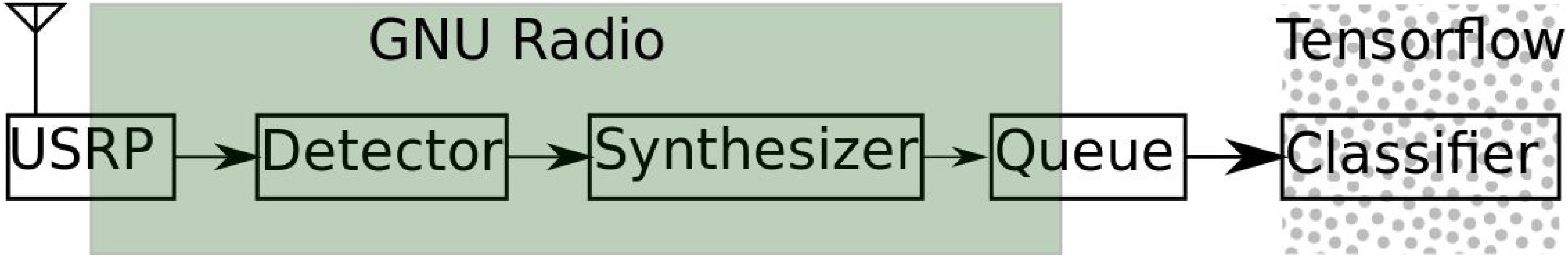
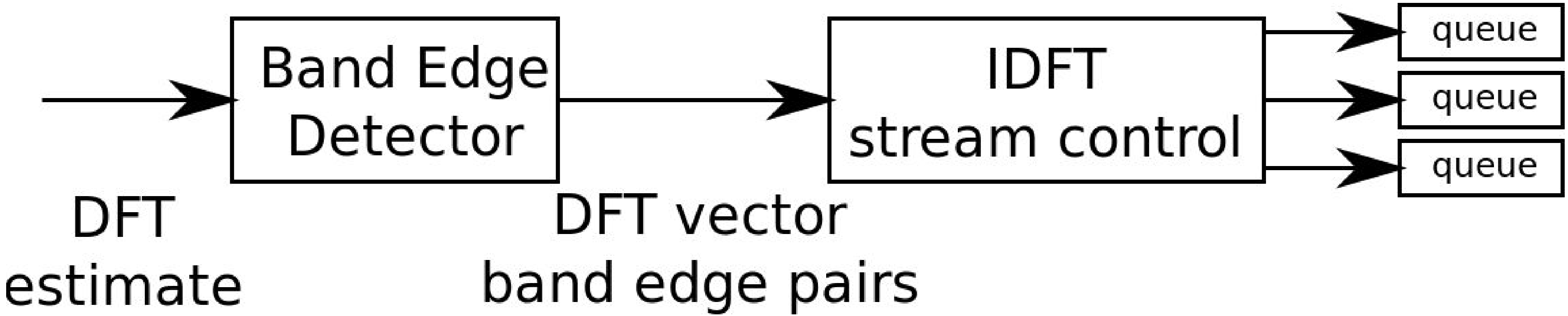
Rapidly Iterating on DSP:

a reflection of my development experience

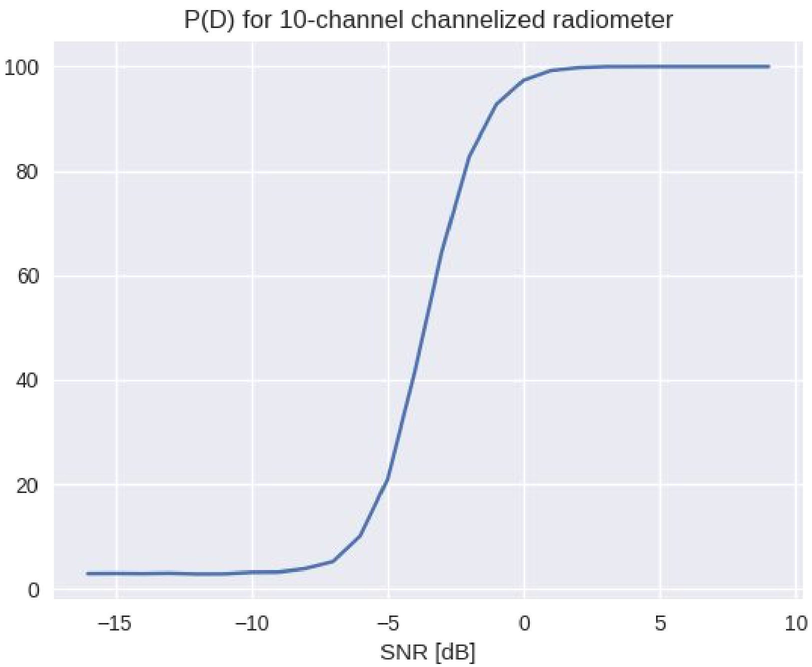
Nathan West (nathan.west@okstate.edu)

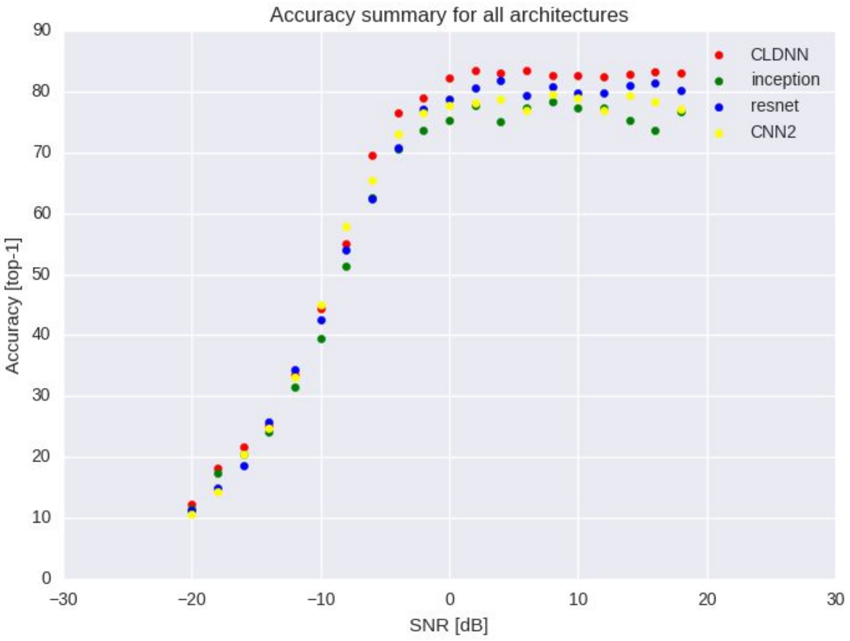
NRL/Oklahoma State University

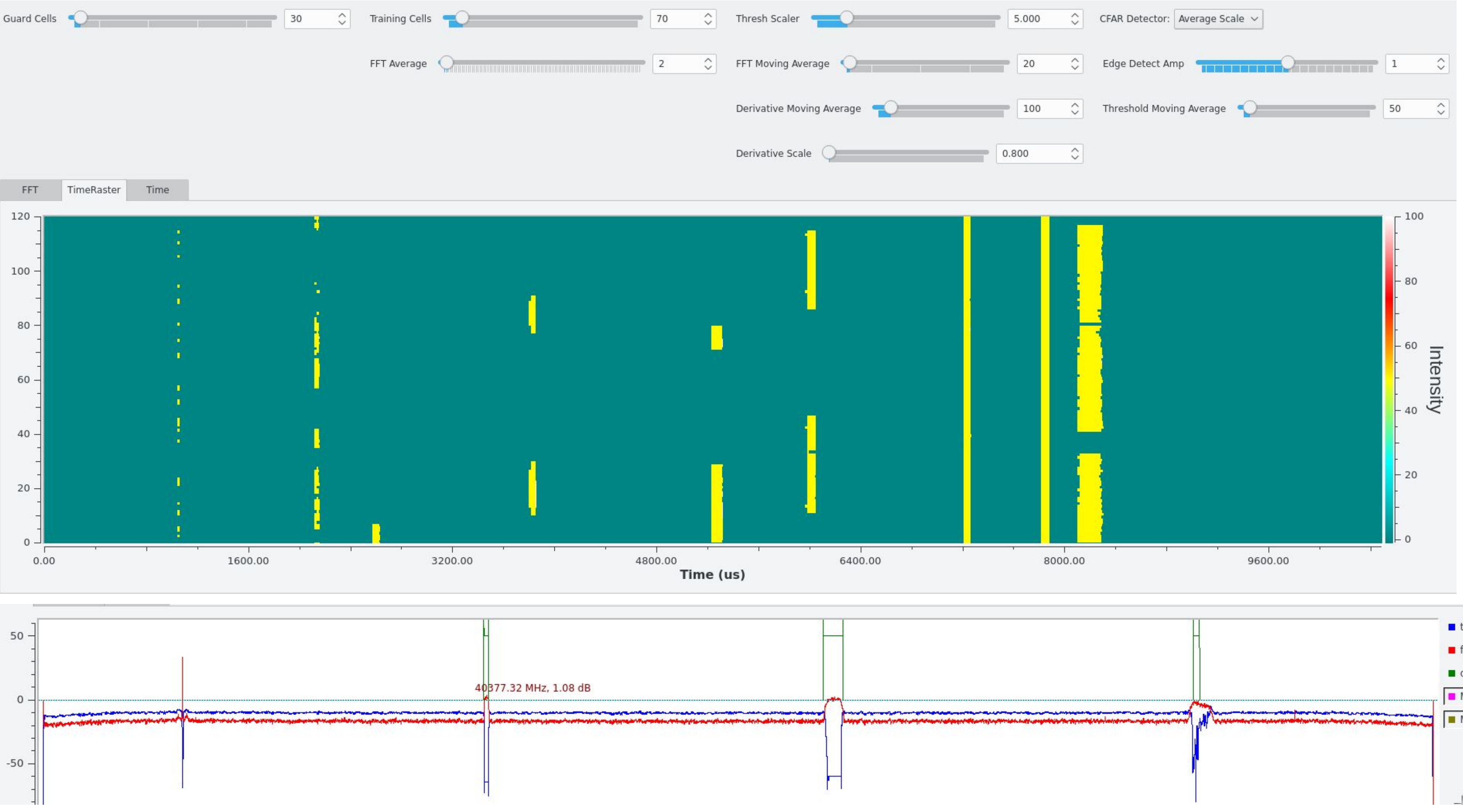

GNU Radio
Tensorflow
USRP
Detector
Synthesizer
Queue
Classifier


Band Edge
Detector
IDFT
stream control
queue
queue
queue
DFT
estimate
DFT vector
band edge pairs

Start with modrec + spectrum sensing







Guard Cells
30
Training Cells
70
Thresh Scaler
5.000
CFAR Detector:
Average Scale
FFT Average
2
FFT Moving Average
20
Edge Detect Amp
1
Derivative Moving Average
100
Threshold Moving Average
50
Derivative Scale
0.800
FFT
TimeRaster
Time
Intensity
Time (us)
40377.32 MHz, 1.08 dB

Many problems...

- Copying tons of data just to move between tools
  - The ML is too far from the samples
  - Memory speed matters!
- Development iteration
  - Exposing and removing parameters
  - Quick changes
  - Change -> compiler -> reload blocks -> run -> wait for interesting point in file -> STOP

- In action problems, streaming all of this from gr-uhd (and synchronization)
- Then there’s the actual DSP

A solution emerges

python

📻 🐍

Problem-pivoting

- Python is actually kinda slow
    - Many ways to port code
- GIL
    - Really poor concurrency
    - Shared memory in multi-processing
- 2 vs 3 (6?!?)
- Cultish adherence to pythonism
- Occasionally awkward syntax and loss of some control

The test of a first-rate intelligence is the ability to hold two opposed ideas in the mind at the same time, and still retain the ability to function.
— *F. Scott Fitzgerald*

Much discussion on accelerating python

Another 20 times speedup! Let us compile our function, with and without static typing.

```
%%cython

def fib_seq_cython(n):
    if n < 2:
        return n
    a,b = 1,0
    for i in range(n-1):
        a,b = a+b,a
    return a

cpdef long fib_seq_cython_type(long n):
    if n < 2:
        return n
    cdef long a,b
    a,b = 1,0
    for i in range(n-1):
        a,b = a+b,a
    return a
```

If you really care about extracting every possible ounce of performance out of Python's scientifi
P. Rougier of Inria: https://www.labri.fr/perso/nrougier/from-python-to-numpy/

Here's a typical example of the kinds of optimizations this guide teaches you, in this case by av

```
# Create two int arrays, each filled with with one billion 1's.
X = np.ones(1000000000, dtype=np.int)
Y = np.ones(1000000000, dtype=np.int)

# Add 2 * Y to X, element by element:

# Slowest
%time X = X + 2.0 * Y
100 loops, best of 3: 3.61 ms per loop

# A bit faster
%time X = X + 2 * Y
100 loops, best of 3: 3.47 ms per loop

# Much faster
%time X += 2 * Y
100 loops, best of 3: 2.79 ms per loop

# Fastest
%time np.add(X, Y, out=X); np.add(X, Y, out=X)
100 loops, best of 3: 1.57 ms per loop
```

That's a 2.3x speed improvement (from 3.61 ms to 1.57 ms) on a simple vector operation (

BLAS really shines when you do matrix multiplication, for element-wise operation
benchmark about seems unrealistic, here are results from my newest MaBook P

```
In [2]: import numpy as np

In [3]: X = np.ones(1000000000, dtype=np.int)

In [4]: Y = np.ones(1000000000, dtype=np.int)

In [5]: %time X = X + 2.0 * Y
CPU times: user 10.4 s, sys: 27.1 s, total: 37.5 s
Wall time: 46 s

In [6]: %time X = X + 2 * Y
CPU times: user 8.66 s, sys: 26 s, total: 34.7 s
Wall time: 42.6 s

In [7]: %time X += 2 * Y
CPU times: user 8.58 s, sys: 23.2 s, total: 31.8 s
Wall time: 37.7 s

In [8]: %time np.add(X, Y, out=X); np.add(X, Y, out=X)
CPU times: user 11.3 s, sys: 25.6 s, total: 36.9 s
Wall time: 42.6 s
```

No surprise, Julia makes nearly the same result:

```
julia> X = ones(Int, 1000000000);
julia> Y = ones(Int, 1000000000);

julia> @btime X .= X .+ 2Y
  34.814 s (6 allocations: 7.45 GiB)
```

(How to Make Python Run as Fast as Julia (2015) (ibm.com))[https://news.ycombinator.com/item?id=15121520]

A wild volk appears

Many ways to write python wrappers

- Boost.python
- Ctypes
- SWIG
- SIP (QT)
- Pyrex/cython
- Python.h (C API)

Another 20 times speedup! Let us compile our function, with and without static typing.

```
%%cython

def fib_seq_cython(n):
    if n < 2:
        return n
    a,b = 1,0
    for i in range(n-1):
        a,b = a+b,a
    return a

cpdef long fib_seq_cython_type(long n):
    if n < 2:
        return n
    cdef long a,b
    a,b = 1,0
    for i in range(n-1):
```

```
cdef extern from "Python.h":    # we will use these C functions below
    void* PyMem_Malloc(int)
    void PyMem_Free(void *p)

cdef class Matrix:
    cdef int *entries
    cdef int p, n

    def __new__(self, int p, int n, entries=None):
        self.p = p; self.n = n
        self.entries = <int*> PyMem_Malloc(sizeof(int)*n*n)    # cast to int pointer

    def __dealloc__(self):
        PyMem_Free(self.entries)          # using a C function

    def __init__(self, int p, int n, entries=None):
        """ p -- prime
            n -- positive integer
            entries -- entries of the matrix (defaults to None, which means 0 matrix).
```

```
static size_t wrap_recv(uhd::rx_streamer *rx_stream,
                        bp::object &np_array,
                        bp::object &metadata)
{
    // Extract the metadata
    bp::extract<uhd::rx_metadata_t&> get_metadata(metadata);
    if (not get_metadata.check())
    {
        return 0;
    }

    // Get a numpy array object from given python object
    // No sanity checking possible!
    PyObject* array_obj = PyArray_FROM_OF(np_array.ptr(), NPY_ARRAY_CARRAY);
    PyArrayObject* array_type_obj = reinterpret_cast<PyArrayObject*>(array_obj);

    // Get dimensions of the numpy array
    const size_t dims = PyArray_NDIM(array_type_obj);
    const npy_intp* shape = PyArray_SHAPE(array_type_obj);

    // How many bytes to jump to get to the next element of this stride
    // (next row)
    const npy_intp* strides = PyArray_STRIDES(array_type_obj);
    const size_t channels = rx_stream->get_num_channels();

    // Check if numpy array sizes are okay
    if ((channels > 1) && (dims != 2)) {
        return 0;
    } else if ((size_t) shape[0] < channels) {
        return 0;
    }

    // Get a pointer to the storage
    std::vector<void*> channel_storage;
    char* data = PyArray_BYTES(array_type_obj);
    for (size_t i = 0; i < channels; ++i)
    {
        channel_storage.push_back((void*)(data + i * strides[0]));
    }

    // Get data buffer and size of the array
    size_t nsamps_per_buff;
    if (dims > 1) {
        nsamps_per_buff = (size_t) shape[1];
```

Just write against Python And numpy C-api

Some concessions made along the way

```c
static PyObject *
multiply_wrapper(PyObject* self, PyObject* args)
{
    PyObject *aArg=NULL, *bArg=NULL;
    PyArrayObject *aArray=NULL, *bArray=NULL;

    if (!PyArg_ParseTuple(args, "OO", &aArg, &bArg)) {
        return NULL;
    }

    bool a_is_array = PyArray_Check(aArg);
    bool b_is_array = PyArray_Check(bArg);

    float scalar = 0;
    float *vec=NULL;
    int vec_length = 0;
    int ndims;
    npy_intp *shape;

    if (a_is_array && !b_is_array) {
        scalar = (float) PyFloat_AsDouble(bArg);
        vec = (float*) PyArray_DATA((PyArrayObject*) aArg);
        shape = PyArray_SHAPE((PyArrayObject*) aArg);
        vec_length = shape[0];
    } else if (!a_is_array && b_is_array) {
        scalar = (float) PyFloat_AsDouble(aArg);
        vec = (float*) PyArray_DATA((PyArrayObject*) bArg);
        shape = PyArray_SHAPE((PyArrayObject*) bArg);
        vec_length = shape[0];
    }

    PyObject *result = PyArray_SimpleNew(1, shape, NPY_FLOAT);
    float *result_data = PyArray_DATA((PyArrayObject*) result);
    volk_32f_s32f_multiply_32f(result_data, vec, scalar, vec_length);

    return result;
```

Radiometer

```
In [25]: X = np.ones(1000000000, dtype=np.float32)

In [26]: Y = np.ones(1000000000, dtype=np.float32)

In [27]: %timeit X + 2.0 * Y
1 loop, best of 3: 5.29 s per loop

In [28]: %timeit np.add(X, 2.0 * Y)
1 loop, best of 3: 4.66 s per loop

In [29]: %timeit pv.add(X, 2.0 * Y)
1 loop, best of 3: 3.98 s per loop

In [30]: %timeit pv.add(X, pv.multiply(2.0, Y))
1 loop, best of 3: 3.88 s per loop
```

```
In [22]: vlen = 100000

In [23]: X = np.random.randn(vlen) + np.random.randn(vlen) * 1.j

In [24]: X = np.complex64(X)

In [25]: %timeit pv.sum(pv.magnitude_squared(X))
10000 loops, best of 3: 42.5 µs per loop

In [26]: %timeit np.sum(np.square(np.abs(X)))
1000 loops, best of 3: 483 µs per loop
```

Getting samples

More or less solved my rapidly prototyping algorithms problem… now how to we build the application around it?

Enter pysdr(uhd)

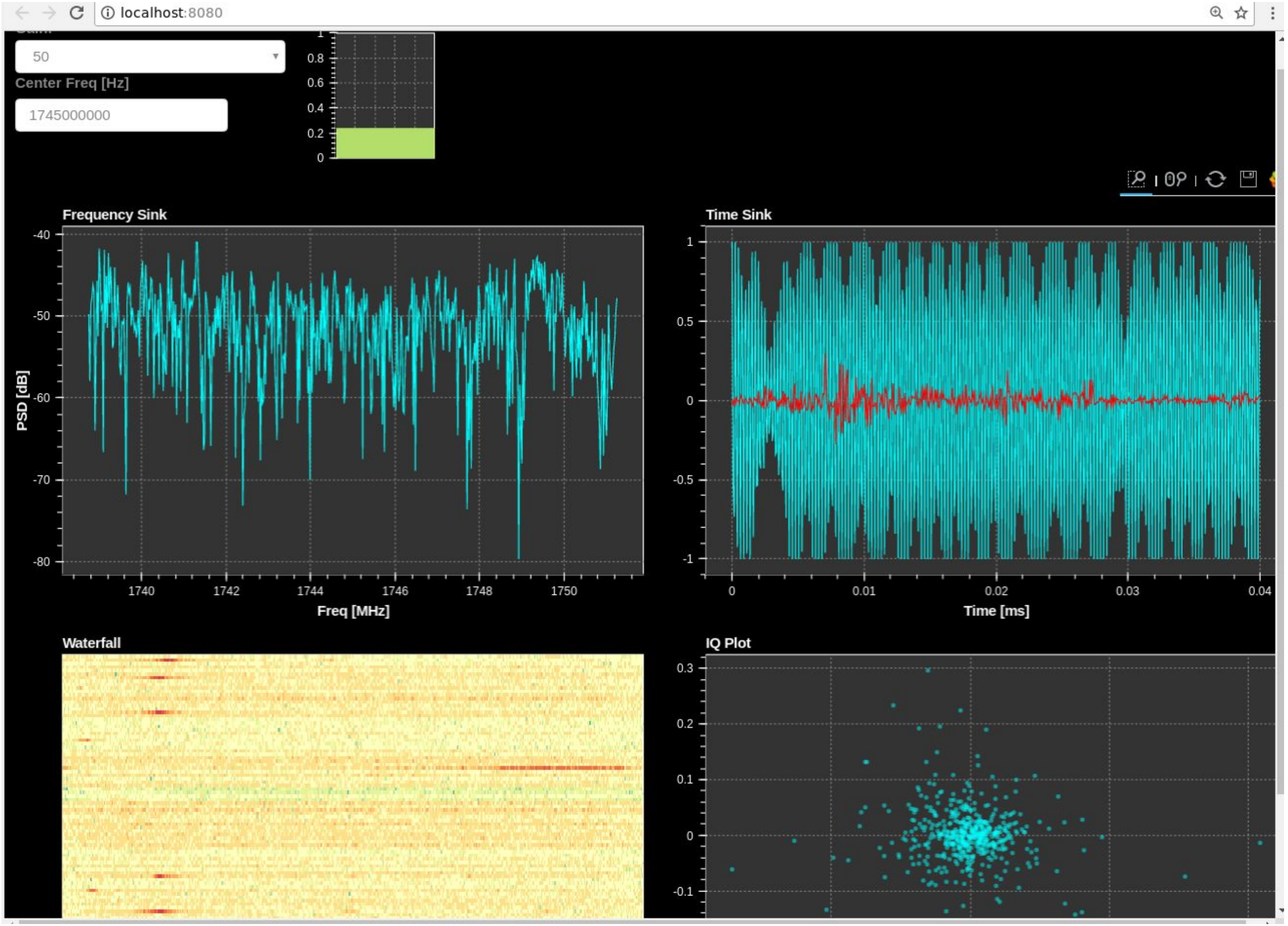
Quick example

```python
import pysdruhd as uhd
import matplotlib.pyplot as plt

center_freq = 100e6
samp_rate = 4e6

usrp = uhd.Usrp(type="b200", streams={"A:A": {'mode':'RX', 'frequency':center_freq, 'gain':70}}, rate=samp_rate, gain=30.0)
usrp.send_stream_command({'now': True})
samples, metadata = usrp.recv()
plt.psd(samples[0], NFFT=512, Fs=samp_rate/1e6, Fc=center_freq/1e6)
plt.show()
```

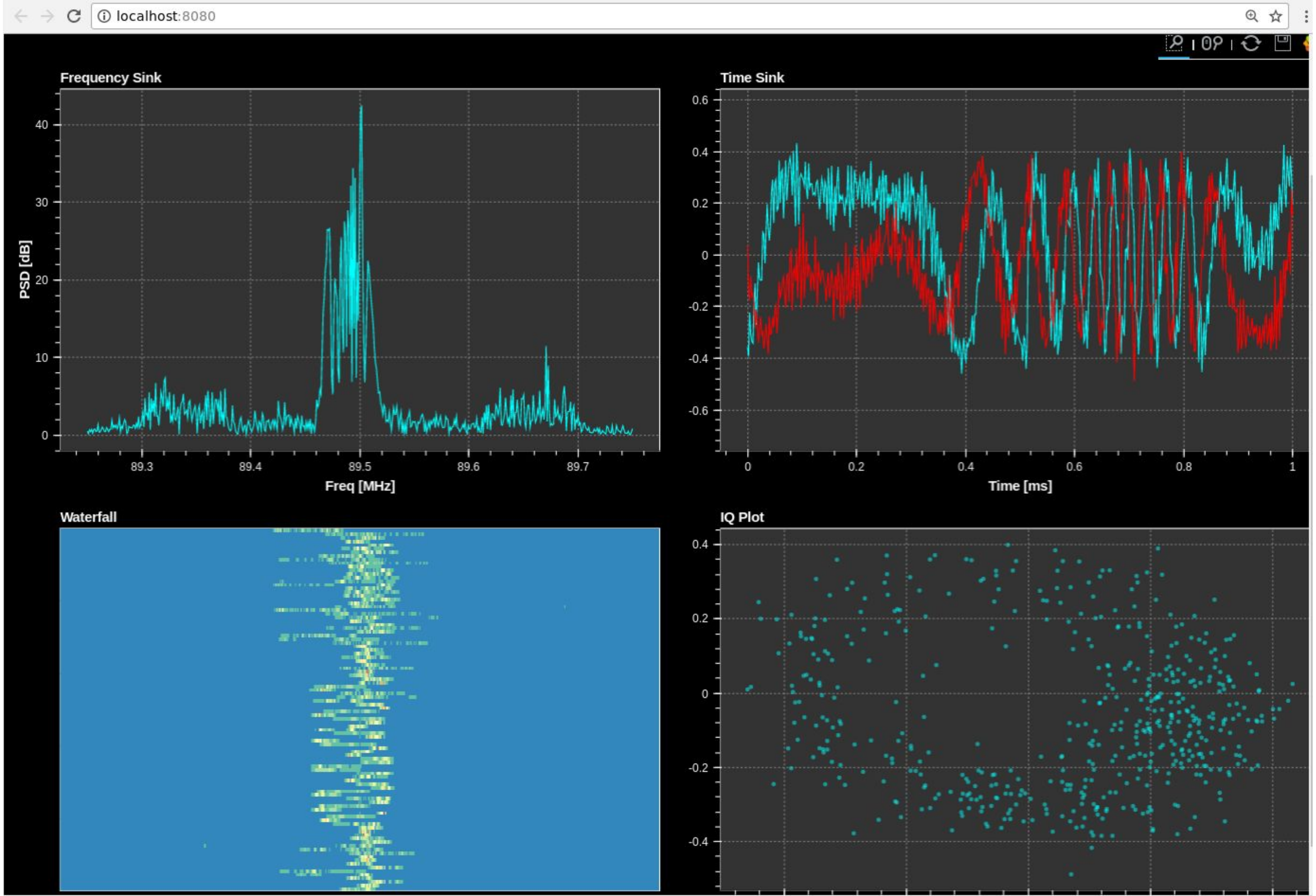
Great, now how do I look at samples?

Streaming into a visualization is important

Enables twiddling things on the fly



localhost:8080
50
Center Freq [Hz]
1745000000
Frequency Sink
PSD [dB]
Freq [MHz]
Time Sink
Time [ms]
Waterfall
IQ Plot


localhost:8080
Frequency Sink
PSD [dB]
Freq [MHz]
89.3
89.4
89.5
89.6
89.7
0
10
20
30
40
Time Sink
Time [ms]
0
0.2
0.4
0.6
0.8
1
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
Waterfall
IQ Plot
0.4
0.2
0
-0.2
-0.4

NumPy

VOLK

Summary

UHD within Python -> fast DSP -> machine learning, all in one language without weird connections

When python is slow, it’s not too bad to write fast C and wrap it. Pyvolk on github later this week

Streaming and sample-specific visualizer

Great way to rapidly iterate while developing