\log_2(n)\rceil$	`liquid_nextpow2(n)`
$\binom{n}{k}$	`liquid_nchoosek(n,k)`

16 math

The `math` module implements several useful functions for digital signal processing including trascendental function not necessarily in the standard C library, windowing functions, and polynomial manipulation methods.

16.1 Transcendental Functions

This section describes the implementation and interface to transcendental functions not in the C standard library including a full arrangement of Gamma and Bessel functions. Table 6 summarizes the interfaces provided in *liquid*.

16.1.1 `liquid_gammaf(z)`, `liquid_lngammaf(z)`

liquid computes $\Gamma(z)$ from $\ln\Gamma(z)$ (see below) due to its steep, exponential response to z. The complete Gamma function is defined as

$$\Gamma(z) \triangleq \int_0^\infty t^{z-1}e^{-t}dt \tag{72}$$

The upper an lower incomplete Gamma functions are described in Sections 16.1.3 and 16.1.2, respectively. The natural log of the complete Gamma function is computed by splitting into discrete

piecewise sections:

$$\ln\left[\Gamma(z)\right] \approx \begin{cases} \text{undefined} & z < 0 \\ \ln\Gamma(z+1) - \ln(z) & 0 \le z < 10 \\ \frac{z}{2}\ln\left(\frac{2\pi}{z}\right)\left(\ln\left(z + \frac{1}{12z - 0.1/z}\right) - 1\right) & z \ge 0.6 \end{cases} \tag{73}$$

16.1.2 `liquid_lowergammaf(z,a)`, `liquid_lnlowergammaf(z,a)` (lower incomplete Gamma)

Like $\Gamma(z)$, *liquid* computes the lower incomplete gamma function $\gamma(z,\alpha)$ from its logarithm $\ln\gamma(z,\alpha)$ due to its steep, exponential response to z. The lower incomplete Gamma function is defined as

$$\gamma(z,\alpha) \triangleq \int_0^\alpha t^{z-1}e^{-t}dt \tag{74}$$

liquid computes the log of lower incomplete Gamma function as

$$\ln\gamma(z,\alpha) = z\ln(\alpha) + \ln\Gamma(z) - \alpha + \ln\left[\sum_{k=0}^{\infty}\frac{\alpha^k}{\Gamma(z+k+1)}\right] \tag{75}$$

16.1.3 `liquid_uppergammaf(z,a)`, `liquid_lnuppergammaf(z,a)` (upper incomplete Gamma)

Like $\Gamma(z)$, *liquid* computes the upper incomplete gamma function $\Gamma(z,\alpha)$ from $\ln\Gamma(z,\alpha)$ due to its steep, exponential response to z. The complete Gamma function is defined as

$$\Gamma(z,\alpha) \triangleq \int_\alpha^\infty t^{z-1}e^{-t}dt \tag{76}$$

By definition the sum of the lower and upper incomplete gamma functions is the complete Gamma function: $\Gamma(z) = \gamma(z,\alpha) + \Gamma(z,\alpha)$. As such, *liquid* computes the upper incomplete Gamma function as

$$\Gamma(z,\alpha) = \Gamma(z) - \gamma(z,\alpha) \tag{77}$$

16.1.4 `liquid_factorialf(n)`

liquid computes $n! = n\cdot(n-1)\cdot(n-2)\cdots 3\cdot 2\cdot 1$ iteratively for small values of n, and with the Gamma function for larger values. Specifically, $n! = \Gamma(n+1)$.

16.1.5 `liquid_nchoosek()`

computes the binomial coefficient

$$\binom{n}{k} = \frac{n!}{(n-k)!k!} \tag{78}$$

16.1.6 `liquid_nextpow2()`

computes $\lceil\log_2(x)\rceil$

16.1.7 `liquid_sinc(z)`

The sinc function is defined as

$$\text{sinc}(z) = \frac{\sin(\pi z)}{\pi z} \tag{79}$$

Simply evaluating the above equation with finite precision for z results in a discontinuity for small z, and is approximated by expanding the first few terms of the series

$$\text{sinc}(z) = \prod_{k=1}^{\infty} \cos\left(2^{-k}\pi z\right) \tag{80}$$

16.1.8 `liquid_lnbesselif()`, `liquid_besselif()`, `liquid_besseli0f()`

$I_\nu(z)$ is the modified Bessel function of the first kind and is particularly useful for filter design. An iterative method for computing I_ν comes from Gross(1995),

$$I_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{k=0}^{\infty} \frac{\left(\frac{1}{4}z^2\right)^k}{k!\Gamma(k+\nu+1)} \tag{81}$$

Due to its steep response to z it is often useful to compute $I_\nu(z)$ by first computing $\ln I_n u(z)$ as

$$\begin{aligned}
\ln I_\nu(z) &= \nu\ln(z/2) + \ln\left[\sum_{k=0}^{\infty} \frac{\left(\frac{1}{4}z^2\right)^k}{k!\Gamma(\nu+k+1)}\right] \\
&= \nu\ln(z/2) + \ln\left[\sum_{k=0}^{\infty} \exp\Big\{2k\ln(z/2) - \ln\Gamma(k+1) - \ln\Gamma(\nu+k+1)\Big\}\right]
\end{aligned}$$

For $\nu = 0$ a good approximation can be derived by using piecewise polynomials,

$$\ln\Big[\ln\left(I_0(z)\right)\Big] \approx c_0 + c_1 t + c_2 t^2 + c_3 t^3 \tag{82}$$

where $t = \ln(z)$ and

$$\{c_0, c_1, c_2, c_3\} = \begin{cases} \{\text{-1.52624, 1.9597, -9.4287e-03, -7.0471e-04}\} & t < 0.5 \\ \{\text{-1.5531, 1.8936, -0.07972, -0.01333}\} & 0.5 \le t < 2.3 \\ \{\text{-1.2958, 1.7693, -0.1175, 0.006341}\} & \text{else.} \end{cases}$$

This is a particularly useful approximation for the Kaiser window in fixed-point math where $w[n]$ is computed as the ratio of two large numbers.

16.1.9 `liquid_lnbesseljf()`, `liquid_besselj0f()`

$J_\nu(z)$ is the Bessel function of the first kind and is found in Doppler filter design. *liquid* computes $J_\nu(z)$ using the series expansion

$$J_\nu(z) = \sum_{k=0}^{\infty} \frac{(-1)^k}{2^{2k+|v|}k!\left(|v|+k\right)!} z^{2k+|v|} \tag{83}$$

Table 7: Summary of Complex Trigonometric Math Interfaces

function	*interface*
$\sqrt{z}$	`liquid_csqrtf(z)`
e^z	`liquid_cexpf(z)`
$\ln(z)$	`liquid_clogf(z)`
$\sin^{-1}(z)$	`liquid_casinf(z)`
$\cos^{-1}(z)$	`liquid_cacosf(z)`
$\tan^{-1}(z)$	`liquid_catanf(z)`

16.2 Complex Trigonometry

This section describes the implementation and interface to complex trigonometric functions not in the C standard library. Table 6 summarizes the interfaces provided in *liquid*.

16.2.1 `liquid_csqrtf()`

The function `liquid_csqrtf(z)` computes the complex square root of a number

$$\sqrt{z} = \sqrt{\frac{r+a}{2}} + j\text{sgn}\big(\Im\{z\}\big)\sqrt{\frac{r-a}{2}} \tag{84}$$

where $r = |z|$, $a = \Re\{z\}$, and $\text{sgn}(t) = t/|t|$.

16.2.2 `liquid_cexpf()`

The function `liquid_cexpf(z)` computes the complex exponential of a number

$$e^z = \exp\big\{a\big\}\big(\cos(b) + j\sin(b)\big) \tag{85}$$

where $a = \Re\{z\}$ and $b = \Im\{z\}$.

16.2.3 `liquid_clogf()`

The function `liquid_clogf(z)` computes the complex natural logarithm of a number.

$$\log(z) = \log(|z|) + j\arg(z) \tag{86}$$

16.2.4 `liquid_cacosf()`

The function `liquid_cacosf(z)` computes the complex arccos of a number

$$\arccos(z) = \begin{cases} -j\log\big(z + \sqrt{z^2-1}\big) & \text{sgn}\big(\Re\{z\}\big) = \text{sgn}\big(\Im\{z\}\big) \\ -j\log\big(z - \sqrt{z^2-1}\big) & \text{otherwise} \end{cases} \tag{87}$$

16.2.5 `liquid_casinf()`

The function `liquid_casinf(z)` computes the complex arcsin of a number

$$\arcsin(z) = \frac{\pi}{2} - \arccos(z) \tag{88}$$

16.2.6 `liquid_catanf()`

The function `liquid_catanf(z)` computes the complex arctan of a number

$$\arctan(z) = \frac{j}{2}\log\left(\frac{1-jz}{1+jz}\right) \tag{89}$$

16.3 Windowing functions

This section describes the various windowing functions in the `math` module. These windowing functions are useful for spectral approximation as they are compact in both the time and frequency domains.

16.3.1 `hamming()`, (Hamming window)

The function `hamming(n,N)` computes the n^{th} of N indices of the Hamming window:

$$w(n) = 0.53836 - 0.46164\cos\left(2\pi n/(N-1)\right) \tag{90}$$

16.3.2 `hann()`, (Hann window)

The function `hann(n,N)` computes the n^{th} of N indices of the Hann window:

$$w(n) = 0.5 - 0.5\cos\left(2\pi n/(N-1)\right) \tag{91}$$

16.3.3 `blackmanharris()`, (Blackman-harris window)

The function `blackmanharris(n,N)` computes the n^{th} of N indices of the Blackman-harris window:

$$w(n) = \sum_{k=0}^{3} a_k \cos\left(2\pi k n/(N-1)\right) \tag{92}$$

where $a_0 = 0.35875$, $a_1 = -0.48829$, $a_2 = 0.14128$, and $a_3 = -0.01168$.

16.3.4 `kaiser()`, (Kaiser-Bessel window)

The function `kaiser(n,N,dt,beta)` computes the n^{th} of N indices of the Kaiser-β window with a shape parameter β:

$$w(n,\beta) = \frac{I_0\left(\pi\beta\sqrt{1-\left(\frac{n}{N/2}\right)^2}\right)}{I_0\left(\pi\beta\right)} \tag{93}$$

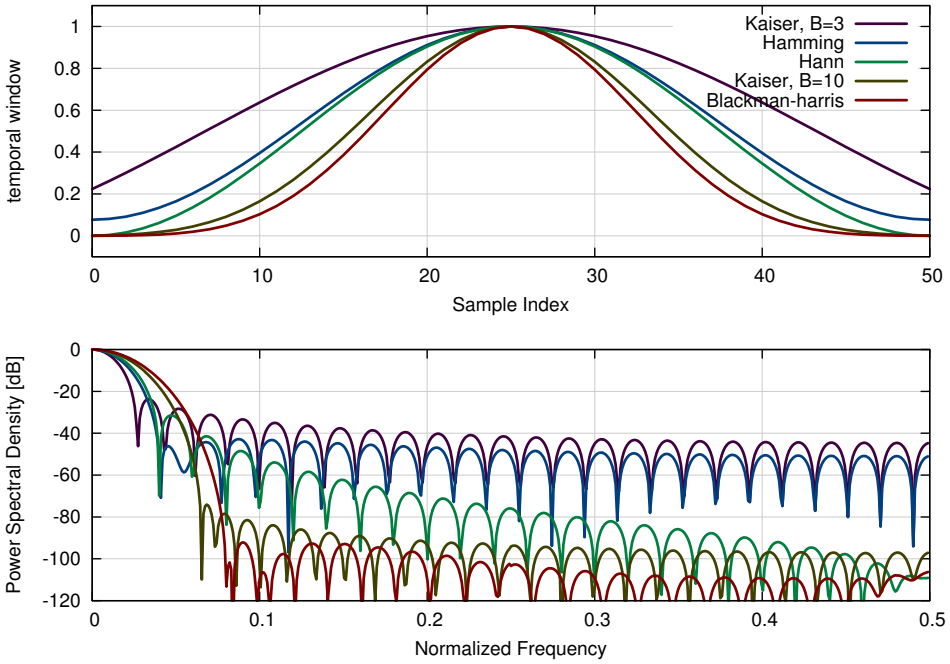


Figure 28: windowing functions, power spectral density

where $I_\nu(z)$ is the modified Bessel function of the first kind of order ν, and β is a parameter controlling the width of the window and its stop-band attenuation. In *liquid*, $I_0(z)$ is computed using `liquid_besseli0f()` (see Section 16.1). A fractional sample offset Δt can be introduced by substituting $\frac{n}{N/2}$ with $\frac{n}{N/2} + \Delta t$ in (93).

16.3.5 `liquid_kbd_window()`, (Kaiser-Bessel derived window)

The function `liquid_kbd_window(n,beta,*w)` computes the n-point Kaiser-Bessel derived window with a shape parameter β storing the result in the n-point array `w`. The length of the window *must* be even.

16.3.6 Examples

Temporal and spectral examples of all these windows can be seen in Figure 28. Notice that as the power spectral density main lobe increases in width, its stop-band attenuation decreases. Furthermore, windows with wider temporal widths exhibit more narrow spectral main lobes, but have less stop-band attenuation.

16.4 Polynomials

A number of *liquid* modules require polynomial manipulations, particularly those involving filter design where transfer functions are represented as the explicit ratio of polynomials in z^{-1}. This sub-module is not intended to be complete, but rather is required for the proper functionality of

other modules. Like matrices, polynomials in *liquid* do not use a particular data type, but are stored as memory arrays.

$$P_n(x) = \sum_{k=0}^{n} c_k x^k = c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n \tag{94}$$

An n^{th}-order polynomial has $n+1$ coefficients ordered in memory in increasing degree.[7] For example, a 2^{nd}-order polynomial $0.1 - 2.4x + 1.3x^2$ stored in an array `float c[]` has `c[0]=0.1`, `c[1]=-2.4`, and `c[2]=1.3`.

Notice that all routines for the type *float* are prefaced with `polyf`. This follows the naming convention of the standard C library routines which append an `f` to the end of methods operating on floating-point precision types. Similar matrix interfaces exist in *liquid* for *double* (`poly`), *double complex* (`polyc`), and *float complex* (`polycf`).

16.4.1 `polyf_val()`

The `polyf_val(*p,k,x)` method evaluates the polynomial $P_n(x)$ at x_0 where the `k` coefficients are stored in the input array `p`. Here is a brief example which evaluates $P_2(x) = 0.2 + 1.0x + 0.4x^2$ at $x = 1.3$:

```
float p[3] = {0.2f, 1.0f, 0.4f};
float x = 1.3f;
float y = polyf_val(p,3,x);
>>> y = 2.17599988
```

16.4.2 `polyf_fit()`

The `polyf_fit(*x,*y,n,*p,k)` method fits data to a polynomial of order $k-1$ from n samples using the least-squares method on the input data vectors $\boldsymbol{x} = [x_0, x_1, \cdots, x_{n-1}]^T$ and $\boldsymbol{y} = [y_0, y_1, \cdots, y_{n-1}]^T$. Internally *liquid* uses matrix algebra to solve the system of equations

$$\boldsymbol{p} = \left(\boldsymbol{X}^T \boldsymbol{X}\right)^{-1} \boldsymbol{X}^T \boldsymbol{y} \tag{95}$$

where

$$\boldsymbol{X} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^k \\ 1 & x_1 & x_1^2 & \cdots & x_1^k \\ & & & & \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^k \end{bmatrix} \tag{96}$$

For example this script fits the 4 data samples to a linear (first-order, two coefficients) polynomial:

```
float x[4] = {0.0f,  1.0f,  2.0f,  3.0f};
float y[4] = {0.85f, 3.07f, 5.07f, 7.16f};
float p[2];
polyf_fit(x,y,4,p,2);
>>> p = {  0.89800072,   2.09299946}
```

[7] Note that this convention is reversed from that used in octave [7].

16.4.3 polyf_fit_lagrange()

The `polyf_fit_lagrange(*x,*y,n,*p)` method fit a dataset of n sample points to exact polynomial of order $n-1$ using Lagrange interpolation. Given input vectors $\boldsymbol{x} = [x_0, x_1, \cdots, x_{n-1}]^T$ and $\boldsymbol{y} = [y_0, y_1, \cdots, y_{n-1}]^T$, the interpolating polynomial is

$$P_{n-1}(x) = \sum_{j=0}^{n-1} \left[y_j \prod_{\substack{k=0 \\ k \neq j}}^{n-1} \frac{x - x_k}{x_j - x_k} \right] \tag{97}$$

For example this script fits the 4 data samples to a cubic (third-order, four coefficients) polynomial:

```
float x[4] = {0.0f,  1.0f,  2.0f,  3.0f};
float y[4] = {0.85f, 3.07f, 5.07f, 7.16f};
float p[4];
polyf_fit_lagrange(x,y,4,p);
>>> p = {  0.85000002,   2.43333268,  -0.26499939,   0.05166650}
```

Notice that `polyf_fit_lagrange(x,y,n,p)` is mathematically equivalent to `polyf_fit(x,y,n,p,n)`, but is computed in fewer steps. See also `polyf_expandroots`.

16.4.4 polyf_interp_lagrange()

The `polyf_interp_lagrange(*x,*y,n,x0)` method uses Lagrange polynomials to find the interpolant $(\dot{x}, \dot{y})$ from a set of n pairs $\boldsymbol{x} = [x_0, x_1, \cdots, x_{n-1}]^T$ and $\boldsymbol{y} = [y_0, y_1, \cdots, y_{n-1}]^T$.

$$\dot{y} = \sum_{j=0}^{n-1} \left[y_j \prod_{\substack{k=0 \\ k \neq j}}^{n-1} \frac{\dot{x} - x_k}{x_j - x_k} \right] \tag{98}$$

For example this script interpolates between the 4 data points

```
float x[4] = {0.0f,  1.0f,  2.0f,  3.0f};
float y[4] = {0.85f, 3.07f, 5.07f, 7.16f};
float x0 = 0.5f;
float y0 = polyf_interp_lagrange(x,y,4,x0);
>>> y0 =   2.00687504
```

See also `polyf_fit_lagrange()`.

16.4.5 polyf_fit_lagrange_barycentric()

The `polyf_fit_lagrange_barycentric(*x,n,*w)` method computes the barycentric weights $\boldsymbol{w}$ of $\boldsymbol{x}$ via

$$w_j = \frac{1}{\prod_{k \neq j} (x_j - x_k)} \tag{99}$$

which can be used to compute the interpolant $(\dot{x}, \dot{y})$ with fewer computations.

```
float x[4] = {0.0f,  1.0f,  2.0f,  3.0f};
float w[4];
polyf_fit_lagrange_barycentric(x,4,w);
>>> w = {  1.00000000,  -3.00000000,   3.00000000,  -1.00000000}
```

16.4.6 polyf_val_lagrange_barycentric()

The `polyf_val_lagrange_barycentric(*x,*y,*w,x0,n)` method computes the interpolant $(\dot{x}, \dot{y})$ given the barycentric weights $\boldsymbol{w}$ (defined above) as

$$\dot{y} = \frac{\sum_{j=0}^{k-1} w_j y_j / (\dot{x} - x_j)}{\sum_{j=0}^{k-1} w_j / (\dot{x} - x_j)} \tag{100}$$

This is the preferred method for computing Lagrange interpolating polynomials, particularly if $\boldsymbol{x}$ is unchanging. The function returns $\dot{y}$ if $\dot{x}$ is equal to any x_j.

```
float x[4] = {0.0f,  1.0f,  2.0f,  3.0f};
float y[4] = {0.85f, 3.07f, 5.07f, 7.16f};
float w[4];
polyf_fit_lagrange_barycentric(x,4,w);
float x0 = 0.5f;
float y0 = polyf_val_lagrange_barycentric(x,y,w,x0,4);
>>> y0 =   2.00687504
```

Lagrange polynomials of the barycentric form are used heavily in *liquid*'s implementation of the Parks-McClellan algorithm (`firdespm`) for filter design (see Section 14.5.3).

16.4.7 polyf_expandbinomial()

The `polyf_expandbinomial(n,*p)` method expands the a polynomial as a binomial series

$$P_n(x) = (x+1)^n = \sum_{k=0}^{n} \binom{n}{k} x^k \tag{101}$$

For example the following script will compute $P_3(x) = (1+x)^3$:

```
float p[4];
polyf_expandbinomial(3,p);
>>> p = {  1.00000000,   3.00000000,   3.00000000,   1.00000000}
```

16.4.8 polyf_expandbinomial_pm()

Expands the a polynomial as an alternating binomial series

$$P_n(x) = (x+1)^m (x-1)^{n-m} = \left\{ \sum_{k=0}^{m} \binom{n}{k} x^k \right\} \left\{ \sum_{k=0}^{n-m} \binom{n}{k} (-x)^k \right\} \tag{102}$$

For example the following script will compute $P_3(x) = (1+x)^2(1-x)$:

```
float p[4];
polyf_expandbinomial_pm(2,1,p);
>>> p = {  1.00000000,   1.00000000,  -1.00000000,  -1.00000000}
```

16.4.9 polyf_expandroots()

The `polyf_expandroots(*r,n,*p)` method expands the a polynomial based on its roots

$$P_n(x) = \prod_{k=0}^{n-1} (x - r_k) \tag{103}$$

where r_k are the roots of $P_n(x)$. For example, this script will expand the polynomial $P_3(x) = (x-1)(x+2)(x-3)$ which has roots $\{1, -2, 3\}$:

```
float roots[3] = {1.0f, -2.0f, 3.0f};
float p[4];
polyf_expandroots(roots,3,p);
>>> p = {  6.00000000,  -5.00000000,  -2.00000000,   1.00000000}
```

16.4.10 polyf_expandroots2()

The `polyf_expandroots2(*a,*b,n,*p)` method expands the a polynomial as

$$P_n(x) = \prod_{k=0}^{n-1} (b_k x - a_k) \tag{104}$$

by first factoring out the b_k terms, invoking `polyf_expandroots()`, and multiplying the result by $\prod_k b_k$. For example, this script will expand the polynomial $P_3(x) = (2x-1)(-3x+2)(-x-3)$:

```
float b[3] = { 2.0f, -3.0f, -1.0f};
float a[3] = { 1.0f, -2.0f,  3.0f};
float p[4];
polyf_expandroots2(b,a,3,p);
>>> p = {  6.00000000,  11.00000000, -19.00000000,   6.00000000}
```

16.4.11 polyf_findroots()

The `polyf_findroots(*p,n,*r)` method finds the n roots of the n^{th}-order polynomial using Bairstow's method. For an n^{th}-order polynomial $P_n(x)$ given by

$$P_n(x) = \prod_{k=0}^{n-1} (x - r_k) \tag{105}$$

there exists at least one quadratic polynomial $p_2(x) = u + vx + x^2$ which exactly divides $P_n(x)$ and has two roots (possibly complex)

$$r_0 = \frac{1}{2}\left(-v - \sqrt{v^2 - 4u}\right), r_1 = \frac{1}{2}\left(-v + \sqrt{v^2 - 4u}\right) \tag{106}$$

If indeed the roots r_0 and r_1 are complex, they are also complex conjugates. Bairstow's method uses Newtonian iterations to find a pair u and v which are both finite and real-valued. This method has several advantages over other methods

- iterations operate on real-valued math, even if the roots are complex

- the algorithm is capable of handling multiple roots (unlike the Durand-Kerner method), i.e. $P_n(x) = (x-2)(x-2)(x-2)\cdots$
- the algorithm does not rely on expanding the full polynomial and is therefore resilient to machine precision

Each iteration of Bairstow's algorithm reduces the original polynomial order by two, eventually collapsing the polynomial. The initial choice of u and v determine both algorithm convergence and speed.

liquid implements Bairstow's method with the `polyf_findroots()` function which accepts an n^{th}-order polynomial in standard expanded form and computes its n roots. The last term of the polynomial (highest order) cannot be zero, otherwise the algorithm will not converge.

16.4.12 `polyf_mul()`

The `polyf_mul(*P,n,*Q,m,*S)` method multiplies two polynomials $P_n(x)$ and $Q_m(x)$ to produce the resulting polynomial $S_{n+m-1}(x)$

17 matrix

Matrices are used for solving linear systems of equations and are used extensively in polynomial fitting, adaptive equalization, and filter design. In *liquid*, matrices are represented as just arrays of a single dimension, and do not rely on special objects for their manipulation. This is to help portability of the code and ease of integration into other libraries. Here is a simple example of the matrix interface:

```
1  // file: doc/listings/matrix.example.c
2  #include <liquid/liquid.h>
3
4  int main() {
5      // designate X as a 4 x 4 matrix
6      float X[16] = {
7         0.84382,  -2.38304,   1.43061,  -1.66604,
8         3.99475,   0.88066,   4.69373,   0.44563,
9         7.28072,  -2.06608,   0.67074,   9.80657,
10        6.07741,  -3.93099,   1.22826,  -0.42142};
11     matrixf_print(X,4,4);
12
13     // L/U decomp (Doolittle's method)
14     float L[16], U[16], P[16];
15     matrixf_ludecomp_doolittle(X,4,4,L,U,P);
16 }
```

Notice that all routines for the type *float* are prefaced with `matrixf`. This follows the naming convention of the standard C library routines which append an `f` to the end of methods operating on floating-point precision types. Similar matrix interfaces exist in *liquid* for *double* (`matrix`), *double complex* (`matrixc`), and *float complex* (`matrixcf`).

17.1 Basic math operations

This section describes the basic matrix math operations, including addition, subtraction, point-wise multiplication and division, transposition, and initializing the identity matrix.

17.1.1 `matrix_access` (access element)

Because matrices in *liquid* are really just one-dimensional arrays, indexing matrix values for storage or retrieval is as straighforward as indexing the array itself. *liquid* also provides a simple macro for ensuring the proper value is returned. `matrix_access(X,R,C,r,c)` will access the element of a $R \times C$ matrix $\boldsymbol{X}$ at row r and column c. This method is really just a pre-processor macro which performs a literal string replacement

```
#define matrix_access(X,R,C,r,c) ((X)[(r)*(C)+(c)])
```

and can be used for both setting and retrieving values of a matrix. For example,

```
X =
  0.406911   0.118444   0.923281   0.827254   0.463265
  0.038897   0.132381   0.061137   0.880045   0.570341
```

```
  0.151206   0.439508   0.695207   0.215935   0.999683
  0.808384   0.601597   0.149171   0.975722   0.205819

float v = matrix_access(X,4,5,0,1);
v =
  0.118444

matrix_access(X,4,5,2,3) = 0;
X =
  0.406911   0.118444   0.923281   0.827254   0.463265
  0.038897   0.132381   0.061137   0.880045   0.570341
  0.151206   0.439508   0.695207   0.0        0.999683
  0.808384   0.601597   0.149171   0.975722   0.205819
```

Because this method is really just a macro, there is no error-checking to ensure that one is accessing the matrix within its memory bounds. Therefore, special care must be taken when programming. Furthermore, `matrix_access()` can be used for all matrix types (`matrixf`, `matrixcf`, etc.).

17.1.2 `matrixf_add`, `matrixf_sub`, `matrixf_pmul`, and `matrixf_pdiv` (scalar math operations)

The `matrixf_add(*x,*y,*z,m,n)`, `matrixf_sub(*x,*y,*z,m,n)`, `matrixf_pmul(*x,*y,*z,m,n)`, and `matrixf_pdiv(*x,*y,*z,m,n)` methods perform point-wise (scalar) addition, subtraction, multiplication, and division of the elements of two $n \times m$ matrices, $\boldsymbol{X}$ and $\boldsymbol{Y}$. That is, $\boldsymbol{Z}_{i,k} = \boldsymbol{X}_{i,k} + \boldsymbol{Y}_{i,k}$ for all i, k. The same holds true for subtraction, multiplication, and division. It is very important to understand the difference between the methods `matrixf_pmul()` and `matrixf_mul()`, as well as `matrixf_pdiv()` and `matrixf_div()`. In each case the latter performs a vastly different operation from `matrixf_mul()` and `matrixf_div()` (see Sections 17.2.3 and 17.3.2, respectively).

```
X =                         Y =
  0.59027   0.83429           0.764108   0.741641
  0.67779   0.19793           0.660932   0.041723
  0.95075   0.33980           0.972282   0.347090

matrixf_pmul(X,Y,Z,2,3);
Z =
  0.4510300   0.6187437
  0.4479731   0.0082582
  0.9243971   0.1179412
```

17.1.3 `matrixf_trans()`, `matrixf_hermitian()` (transpose matrix)

The `matrixf_trans(X,m,n,XT)` method performs the conjugate matrix transpose operation on an $m \times n$ matrix $\boldsymbol{X}$. That is, the matrix is flipped on its main diagonal and the conjugate of each element is taken. Formally, $\boldsymbol{A}^T_{i,j} = \boldsymbol{A}^*_{j,i}$. Here's a simple example:

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix}^T = \begin{bmatrix} 0 & 3 \\ 1 & 4 \\ 2 & 5 \end{bmatrix}$$

Similarly, the `matrixf_hermitian(X,m,n,XH)` computes the Hermitian transpose which is identical to the regular transpose but without the conjugation operation, viz $\boldsymbol{A}^H_{i,j} = \boldsymbol{A}_{j,i}$.

17.1.4 `matrixf_eye()` (identity matrix)

The `matrixf_eye(*x,n)` method generates the $n \times n$ identity matrix $\boldsymbol{I}_n$:

$$\boldsymbol{I}_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ & & & \\ 0 & 0 & \cdots & 1 \end{bmatrix} \tag{107}$$

17.2 Elementary math operations

This section describes elementary math operations for linear systems of equations.

17.2.1 `matrixf_swaprows()` (swap rows)

Matrix row-swapping is often necessary to express a matrix in its row-reduced echelon form. The `matrixf_swaprows(*X,m,n,i,j)` method simply swaps rows i and j of an $m \times n$ matrix $\boldsymbol{X}$, viz

```
x =
  0.84381998 -2.38303995  1.43060994 -1.66603994
  3.99475002  0.88066000  4.69372988  0.44563001
  7.28072023 -2.06608009  0.67074001  9.80657005
  6.07741022 -3.93098998  1.22826004 -0.42142001

matrixf_swaprows(x,4,4,0,2);
  7.28072023 -2.06608009  0.67074001  9.80657005
  3.99475002  0.88066000  4.69372988  0.44563001
  0.84381998 -2.38303995  1.43060994 -1.66603994
  6.07741022 -3.93098998  1.22826004 -0.42142001
```

17.2.2 `matrixf_pivot()` (pivoting)

[NOTE: terminology for "pivot" is different from literature.] Given an $n \times m$ matrix $\boldsymbol{A}$,

$$\boldsymbol{A} = \begin{bmatrix} A_{0,0} & A_{0,1} & \cdots & A_{0,m-1} \\ A_{1,0} & A_{1,1} & \cdots & A_{1,m-1} \\ & & & \\ A_{n-1,0} & A_{n-1,1} & \cdots & A_{n-1,m-1} \end{bmatrix}$$

pivoting $\boldsymbol{A}$ around $\boldsymbol{A}_{a,b}$ gives

$$\boldsymbol{B}_{i,j} = \left(\frac{\boldsymbol{A}_{i,b}}{\boldsymbol{A}_{a,b}} \right) \boldsymbol{A}_{a,j} - \boldsymbol{A}_{i,j} \forall i \neq a$$

The pivot element must not be zero. Row a is left unchanged in $\boldsymbol{B}$. All elements of $\boldsymbol{B}$ in column b are zero except for row a. This is accomplished in *liquid* with the `matrixf_pivot(*A,m,n,i,j)` method. For our example 4×4 matrix $\boldsymbol{x}$, pivoting around $\boldsymbol{x}_{1,2}$ gives:

```
matrixf_pivot(x,4,4,1,2);
  0.37374675  2.65145779  0.00000000  1.80186427
  3.99475002  0.88066000  4.69372988  0.44563001
 -6.70986557  2.19192743  0.00000000 -9.74288940
 -5.03205967  4.16144180  0.00000000  0.53803295
```

17.2.3 `matrixf_mul()` (multiplication)

Multiplication of two input matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ is accomplished with the `matrixf_mul(*A,ma,na,*B,mb,nb,*C,mc,n` method, and is not to be confused with `matrixf_pmul()` in Section 17.1.2. If $\boldsymbol{A}$ is $m \times n$ and $\boldsymbol{B}$ is $n \times p$, then their product is computed as

$$\left(\boldsymbol{AB}\right)_{i,j} = \sum_{r=0}^{n-1} \boldsymbol{A}_{i,r}\boldsymbol{B}_{r,j} \tag{108}$$

Note that the number of columns of $\boldsymbol{A}$ must be equal to the number of rows of $\boldsymbol{B}$, and that the resulting matrix is of size $m \times p$ (the number of rows in $\boldsymbol{A}$ and columns in $\boldsymbol{B}$).

```
A =                 B =
    1   2   3           1   2   3
    4   5   6           4   5   6
                        7   8   9
matrixf_mul(A,2,3, B,3,3, C,2,3);

C =
    30  36  42
    66  81  96
```

17.2.4 Transpose multiplication

liquid also implements transpose-multiplication operations on an $m \times n$ matrix $\boldsymbol{X}$, commonly used in signal processing. Section 17.1.3 describes the difference between the $(\cdot)^T$ and $(\cdot)^H$ operations. The interface for transpose-multiplications in *liquid* is tabulated below for an input $m \times n$ matrix $\boldsymbol{X}$.

operation	*output dimensions*	*interface*
$\boldsymbol{X}\boldsymbol{X}^T$	$m \times m$	`matrixcf_mul_transpose(x,m,n,xxT)`
$\boldsymbol{X}\boldsymbol{X}^H$	$m \times m$	`matrixcf_mul_hermitian(x,m,n,xxH)`
$\boldsymbol{X}^T\boldsymbol{X}$	$n \times n$	`matrixcf_transpose_mul(x,m,n,xTx)`
$\boldsymbol{X}^H\boldsymbol{X}$	$n \times n$	`matrixcf_transpose_mul(x,m,n,xHx)`

17.3 Complex math operations

More complex math operations are described here, including matrix inversion, square matrix determinant, Gauss-Jordan elimination, and lower/upper decomposition routines using both Crout's and Doolittle's methods.

17.3.1 `matrixf_inv` (inverse)

Matrix inversion is accomplished with the `matrixf_inv(*X,m,n)` method.[8] Given an $n \times n$ matrix $\boldsymbol{A}$, *liquid* augments with $\boldsymbol{I}_n$:

$$[\boldsymbol{A}|\boldsymbol{I}_n] = \left[\begin{array}{cccc|cccc} A_{0,0} & A_{0,1} & \cdots & A_{0,m-1} & 1 & 0 & \cdots & 0 \\ A_{1,0} & A_{1,1} & \cdots & A_{1,m-1} & 0 & 1 & \cdots & 0 \\ & & & & & & & \\ A_{n-1,0} & A_{n-1,1} & \cdots & A_{n-1,m-1} & 0 & 0 & \cdots & 1 \end{array}\right]$$

Next *liquid* performs elementary operations to convert to its row-reduced echelon form. The resulting matrix has the identity matrix on the left and $\boldsymbol{A}^{-1}$ on its right, viz

$$\left[\boldsymbol{I}_n|\boldsymbol{A}^{-1}\right] = \left[\begin{array}{cccc|cccc} 1 & 0 & \cdots & 0 & A^{-1}_{0,0} & A^{-1}_{0,1} & \cdots & A^{-1}_{0,m-1} \\ 0 & 1 & \cdots & 0 & A^{-1}_{1,0} & A^{-1}_{1,1} & \cdots & A^{-1}_{1,m-1} \\ & & & & & & & \\ 0 & 0 & \cdots & 1 & A^{-1}_{n-1,0} & A^{-1}_{n-1,1} & \cdots & A^{-1}_{n-1,m-1} \end{array}\right]$$

The `matrixf_inv()` method uses Gauss-Jordan elmination (see `matrixf_gjelim()`) for row reduction and back-substitution. Pivot elements in $\boldsymbol{A}$ with the largest magnitude are chosen to help stability in floating-point arithmetic.

```
matrixf_inv(x,4,4);
 -0.33453920  0.04643385 -0.04868321  0.23879384
 -0.42204019  0.12152659 -0.07431178  0.06774280
  0.35104612  0.15256262  0.04403552 -0.20177667
  0.13544561 -0.01930523  0.11944833 -0.14921521
```

17.3.2 `matrixf_div()`

The `matrixf_div(*X,*Y,*Z,*n)` method simply computes $\boldsymbol{Z} = \boldsymbol{Y}^{-1}\boldsymbol{X}$ where $\boldsymbol{X}$, $\boldsymbol{Y}$, and $\boldsymbol{Z}$ are all $n \times n$ matrices.

17.3.3 `matrixf_linsolve()` (solve linear system of equations)

The `matrixf_linsolve(*A,n,*b,*x,opts)` method solves a set of n linear equations $A\boldsymbol{x} = \boldsymbol{b}$ where A is an $n \times n$ matrix, and $\boldsymbol{x}$ and $\boldsymbol{b}$ are $n \times 1$ vectors. The `opts` argument is reserved for future development and can be ignored by setting to `NULL`.

17.3.4 `matrixf_det()` (determinant)

The `matrixf_det(*X,m,n)` method computes the determinant of an $n \times n$ matrix $\boldsymbol{X}$. In *liquid*, the determinant is computed by L/U decomposition of $\boldsymbol{A}$ using Doolittle's method (see `matrixf_ludecomp_doolittle`) and then computing the product of the diagonal elements of $\boldsymbol{U}$, viz

$$\det(\boldsymbol{A}) = |\boldsymbol{A}| = \prod_{k=0}^{n-1} \boldsymbol{U}_{k,k}$$

[8] While matrix inversion requires a square matrix, *liquid* internally checks to ensure $m = n$ on the input size for $\boldsymbol{X}$.

This is equivalent to performing L/U decomposition using Crout's method and then computing the product of the diagonal elements of $\boldsymbol{L}$.

```
matrixf_det(X,4,4) = 585.40289307
```

17.3.5 `matrixf_ludecomp_crout()` (LU Decomposition, Crout's Method)

Crout's method decomposes a non-singular $n \times n$ matrix $\boldsymbol{A}$ into a product of a lower triangular $n \times n$ matrix $\boldsymbol{L}$ and an upper triangular $n \times n$ matrix $\boldsymbol{U}$. In fact, $\boldsymbol{U}$ is a unit upper triangular matrix (its values along the diagonal are 1). The `matrixf_ludecomp_crout(*A,m,n,*L,*U,*P)` implements Crout's method.

$$\boldsymbol{L}_{i,k} = \boldsymbol{A}_{i,k} - \sum_{t=0}^{k-1} \boldsymbol{L}_{i,t}\boldsymbol{U}_{t,k} \; \forall k \in \{0, n-1\}, i \in \{k, n-1\}$$

$$\boldsymbol{U}_{k,j} = \left[\boldsymbol{A}_{k,j} - \sum_{t=0}^{k-1} \boldsymbol{L}_{k,t}\boldsymbol{U}_{t,j}\right] / \boldsymbol{L}_{k,k} \; \forall k \in \{0, n-1\}, j \in \{k+1, n-1\}$$

```
matrixf_ludecomp_crout(X,4,4,L,U,P)
L =
  0.84381998  0.00000000  0.00000000  0.00000000
  3.99475002 12.16227055  0.00000000  0.00000000
  7.28072023 18.49547005 -8.51144791  0.00000000
  6.07741022 13.23228073 -6.81350422 -6.70173073
U =
  1.00000000 -2.82410932  1.69539714 -1.97440207
  0.00000000  1.00000000 -0.17093502  0.68514121
  0.00000000  0.00000000  1.00000000 -1.35225296
  0.00000000  0.00000000  0.00000000  1.00000000
```

17.3.6 `matrixf_ludecomp_doolittle()` (LU Decomposition, Doolittle's Method)

Doolittle's method is similar to Crout's except it is the lower triangular matrix that is left with ones on the diagonal. The update algorithm is similar to Crout's but with a slight variation: the upper triangular matrix is computed first. The `matrixf_ludecomp_doolittle(*A,m,n,*L,*U,*P)` implements Doolittle's method.

$$\boldsymbol{U}_{k,j} = \boldsymbol{A}_{k,j} - \sum_{t=0}^{k-1} \boldsymbol{L}_{k,t}\boldsymbol{U}_{t,j} \; \forall k \in \{0, n-1\}, j \in \{k, n-1\}$$

$$\boldsymbol{L}_{i,k} = \left[\boldsymbol{A}_{i,k} - \sum_{t=0}^{k-1} \boldsymbol{L}_{i,t}\boldsymbol{U}_{t,k}\right] / \boldsymbol{U}_{k,k} \; \forall k \in \{0, n-1\}, i \in \{k+1, n-1\}$$

Here is a simple example:

```
matrixf_ludecomp_doolittle(X,4,4,L,U,P)
L =
  1.00000000  0.00000000  0.00000000  0.00000000
```

```
  4.73412609  1.00000000  0.00000000  0.00000000
  8.62828636  1.52072513  1.00000000  0.00000000
  7.20225906  1.08797777  0.80051047  1.00000000
U =
  0.84381998 -2.38303995  1.43060994 -1.66603994
  0.00000000 12.16227150 -2.07895803  8.33287334
  0.00000000  0.00000000 -8.51144791 11.50963116
  0.00000000  0.00000000  0.00000000 -6.70172977
```

17.3.7 `matrixf_qrdecomp_gramschmidt()` (QR Decomposition, Gram-Schmidt algorithm)

liquid implements Q/R decomposition with the `matrixf_qrdecomp_gramschmidt(*A,m,n,*Q,*R)` method which factors a non-singular $n \times n$ matrix $\boldsymbol{A}$ into product of an orthogonal matrix $\boldsymbol{Q}$ and an upper triangular matrix $\boldsymbol{R}$, each $n \times n$. That is, $\boldsymbol{A} = \boldsymbol{QR}$ where $\boldsymbol{Q}^T\boldsymbol{Q} = \boldsymbol{I}_n$ and $\boldsymbol{R}_{i,j} = 0 \; \forall_{i>j}$. Building on the previous example for our test 4×4 matrix $\boldsymbol{X}$, the Q/R factorization is

```
matrixf_qrdecomp_gramschmidt(X,4,4,Q,R)
Q =
  0.08172275 -0.57793844  0.57207584  0.57622749
  0.38688579  0.63226062  0.66619849 -0.08213031
  0.70512730  0.13563085 -0.47556636  0.50816941
  0.58858842 -0.49783322  0.05239720 -0.63480729
R =
 10.32539940 -3.62461853  3.12874746  6.70309162
  0.00000000  3.61081028  1.62036073  2.78449297
  0.00000000  0.00000000  3.69074893 -5.34197950
  0.00000000  0.00000000  0.00000000  4.25430155
```

17.3.8 `matrixf_gjelim()` (Gauss-Jordan Elimination)

The `matrixf_gjelim(*X,m,n)` method in *liquid* performs the Gauss-Jordan elimination on a matrix $\boldsymbol{X}$. Gauss-Jordan elimination converts a $m \times n$ matrix into its row-reduced echelon form using elementary matrix operations (e.g. pivoting). This can be used to solve a linear system of n equations $\boldsymbol{Ax} = \boldsymbol{b}$ for the unknown vector $\boldsymbol{x}$

$$\begin{bmatrix} A_{0,0} & A_{0,1} & \cdots & A_{0,n-1} \\ A_{1,0} & A_{1,1} & \cdots & A_{1,n-1} \\ & & & \\ A_{n-1,0} & A_{n-1,1} & \cdots & A_{n-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \\ b_{n-1} \end{bmatrix}$$

The solution for $\boldsymbol{x}$ is given by inverting $\boldsymbol{A}$ and multiplying by $\boldsymbol{b}$, viz

$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$$

This is also equivalent to augmenting $\boldsymbol{A}$ with $\boldsymbol{b}$ and converting it to its row-reduced echelon form. If $\boldsymbol{A}$ is non-singular the resulting $n \times n+1$ matrix will hold $\boldsymbol{x}$ in its last column. The row-reduced echelon form of a matrix is computed in *liquid* using the Gauss-Jordan elimination algorithm, and can be invoked as such:

```
Ab =
  0.84381998 -2.38303995  1.43060994 -1.66603994  0.91488999
  3.99475002  0.88066000  4.69372988  0.44563001  0.71789002
  7.28072023 -2.06608009  0.67074001  9.80657005  1.06552994
  6.07741022 -3.93098998  1.22826004 -0.42142001 -0.81707001
matrixf_gjelim(Ab,4,5)
  1.00000000 -0.00000000  0.00000000 -0.00000000 -0.51971692
 -0.00000000  1.00000000  0.00000000  0.00000000 -0.43340963
 -0.00000000 -0.00000000  1.00000000 -0.00000000  0.64247853
  0.00000000 -0.00000000 -0.00000000  0.99999994  0.35925382
```

Notice that the result contains $\boldsymbol{I}_n$ in its first n rows and n columns (to within machine precision).[9]

[9] row permutations (swapping) might have occurred.

18 modem

The modem module implements a set of (mod)ulation/(dem)odulation schemes for encoding information into signals. For the analog modems, samples are encoded according to frequency or analog modulation. For the digital modems, data bits are encoded into symbols representing carrier frequency, phase, amplitude, etc. This section gives a brief overview of modulation schemes available in *liquid*, and provides a brief description of the interfaces.

18.1 Analog modulation schemes

This section describes the two basic analog modulation schemes available in *liquid*: frequency modulation and amplitude modulation implemented with the respective `freqmodem` and `ampmodem` objects.

18.1.1 `freqmodem` (analog FM)

The `freqmodem` object implements an analog frequency modulation (FM) modulator and demodulator. Given an input message signal $-1 \le s(t) \le 1$, the transmitted signal is

$$s(t) = \exp\left\{ j2\pi k f_c \int_0^t s(\tau) d\tau \right\} \tag{109}$$

where f_c is the carrier frequency, and k is the modulation index. The modulation index governs the relative bandwidth of the signal. Two options for demodulation are possible: observing the instantaneous frequency on the output of a phase-locked loop, or computing the instantaneous frequency using the delay-conjugate method. An example of the `freqmodem` interface is listed below.

```
1  // file: doc/listings/freqmodem.example.c
2  #include <liquid/liquid.h>
3
4  int main() {
5      float mod_index = 0.1f; // modulation index (bandwidth)
6      float fc = 0.0f;        // FM carrier
7      liquid_fmtype type = LIQUID_MODEM_FM_DELAY_CONJ;
8
9      // create mod/demod objects
10     freqmodem mod   = freqmodem_create(mod_index,fc,type);
11     freqmodem demod = freqmodem_create(mod_index,fc,type);
12
13     float s;                // input message
14     float complex x;        // modulated
15     float y;                // output/demodulated message
16
17     // repeat as necessary
18     {
19         // modulate signal
20         freqmodem_modulate(mod, s, &x);
21
22         // demodulate signal
```

```
23          freqmodem_demodulate(demod, x, &y);
24      }
25
26      // clean up objects
27      freqmodem_destroy(mod);
28      freqmodem_destroy(demod);
29  }
```

A more detailed example can be found in `examples/freqmodem_example.c` located under the main *liquid* project source directory. Listed below is the full interface to the `freqmodem` object for analog frequency modulation/demodulation.

`freqmodem_create(k,fc,type)` creates and returns an `freqmodem` object with a modulation index k, a carrier frequency $-0.5 < f_c < 0.5$, and a demodulation type defined by `type`. The demodulation type can either be `LIQUID_MODEM_FM_PLL` which uses a phased-locked loop or `LIQUID_MODEM_FM_DELAY_CONJ` which uses the delay conjugate method.

`freqmodem_destroy(q)` destroys an `freqmodem` object, freeing all internally-allocated memory.

`freqmodem_reset(q)` resets the state of the `freqmodem` object.

`freqmodem_print(q)` prints the internal state of the `freqmodem` object.

`freqmodem_modulate(q,x,*y)` modulates the input sample x storing the output to y.

`freqmodem_demodulate(q,y,*x)` demodulates the input sample y storing the output to x.

18.1.2 `ampmodem` (analog AM)

The `ampmodem` object implements an analog amplitude modulation (AM) modulator/demodulator pair. Two basic transmission schemes are available: single side-band (SSB), and double side-band (DSB). For an input message signal $-1 \le s(t) \le 1$, the double side-band transmitted signal is

$$x_{DSB}(t) = \begin{cases} s(t)e^{j2\pi f_c t} & \text{suppressed carrier} \\ \frac{1}{2}\big(1 + ks(t)\big)e^{j2\pi f_c t} & \text{unsuppressed carrier} \end{cases} \tag{110}$$

where f_c is the carrier frequency, and k is the modulation index. For single side-band, only the upper (USB) or lower half (LSB) of the spectrum is transmitted. The opposing half of the spectrum is rejected using a Hilbert transform (see Section 14.6). Let $\dot{s}(t)$ represent the Hilbert transform of the message signal $s(t)$ such that its Fourier transform is non-zero only for positive frequency components, viz

$$\dot{S}(\omega) = \mathcal{F}\left\{\dot{s}(t)\right\} = \begin{cases} S(\omega) = \mathcal{F}\left\{s(t)\right\} & \omega > 0 \\ 0 & \omega \le 0 \end{cases} \tag{111}$$

Consequently the transmitted upper side-band signal is

$$x_{USB}(t) = \begin{cases} \dot{s}(t)e^{j2\pi f_c t} & \text{suppressed carrier} \\ \frac{1}{2}\big(1 + k\dot{s}(t)\big)e^{j2\pi f_c t} & \text{unsuppressed carrier} \end{cases} \tag{112}$$

For lower single side-band, $\dot{s}(t)$ is simply conjugated. For suppressed carrier modulation the receiver uses a phase-locked loop for carrier frequency and phase tracking. When the carrier is not suppressed the receiver demodulates using a simple peak detector and IIR bias removal filter. An example of the `freqmodem` interface is listed below.

```
// file: doc/listings/ampmodem.example.c
#include <liquid/liquid.h>

int main() {
    float mod_index = 0.1f;         // modulation index (bandwidth)
    liquid_modem_amtype type = LIQUID_MODEM_AM_USB;
    int suppressed_carrier = 0;     // suppress the carrier?

    // create mod/demod objects
    ampmodem mod   = ampmodem_create(mod_index, type, suppressed_carrier);
    ampmodem demod = ampmodem_create(mod_index, type, suppressed_carrier);

    float s;                // input message
    float complex x;        // modulated
    float y;                // output/demodulated message

    // repeat as necessary
    {
        // modulate signal
        ampmodem_modulate(mod, s, &x);

        // demodulate signal
        ampmodem_demodulate(demod, x, &y);
    }

    // clean up objects
    ampmodem_destroy(mod);
    ampmodem_destroy(demod);
}
```

A more detailed example can be found in `examples/ampmodem_example.c` located under the main *liquid* project source directory. Listed below is the full interface to the `ampmodem` object for analog frequency modulation/demodulation.

`ampmodem_create(k,type,suppressed_carrier)` creates and returns an `ampmodem` object with a modulation index k, a modulation scheme defined by `type`, and a binary flag specifying whether the carrier should be suppressed. The modulation type can either be `LIQUID_MODEM_AM_DSB` (double side-band), `LIQUID_MODEM_FM_USB` (single upper side-band), or `LIQUID_MODEM_FM_LSB` (single lower side-band). method.

`ampmodem_destroy(q)` destroys an `ampmodem` object, freeing all internally-allocated memory.

`ampmodem_reset(q)` resets the state of the `ampmodem` object.

`ampmodem_print(q)` prints the internal state of the `ampmodem` object.

Table 8: Linear Modulation Schemes Available in *liquid*

scheme	*depth range*	*description*
LIQUID_MODEM_UNKNOWN	-	unknown/unsupported scheme
LIQUID_MODEM_PSK	1—8	phase-shift keying
LIQUID_MODEM_DPSK	1—8	differential phase-shift keying
LIQUID_MODEM_ASK	1—8	amplitude-shift keying
LIQUID_MODEM_QAM	2—8	quadrature amplitude-shift keying
LIQUID_MODEM_APSK	2—7	amplitude/phase-shift keying
LIQUID_MODEM_ARB	1—8	arbitrary signal constellation
LIQUID_MODEM_BPSK	1	binary phase-shift keying
LIQUID_MODEM_QPSK	2	quaternary phase-shift keying
LIQUID_MODEM_APSK4	2	amplitude/phase-shift keying, $M = 4$, (1,3)
LIQUID_MODEM_APSK8	3	amplitude/phase-shift keying, $M = 8$, (1,7)
LIQUID_MODEM_APSK16	4	amplitude/phase-shift keying, $M = 16$, (4,12)
LIQUID_MODEM_APSK32	5	amplitude/phase-shift keying, $M = 32$, (4,12,16)
LIQUID_MODEM_APSK64	6	amplitude/phase-shift keying, $M = 64$, (4,14,20,26)
LIQUID_MODEM_APSK128	7	amplitude/phase-shift keying, $M = 128$, (8,18,24,36,42)
LIQUID_MODEM_V29	4	V.29 star modem
LIQUID_MODEM_ARB16OPT	4	optimal 16-QAM
LIQUID_MODEM_ARB32OPT	4	optimal 32-QAM
LIQUID_MODEM_ARB64VT	6	Virginia Tech logo

`ampmodem_modulate(q,x,*y)` modulates the input sample x storing the output to y.

`ampmodem_demodulate(q,y,*x)` demodulates the input sample y storing the output to x.

18.2 Linear digital modulation schemes

The `modem` object realizes the linear digital modulation library in which the information from a symbol is encoded into the amplitude and phase of a sample. The modem structure implements a variety of common modulation schemes, including (differential) phase-shift keying, and (quadrature) amplitude-shift keying. The input/output relationship for modulation/demodulation for the `modem` object is strictly one-to-one and is independent of any pulse shaping, or interpolation.

In general, linear modems demodulate by finding the closest of M symbols in the set $\mathcal{S}_M$ to the received symbol r, viz

$$\arg\min_{k\in\mathcal{S}_M}\left\{\|r - s_k\|\right\} \tag{113}$$

For arbitrary modulation schemes a linear search over all symbols in $\mathcal{S}$ is required which has a complexity of $\mathcal{O}(M^2)$, however one may take advantage of symmetries in certain constellations to reduce this.

18.2.1 Interface

`modem_create(scheme,bps)` creates a linear modulator/demodulator `modem` object with one of the schemes defined in Table 8 with `bps` bits per symbol.

`modem_destroy(q)` destroys a `modem` object, freeing all internally-allocated memory.

`modem_print(q)` prints the internal state of the object.

`modem_reset(q)` resets the internal state of the object. This method is really only relevant to `LIQUID_MODEM_DPSK` (differential phase-shift keying) which retains the phase of the previous symbol in memory. All other modulation schemes are memoryless.

`modem_arb_init(q,*map,n)` initializes an arbitrary modem (`LIQUID_MODEM_ARB`) with the n-point constellation map. The resulting constellation is normalized such that it is centered at zero and has unity energy.

`modem_arb_init_file(q,*filename)` initializes an arbitrary modem (`LIQUID_MODEM_ARB`) with a constellation map defined in an external file. The file includes one line per symbol with the in-phase and quadrature components separated by white space, e.g.

```
 1.46968   0.13529
 1.69067   0.71802
-0.85603   0.43542
-0.56563   1.50369
 0.45232   0.42128
    ...
```

The resulting constellation is normalized such that it is centered at zero and has unity energy.

`modem_modulate(q,symbol,*x)` modulates the integer symbol storing the result in the output value of x. The input symbol value must be less than the constellation size M.

`modem_demodulate(q,x,*symbol)` finds the closest integer symbol which matches the input sample x. The exact method by which *liquid* performs this computation is dependent upon the modulation scheme. For example, while `LIQUID_MODEM_QAM` ($M = 4$), and `LIQUID_MODEM_PSK` ($M = 4$) are effectively equivalent (four points on the unit circle) they are demodulated differently.

`modem_get_demodulator_phase_error(q)` returns an angle proportional to the phase error after demodulation. This value can be used in a phase-locked loop (see Section 19.2) to correct for carrier phase recovery.

`modem_get_demodulator_evm(q)` returns a value equal to the error vector magnitude after demodulation. The error vector is the difference between the received symbol and the estimated transmitted symbol, $e = r - \hat{s}$. The magnitude of the error vector is an indication to the signal-to-noise/distortion ratio at receiver.

While the same modem structure may be used for both modulation and demodulation for most schemes, it is important to use separate objects for differential-mode modems (e.g. `LIQUID_MODEM_DPSK`) as the internal state will change after each symbol. It is usually good practice to keep separate instances of modulators and demodulators. This holds true for most any encoder/decoder pair in *liquid*. An example of the `qmodem` interface is listed below.

```c
// file: doc/listings/agc.example.c
#include <liquid/liquid.h>

int main() {
    // create mod/demod objects
    unsigned int bps=2;
    modulation_scheme ms = LIQUID_MODEM_PSK;

    // create the modem objects
    modem mod   = modem_create(ms, bps);    // modulator
    modem demod = modem_create(ms, bps);    // demodulator
    modem_print(mod);

    unsigned int sym_in;    // input symbol
    float complex x;        // modulated sample
    unsigned int sym_out;   // demodulated symbol

    // ...repeat as necessary...
    {
        // modulate symbol
        modem_modulate(mod, sym_in, &x);

        // demodulate symbol
        modem_demodulate(demod, x, &sym_out);
    }

    // destroy modem objects
    modem_destroy(mod);
    modem_destroy(demod);
}
```

18.2.2 Gray coding

In order to reduce the number of bit errors in a digital modem, all symbols are automatically Gray encoded such that adjacent symbols in a constellation differ by only one bit. For example, the binary-coded decimal (BCD) value of 183 is `10110111`. It has adjacent symbol 184 (`10111000`) which differs by 4 bits. Assume the transmitter sends 183 without encoding. If noise at the receiver were to cause it to demodulate the nearby symbol 184, the result would be 4 bit errors. Gray encoding is computed to the binary-coded decimal symbol by applying an exclusive OR bitmask of itself shifted to the right by a single bit.

```
          10110111    bcd_in (183)        10111000    bcd_in (184)
          .1011011    bcd_in >> 1         .1011100    bcd_in >> 1
    xor : --------                        --------
          11101100    gray_out (236)      11100100    gray_out (228)
```

Notice that the two encoded symbols 236 (`11101100`) and 228 (`11100100`) differ by only one bit. Now if noise caused the receiver were to demodulate a symbol error, it would result in only a single bit error instead of 4 without Gray coding.

Reversing the process (decoding) is similar to encoding but slightly more involved. Gray decoding is computed on an encoded input symbol by adding to it (modulo 2) as many shifted versions

of itself as it has bits. In our previous example the receiver needs to map the received encoded symbol back to the original symbol before encoding:

```
          11101100    gray_in (236)       11100100    gray_in (228)
          .1110110    gray_in >> 1        .1110010    gray_in >> 1
          ..111011    gray_in >> 2        ..111001    gray_in >> 2
          ...11101    gray_in >> 3        ...11100    gray_in >> 3
          ....1110    gray_in >> 4        ....1110    gray_in >> 4
          .....111    gray_in >> 5        .....111    gray_in >> 5
          ......11    gray_in >> 6        ......11    gray_in >> 6
          .......1    gray_in >> 7        .......1    gray_in >> 7
xor :     --------                        --------
          10110111    gray_out (183)      10111000    gray_out (184)
```

There are a few interesting characteristics of Gray encoding:

- the first bit never changes in encoding/decoding
- there is a unique mapping between input and output symbols

It is also interesting to note that in linear modems (e.g. PSK), the `decoder` is actually applied to the symbol at the transmitter while the `encoder` is applied to the received symbol at the receiver. In *liquid*, Gray encoding and decoding are computed with the `gray_encode()` `gray_decode()` methods, respectively.

18.2.3 `LIQUID_MODEM_PSK` (phase-shift keying)

With phase-shift keying the information is stored in the absolute phase of the modulated signal. This means that each of $M = 2^m$ symbols in the constellation are equally spaced around the unit circle. Figure 29 depicts the constellation of PSK up to $M = 16$ with the bits gray encoded. While *liquid* supports up to $M = 256$, values greater than $M = 32$ are typically avoided due to error rates for practical signal-to-noise ratios. For an M-symbol constellation, the k^{th} symbol is

$$s_k = e^{j2\pi k/M} \tag{114}$$

where $k \in \{0, 1, \ldots, M-1\}$. Specific schemes include BPSK ($M = 2$),

$$s_k = e^{j\pi k} = \begin{cases} +1 & k = 0 \\ -1 & k = 1 \end{cases} \tag{115}$$

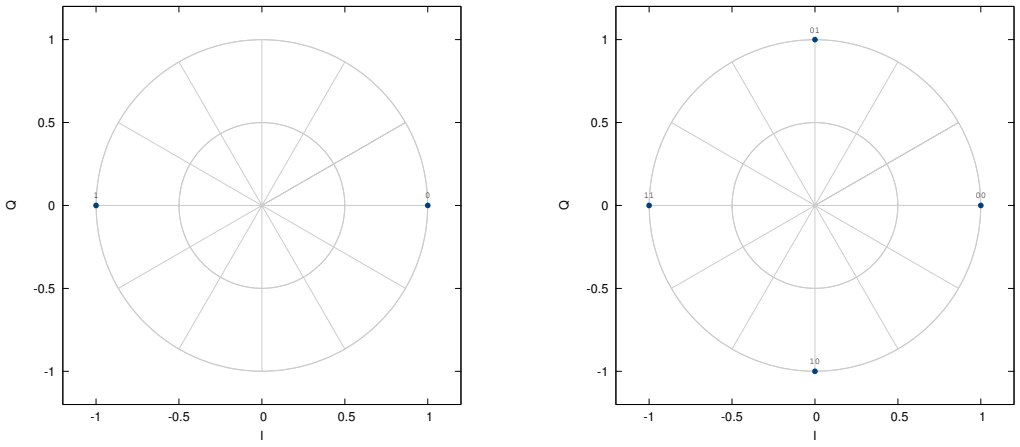
and QPSK ($M = 4$)

$$s_k = e^{j\left(\pi k/4 + \frac{\pi}{4}\right)} \tag{116}$$

Demodulation is performed independent of the signal amplitude for coherent PSK.

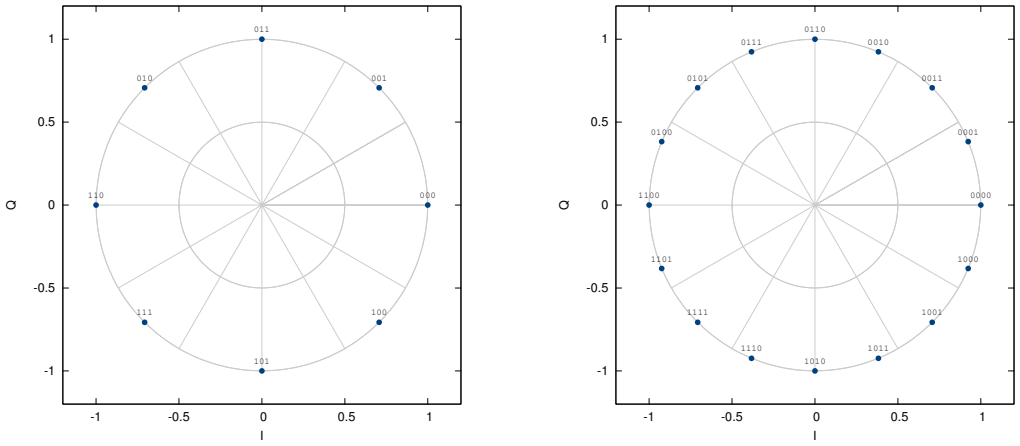
18.2.4 `LIQUID_MODEM_DPSK` (differential phase-shift keying)

Differential PSK (DPSK) encodes information in the phase change of the carrier. Like regular PSK demodulation is performed independent of the signal amplitude; however because the data are encoded using phase transitions rather than absolute phase, the receiver does not have to know



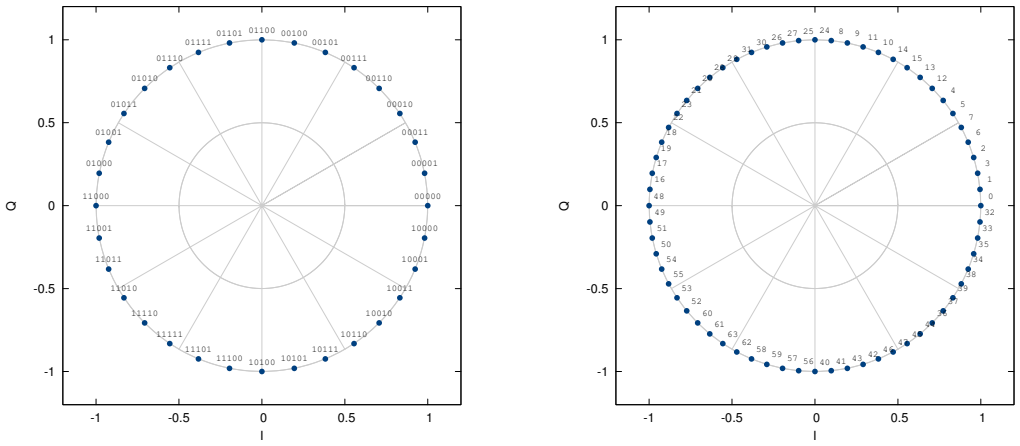
(a) 2-PSK (generic BPSK)

(b) 4-PSK (generic QPSK)



(c) 8-PSK

(d) 16-PSK



(e) 32-PSK

(f) 64-PSK

Figure 29: Phase-shift keying (PSK) modem constellation map. Note that BPSK and QPSK are customized implementations of 2-PSK and 4-PSK. While only PSK up to $M = 64$ are shown, *liquid* supports up to 256-PSK.

the absolute phase of the transmitter. This allows the receiver to demodulate incoherently, but at a quality degradation of 3dB. As such the n^{th} transmitted symbol $k(n)$ depends on the previous symbol, viz

$$s_k(n) = \exp\left\{\frac{j2\pi\Big(k(n) - k(n-1)\Big)}{M}\right\} \tag{117}$$

18.2.5 `LIQUID_MODEM_APSK` (amplitude/phase-shift keying

Amplitude/phase-shift keying (APSK) is a specific form of quadrature amplitude modulation where constellation points lie on concentric circles. The constellation points are further apart than those of PSK/DPSK, resulting in an improved error performance. Furthermore the phase recovery for APSK is improved over regular QAM as the constellation points are less sensitive to phase noise. This improvement comes at the cost of an increased computational complexity at the receiver. Demodulation follows as a two-step process: first, the amplitude of the received signal is evaluated to determine in which level ("ring") the transmitted symbol lies. Once the level is determined, the appropriate symbol is chosen based on its phase, similar to PSK demodulation. Demodulation of APSK consumes slightly more clock cycles than the PSK and QAM demodulators. Figure 30 depicts the available APSK signal constellations for M up to 128. The constellation points and bit mappings have been optimized to minimize the bit error rate in 10 dB SNR.

18.2.6 `LIQUID_MODEM_ASK` (amplitude-shift keying)

Amplitude-shift keying (ASK) is a simple form of amplitude modulation by which the information is encoded entirely in the in-phase component of the baseband signal. The encoded symbol is simply

$$s_k = \alpha\big(2k - M - 1\big) \tag{118}$$

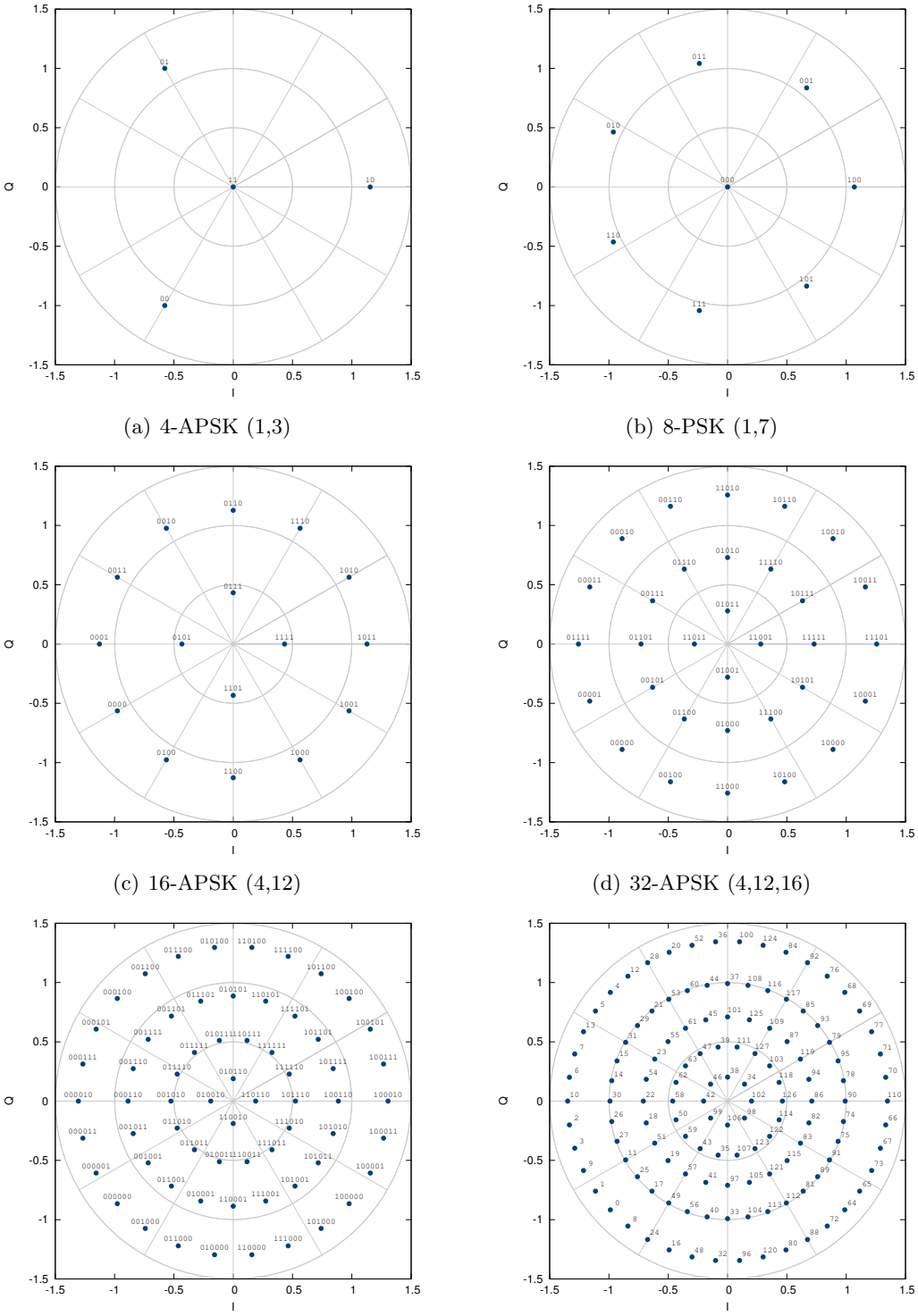
where α is a scaling factor to ensure $E\{s_k^2\} = 1$,

$$\alpha = \begin{cases} 1 & M = 2 \\ 1/\sqrt{5} & M = 4 \\ 1/\sqrt{21} & M = 8 \\ 1/\sqrt{85} & M = 16 \\ 1/\sqrt{341} & M = 32 \\ \sqrt{3}/M & M > 32 \end{cases} \tag{119}$$

Figure 31 depicts the ASK constellation map for M up to 16. Due to the poor error rate performance of ASK values of M greater than 16 are not recommended.

18.2.7 `LIQUID_MODEM_QAM` (quadrature amplitude modulation)

Also known as quadrature amplitude-shift keying, QAM modems encode data using both the in-phase and quadrature components of a signal amplitude. In fact, the symbol is split into independent in-phase and quadrature symbols which are encoded separately as `LIQUID_MODEM_ASK` symbols. Gray encoding is applited to both the I and Q symbols separately to help ensure minimal



(a) 4-APSK (1,3)

(b) 8-PSK (1,7)

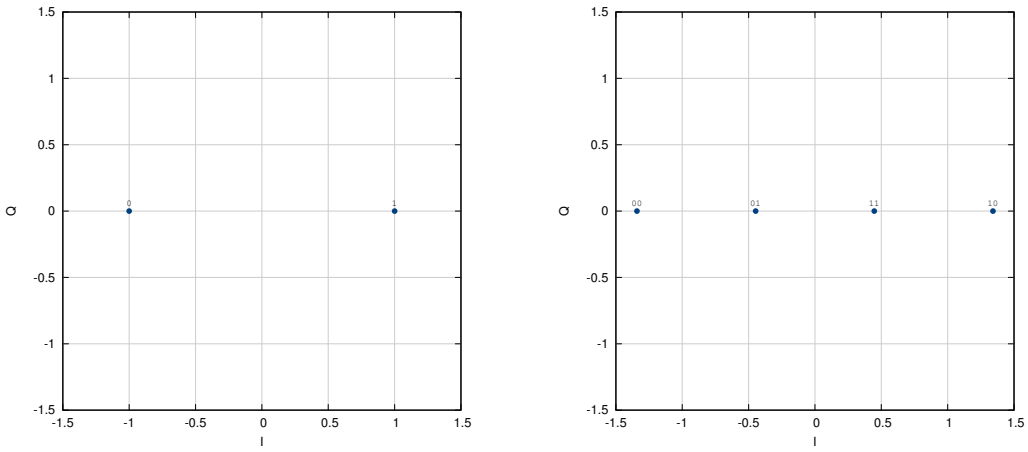
(c) 16-APSK (4,12)

(d) 32-APSK (4,12,16)

(e) 64-APSK (4,14,20,26)

(f) 128-APSK (8,18,24,36,42)

Figure 30: Amplitude/phase-shift keying (APSK) modem



(a) 2-ASK

(b) 4-ASK

(c) 8-ASK

(d) 16-ASK

Figure 31: Pulse-amplitude modulation (ASK) modem

bit changes between adjacent samples across both in-phase and quadrature-phase dimensions. This is made evident in Figure 32(d) where one can see that the first three bits of the symbol encode the in-phase component of the sample, and the last thre bits encode the quadrature component of the sample. We may formally describe the encoded sample is

$$s_k = \alpha\Big\{(2k_i - M_i - 1) + j(2k_q - M_q - 1)\Big\} \tag{120}$$

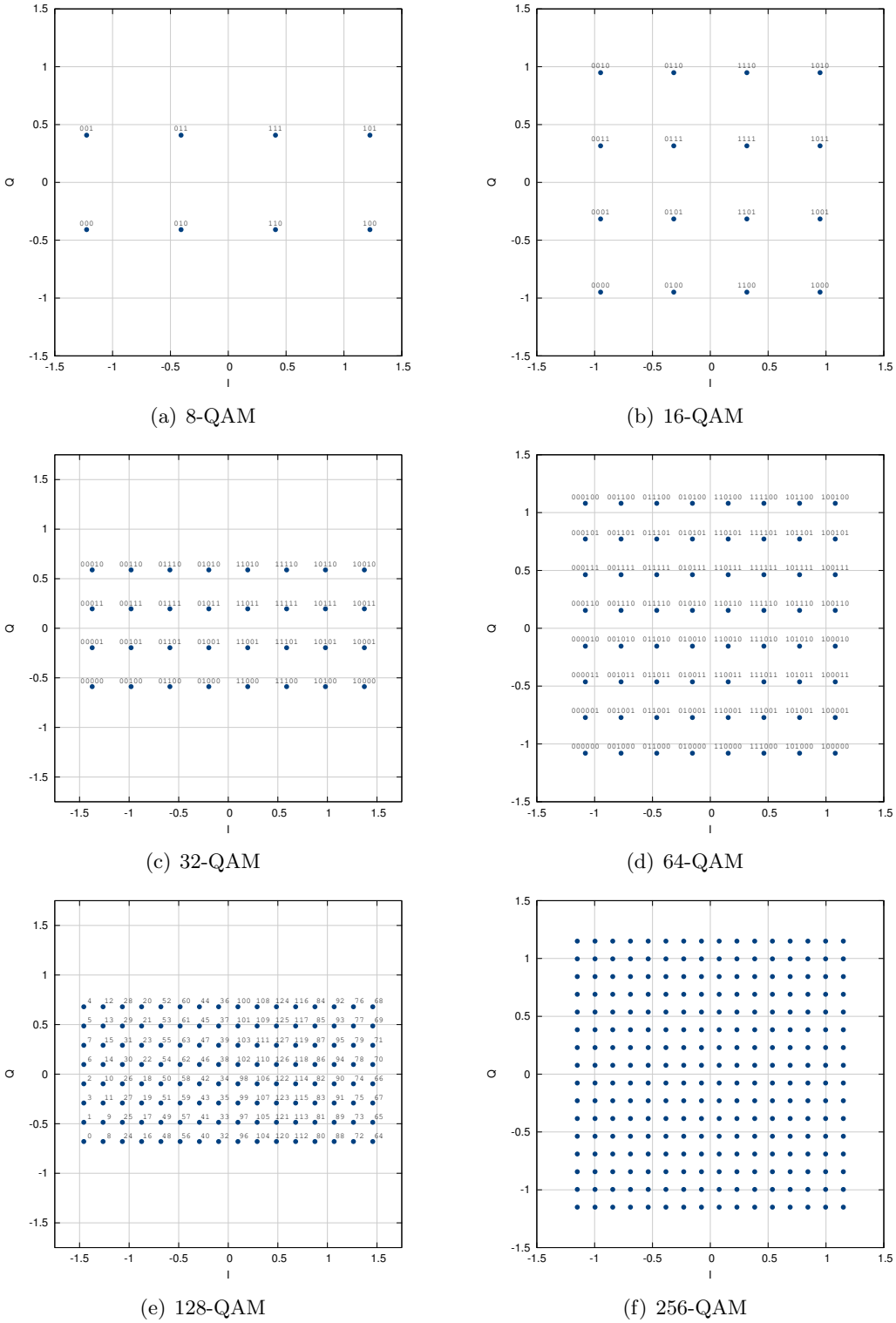
where k_i is the in-phase symbol, k_q is the quadrature symbol, $M_i = 2^{m_i}$ and $M_q = 2^{m_q}$, are the number of respective in-phase and quadrature symbols, $m_i = \lceil \log_2(M) \rceil$ and $m_q = \lfloor \log_2(M) \rfloor$ are the number of respective in-phase and quadrature bits, and α is a scaling factor to ensure $E\{s_k^2\} = 1$,

$$\alpha = \begin{cases} 1/\sqrt{2} & M = 4 \\ 1/\sqrt{6} & M = 8 \\ 1/\sqrt{10} & M = 16 \\ 1/\sqrt{26} & M = 32 \\ 1/\sqrt{42} & M = 64 \\ 1/\sqrt{106} & M = 128 \\ 1/\sqrt{170} & M = 256 \\ 1/\sqrt{426} & M = 512 \\ 1/\sqrt{682} & M = 1024 \\ 1/\sqrt{1706} & M = 2048 \\ 1/\sqrt{2730} & M = 4096 \\ \sqrt{2/M} & \text{else} \end{cases} \tag{121}$$

Figure 32 depicts the arbitrary rectangular QAM modem constellation maps for M up to 256. Notice that all the symbol points are gray encoded to minimize bit errors between adjacent symbols.

18.2.8 `LIQUID_MODEM_ARB` (arbitrary modem)

liquid also allows the user to create their own modulation schemes by designating the number of bits per symbol, and the full signal constellation. Several pre-loaded arbitrary signal constellations are available. Figure 33 depicts several available arbitrary constellation maps; however the user can create any arbitrary constellation map so long as no two points overlap (see `modem_arb_init()` and `modem_arb_init_file()` in Section 18.2.1).



(a) 8-QAM

(b) 16-QAM

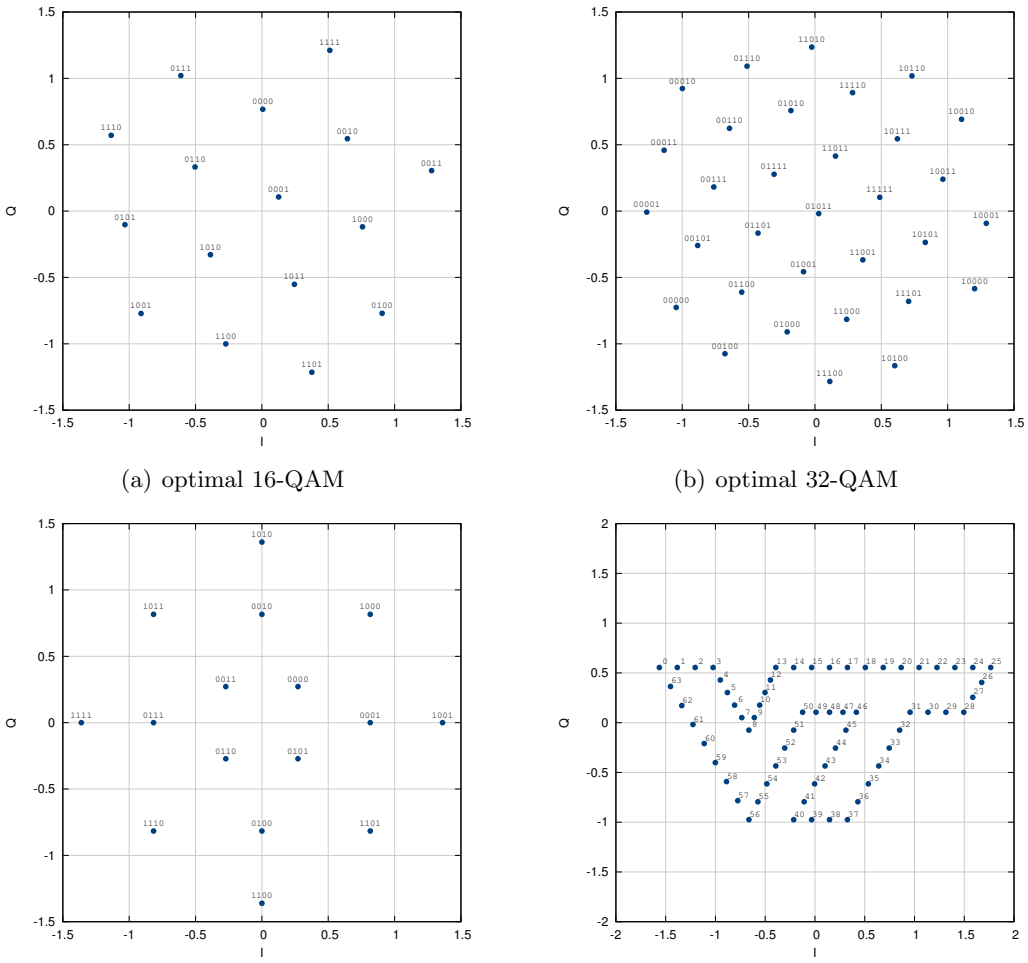
(c) 32-QAM

(d) 64-QAM

(e) 128-QAM

(f) 256-QAM

Figure 32: Rectangular quaternary-amplitude modulation (QAM) modem



(a) optimal 16-QAM

(b) optimal 32-QAM

(c) V.29

(d) 64-VT

Figure 33: Arbitrary constellation (ARB) modem

19 nco (numerically-controlled oscillator)

This section describes the numerically-controlled oscillator (NCO) for carrier synchronization.

19.1 nco object

The nco object implements an oscillator with two options for internal phase precision: LIQUID_NCO and LIQUID_VCO. The LIQUID_NCO implements a numerically-controlled oscillator that uses a look-up table to generate a complex sinusoid while the LIQUID_VCO implements a "voltage-controlled" oscillator that uses the sinf and cosf standard math functions to generate a complex sinusoid.

19.1.1 Description of operation

The nco object maintains its phase and frequency states internally. Various computations–such as mixing–use the phase state for generating complex sinusoids. The phase θ of the nco object is updated using the nco_crcf_step() method which increments θ by $\Delta\theta$, the frequency. Both the phase and frequency of the nco object can be manipulated using the appropriate nco_crcf_set and nco_crcf_adjust methods. Here is a minimal example demonstrating the interface to the nco object:

```
// file: doc/listings/nco_pll.example.c
#include <liquid/liquid.h>

int main() {
    // create nco objects
    nco_crcf nco_tx = nco_crcf_create(LIQUID_VCO);    // transmit NCO
    nco_crcf nco_rx = nco_crcf_create(LIQUID_VCO);    // receive NCO

    // ... initialize objects ...

    float complex * x;
    unsigned int i;
    // loop as necessary
    {
        // tx : generate complex sinusoid
        nco_crcf_cexpf(nco_tx, &x[i]);

        // compute phase error
        float dphi = nco_crcf_get_phase(nco_tx) -
                     nco_crcf_get_phase(nco_rx);

        // update pll
        nco_crcf_pll_step(nco_rx, dphi);

        // update nco objects
        nco_crcf_step(nco_tx);
        nco_crcf_step(nco_rx);
    }

    // destry nco object
```

```
31      nco_crcf_destroy(nco_tx);
32      nco_crcf_destroy(nco_rx);
33  }
```

19.1.2 Interface

Listed below is the full interface to the `nco` family of objects.

`nco_crcf_create(type)` creates an `nco` object of type `LIQUID_NCO` or `LIQUID_VCO`.

`nco_crcf_destroy(q)` destroys an `nco` object, freeing all internally-allocated memory.

`nco_crcf_print(q)` prints the internal state of the `nco` object to the standard output.

`nco_crcf_reset(q)` clears in internal state of an `nco` object.

`nco_crcf_set_frequency(q,f)` sets the frequency f (equal to the phase step size $\Delta\theta$).

`nco_crcf_adjust_frequency(q,df)` increments the frequency by Δf.

`nco_crcf_set_phase(q,theta)` sets the internal `nco` phase to θ.

`nco_crcf_adjust_phase(q,dtheta)` increments the internal `nco` phase by $\Delta\theta$.

`nco_crcf_step(q)` increments the internal `nco` phase by its internal frequency, $\theta \leftarrow \theta + \Delta\theta$

`nco_crcf_get_phase(q)` returns the internal phase of the `nco` object, $-\pi \le \theta < \pi$.

`nco_crcf_get_frequency(q)` returns the internal frequency (phase step size)

`nco_crcf_sin(q)` returns $\sin(\theta)$

`nco_crcf_cos(q)` returns $\cos(\theta)$

`nco_crcf_sincos(q,*sine,*cosine)` computes $\sin(\theta)$ and $\cos(\theta)$

`nco_crcf_cexpf(q,*y)` computes $y = e^{j\theta}$

`nco_crcf_mix_up(q,x,*y)` rotates an input sample x by $e^{j\theta}$, storing the result in the output sample y.

`nco_crcf_mix_down(q,x,*y)` rotates an input sample x by $e^{-j\theta}$, storing the result in the output sample y.

`nco_crcf_mix_block_up(q,*x,*y,n)` rotates an n-element input array $\boldsymbol{x}$ by $e^{j\theta k}$ for $k \in \{0, 1, \ldots, n-1\}$, storing the result in the output vector $\boldsymbol{y}$.

`nco_crcf_mix_block_down(q,*x,*y,n)` rotates an n-element input array $\boldsymbol{x}$ by $e^{-j\theta k}$ for $k \in \{0, 1, \ldots, n-1\}$, storing the result in the output vector $\boldsymbol{y}$.

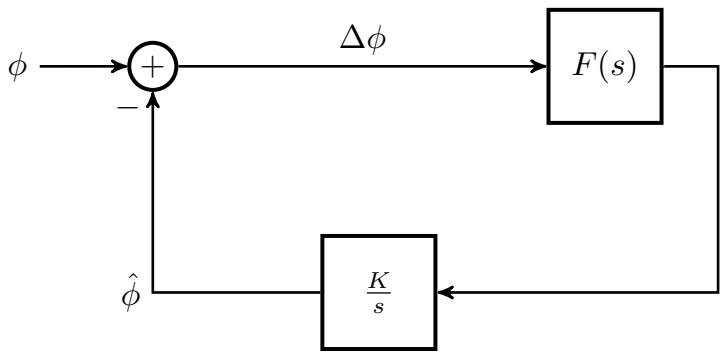


Figure 34: PLL block diagram

19.2 PLL (phase-locked loop)

The phase-locked loop object provides a method for synchronizing oscillators on different platforms. It uses a second-order integrating loop filter to adjust the frequency of its `nco` based on an instantaneous phase error input. As its name implies, a PLL locks the phase of the `nco` object to a reference signal. The PLL accepts a phase error and updates the frequency (phase step size) of the `nco` to track to the phase of the reference. The reference signal can be another `nco` object, or a signal whose carrier is modulated with data. The PLL consists of three components: the phase detector, the loop filter, and the integrator. A block diagram of the PLL can be seen in Figure 34 in which the phase detector is represented by the summing node, the loop filter is $F(s)$, and the integrator has a transfer function $G(s) = K/s$. For a given loop filter $F(s)$, the closed-loop transfer function becomes

$$H(s) = \frac{G(s)F(s)}{1 + G(s)F(s)} = \frac{KF(s)}{s + KF(s)} \tag{122}$$

where the loop gain K absorbs all the gains in the loop. There are several well-known options for designing the loop filter $F(s)$, which is, in general, a first-order low-pass filter. In particular we are interested in getting the denominator of $H(s)$ to the standard form $s^2 + 2\zeta\omega_n s + \omega_n^2$ where ω_n is the natural frequency of the filter and ζ is the damping factor. This simplifies analysis of the overall transfer function and allows the parameters of $F(s)$ to ensure stability.

19.2.1 Active lag design

The active lag PLL [4] has a loop filter with a transfer function $F(s) = (1 + \tau_2 s)/(1 + \tau_1 s)$ where τ_1 and τ_2 are parameters relating to the damping factor and natural frequency. This gives a closed-loop transfer function

$$H(s) = \frac{\frac{K}{\tau_1}(1 + s\tau_2)}{s^2 + s\frac{1+K\tau_2}{\tau_1} + \frac{K}{\tau_1}} \tag{123}$$

Converting the denominator of (123) into standard form yeilds the following equations for τ_1 and τ_2:

$$\omega_n = \sqrt{\frac{K}{\tau_1}} \quad \zeta = \frac{\omega_n}{2}\left(\tau_2 + \frac{1}{K}\right) \rightarrow \tau_1 = \frac{K}{\omega_n^2} \quad \tau_2 = \frac{2\zeta}{\omega_n} - \frac{1}{K} \tag{124}$$

The open-loop transfer function is therefore

$$H'(s) = F(s)G(s) = K\frac{1+\tau_2 s}{s+\tau_1 s^2} \tag{125}$$

Taking the bilinear z-transform of $H'(s)$ gives the digital filter:

$$H'(z) = H'(s)\Big|_{s=\frac{1}{2}\frac{1-z^{-1}}{1+z^{-1}}} = 2K\frac{(1+\tau_2/2)+2z^{-1}+(1-\tau_2/2)z^{-2}}{(1+\tau_1/2)-\tau_1 z^{-1}+(-1+\tau_1/2)z^{-2}} \tag{126}$$

A simple 2^{nd}-order active lag IIR filter can be designed using the following method:

```
void iirdes_pll_active_lag(float _w,    // filter bandwidth
                           float _zeta, // damping factor
                           float _K,    // loop gain (1,000 suggested)
                           float * _b,  // output feed-forward coefficients [size: 3 x 1]
                           float * _a); // output feed-back coefficients [size: 3 x 1]
```

19.2.2 Active PI design

Similar to the active lag PLL design is the active "proportional plus integration" (PI) which has a loop filter $F(s) = (1+\tau_2 s)/(\tau_1 s)$ where τ_1 and τ_2 are also parameters relating to the damping factor and natural frequency, but are different from those in the active lag design. The above loop fitler yields a closed-loop transfer function

$$H(s) = \frac{\frac{K}{\tau_1}(1+s\tau_2)}{s^2 + s\frac{K\tau_2}{\tau_1} + \frac{K}{\tau_1+\tau_2}} \tag{127}$$

Converting the denominator of (127) into standard form yeilds the following equations for τ_1 and τ_2:

$$\omega_n = \sqrt{\frac{K}{\tau_1}} \quad \zeta = \frac{\omega_n\tau_2}{2} \rightarrow \tau_1 = \frac{K}{\omega_n^2} \quad \tau_2 = \frac{2\zeta}{\omega_n} \tag{128}$$

The open-loop transfer function is therefore

$$H'(s) = F(s)G(s) = K\frac{1+\tau_2 s}{\tau_1 s^2} \tag{129}$$

Taking the bilinear z-transform of $H'(s)$ gives the digital filter

$$H'(z) = H'(s)\Big|_{s=\frac{1}{2}\frac{1-z^{-1}}{1+z^{-1}}} = 2K\frac{(1+\tau_2/2)+2z^{-1}+(1-\tau_2/2)z^{-2}}{\tau_1/2-\tau_1 z^{-1}+(\tau_1/2)z^{-2}} \tag{130}$$

A simple 2^{nd}-order active PI IIR filter can be designed using the following method:

```
void iirdes_pll_active_PI(float _w,    // filter bandwidth
                          float _zeta, // damping factor
                          float _K,    // loop gain (1,000 suggested)
                          float * _b,  // output feed-forward coefficients [size: 3 x 1]
                          float * _a); // output feed-back coefficients [size: 3 x 1]
```

19.2.3 PLL Interface

The `nco` object has an internal PLL interface which only needs to be invoked before the `nco_crcf_step()` method (see Section 19.1.2) with the appropriate phase error estimate. This will permit the `nco` object to automatically track to a carrier offset for an incoming signal. The `nco` object has the following PLL method extensions to enable a simplified phase-locked loop interface.

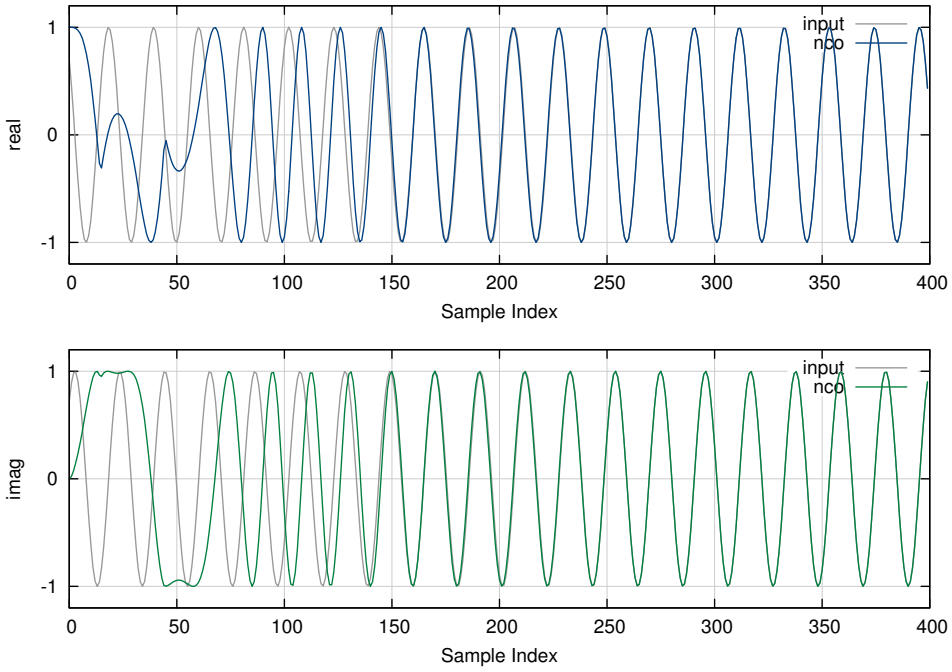
`nco_crcf_pll_set_bandwidth(q,w)` sets the bandwidth of the loop filter of the `nco` object's internal PLL to ω.

`nco_crcf_pll_step(q,dphi)` advances the `nco` object's internal phase with a phase error $\Delta\phi$ to the loop filter. This method only changes the frequency of the `nco` object and does not update the phase until `nco_crcf_step()` is invoked. This is useful if one wants to only run the PLL periodically and ignore several samples. See the example code below for help.

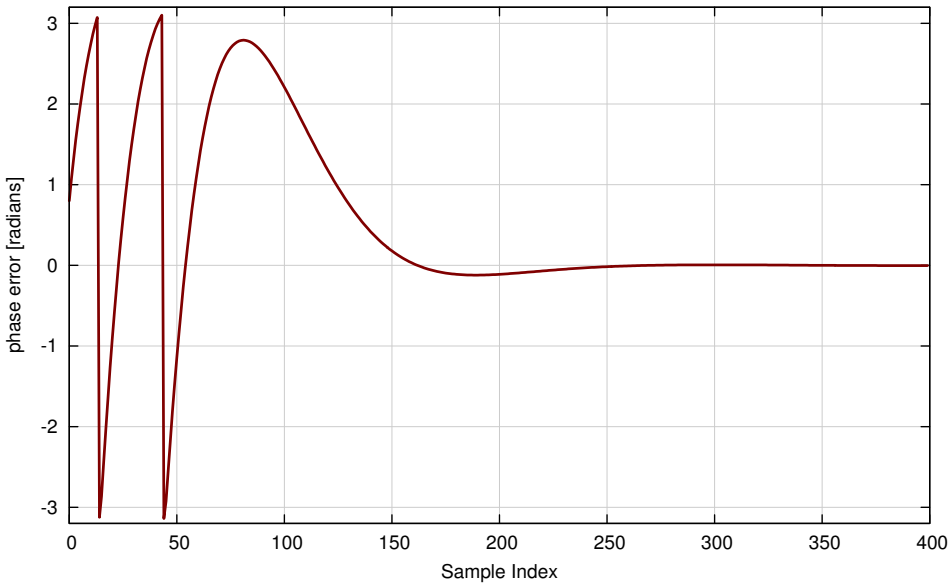
Here is a minimal example demonstrating the interface to the `nco` object and the internal phase-locked loop:

```
1  // file: doc/listings/nco_pll.example.c
2  #include <liquid/liquid.h>
3
4  int main() {
5      // create nco objects
6      nco_crcf nco_tx = nco_crcf_create(LIQUID_VCO);    // transmit NCO
7      nco_crcf nco_rx = nco_crcf_create(LIQUID_VCO);    // receive NCO
8
9      // ... initialize objects ...
10
11     float complex * x;
12     unsigned int i;
13     // loop as necessary
14     {
15         // tx : generate complex sinusoid
16         nco_crcf_cexpf(nco_tx, &x[i]);
17
18         // compute phase error
19         float dphi = nco_crcf_get_phase(nco_tx) -
20                      nco_crcf_get_phase(nco_rx);
21
22         // update pll
23         nco_crcf_pll_step(nco_rx, dphi);
24
25         // update nco objects
26         nco_crcf_step(nco_tx);
27         nco_crcf_step(nco_rx);
28     }
29
30     // destry nco object
31     nco_crcf_destroy(nco_tx);
32     nco_crcf_destroy(nco_rx);
33 }
```

See also `examples/nco_pll_example.c` and `examples/nco_pll_modem_example.c` located in the main *liquid* project directory. An example of the PLL can be seen in Figure 35. Notice that during the first 150 samples the NCO's output signal is misaligned to the input; eventually, however, the PLL acquires the phase of the input sinusoid and the phase error of the NCO's output approaches zero.



(a) nco output



(b) phase error

Figure 35: `nco` phase-locked loop demonstration

20 optim (optimization)

The *optim* module in *liquid* implements several non-linear optimization algorithms including a gradient descent search, a quasi-Newton search (experimental: see Section 24.5) and an evolutionary algorithm.

20.1 `gradsearch` (gradient search)

This module implements a gradient or "steepest-descent" search. Given a function f which operates on a vector $\boldsymbol{x} = [x_0, x_1, \ldots, x_{N-1}]^T$ of N parameters, the gradient search method seeks to find the optimum $\boldsymbol{x}$ which minimizes $f(\boldsymbol{x})$.

20.1.1 Theory

The gradient search is an iterative method and adjusts $\boldsymbol{x}$ proportional to the negative of the gradient of f evaluated at the current location. The vector $\boldsymbol{x}$ is adjusted by

$$\Delta\boldsymbol{x}[n+1] = -\gamma[n]\nabla f(\boldsymbol{x}[n])$$

where $\gamma[n]$ is the step size and $\nabla f(\boldsymbol{x}[n])$ is the gradient of f at $\boldsymbol{x}$, at the n^{th} iteration. The gradient is a vector field which points to the greatest rate of increase, and is computed at $\boldsymbol{x}$ as

$$\nabla f(\boldsymbol{x}) = \left(\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_{N-1}}\right)$$

In most non-linear optimization problems, $\nabla f(\boldsymbol{x})$ is not known, and must be approximated for each value of $\boldsymbol{x}[n]$ using the finite element method. The partial derivative of the k^{th} component is estimated by computing the slope of f when x_k is increased by a small amount Δ while holding all other elements of $\boldsymbol{x}$ constant. This process is repeated for all elements in $\boldsymbol{x}$ to compute the gradient vector. Mathematically, the k^{th} component of the gradient is approximated by

$$\frac{\partial f(\boldsymbol{x})}{\partial x_k} \approx \frac{f(x_0, \ldots, x_k + \Delta, \ldots, x_{N-1}) - f(\boldsymbol{x})}{\Delta}$$

Once $\nabla f(\boldsymbol{x}[n])$ is known, $\Delta\boldsymbol{x}[n+1]$ is computed and the optimizing vector is updated via

$$\boldsymbol{x}[n+1] = \boldsymbol{x}[n] + \Delta\boldsymbol{x}[n+1]$$

20.1.2 Momentum constant

When $f(\boldsymbol{x})$ is flat (i.e. $\nabla f(\boldsymbol{x}) \approx \boldsymbol{0}$), convergence will be slow. This effect can be mitigated by permitting the update vector equation to retain a small portion of the previous step vector. The updated vector at time $n+1$ is

$$\boldsymbol{x}[n+1] = \boldsymbol{x}[n] + \Delta\boldsymbol{x}[n+1] + \alpha\Delta\boldsymbol{x}[n]$$

where $\Delta\boldsymbol{x}[0] = \boldsymbol{0}$. The effective update at time $n+1$ is

$$\boldsymbol{x}[n+1] = \sum_{k=0}^{n+1} \alpha^k \Delta\boldsymbol{x}[n+1-k]$$

which is stable only for $0 \leq \alpha < 1$. For flat regions, the gradient vector $\nabla f(\boldsymbol{x})$ is approximately a constant $\Delta\boldsymbol{x}$, and $\boldsymbol{x}[n]$ therefore becomes a geometric series converging to $\Delta\boldsymbol{x}/(1-\alpha)$. This accelerates the algorithm across relatively flat regions of f. The momentum constant additionally adds some stability for regions where the gradient method tends to oscillate, such as steep valleys in f.

20.1.3 Step size adjustment

In *liquid*, the gradient is normalized to unity (orthonormal). That is $\|\nabla f(\boldsymbol{x}[n])\| = 1$. Furthermore, γ is slightly reduced each epoch by a multiplier μ

$$\gamma[n+1] = \mu\gamma[n]$$

This helps improve stability and convergence over regions where the algorithm might oscillate due to steep values of f.

20.1.4 Interface

Here is a summary of the parameters used in the gradient search algorithm and their default values:

- Δ : step size in computing the gradient (default 10^{-6})
- γ : step size in updating $\boldsymbol{x}[n]$ (default 0.002)
- α : momentum constant (default 0.1)
- μ : iterative γ adjustment factor (default 0.99)

`gradsearch_create(*userdata,*v,n,utility,min/max)` creates a gradient search object designed to optimize an n-point vector $\boldsymbol{v}$. The user-defined utility function and userdata structures define the search, as well as the `min/max` flag which can be either `LIQUID_OPTIM_MINIMIZE` or `LIQUID_OPTIM_MAXIMIZE`. When run the `gradsearch` object will update the "optimal" value in the input vector $\boldsymbol{v}$ specified during `create()`.

`gradsearch_destroy(q)` destroys a `gradsearch` object, freeing all internally-allocated memory.

`gradsearch_print(q)` prints the internal state of the gradient search object.

`gradsearch_reset(q)` resets the internal state of the gradient search object.

`gradsearch_step(q)` steps through a single iteration of the gradient search. The result is stored in the original input vector $\boldsymbol{v}$ specified during the `create()` method.

`gradsearch_execute(q,n,target_utility)` runs multiple iterations of the search algorithm, stopping after either n iterations or if the target utility is met.

Here is an example of how the `gradient_search` is used:

```c
// file: doc/listings/gradsearch.example.c
#include <liquid/liquid.h>

// user-defined utility callback function
float myutility(void * _userdata, float * _v, unsigned int _n)
{
    float u = 0.0f;
    unsigned int i;
    for (i=0; i<_n; i++)
        u += fabsf(_v[i]);
    return u;
}

int main() {
    unsigned int num_parameters = 8;    // search dimensionality
    unsigned int num_iterations = 100;  // number of iterations to run
    float target_utility = 0.01f;       // target utility

    float v[num_parameters];            // optimum vector

    // ... intialize v ...

    // create gradsearch object
    gradsearch gs = gradsearch_create(NULL,
                                      v,
                                      num_parameters,
                                      &myutility,
                                      LIQUID_OPTIM_MINIMIZE);

    // execute batch search
    gradsearch_execute(gs, num_iterations, target_utility);

    // clean it up
    gradsearch_destroy(gs);
}
```

Notice that the utility function is a callback that is completely defined by the user. Figure 36 depicts the performance of the gradient search for the Rosenbrock function, defined as $f(x,y) = (1-x)^2 + 100(y-x^2)^2$ for input parameters x and y. The Rosenbrock function has a minimum at $(x,y) = (1,1)$; however the minimum lies in a deep valley which can be difficult to navigate. From the figure it is apparent that finding the valley is trivial, but convergence to the minimum is slow.

20.2 `gasearch` genetic algorithm search

The `gasearch` object implements an evolutionary (genetic) algorithm search in *liquid*. The search uses a binary string of traits called a *chromosome* (see Section 20.2.1, below) to represent a potential solution. A *population* of chromosomes is generated and their appropriate fitnesses are calculated. With each evolution of the population the best chromosomes are retained and the worst are discarded; this process is known as *selection*. The population is restored by computing new potential solutions by splitting traits of the better chromosomes into a new member (*crossover*) as well as

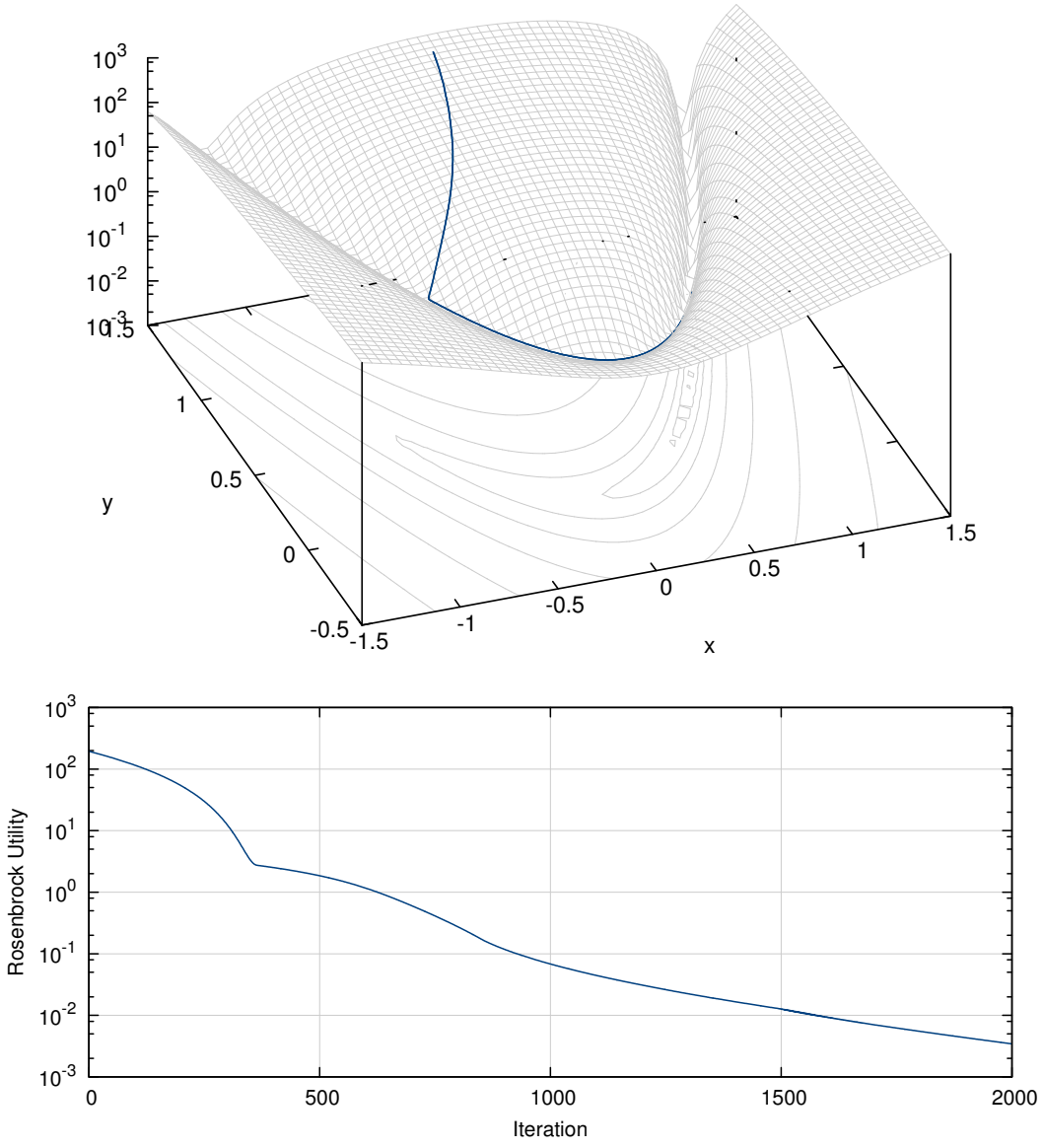


Figure 36: `gradsearch` performance for 2-parameter Rosenbrock function $f(x, y) = (1 - x)^2 + 100(y - x^2)^2$ with a starting point of $(x_0, y_0) = (-0.2, 1.4)$. The minimum is located at $(1, 1)$.

randomly flipping some of the bits in each chromosome (*mutation*).

20.2.1 `chromosome`, solution representation

The `chromosome` object in *liquid* realizes a binary string of traits used in the `gasearch` object. A chromosome has a fixed number of traits as well as a fixed number of bits to represent each trait; however the number of bits representing each trait does not necessarily need to be the same for the chromosome. That is to say a chromosome may have a number of traits, each with a different number of bits representing them; however once a `chromosome` object is created, the number of bits representing each trait is not allowed to be changed.

Because of the many ways a chromosome can represent information *liquid* provides a number of methods for creating and initializing chromosomes.

`chromosome_create()` creates a chromosome where the number of bits per each trait are specified.

`chromosome_create_basic()` creates a chromosome where the number of bits per trait is the same for all traits.

`chromosome_create_clone()` clones a chromosome from another one, including its representation of traits, the number of bits per trait, as well as the values of the traits themselves.

Furthermore, the value of all the chromosome's traits may be set with the appropriate `init()` method:

`chromosome_init()` initializes a chromosome's discrete trait values to the input array. The trait values are in the range $[0, 2^{n_k} - 1]$ where n represents the number of bits in the k^{th} trait.

`chromosome_initf()` initializes a chromosome's continuous trait values to the input array. The trait values are in the range $[0, 1]$ and are represented by floating-point values. Because each trait has a discrete number of values (limited bit resolution), the value of the trait is quantized to its nearest representation.

`chromosome_init_random()` initializes a chromosome's trait values to a random number.

The values of specific traits can be retrieved using the `value()` methods. They are useful for evaluating the fitness of the chromosome in the search algorithm's callback function.

`chromosome_value()` returns the value of the k^{th} trait (integer representation).

`chromosome_valuef()` returns the value of the k^{th} trait (floating-point representation).

Finally the methods for use in the `gasearch` algorithm are described:

`chromosome_mutate()` flips the k^{th} bit of the chromosome.

`chromosome_crossover()` copies the first k bits of the first parent and the remaining bits of the second parent to the child chromosome.

20.2.2 Interface

Listed below is a description for the `gasearch` object in *liquid*.

`gasearch_create(*utility,*userdata,parent,min/max)` creates a `gasearch` object, initialized on the specified parent chromosome. The user-defined utility function and userdata structures define the search, as well as the `min/max` flag which can be either `LIQUID_OPTIM_MINIMIZE` or `LIQUID_OPTIM_MAXIMIZE`.

`gasearch_destroy(q)` destroys a `gasearch` object, freeing all internally-allocated memory.

`gasearch_print(q)` prints the internal state of the `gasearch` object

`gasearch_set_mutation_rate(q,rate)` sets the mutation rate

`gasearch_set_population_size(q,population,selection)` sets both the population size as well as the selection size of the evolutionary algorithm

`gasearch_run(q,n,target_utility)` runs multiple iterations of the search algorithm, stopping after either n iterations or if the target utility is met.

`gasearch_evolve(q)` steps through a sigle iteration of the search.

`gasearch_getopt(q,*chromosome,*u)` produces the best chromosome over the coarse of the search evolution, as well as its utility.

20.2.3 Example Code

An example of the `gasearch` interface is given below:

```
// file: doc/listings/gasearch.example.c
#include <liquid/liquid.h>

// user-defined utility callback function
float myutility(void * _userdata, chromosome _c)
{
    // compute utility from chromosome
    float u = 0.0f;
    unsigned int i;
    for (i=0; i<chromosome_get_num_traits(_c); i++)
        u += chromosome_valuef(_c,i);
    return u;
}

int main() {
    unsigned int num_parameters = 8;        // dimensionality of search (minimum 1)
    unsigned int num_iterations = 100;      // number of iterations to run
    float target_utility = 0.01f;           // target utility

    unsigned int bits_per_parameter = 16;   // chromosome parameter resolution
    unsigned int population_size = 100;     // GA population size
    float mutation_rate = 0.10f;            // GA mutation rate
```

```
23
24      // create prototype chromosome
25      chromosome prototype = chromosome_create_basic(num_parameters, bits_per_parameter);
26
27      // create gasearch object
28      gasearch ga = gasearch_create_advanced(
29                                      &myutility,
30                                      NULL,
31                                      prototype,
32                                      LIQUID_OPTIM_MINIMIZE,
33                                      population_size,
34                                      mutation_rate);
35
36      // execute batch search
37      gasearch_run(ga, num_iterations, target_utility);
38
39      // execute search one iteration at a time
40      unsigned int i;
41      for (i=0; i<num_iterations; i++)
42          gasearch_evolve(ga);
43
44      // clean up objects
45      chromosome_destroy(prototype);
46      gasearch_destroy(ga);
47  }
```

Evolutionary algorithms are well-suited for discrete optimization problems, particularly where a large number of parameters only hold a few values. The classic example is the knapsack problem (constrained, non-linear) in which the selection of items with different weights and values must be chosen to maximize the total value without exceeding a prescribed weight capacity. An example of using the `gasearch` object in *liquid* to search over the solution space of the knapsack problem can be found in the `examples` directory as `examples/gasearch_knapsack_example.c`.

21 random

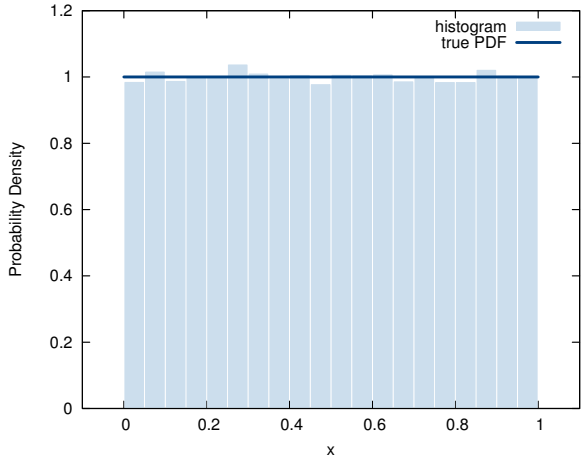
The random module in *liquid* includes a comprehensive set of random number generators useful for simulation of wireless communications channels, particularly for generating noise as well as fading channels. This includes the uniform, normal, circular (complex) Gaussian, Rice-K, and Weibull distributions.

21.1 Uniform

The uniform random variable generator in *liquid* simply generates a number evenly distributed in $[0, 1)$. Internally *liquid* uses the standard `rand()` method for generating random integers and then divides by `RAND_MAX`, the maximum number that can be generated. The probability density function is defined as

$$f_X(x) = \begin{cases} 1 & \text{if } 0 \le x < 1 \\ 0 & \text{else.} \end{cases} \tag{131}$$

The uniform random number generator is the basis for generating most other distributions in *liquid*.



Uniform random number generator interface:

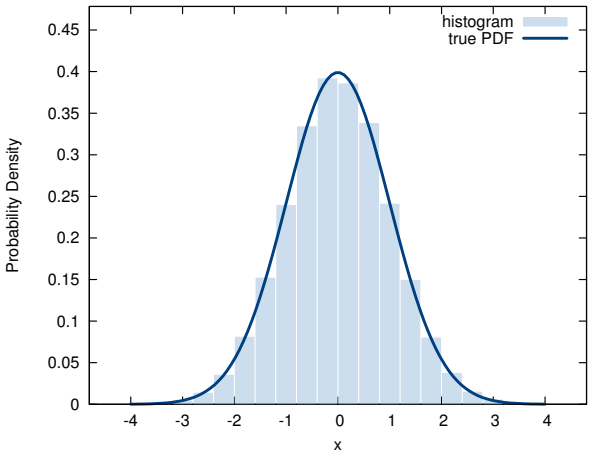
```
float randf();
float randf_pdf(float _x);
float randf_cdf(float _x);
```

21.2 Normal (Gaussian)

The normal (or Gauss) distribution has a probability density function defined as

$$f_X(x;\sigma,\eta) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\eta)^2/2\sigma^2} \tag{132}$$

liquid generates normal random variables using the Box-Muller method. If U_1 and U_2 are uniform random variables with a distribution defined by (131), then $X_1 = \sqrt{-2\ln(U_1)}\sin(2\pi U_2)$ and $X_2 = \sqrt{-2\ln(U_1)}\cos(2\pi U_2)$ are independent normal random variables with a mean of zero and a unity standard deviation ($X_1, X_2 \sim N(0,1)$).



Normal (Gauss) random number generator interface:

```
float randnf();
float randnf_pdf(float _x,
                 float _eta,
                 float _sigma);
float randnf_cdf(float _x,
                 float _eta,
                 float _sigma);
```

21.3 Exponential

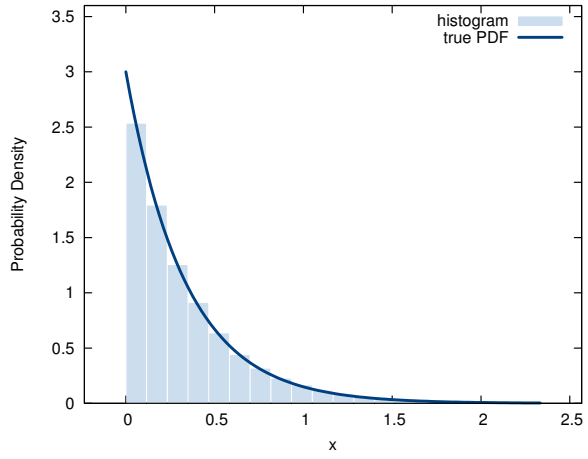
The exponential distribution has a probability density function defined as

$$f_X(x;\lambda) = \lambda e^{-\lambda x} \tag{133}$$

liquid generates exponential random variables by inverting the cumulative distribution function, viz

$$F_X(x;\lambda) = 1 - e^{-\lambda x} \tag{134}$$

Specifically if U is uniform random variable with a distribution defined by (131) then $X = -\ln U/\lambda$ has an exponential distribution defined by (134).



Exponential random number generator interface:

```
float randexpf(float _lambda);
float randexpf_pdf(float _x, float _lambda);
float randexpf_cdf(float _x, float _lambda);
```

21.4 Weibull

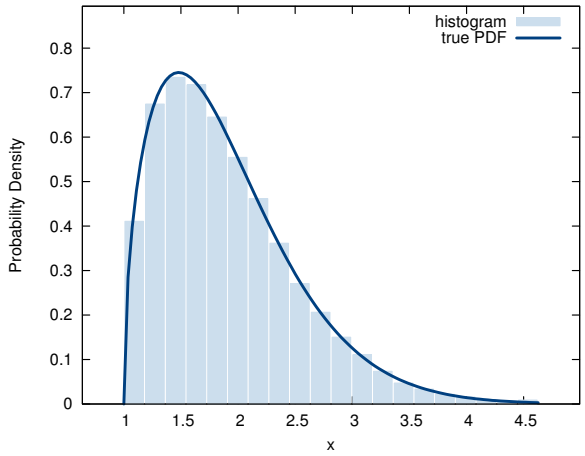
The Weibull distrubtion has a probability denisty function defined by

$$f_X(x;\alpha,\beta,\gamma) = \begin{cases} \frac{\alpha}{\beta}\left(\frac{x-\gamma}{\beta}\right)^{\alpha-1} \exp\left\{-\left(\frac{x-\gamma}{\beta}\right)^{\alpha}\right\} & x \geq \gamma \\ 0 & \text{else.} \end{cases} \tag{135}$$

where α is the shape parameter, β is the scale parameter, and γ is the threshold parameter. *liquid* generates Weibull random variables by inverting the cumulative ditribution function, viz

$$F_X(x;\alpha,\beta,\gamma) = \begin{cases} 1 - \exp\left\{-\left(\frac{x-\gamma}{\beta}\right)^{\alpha}\right\} & x \geq \gamma \\ 0 & \text{else.} \end{cases} \tag{136}$$

Specifically if U is uniform random variable with a distribution defined by (131) then $X = \gamma + \beta\left[\ln\left(1-U\right)\right]^{1/\alpha}$ has a Weibull distribution defined by (136).



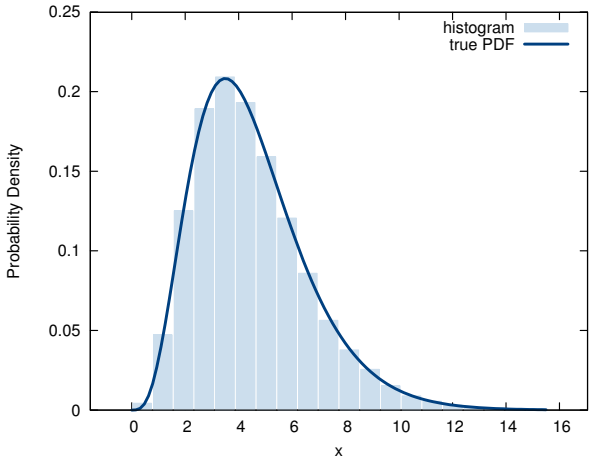
Weibull random number generator interface:

```
float randweibf(float _alpha,
                float _beta,
                float _gamma);
float randweibf_pdf(float _x,
                    float _alpha,
                    float _beta,
                    float _gamma);
float randweibf_cdf(float _x,
                    float _alpha,
                    float _beta,
                    float _gamma);
```

21.5 Gamma

The gamma distribution has a probability density function defined by

$$f_X(x;\alpha,\beta) = \begin{cases} \frac{x^{\alpha-1}}{\Gamma(\alpha)\beta^{\alpha}} e^{-x/\beta} & x \geq 0 \\ 0 & \text{else.} \end{cases} \tag{137}$$



Gamma random number generator interface:

```
float randgammaf(float _alpha,
                 float _beta);
float randgammaf_pdf(float _x,
                     float _alpha,
                     float _beta);
float randgammaf_cdf(float _x,
                     float _alpha,
                     float _beta);
```

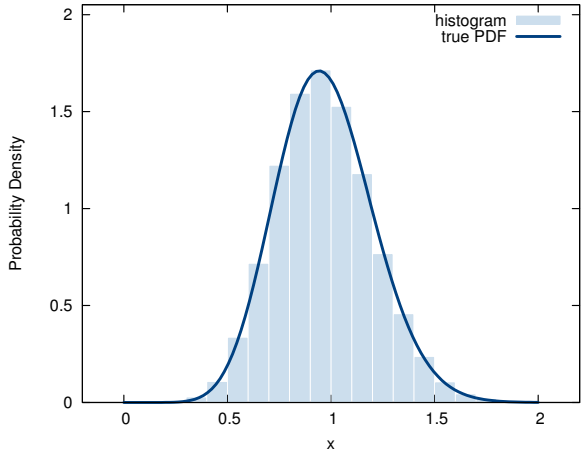
21.6 Nakagami-m

The Nakagami-m distribution is a versatile stochastic model for modeling radio links [5] and has often been regarded as the best distribution to model land mobile propagation due to its ability

to describe fading situations worse than Rayleigh, including one-sided Gaussian [27]. Empirical evidence regarding the efficacy the Nakagami-m distribution has on fading profiles been presented in [29, 28]. Thus statistical inference of the Nakagami-m fading parameters are of interest in the design of adaptive radios such as optimized transmit diversity modes [6, 14] and adaptive modulation schemes [26]. The Nakagami-m probability density function is given by [22]

$$f_X(x;m,\Omega) = \begin{cases} \frac{2}{\Gamma(m)}\left(\frac{m}{\Omega}\right)^m x^{2m-1}e^{-(m/\Omega)x^2} & x \geq 0 \\ 0 & \text{else.} \end{cases} \tag{138}$$

where $m \geq 1/2$ is the shape parameter and $\Omega > 0$ is the spread parameter. Nakagami-m random numbers are generated from the gamma distribution. Specifically if R follows a gamma distribution defined by (137) with parameters α and β, then $X = \sqrt{R}$ has a Nakagami-m distribution with $m = \alpha$ and $\Omega = \beta/\alpha$.



Nakagami random number generator interface:

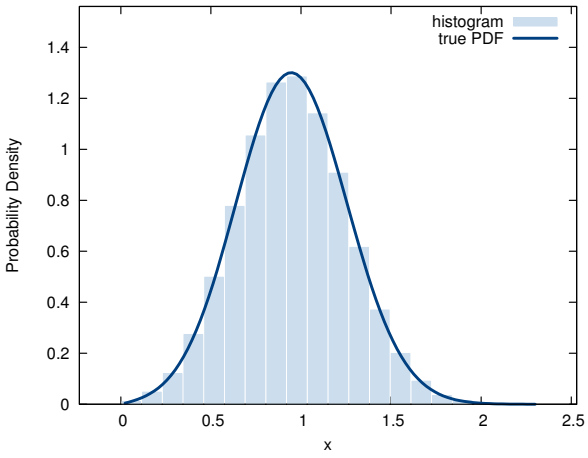
```
float randnakmf(float _m,
                float _omega);
float randnakmf_pdf(float _x,
                    float _m,
                    float _omega);
float randnakmf_cdf(float _x,
                    float _m,
                    float _omega);
```

21.7 Rice-K

The Rice-K multipath channel models a fading envelope by assuming a line of sight (LoS) component to the multipath elements summed at the receiver. The complex path gain at a particular frequency consists of a fixed (LoS) and fluctuating (diffuse) components. When assuming a narrowband complex Gaussian stochasitc process, the time-varying envelope will exhibit a Rice distribution where the K factor is the power ratio of the LoS and diffuse components (often referred to in dB) and thus is commonly used to describe fading environments. The Rice-K distribution has a probability density function defined as

$$f_R(r;K,\Omega) = \frac{2(K+1)r}{\Omega}\exp\left\{-K - \frac{(K+1)r^2}{\Omega}\right\} I_0\left(2r\sqrt{\frac{K(K+1)}{\Omega}}\right) \tag{139}$$

where $\Omega = E\left\{R^2\right\}$ is the average signal power and K is the fading factor (shape parameter). *liquid* generates Rice-K random variables using two independent normal random variables. Specifically if $X_0 \sim N(0,\sigma)$ and $X_1 \sim N(s,\sigma)$ then $R = \sqrt{X_0^2 + X_1^2}$ has follows a Rice-K distribution defined by (139) where $s = \sqrt{\frac{\Omega K}{K+1}}$ and $\sigma = \sqrt{\frac{\Omega}{2(K+1)}}$.



Nakagami random number generator interface:

```
float randricekf(float _m,
                 float _omega);
float randricekf_pdf(float _x,
                     float _K,
                     float _omega);
float randricekf_cdf(float _x,
                     float _K,
                     float _omega);
```

21.8 Data scrambler

Physical layer synchronization of received waveforms relies on independent and identically distributed underlying data symbols. If the message sequence, however, is too repetitive (such as '`00000....`' or '`11001100....`') and the modulation scheme is BPSK, the synchronizer probably won't be able to recover the symbol timing because adjacent symbols are too similar. This is a result of spectral correlation introduced which can prevent physical layer synchronization techniques from tracking or even acquisition. Having said that, certain patterns *are* beneficial to synchronization and actually help the receiver track to the incoming signal, however these are usually only introduced as a preamble to a frame or packet where the receiver knows what to expect. It is therefore imperative to increase the short-term entropy of the underlying data to prevent the receiver from losing its lock on the signal. The data scrambler routine attempts to "whiten" the data sequence with a bit mask in order to achieve maximum entropy.

21.8.1 interface

The data scrambler has two methods, described here:

`scramble_data()` takes an input sequence of data and scrambles the bits by applying a periodic mask. The first argument is a pointer to the input data array; the second argument is its length (number of bytes).

`unscramble_data()` takes an input sequence of data and unscrambles the bits by applying the reverse mask applied by `scramble_data()`. Just like `scramble_data()`, the first argument is a pointer to the input data array; the second argument is its length (number of bytes).

See `examples/scramble_example.c` for a full example of the interface.

22 sequence

The sequence module implements a number of binary sequencing objects useful for communications, including generic binary shift registers, linear feedback shift registers, maximal length codes (m-sequences), and complementary codes.

22.1 bsequence, generic binary sequence

The `bsequence` object implmements a generic binary shift register and is particularly useful in wireless communications for correlating long bit sequences in seeking frame preambles and packet headers. The `bsequence` object internally stores its sequence of bits as an array of bytes which handles shifting values even faster than the `window` family of objects. Listed below is the basic interface to the `bseqeunce` object:

`bsequence_create(n)` creates a `bsequence` object with n bits, filled initially with zeros.

`bsequence_destroy(q)` destroys the object, freeing all internally-allocated memory.

`bsequence_clear(q)` resets the sequence to all zeros.

`bsequence_init(q,*v)` initializes the sequence on an external array of bytes, compactly representing a string of bits.

`bsequence_print(q)` prints the contents of the sequence to the screen.

`bsequence_push(q,bit)` pushes a bit into the back (right side) of a binary sequence, and in turn drops the left-most bit. Only the right-most (least-significant) bit of the input is regarded. For example, pushing a `1` into the sequence `0010011` results in `0100111`.

`bsequence_circshift(q)` circularly shifts a binary sequence left, pushing the left-most bit back into the right-most position. For example, invoking a circular shift on the sequence `1001110` results in `0011101`.

`bsequence_correlate(q0,q1)` runs a binary correlation of two `bsequence` objects `q0` and `q1`, returning the number of similar bits in both sequences. For example, correlating the sequence `11110000` with `11001100` yields `4`.

`bsequence_add(q0,q1,q2)` computes the binary addition of two sequences `q0` and `q1` storing the result in a third sequence `q2`. Binary addition of two bits is equivalent to their logical *exclusive or*, $\oplus$. For example, the binary addition of `01100011` and `11011001` is `10111010`.

`bsequence_mul(q0,q1,q2)` computes the binary multiplication of two sequences `q0` and `q1` storing the result in a third sequence `q2`. Binary multiplication of two bits is equivalent to their logical *and*, $\wedge$. For example, the binary multiplication of `01100011` and `11011001` is `01000001`.

`bsequence_accumulate(q)` returns the `1`s in a binary sequence.

`bsequence_get_length(q)` returns the length of the sequence (number of bits).

`bsequence_index(q,i)` returns the bit at a particular index of the sequence, starting from the right-most bit. For example, indexing the sequence `00000001` at index `0` gives the value `1`.

Table 9: Default m-sequence generator polynomials in *liquid*

m	n	g (hex)	g (octal)	g (binary)
2	3	0x0007	000007	111
3	7	0x000b	000013	1011
4	15	0x0013	000023	10011
5	31	0x0025	000045	100101
6	63	0x0043	000103	1000011
7	127	0x0089	000211	10001001
8	255	0x012d	000455	100101101
9	511	0x0211	001021	1000010001
10	1023	0x0409	002011	10000001001
11	2047	0x0805	004005	100000000101
12	4095	0x1053	010123	1000001010011
13	8191	0x201b	020033	10000000011011
14	16383	0x402b	040053	100000000101011
15	32767	0x8003	100003	1000000000000011

22.2 msequence, m-sequence (linear feedback shift register)

The `msequence` object in *liquid* is really just a linear feedback shift register (LFSR), efficiently implemented using unsigned integers. The LFSR consists of an m-bit shift register, v, and generator polynomial g. For primitive polynomials, the output sequence has a length $n = 2^m - 1$ before repeating. This sequence is known as a maximal-length P/N (positive/negative) sequence, and consists of several useful properties:

1. the output sequence has very good auto-correlation properties; when aligned, the sequence, of course, correlates perfectly to 1. When misaligned by any amount, however, the sequence correlates to exactly $-1/n$.

2. the sequence is easily generated using a linear feedback shift register

Only a certain subset of all possible generator polynomials produce this maximal length sequence. The default generator polynomials are listed in Table 9, however many more exist.[10] Notice that both the first and last bit of each generator polynomial is a `1`. This holds true for all m-sequence generator polynomials. All generator polymomials of length $m = 2$ ($n = 3$) through $m = 15$ ($n = 32767$) are given in the `data/msequence/` subdirectory of this documentation directory.

Here is a brief description of the `msequence` object's interface in *liquid*:

`msequence_create(m,g,a)` creates an `msequence` object with an internal shift register length of m bits using a generator polynomial g and the initial state of the shift register a.

`msequence_create_default(m)` creates an `msequence` object with m bits in the shift register using the default generator polynomial (e.g. `LIQUID_MSEQUENCE_GENPOLY_M6`). The initial state is set to `000...001`.

[10] A list of all m-sequence generator polynomials are provided in `doc/data/msequence` located in the main *liquid* project directory.

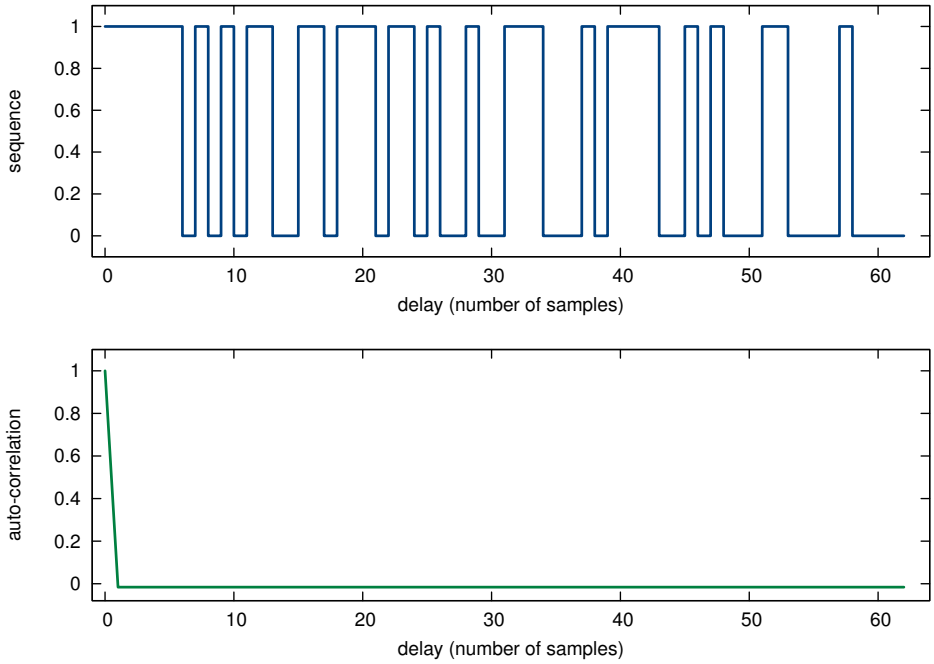


Figure 37: `msequence` autocorrelation, $m = 6$ ($n = 63$), $g =$`1000011`

`msequence_destroy(ms)` destroys the object `ms`, freeing all internal memory.

`msequence_print(ms)` prints the contents of the sequence to the screen.

`msequence_advance(ms)` advances the `msequence` object's shift register by computing the binary dot product of the register with the generator polynomial. The resulting bit is sum of `1`s modulo 2 of the dot product and is fed back into the end of the shift register, as well as returned to the user.

`msequence_generate_symbol(ms,bps)` generates a pseudo-random `bps`-bit symbol from the shift register. This is accomplished by advancing the `msequence` object `bps` times and shifting the result back into the symbol. It is important to note that because the sequence repeats every n bits, if the random number is an even multiple of n, the random sequence will repeat every `bps` symbols. For example, if $m = 4$ ($n = 15$) and `_bps` is 3, then the sequence will repeat 5 times.

`msequence_reset(ms)` resets the `msequence` object's internal shift register to the original state (typically `000...001`).

`msequence_get_length(ms)` returns the length of the sequence, n

`msequence_get_state(ms)` returns the internal state of the sequence, v

The auto-correlation of the m-sequence with generator polynomial $g =$`1000011` can be seen in Figure 37. The shift register has six bits ($m = 6$) and therefore the output sequence is of length

Table 10: Default complementary codes in *liquid*

1	$\boldsymbol{a}_0 =$ `1` $\boldsymbol{b}_0 =$ `0`
2	$\boldsymbol{a}_1 =$ `10` $\boldsymbol{b}_1 =$ `11`
4	$\boldsymbol{a}_2 =$ `1011` $\boldsymbol{b}_2 =$ `1000`
8	$\boldsymbol{a}_3 =$ `10111000` $\boldsymbol{b}_3 =$ `10110111`
16	$\boldsymbol{a}_4 =$ `10111000 10110111` $\boldsymbol{b}_4 =$ `10111000 01001000`
32	$\boldsymbol{a}_5 =$ `10111000 10110111 10111000 01001000` $\boldsymbol{b}_5 =$ `10111000 10110111 01000111 10110111`
64	$\boldsymbol{a}_6 =$ `10111000 10110111 10111000 01001000 10111000 10110111 01000111 10110111` $\boldsymbol{b}_6 =$ `10111000 10110111 10111000 01001000 01000111 01001000 10111000 01001000`

$n = 2^m - 1 = 63$. Notice that the autocorrelation is equal to unity with no delay, and nearly zero ($-1/63$) for all other delays.

22.3 complementary codes

In addition to m-sequences, *liquid* also implements complementary codes: P/N sequence pairs which have similar properties to m-sequences. A complementary code pair is one in which the sum of individual auto-correlations is identically zero for all delays except for the zero-delay which provides an auto-correlation of unity. The two codes $\boldsymbol{a}$ and $\boldsymbol{b}$ are generated recursively as

$$\begin{aligned} \boldsymbol{a}_{k+1} &= \left[\boldsymbol{a}_k \;\; \boldsymbol{b}_k\right] \\ \boldsymbol{b}_{k+1} &= \left[\boldsymbol{a}_k \;\; \bar{\boldsymbol{b}}_k\right] \end{aligned}$$

where $[\cdot,\cdot]$ represents concatenation and $\bar{(\cdot)}$ denotes a binary inversion. Table 10 shows the first several iterations of the sequence. Notice that the sequence length doubles for each iteration, and that (with the exception of $k = 0$) the first half of $\boldsymbol{a}_k$ and $\boldsymbol{b}_k$ are identical. Figure 38 shows that the auto-correlation of the two sequences is non-zero for delays other than zero, but that they indeed do sum to zero.

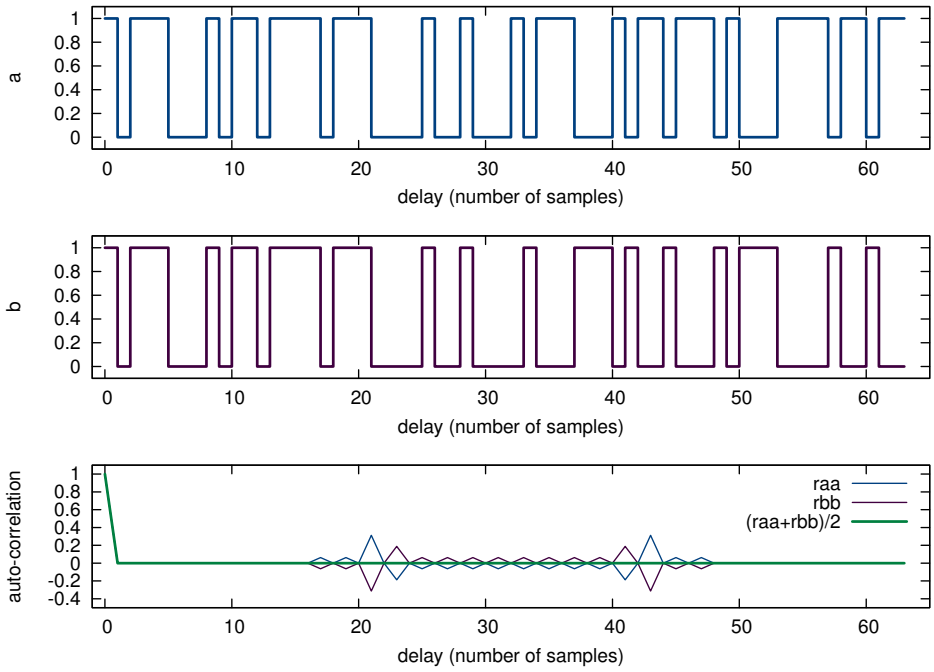


Figure 38: Complementary codes autocorrelation, $n = 64$

23 utility

The utility module contains useful functions, primarily for bit fast bit manipulation. This includes packing/unpacking byte arrays, counting ones in an integer, computing a binary dot-product, and others.

23.1 `liquid_pack_bytes()`, `liquid_unpack_bytes()`, and `liquid_repack_bytes()`

Byte packing is used extensively in the `fec` (Section 12) and `framing` (Section 15) modules. These methods resize symbols represented by various numbers of bits. This is necessary to move between raw data arrays which use full bytes (eight bits per symbol) to methods expecting symbols of different sizes. In particular, the `liquid_repack_bytes()` method is useful when one wants to transmit a block of 64 data bytes using an 8-PSK modem which requires a 3-bit input symbol. For example repacking two 8-bit symbols `00000000,11111111` into six 3-bit symbols gives `000,000,001,111,111,100`. Because 16 bits cannot be divided evenly among 3-bit symbols, the last symbol is padded with zeros.

23.2 `liquid_pack_array()`, `liquid_unpack_array()`

The `liquid_pack_array()` and `liquid_unpack_array()` methods pack an array with symbols of arbitrary length. These methods are similar to those in Section 23.1 but are capable of packing symbols of any arbitrary length. These are convenient for digital modulation and demodulation of a block of symbols with different modulation schemes. For example packing an array with five

symbols 1000,011,11010,1,000 yields two bytes: 10000111,10101000. Here are the basic interfaces for packing and unpacking arrays:

```
// pack binary array with symbol(s)
void liquid_pack_array(unsigned char * _src,        // source array [size: _n x 1]
                       unsigned int _n,             // input source array length
                       unsigned int _k,             // bit index to write in _src
                       unsigned int _b,             // number of bits in input symbol
                       unsigned char _sym_in);      // input symbol

// unpack symbols from binary array
void liquid_unpack_array(unsigned char * _src,      // source array [size: _n x 1]
                         unsigned int _n,           // input source array length
                         unsigned int _k,           // bit index to write in _src
                         unsigned int _b,           // number of bits in output symbol
                         unsigned char * _sym_out); // output symbol
```

Listed below is a simple example of packing symbols of varying lengths into a fixed array of bytes;

```
1  // file: doc/listings/pack_array.example.c
2  #include <liquid/liquid.h>
3
4  int main() {
5      unsigned int sym_size[9] = {8, 2, 3, 6, 1, 3, 3, 4, 3};
6      unsigned char input[9] = {
7          0x81,   // 1000 0001
8          0x03,   //        11
9          0x05,   //       101
10         0x3a,   //   11 1010
11         0x01,   //         1
12         0x07,   //       111
13         0x06,   //       110
14         0x0a,   //      1010
15         0x04    //     10[0] <- last bit is stripped
16     };
17
18     unsigned char output[4];
19
20     unsigned int k=0;
21     unsigned int i;
22     for (i=0; i<9; i++) {
23         liquid_pack_array(output, 4, k, sym_size[i], input[i]);
24         k += sym_size[i];
25     }
26     // output       : 1000 0001 1110 1111 0101 1111 1010 1010
27     // symbol       : 0000 0000 1122 2333 3334 5556 6677 7788
28     // output is now {0x81, 0xEF, 0x5F, 0xAA};
29 }
```

23.3 `liquid_lbshift()`, `liquid_rbshift()`

Binary shifting.

`liquid_lbshift()`

```
// input        : 1000 0001 1110 1111 0101 1111 1010 1010
// output [0]   : 1000 0001 1110 1111 0101 1111 1010 1010
// output [1]   : 0000 0011 1101 1110 1011 1111 0101 0100
// output [2]   : 0000 0111 1011 1101 0111 1110 1010 1000
// output [3]   : 0000 1111 0111 1010 1111 1101 0101 0000
// output [4]   : 0001 1110 1111 0101 1111 1010 1010 0000
// output [5]   : 0011 1101 1110 1011 1111 0101 0100 0000
// output [6]   : 0111 1011 1101 0111 1110 1010 1000 0000
// output [7]   : 1111 0111 1010 1111 1101 0101 0000 0000
```

`liquid_rbshift()`

```
// input        : 1000 0001 1110 1111 0101 1111 1010 1010
// output [0]   : 1000 0001 1110 1111 0101 1111 1010 1010
// output [1]   : 0100 0000 1111 0111 1010 1111 1101 0101
// output [2]   : 0010 0000 0111 1011 1101 0111 1110 1010
// output [3]   : 0001 0000 0011 1101 1110 1011 1111 0101
// output [4]   : 0000 1000 0001 1110 1111 0101 1111 1010
// output [5]   : 0000 0100 0000 1111 0111 1010 1111 1101
// output [6]   : 0000 0010 0000 0111 1011 1101 0111 1110
// output [7]   : 0000 0001 0000 0011 1101 1110 1011 1111
```

23.4 `liquid_lbcircshift()`, `liquid_rbcircshift()`

Binary circular shifting.

`liquid_lbcircshift()`

```
// input        : 1001 0001 1110 1111 0101 1111 1010 1010
// output [0]   : 1001 0001 1110 1111 0101 1111 1010 1010
// output [1]   : 0010 0011 1101 1110 1011 1111 0101 0101
// output [2]   : 0100 0111 1011 1101 0111 1110 1010 1010
// output [3]   : 1000 1111 0111 1010 1111 1101 0101 0100
// output [4]   : 0001 1110 1111 0101 1111 1010 1010 1001
// output [5]   : 0011 1101 1110 1011 1111 0101 0101 0010
// output [6]   : 0111 1011 1101 0111 1110 1010 1010 0100
// output [7]   : 1111 0111 1010 1111 1101 0101 0100 1000
```

`liquid_rbcircshift()`

```
// input        : 1001 0001 1110 1111 0101 1111 1010 1010
// output [0]   : 1001 0001 1110 1111 0101 1111 1010 1010
// output [1]   : 0100 1000 1111 0111 1010 1111 1101 0101
// output [2]   : 1010 0100 0111 1011 1101 0111 1110 1010
// output [3]   : 0101 0010 0011 1101 1110 1011 1111 0101
// output [4]   : 1010 1001 0001 1110 1111 0101 1111 1010
// output [5]   : 0101 0100 1000 1111 0111 1010 1111 1101
// output [6]   : 1010 1010 0100 0111 1011 1101 0111 1110
// output [7]   : 0101 0101 0010 0011 1101 1110 1011 1111
```

23.5 `liquid_lshift()`, `liquid_rshift()`

Byte-wise shifting.

`liquid_lshift()`

```
// input        : 1000 0001 1110 1111 0101 1111 1010 1010
// output [0]   : 1000 0001 1110 1111 0101 1111 1010 1010
// output [1]   : 1110 1111 0101 1111 1010 1010 0000 0000
// output [2]   : 0101 1111 1010 1010 0000 0000 0000 0000
// output [3]   : 1010 1010 0000 0000 0000 0000 0000 0000
// output [4]   : 0000 0000 0000 0000 0000 0000 0000 0000
```

`liquid_rshift()`

```
// input        : 1000 0001 1110 1111 0101 1111 1010 1010
// output [0]   : 1000 0001 1110 1111 0101 1111 1010 1010
// output [1]   : 0000 0000 1000 0001 1110 1111 0101 1111
// output [2]   : 0000 0000 0000 0000 1000 0001 1110 1111
// output [3]   : 0000 0000 0000 0000 0000 0000 1000 0001
// output [4]   : 0000 0000 0000 0000 0000 0000 0000 0000
```

23.6 `liquid_lcircshift()`, `liquid_rcircshift()`

Byte-wise circular shifting.

`liquid_lcircshift()`

```
// input        : 1000 0001 1110 1111 0101 1111 1010 1010
// output [0]   : 1000 0001 1110 1111 0101 1111 1010 1010
// output [1]   : 1110 1111 0101 1111 1010 1010 1000 0001
// output [2]   : 0101 1111 1010 1010 1000 0001 1110 1111
// output [3]   : 1010 1010 1000 0001 1110 1111 0101 1111
// output [4]   : 1000 0001 1110 1111 0101 1111 1010 1010
```

`liquid_rcircshift()`

```
// input        : 1000 0001 1110 1111 0101 1111 1010 1010
// output [0]   : 1000 0001 1110 1111 0101 1111 1010 1010
// output [1]   : 1010 1010 1000 0001 1110 1111 0101 1111
// output [2]   : 0101 1111 1010 1010 1000 0001 1110 1111
// output [3]   : 1110 1111 0101 1111 1010 1010 1000 0001
// output [4]   : 1000 0001 1110 1111 0101 1111 1010 1010
```

23.7 miscellany

This section describes the bit-counting methods which are used extensively throughout *liquid*, particularly the `fec` (Section 12) and `sequence` (Section 22) modules. Integer sizes vary for different machines; when *liquid* is initially configured (see Chapter 25), the size of the integer is computed such that the fastest method can be computed without performing unnecessary loop iterations or comparisons.

`liquid_count_ones(x)` counts the number of `1`s that exist in the integer x. For example, the number `237` is represented in binary as `11101101`, therefore `liquid_count_ones(237)` returns `6`.

`liquid_count_ones_mod2(x)` counts the number of `1`s that exist in the integer x, modulo 2; in other words, it returns `1` if the number of ones in x is odd, `0` if the number is even. For example, `liquid_count_ones_mod2(237)` return `0`.

`liquid_bdotprod(x,y)` computes the binary dot-product between two itegers x and y as the sum of ones in $x \wedge y$, modulo 2 (where $\wedge$ is the logical 'and' operation). This is useful in linear feedback shift registers (see Section 22.2 on m-sequences) as well as certain forward error-correction codes (see Section 12.2 on Hamming codes). For example, the binary dot product between `10110011` and `11101110` is `1` because `10110011` $\wedge$ `11101110` = `10100010` which has an odd number of `1`s.

`liquid_count_leading_zeros(x)` counts the number of leading zeros in the integer x. This is dependent upon the size of the integer for the target machine which is usually either two or four bytes.

`liquid_msb_index(x)` computes the index of the most-significant bit in the integer x. The function will return `0` for $x = 0$. For example if $x = 129$ (`10000001`), the function will return `8`.

24 experimental

The experimental module is a placeholder for modules which haven't yet been approved for release, but might eventually be incorporated into the library. By default the `experimental` module is disabled and none of its modules are compiled or installed. It is enabled using the `configure` flag `--enable-experimental` and includes the internal header file `include/liquid.experimental.h`.

24.1 `fbasc` (filterbank audio synthesizer codec)

The fbasc audio codec implements an AAC-like compression algorithm, using the modified discrete cosine transform as a loss-less channelizer. The resulting channelized data are then quantized based on their spectral energy levels and then packed into a frame which the decoder can then interpret. The result is a lossy encoder (as a result of quantization) whose compression/quality levels can be easily varied.

Specifically, fbasc uses sub-band coding to allocate quantization bits to each channel in order to minimize distortion of the reconstructed signal. Sub-bands with higher variance (signal 'energy') are assigned more bits. This is the heart of the codec, which exploits several components typical of audio signals and aspects of human hearing and perception:

1. The majority of audio signals (including music and voice) have a strong time-frequency localization; that is, they only occupy a small fraction of audible frequencies for a short duration. This is particularly true for voiced signals (e.g. vowel sounds).

2. The human ear (and brain) tends to be quite forgiving of spectral compression and often cannot easily distinguish between neighboring frequency components.

There are several benefits to using fbasc over other compression algorithms such as CVSD (see src/audio/readme.cvsd.txt) and auto-regressive models, the main being that the algorithm is theoretically lossless (i.e. perfect reconstruction) as the bit rate increases. As a result, the codec is limited only by the quantization noise on each channel.

Here are some useful definitions, as used in the fbasc code:

MDCT the modified discrete cosine transform is a lapped discrete cosine transform which uses a special windowing function to ensure perfect reconstruction on its inverse. The transform operates on $2M$ time-domain samples (overlapped by M) to produce M frequency-domain samples. Conversely, the inverse MDCT accepts M frequency-domain samples and produces $2M$ time-domain samples which are windowed and then overlapped to reconstruct the original signal. For convenience, we may refer to M time-domain samples as a 'symbol.'

symbol one block of M time-domain samples upon which the MDCT operates.

channel one of the M frequency-domain components as a result of applying the MDCT. This is somewhat equivalent to a discrete Fourier transform 'bin.' Note than M is equal to the number of channels in analysis.

frame a set of MDCT symbols upon which the fbasc codec runs its analysis. Because the codec uses time-frequency localization for its encoding, it is necessary for the codec to gain enough statistical information about the original signal without losing temporal stationarity. The

codec typically operates on several symbols, however, the exact number depends on the application.

24.1.1 Interface

`fbasc_create()` creates an fbasc encoder/decoder object, allocating memory as necessary, and computing internal parameters appropriately.

`fbasc_destroy()` destroys an fbasc encoder/decoder object, freeing internally-allocated memory.

`fbasc_encode()` encode a frame of data, storing the header and frame data separately. This separation allows the user to use different forward error-correction codes (if desired) to protect the header differently than the rest of the frame. It is important to keep the two together, however, as the header is a description of how to decode the frame.

`fbasc_decode()` decodes a frame of data, generating the reconstructed time series.

24.1.2 Useful properties

- Because of the nature of the MDCT, frames will overlap by M samples (one symbol). This introduces a reconstruction delay of M samples, noticeable at the decoder.

24.2 gport

The `gport` object implements a generic port to share data between asynchronous threads. The port itself is really just a circular (ring) buffer containing a mutually-exclusive locking mechanism to allow processes running on independent threads to access its data. Because no other modules rely on the `gport` object and because it requires the pthread library, it is likely to be removed from *liquid* in the near future and likely put into another library, e.g. *liquid-threads*.

There are two ways to access the data in the `gport` object: direct memory access and indirect (copied) memory acess, each with distinct advantages and distadvantages. Regardless of which interface you use, the model is equivalent: a buffer of data (initially empty) is created. The *producer* is the method in charge of writing to the buffer (or "producing" the data). The *consumer* is the method in charge of reading the data from the buffer (or "consuming" it). The producer and consumer methods can exist on completely separate threads, and do not need to be externally synchronized. The `gport` object synchronizes the data between the ports.

24.2.1 Direct Memory Access

Using `gport` via direct memory access is a multi-step process, equivalent for both the producer and consumer threads. For the sake of simplicity, we will describe the process for writing data to the port on the producer side; the consumer process is identical.

1. the producer requests a lock on the port of a certain number of samples.

2. once the request is serviced, the port returns a pointer to an array of data allocated internally by the port itself.

3. the producer writes its data at this location, not exceeding the original number of samples requested.

4. the producer then unlocks the port, indicating how many samples were actually written to the buffer. This allows the consumer thread to read data from the buffer.

5. this process is repeated as necessary.

Listed below is a minimal example demonstrating the direct memory access method for the `gport` object.

```
// file: doc/listings/gport.direct.example.c
#include <liquid/liquid.h>

int main() {
    // create the port
    //     size :   1024
    //     type :   int
    gport p = gport_create(1024,sizeof(int));

    // producer requests 256 samples (blocking)
    int * w;
    w = (int*) gport_producer_lock(p,256);

    // producer writes data to w here

    // once data are written, producer unlocks the port
    gport_producer_unlock(p,256);

    // repeat as necessary

    // destroy the port object
    gport_destroy(p);
}
```

24.2.2 Indirect/Copied Memory Access

Indirect (or "copied") memory access appears similar...

```
// file: doc/listings/gport.indirect.example.c
#include <liquid/liquid.h>

int main() {
    // create the port
    //     size :   1024
    //     type :   int
    gport p = gport_create(1024,sizeof(int));

    // create buffer for writing
    int w[256];

```

```
13      // producer writes data to w here
14
15      // producer writes 256 values to port
16      gport_produce(p,(void*)w,256);
17
18      // repeat as necessary
19
20      // destroy the port object
21      gport_destroy(p);
22  }
```

24.2.3 Key differences between memory access modes

While the direct memory access method provides a simpler interface–in the sense that no external buffers are required–the user must take care in not writing outside the bounds of the memory requested. That is, if 256 samples are locked, only 256 values are available. Writing more data will produce unexpected results, and could likely result in a segmentation fault. Furthermore, the buffer must wait until the entire requested block is available before returning. This could potentially increase the amount of time that each process is waiting on the port. Additionally, if one requests too many samples on both the producer and consumer sides, the port could wait forever. For example, assume one initially creates a `gport` with 100 elements and the producer initially writes 30 samples. Immediately following, the consumer requests a lock for 100 samples which isn't serviced because only 30 are available. Following that, the producer requests a lock for 100 samples which isn't serviced because only 70 are available. This is a deadlock condition where both threads are waiting for data, and neither request will be serviced. The solution to this problem is actually fairly simple; the port should be initially created as the sum of maximum size of the producer's and consumer's requests. That is, if the producer will at most ever request a lock on 50 samples and the consumer will at most request a lock of 70 samples, then the port should be initially created with a buffer size of 120. This guarantees that the deadlock condition will never occur.

Alternatively one may use the indirect memory access method which guarantees that the deadlock condition will never occur, even if the buffer size is 1 and the producer writes 1000 samples while the consumer reads 1000. This is because both the internal producer and consumer methods will write the data as it becomes available, and do not have to wait internally until an entire block of the requested size is ready. This is the benefit of using the indirect memory access interface of the `gport` object. Indirect memory access, however, requires the use of memory allocated externally to the port.

It is important to stay consistent with the memory access mode used within a thread, however mixed memory access modes can be used between threads on the same port. For example, the producer thread may use the direct memory access mode while the consumer uses the indrect memory acess mode.

24.2.4 Interface

`gport_create()` creates a new `gport` object with an internal buffer of a certain length.

`gport_destroy()` destroys a `gport` object, signaling *end of message* to any connected ports.

`gport_producer_lock()` locks a requested number of samples for producing, returning a `void` pointer to the locked data array directly. Invoking this method can be thought of as asking the port to allocate a certain number of samples for writing. Special care must be taken by the user not to write more elements to the buffer than were requested. This function is a blocking call and waits for the data to become available or an *end of message* signal to be received. The data are locked until `gport_producer_unlock()` is invoked. The number of unlocked samples does not have to match but cannot exceed those which are locked.

`gport_producer_unlock()` unlocks a requested number of samples from the port. Use in conjunction with `gport_producer_lock()`. Invoking this method can be thought of as telling the port "I have written n samples to the buffer you gave me earlier; release them to the consumer for reading." The number of samples written to the port cannot exceed the initial request (e.g. if you request a lock for 100 samples, you should never try to unlock more than 100). There is no internal error-checking to ensure this. Failure to comply could result in over-writing data internally, and corrupt the consumer side.

`gport_produce()` produces n samples to the port from an external buffer. This method is a blocking call and waits for the requested data to become available or an *end of message* signal to be received.

`gport_produce_available()` operates just like `gport_produce()` except will write as many samples as are available when the function is called. Invoking this method is like telling the buffer "I have n samples, so write as many as you can right now." It will always wait for at least one sample to become available and blocks until this condition is met.

`gport_consumer_lock()` locks a requested number of samples for consuming, returning a `void` pointer to the locked data array directly. Invoking this method can be thought of as asking the port to wait for a certain number of samples to be read. Special care must be taken by the user not to read more elements to the buffer than were requested. This function is a blocking call and waits for enough samples to become available or an *end of message* signal to be received. The data will be locked until `gport_consumer_unlock()` is invoked. The number of unlocked samples does not have to match but cannot exceed those which are locked.

`gport_consumer_unlock()` unlocks a requested number of samples from the port. Use in conjunction with `gport_consumer_lock()`. Invoking this method can be though of as telling the port "I have read n samples from the buffer you gave me earlier; release them to the producer for writing." The number of samples read from the port cannot exceed the initial request (e.g. if you request a lock for 100 samples, you should never try to unlock more than 100).

`gport_consume()` consumes n samples from the port and writes to an external buffer. This method is a blocking call and waits for the requested data to become available or an *end of message* signal to be received.

`gport_consume_available()` operates just like `gport_consume()` except will read as many samples as are available when the function is called. Invoking this method is like telling the buffer "I have a buffer of n samples, so write to it as many as you can right now." It will always wait for at least one sample to become available and blocks until this condition is met.

`gport_signal_eom()` signals *end of message* to any connected `gport`. This tells the port to stop waiting for data (on both the producer and consumer side) and return. This method prevents lock conditions where, e.g., the producer is waiting for several samples to become available, but the consumer has finished its process. This method is normally invoked only during `gport_destroy()`.

`gport_clear_eom()` (*untested*) clears the *end of message* signal.

24.2.5 Problem areas

When using the direct memory access method, the size of the data request during lock is limited by the size of the port. [[race/lock conditions?]]

24.3 `dds` (direct digital synthesizer)

24.4 `qmfb` (quadrature mirror filter bank)

24.5 `quasinewton_search`

The `quasinewton_search` object in *liquid* implements the Quasi-Newton search algorithm which uses the first- and second-order derivates (gradient vector and Hessian matrix) in its update equation. Newtonian-based search algorithms approximate the function to be nearly quadratic near its optimum which requires the second partial derivative (Hessian matrix) to be computed or estimated at each iteration. Quasi-Newton methods circumvent this by approximating the Hessian with successive gradient computations (or estimations) with each step. The Quasi-Newton method is usually faster than the gradient search due in part to its second-order (rather than a first-order) Taylor series expansion about the function of interest, however its update criteria is significantly more involved. In particular the step size must be sufficiently conditioned otherwise the algorithm can result in instability.

An example of the `quasinewton_search` interface is listed below. Notice that its interface is virtually identical to that of `gradient_search`.

```
1  // file: doc/listings/quasinewton_search.example.c
2  #include <liquid/liquid.h>
3
4  int main() {
5      unsigned int num_parameters = 8;    // search dimensionality
6      unsigned int num_iterations = 100;  // number of iterations to run
7      float target_utility = 0.01f;       // target utility
8
9      float optimum_vect[num_parameters];
10
11     // ...
12
13     // create quasinewton_search object
14     quasinewton_search q = quasinewton_search_create(NULL,
15                                                      optimum_vect,
16                                                      num_parameters,
17                                                      &rosenbrock,
```

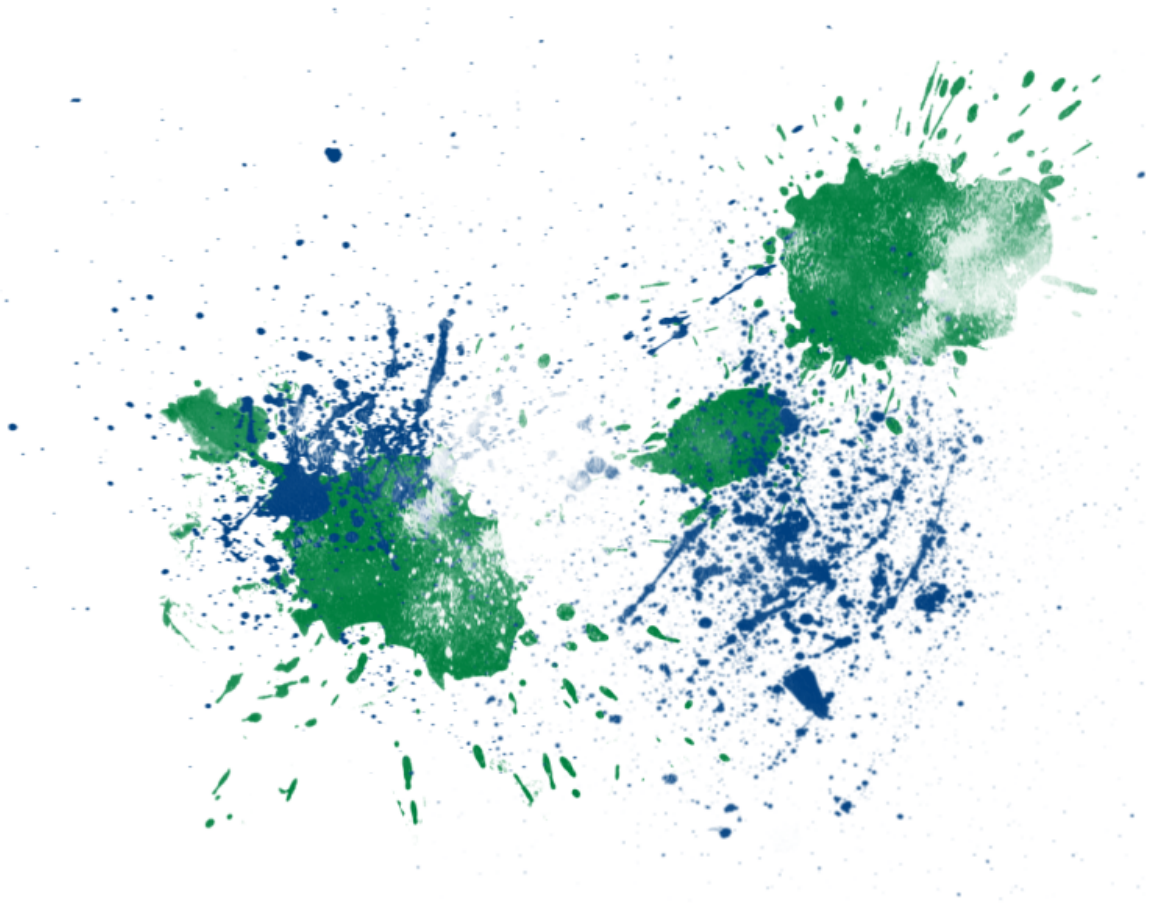
```
                                                    LIQUID_OPTIM_MINIMIZE);

    // execute batch search
    quasinewton_search_execute(q, num_iterations, target_utility);

    // execute search one iteration at a time
    unsigned int i;
    for (i=0; i<num_iterations; i++)
        quasinewton_search_step(q);

    // clean it up
    quasinewton_search_destroy(q);
}
```

Part IV
Installation



25 Installation Guide

The *liquid* DSP library can be easily built from source and is available from several places. The two most typical means of distribution are a compressed archive (a *tarball*) and cloning the source repository. Tarballs are generated with each stable release and are recommended for users not requiring bleeding edge development. Users wanting the very latest version (in addition to every other version) should clone the *liquid* Git repository.

25.1 Building & Dependencies

The *liquid* signal processing library was intended to be universally deployable to a number of platforms by eliminating dependencies on external libraries and programs. That being said, *liquid* still does require a bare minimum build environment to operate. As such the library requires only the following:

- `gcc`, the GNU compiler collection (or equivalent)
- `libc`, the standard C library
- `libm`, the standard math library (eventually will be phased out to optional)

While *liquid* was designed to be portable, requiring a minimal amount of dependencies, the project will take advantage of other libraries if they are installed on the target machine. These optional packages are:

- `fftw3` for computationally efficient fast Fourier transforms
- `libfec` for an extended number of forward error-correction codecs (including convolutional and Reed-Solomon)
- `liquid-fpm` (liquid fixed-point math library)

The build system checks to see if they are installed during the `configure` process and will generate an appropriate `config.h` if they are.

25.2 Building from an archive

Download the compressed archive `liquid-dsp-v.v.v.tar.gz` to your local machine where `v.v.v` denotes the version of the release (e.g. `liquid-dsp-1.0.0.tar.gz`). Check the validity of the tarball with the provided MD5 or SHA1 key. Unpack the tarball

```
$ tar -xvf liquid-dsp-v.v.v.tar.gz
```

Move into the directory and run the configure script and make the library.

```
$ cd liquid-dsp-v.v.v
$ ./configure
$ make
# make install
```

25.3 Building from the Git repository

Development of *liquid* uses Git [11], a free and open-source distributed version control system. The benefits of Git over many other version control systems are numerous and the list is too long to give here; however one of the most important aspects is that each clone holds a copy of the entire repository with a complete history and record of each revision. The main repository for *liquid* is hosted online by *github* [12] and can be cloned on your local machine via

```
$ git clone git://github.com/jgaeddert/liquid-dsp.git
```

Move into the directory and build as before with the archive, but with the additional bootstrapping step:

```
$ cd liquid-dsp.git
$ ./reconf
$ ./configure
$ make
# make install
```

26 Targets

This section lists the specific targets in the main *liquid* project. A basic list can be printed by invoking "`make help`" on the command line. This prints the following to the standard output:

```
all - build shared library (default)
help - print list of targets (see documentation for more)
install - installs the libraries and header files in the host system
uninstall - uninstalls the libraries and header files in the host system
check - build and run autotest scripts
bench - build and run all benchmarks
examples - build all examples binaries
sandbox - build all sandbox binaries
doc - build documentation (doc/liquid.pdf)
todolist - generate tasks list (todolist.txt)
clean - clean build (objects, dependencies, libraries, etc.)
```

The remainder of this section discusses some of the more important and relevant targets.

26.1 Examples (`make examples`)

All examples are built as stand-alone programs not build by the target `all` by default. You may build all of the example binaries at one time by running

```
make examples
```

Sometimes, however, it is useful to build one example individually. This can be accomplished by directly targeting its binary (e.g. "`make examples/modem_example`"). The example then can be run at the command line (e.g. "`./examples/modem_example`").

The examples are probably the best way to understand how each signal processing element works. Each example targets a specific functionality of *liquid*, such as FIR filtering, forward error

correction, digital demodulation, etc. A number of the example programs when run will generate an output `.m` file which can be run directly in Octave [7]. This is particularly useful for visualizing filtering operations. Most of the examples have a brief description at the top of the file; these descriptions are also available in the `examples/README` file for convenience. Some of the examples are experimental and will not be built by default (see Section 24).

26.2 Autotests (`make check`)

Source code validation is a critical step in any software library, particularly for verifying the portability of code to different processors and platforms. Packaged with *liquid* are a number of automatic test scripts to validate the correctness of the source code. The test scripts are located under each module's `tests` directory and take the form of a C header file. The testing framework operates similarly to CppUnit [1] and cxxtest [2], however it is written in C. The generator script `scripts/autoscript` parses these header files looking for the key "`void autotest_`" which corresponds to a specific test. The script generates the header file `autotest_include.h` which includes all the modules' test headers as well as several organizing structures for keeping track of which tests have passed or failed. The result is an executable file, `xautotest`, which can be run to validate the functional correctess of *liquid* on your target platform.

26.2.1 Macros

Each module contains a number of autotest scripts which use pre-processor macros for asserting the functional correctness of the souce code.

`CONTEND_EQUALITY`(x,y) asserts that $x == y$ and fails if false.

`CONTEND_INEQUALITY`(x,y) asserts that x differs from y.

`CONTEND_GREATER_THAN`(x,y) asserts that $x > y$.

`CONTEND_LESS_THAN`(x,y) asserts that $x < y$.

`CONTEND_DELTA`(x,y,Δ) asserts that $|x-y| < \Delta$

`CONTEND_EXPRESSION`$(expr)$ asserts that some expression is true.

`CONTEND_SAME_DATA`$(ptrA,ptrB,n)$ asserts that each of n byte values in the arrays referenced by $ptrA$ and $ptrB$ are equal.

`AUTOTEST_PASS`() passes unconditionally.

`AUTOTEST_FAIL`$(string)$ prints *string* and fails unconditionaly.

`AUTOTEST_WARN`$(string)$ simply prints a warning. The autotest program will keep track of which tests elicit warnings and add them to the list of unstable tests.

Here are some examples:

`CONTEND_EQUALITY`(1,1) will *pass*

`CONTEND_EQUALITY`(1,2) will *fail*

26.2.2 Running the autotests

The result is an executable file named `xautotest` which has several options for running. These options may be viewed with either the `-h` or `-u` flags (for help/usage information).

```
$ ./xautotest -h
autotest options:
  -h,-u : prints this help file
  -t<n> : run specific test
  -p<n> : run specific package
  -L    : lists all autotests
  -l    : lists all packages
  -s    : stop on fail
  -v    : verbose
  -q    : quiet
```

Simply running the program without any arguments executes all the tests and displays the results to the screen. The is the default response of the target `make check`.

26.3 Benchmarks (`make bench`)

Packaged with *liquid* are benchmarks to determine the speed each signal processing element can run on your machine. You can build the benchmark program with `make benchmark`, and view the execution options with a `-u` or `-h` flag for usage/help information:

```
$ ./benchmark -h
bench options:
  -h : prints this help file
  -e : estimate cpu clock frequency and exit
  -c : set cpu clock frequency (Hz)
  -n<num_trials>
  -p<package_index>
  -b<benchmark_index>
  -t<time> minimum execution time (ms)
  -l : lists available packages
  -L : lists all available benchmarks
  -v : verbose
  -q : quiet
  -o<output filename>
```

By default, running "`make bench`" is equivalent to simply executing the `./benchmark` program which runs all of the benchmarks sequentially. Initially the tool provides an estimate of the processor's clock frequency; while not necessarily accurate, this is necessary to gauge the relative speed by which the benchmarks will run. The tool will then estimate the number of trials so that each benchmark will take between 50 and 500 ms to run. Listed below is the output of the first several benchmarks:

```
  estimating cpu clock frequency...
  performed 67108864 trials in 650.0 ms
  estimated clock speed:   2.468 GHz
  setting number of trials to 246754
0: null
```

```
    0  : null                 :  23.59 M trials in 220.00 ms (107.212 M t/s,  22.00   cycles/t)
1: agc
    1  : agc_crcf_default      : 943.47 k trials in 140.00 ms (  6.739 M t/s, 350.00   cycles/t)
    2  : agc_crcf_log          : 943.47 k trials in 150.00 ms (  6.290 M t/s, 375.00   cycles/t)
    3  : agc_crcf_exp          : 943.47 k trials in 140.00 ms (  6.739 M t/s, 350.00   cycles/t)
    4  : agc_crcf_true         : 943.47 k trials in 110.00 ms (  8.577 M t/s, 275.00   cycles/t)
    5  : agc_crcf_default_D4   :   3.77 M trials in 200.00 ms ( 18.869 M t/s, 125.00   cycles/t)
    6  : agc_crcf_log_D4       :   3.77 M trials in 220.00 ms ( 17.154 M t/s, 137.50   cycles/t)
    7  : agc_crcf_exp_D4       :   3.77 M trials in 220.00 ms ( 17.154 M t/s, 137.50   cycles/t)
    8  : agc_crcf_true_D4      :   3.77 M trials in 170.00 ms ( 22.199 M t/s, 106.25   cycles/t)
    9  : agc_crcf_default_sq   : 943.47 k trials in 150.00 ms (  6.290 M t/s, 375.00   cycles/t)
    10 : agc_crcf_default_sq_D4:   3.77 M trials in 240.00 ms ( 15.724 M t/s, 150.00   cycles/t)
    11 : agc_crcf_locked       :   3.77 M trials in  80.00 ms ( 47.173 M t/s,  50.00   cycles/t)
2: window
    12 : windowcf_n16          :   7.55 M trials in 260.00 ms ( 29.029 M t/s,  81.25   cycles/t)
    13 : windowcf_n32          :   7.55 M trials in 260.00 ms ( 29.029 M t/s,  81.25   cycles/t)
    14 : windowcf_n64          :   7.55 M trials in 270.00 ms ( 27.954 M t/s,  84.38   cycles/t)
    15 : windowcf_n128         :   7.55 M trials in 260.00 ms ( 29.029 M t/s,  81.25   cycles/t)
    16 : windowcf_n256         :   7.55 M trials in 260.00 ms ( 29.029 M t/s,  81.25   cycles/t)
3: dotprod_cccf
    17 : dotprod_cccf_4        :   1.89 M trials in 320.00 ms (  5.897 M t/s, 400.00   cycles/t)
    18 : dotprod_cccf_16       : 471.73 k trials in 320.00 ms (  1.474 M t/s,   1.60 k cycles/t)
    19 : dotprod_cccf_64       : 117.93 k trials in 300.00 ms (393.107 k t/s,   6.00 k cycles/t)
    20 : dotprod_cccf_256      :  29.48 k trials in 300.00 ms ( 98.267 k t/s,  24.00 k cycles/t)
4: dotprod_crcf
    21 : dotprod_crcf_4        :   1.89 M trials in  20.00 ms ( 94.347 M t/s,  25.00   cycles/t)
    22 : dotprod_crcf_16       : 471.73 k trials in  10.00 ms ( 47.173 M t/s,  50.00   cycles/t)
    23 : dotprod_crcf_64       : 117.93 k trials in   0.00 ps (    inf T t/s,   0.00 p cycles/t)
    24 : dotprod_crcf_256      :  29.48 k trials in  20.00 ms (  1.474 M t/s,   1.60 k cycles/t)
```

For this run the clock speed was estimated to be 2.468 GHz. Benchmarks are sub-divided into *packages* which group similar signal processing algorithms together. For example, package `3` above refers to benchmarking the `dotprod_cccf` object which computes the vector dot product between two n-point arrays of complex floats. Specifically, benchmark `19` refers to the speed of an $n = 64$-point dot product. In this run the benchmarking tool computed approximately 117,930 64-point complex dot products in 300 ms (about 393,107 trials per second). For the estimated clock rate this means that the algorithm requires approximately 6,000 clock cycles to compute a single 64-point complex vector dot product.

26.4 Documentation (`make doc`)

Specifically, "`make doc`" builds this `.pdf` file you're reading right now. The documentation requires a few additional packages to build from scratch:

- `pdflatex`, the LaTeXengine responsible for making this document with all those pretty equtions
- `bibtex`, the package for creating the bibliography
- `gnuplot`, a program for plotting graphics
- `epstopdf`, conversion from `.eps` to `.pdf`, required for the figures created with `gnuplot`
- `pygments`, the syntax highlighting engine responsible for generating all the fancy code listings given throughout this document. The command-line equivalent is called `pygmentize`.

References

[1] CppUnit website. http://sourceforge.net/projects/cppunit/, 2010.

[2] cxxtest website. http://cxxtest.tigris.org/, 2010.

[3] GNU Radio (official website). http://gnuradio.org/, 2 2011.

[4] Roland E. Best. *Phase-Locked Loops: Design, Simulation, and Applications*. McGraw-Hill, 3 edition, 1997.

[5] W. R. Braun and U. Dersch. A physical mobile radio channel model. *IEEE Transactions on Vehicular Technology*, 40:472–482, February 1991.

[6] J. K. Cavers. Optimized use of diversity modes in transmitter diversity systems. In *Vehicular Technology Conference*, pages 1768–1773, April 1999.

[7] John W. Eaton. Octave Website. http://www.gnu.org/software/octave/, 2010.

[8] fredric j. harris. *Multirate Signal Processing for Communication Systems*. Prentice Hall, 2004.

[9] fredric j. harris, Chris Dick, Sridhar Seshagiri, and Karl Moerder. An Improved Square-Root Nyquist Shaping Filter. In *Proceedings of the Software-Defined Radio Forum*, 2005.

[10] GCC, the GNU Compiler Collection (official website). http://gcc.gnu.org/, 2011.

[11] official Git website. http://git-scm.com/, 2011.

[12] official github website. http://github.com/, 2011.

[13] Simon S. Haykin. *Adaptive Filter Theory*. Prentice Hall, Upper Saddle River, N.J., 4 edition, 2002.

[14] Y. C. Ko and M. Alouini. Estimation of Nakagami-m fading channel parameters with application to optimized transmitter diversity systems. *IEEE Transactions on Wireless Communications*, 2(2):250–259, March 2003.

[15] The Linux Kernel Archives—Official Website. http://www.kernel.org/, 2011.

[16] Matteo Frigo. FFTW Website. www.fftw.org/, 2010.

[17] Umberto Mengali and Aldo N. D'Andrea. *Synchronization Techniques for Digital Receivers*. Applications of Communications Theory. Springer, New York, 1 edition, 1997.

[18] Kurt H. Mueller and Markus Müller. Timing Recovery in Digital Synchronous Data Receivers. *IEEE Transactions on Communications*, COM-24(5), May 1976.

[19] H. J. Orchard. The Roots of the Maximally Flat-Delay Polynomials. In *IEEE Transactions on Circuit Theory*, September 1965.

[20] Sophocles J. Orfanidis. Lecture Notes on Elliptic Filter Design. http://www.ece.rutgers.edu/ orfanidi/hpeq, 2006.

[21] Open Source SCA Implementation::Embedded (official website). http://ossie.wireless.vt.edu/, 2 2011.

[22] A. Papoulis and S. U. Pillai. *Probability, Random Variables and Stochastic Processes*. McGraw-Hill, Boston, 4 edition, 2002.

[23] Phil Karn. libfec website. http://www.ka9q.net/code/fec/, 2007.

[24] J. G. Proakis. *Digital Communications*. McGraw-Hill, New York, 4 edition, 2001.

[25] Python Programming Language—Official Website. http://python.org/, 2011.

[26] V. Erceg S. Catreux and R. Heath. Adaptive Modulation and MIMO Coding for Broadband Wireless Data Networks. *IEEE Communications Magazine*, pages 108–115, June 2002.

[27] M. K. Simon and M. S. Alouini. A Unified Approach to the Performance Analysis of Digital Communication over Generalized Fading Channels. *Proceedings of the IEEE*, 86(9), September 1998.

[28] H. Suzuki. A Statistical model for Urban Radio Propagation. *IEEE Transactions on Communications*, COM-25:673–80, July 1977.

[29] G. L. Turin. Introduction to Spread-Spectrum Anti-Multipath Techniques and Their Applications to Urban Digital Radio. *Proceedings of the IEEE*, 68:328–53, March 1980.

[30] P. P. Vaidyanathan. *Multirate Systems and Filter Banks*. Prentice Hall Signal Processing Series. Prentice Hall, New Jersey, 1993.

[31] official vim website. http://www.vim.org/, 2011.