peraton.com

Peraton

Untangling the Airwaves: Implementing and Evaluating Blind Source Separation Techniques in GNU Radio

GNU Radio Conference 2025
September 12th, 2025

Outline

- The problem solved by blind source separation

- Common techniques and pitfalls (ICA, PCA)
- Performance comparison and GNU Radio implementation

- Adaptive event processing (AEP) for wireless communications
- Accelerating the GNU Radio implementation

The Cocktail Party Problem

Why we need blind source separation

The Cocktail Party Problem

- Goal: Given recordings of an environment with multiple (sources), separate out each individual source.
- You have n independent source signals: $s(t) = [s_0(t), s_1(t), \ldots, s_{n-1}(t)]$
- You record these via m sensors: $\boldsymbol{x}(t) = [x_0(t), x_1(t), \ldots, x_{m-1}\,(t)]$
- Assume each mic records a weighted sum of sources $\boldsymbol{x}(t)$=$\boldsymbol{As}(t)$, where $\boldsymbol{A}$ is an unknown full-rank matrix called the mixing matrix.
- We want to determine both $\boldsymbol{s}(t)$ and $\boldsymbol{A}$ using only $\boldsymbol{x}(t)$, without knowing $\boldsymbol{A}$ — hence “blind.”

Common Techniques and Pitfalls

How we attempt to solve this problem

Common Techniques

Principal Component Analysis

- All BSS techniques make assumptions that establish constraints on $\boldsymbol{x}(t)$ and $\boldsymbol{s}(t)$, which often transforms this into an optimization problem.

- PCA: Assume components of $\boldsymbol{s}(t)$ have minimal linear correlation.
  - Practically, this is accomplished via eigen-decomposition.

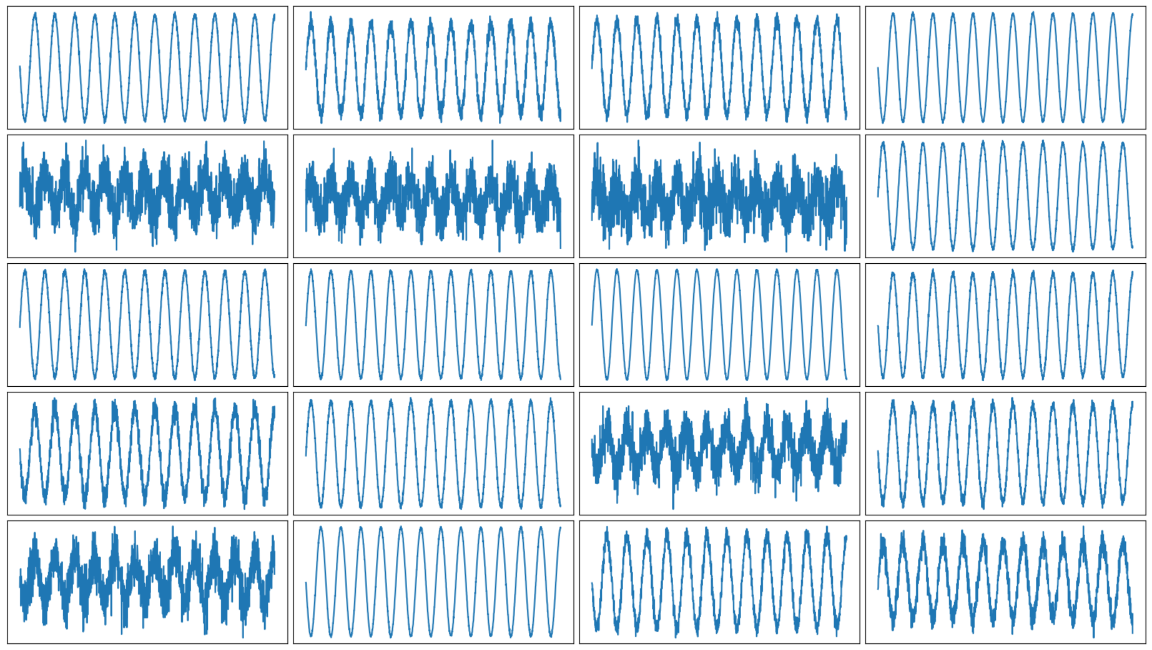
- Common Use-cases
  - Dimensionality reduction (source signals are captured multiple times)
  - De-noising

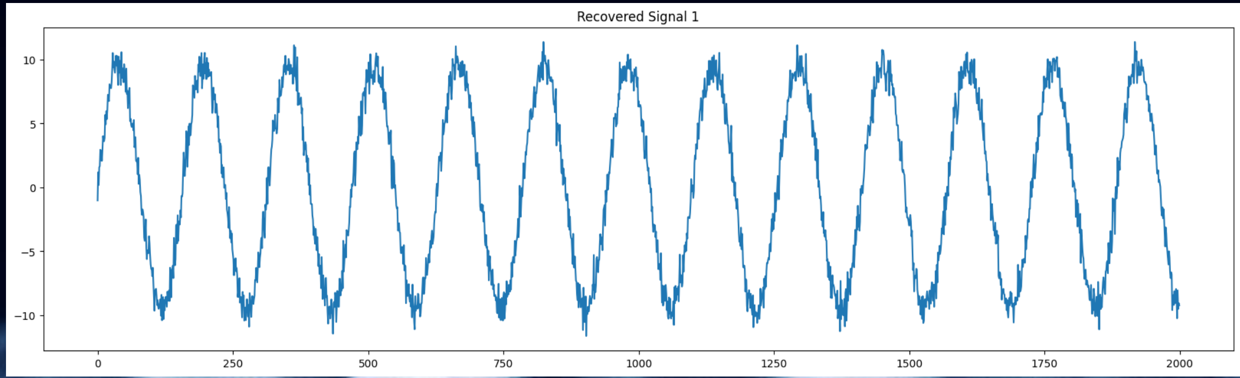
PCA Example

Noise Reduction

- PCA is not great at separating comms signals in practice, but can work as a de-noiser

- As an example: 16 sinusoids with independent noise





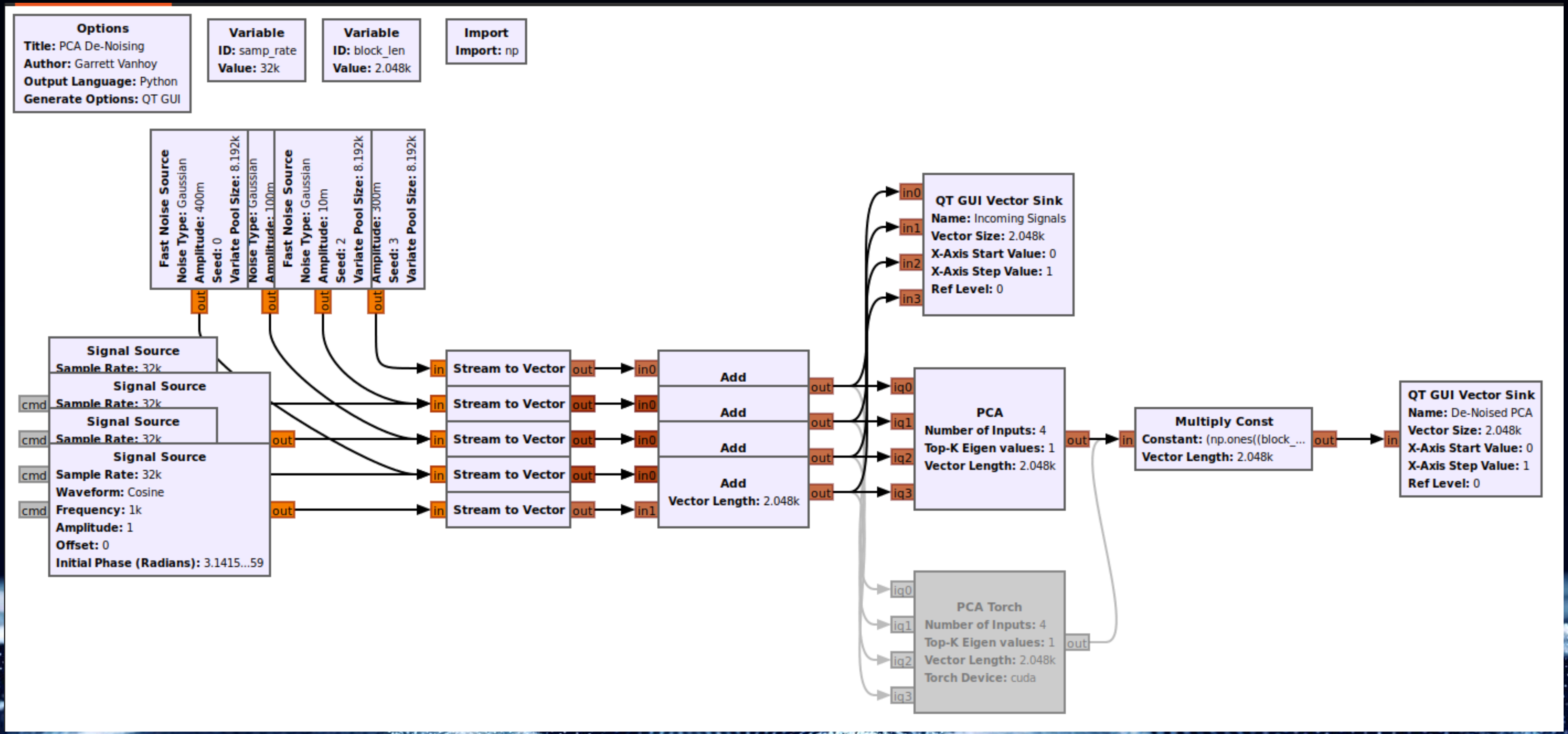
PCA CPU Implementation

C++ implementation in GNU Radio using armadillo

```
using arma::fmat; // float matrix (column-major in memory)

// Materialize X (n_samples x n_features) in column-major layout
fmat X(n_samples, n_features, arma::fill::none);
for (int j = 0; j < n_features; ++j) {
    auto src_col = static_cast<const input_type*>(in[j]);
    std::memcpy(X.colptr(j), src_col, static_cast<size_t>(n_samples) * sizeof(float));
}

// Center columns
fmat Xc = X.each_row() - arma::mean(X, /*dim=*/0);

// Covariance and eigendecomposition
fmat Cov = (Xc.t() * Xc) / static_cast<float>(n_samples - 1);
arma::Col<float> eigvals;
fmat eigvecs;
if (!arma::eig_sym(eigvals, eigvecs, Cov)) {
    throw std::runtime_error("pca_project_from_column_ptrs: eig_sym failed");
}

// Top-k eigenvectors (largest eigenvalues)
fmat Vk = eigvecs.cols(eigvecs.n_cols - k, eigvecs.n_cols - 1);

// Projection Z (n_samples x k)
fmat Z = Xc * Vk;

// Write to contiguous column-major `out`
for (int j = 0; j < k; ++j) {
    auto src_col = static_cast<output_type*>(out[j]);
    std::memcpy(src_col, Z.colptr(j), static_cast<size_t>(n_samples) * sizeof(float));
}
```

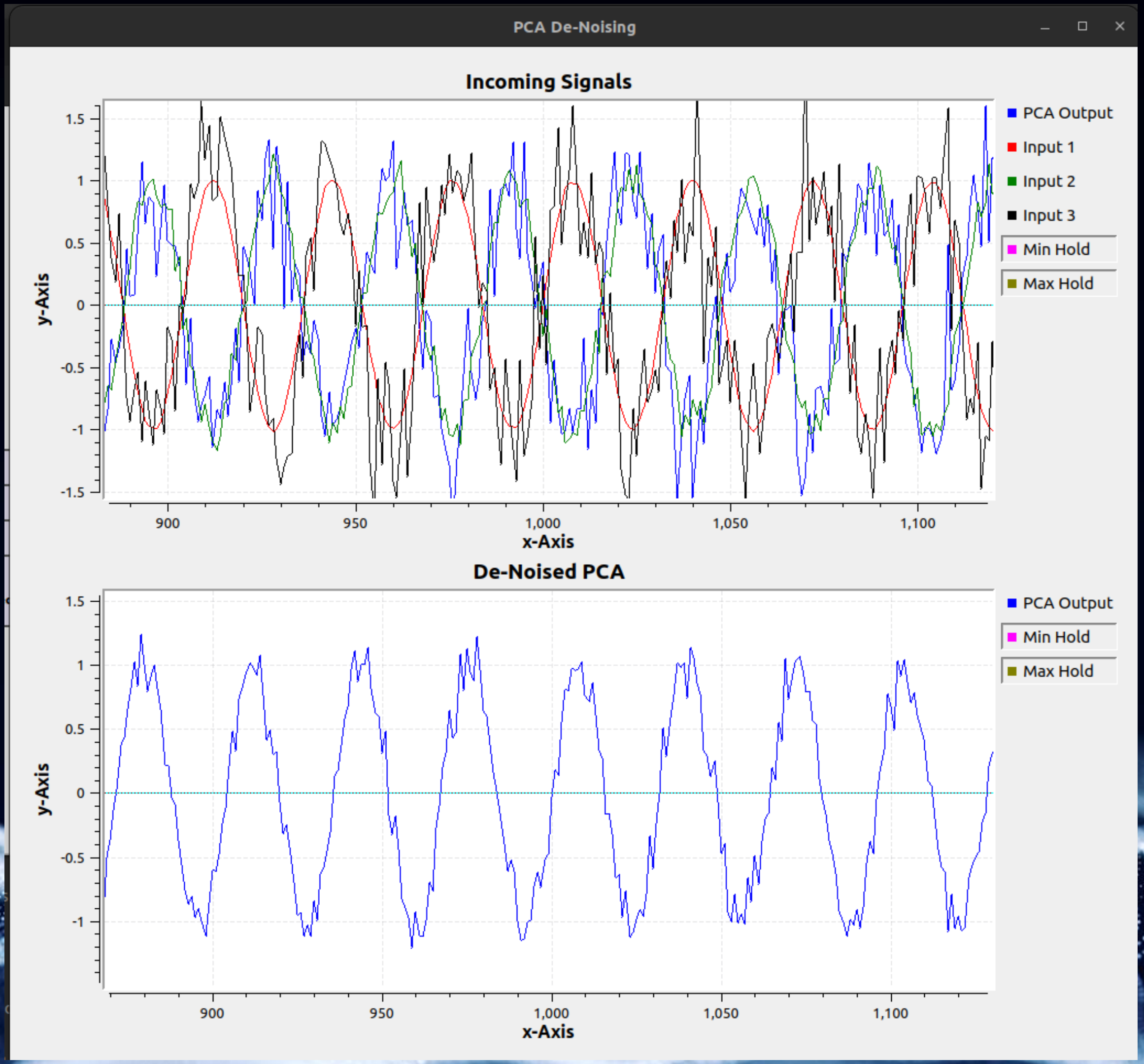
PCA CPU Implementation

C++ implementation in GNU Radio using armadillo



PCA CPU Implementation

C++ implementation in GNU Radio using armadillo



Common Techniques

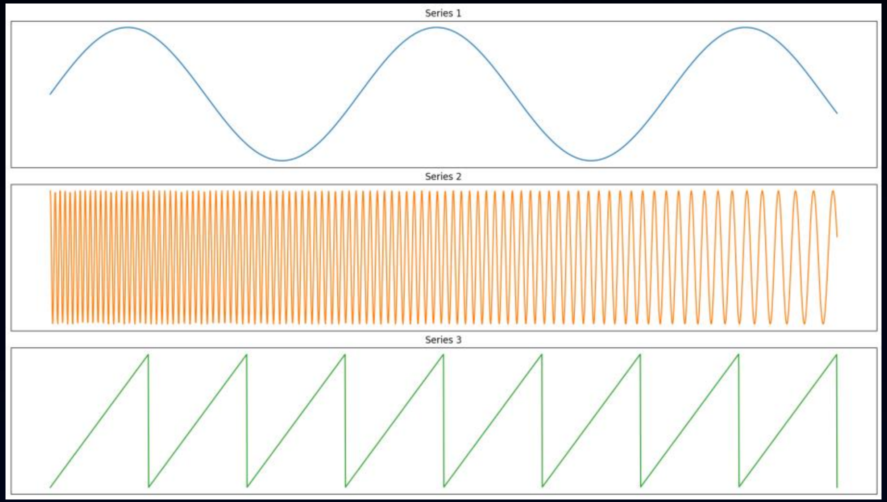
Independent Component Analysis

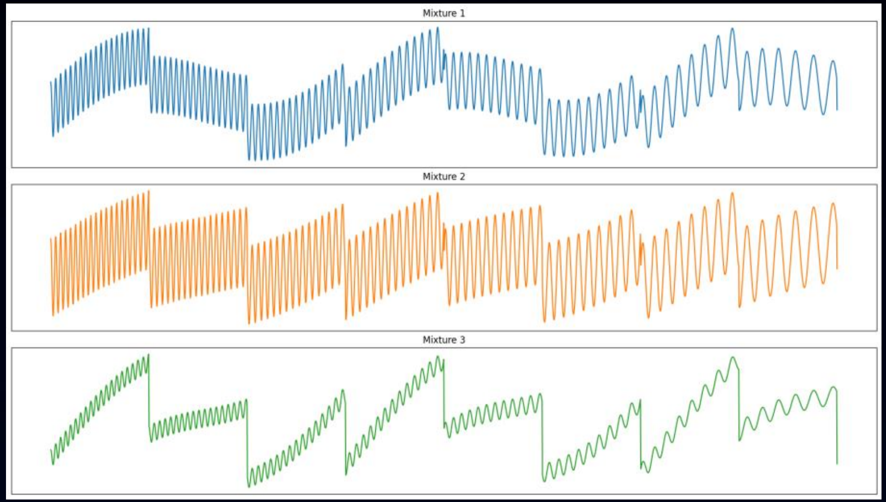
- ICA: Assume components of $\boldsymbol{s}(t)$ are statistically independent. At most one component is a Gaussian process.
  - Practically, this is done via minimizing mutual information through iterative optimization of $\boldsymbol{A}$.

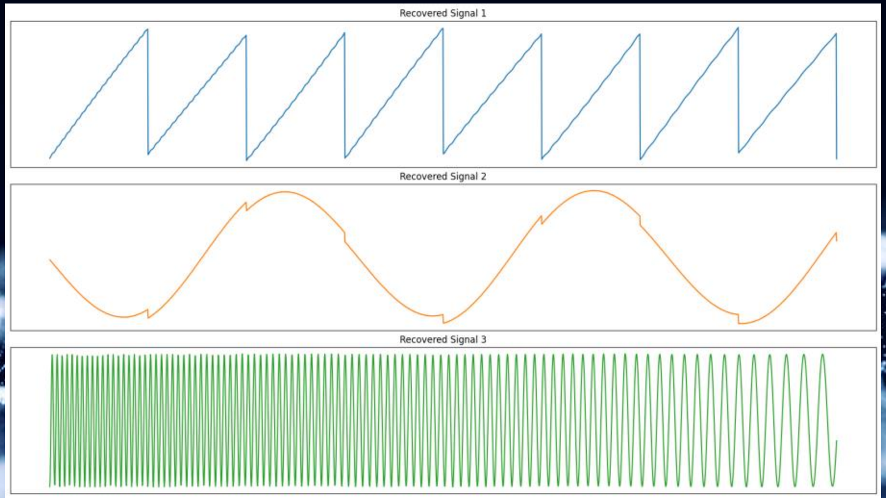
- Common use-case
  - Separating a mixture of unique signals for adaptive beamforming

ICA Example

Signal Separation







ICA CPU Implementation

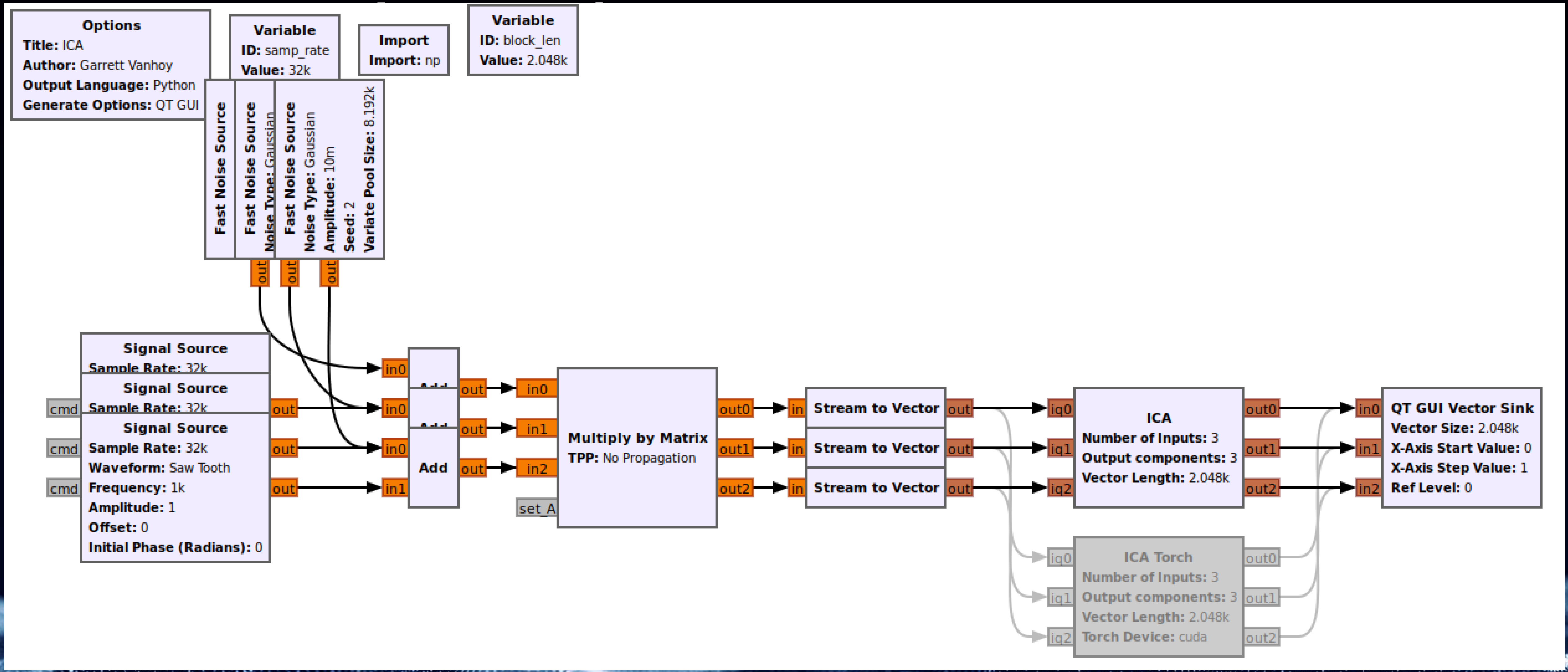
C++ implementation in GNU Radio using armadillo

```
using fmat = arma::fmat;

// Assemble X (n_samples x n_features)
fmat X(n_samples, n_features, arma::fill::none);
for (int j = 0; j < n_features; ++j) {
    auto src = static_cast<const input_type*>(in[j]);
    std::memcpy(X.colptr(j), src, static_cast<size_t>(n_samples) * sizeof(float)
}

// Center columns
X.each_row() -= arma::mean(X, 0);

// Whitening: Xw = X * V * D^{-1/2}
fmat Cov = (X.t() * X) / static_cast<float>(n_samples - 1);
arma::fvec eigval;
fmat eigvec;
arma::eig_sym(eigval, eigvec, Cov);
fmat D = arma::diagmat(1.0f / arma::sqrt(eigval + 1e-5f));
fmat V = eigvec;
fmat Xw = X * V * D * V.t();

// FastICA fixed-point iterations
fmat W(n_features, n_features, arma::fill::randu);
// Normalise each column vector of W to unit norm
for (int j = 0; j < n_features; ++j) {
    W.col(j) = arma::normalise(W.col(j));
}
```

ICA CPU Implementation

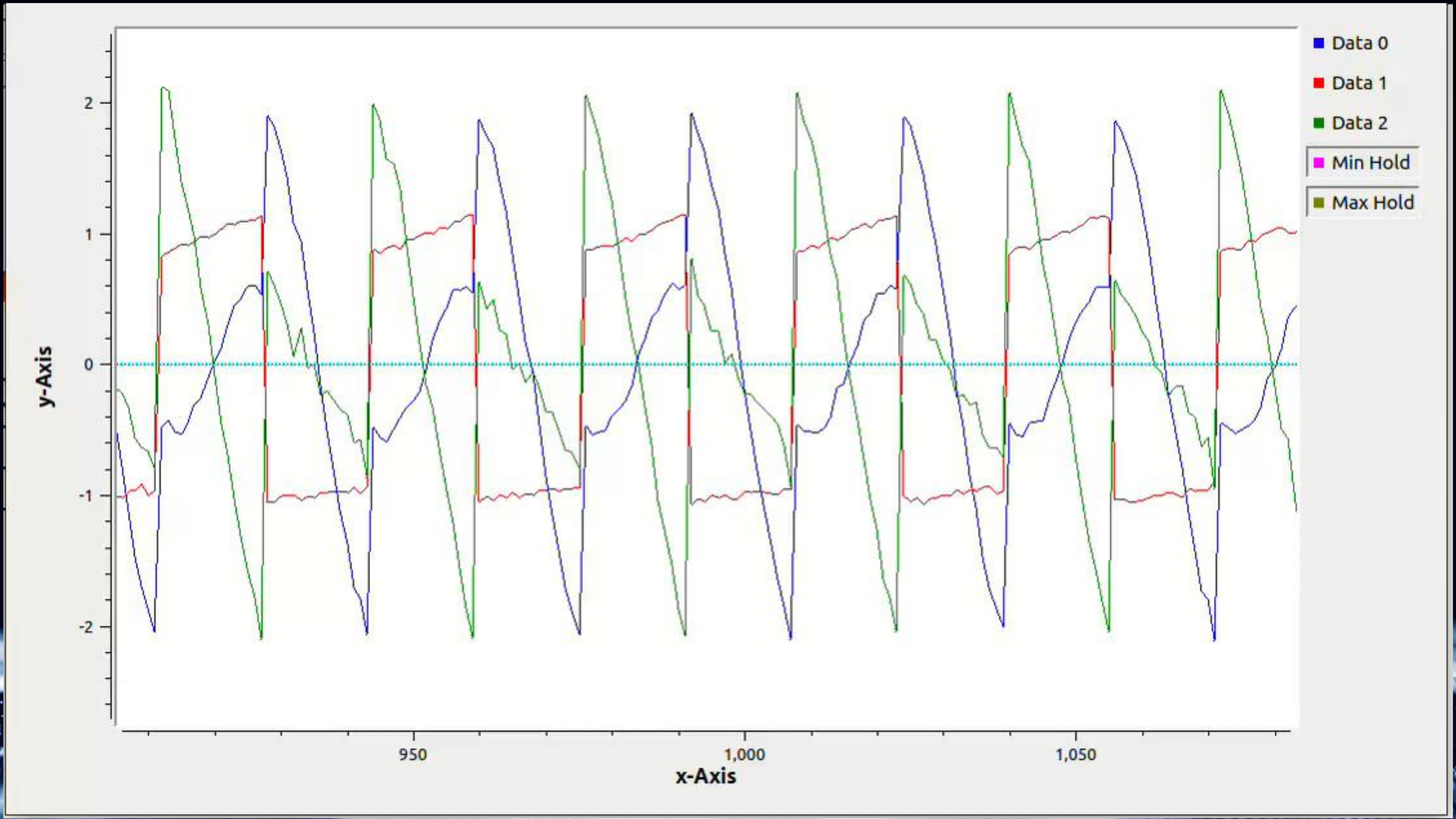
C++ implementation in GNU Radio using armadillo

```
for (int p = 0; p < n_features; ++p) {
    arma::fvec w = W.col(p);
    for (int it = 0; it < max_iter; ++it) {
        arma::fvec wx = Xw * w;
        arma::fvec gwx = g(wx);
        arma::fvec gpx = gprime(wx);

        arma::fvec w_new =
            (Xw.t() * gwx) / static_cast<float>(n_samples) - arma::mean(gpx) * w;

        // Decorrelate against previous components
        if (p > 0) {
            arma::fmat Wprev = W.cols(0, p - 1);
            w_new -= Wprev * (Wprev.t() * w_new);
        }

        w_new = arma::normalise(w_new);

        if (arma::norm(w_new - w) < tol || arma::norm(w_new + w) < tol) {
            w = w_new;
            break;
        }
        w = w_new;
    }
    W.col(p) = w;
}

// Independent components: S = Xw * W
fmat S = Xw * W;

// Copy each component column into its corresponding output buffer
for (int j = 0; j < n_features; ++j) {
    auto src = static_cast<output_type*>(out[j]);
    std::memcpy(src, S.colptr(j), static_cast<size_t>(n_samples) * sizeof(float));
}
```


Options
Title: ICA
Author: Garrett Vanhoy
Output Language: Python
Generate Options: QT GUI
Variable
ID: samp_rate
Value: 32k
Import
Import: np
Variable
ID: block_len
Value: 2.048k
Fast Noise Source
Fast Noise Source
Fast Noise Source
Noise Type: Gaussian
Amplitude: 10m
Seed: 2
Variate Pool Size: 8.192k
out
Signal Source
Signal Source
Signal Source
Sample Rate: 32k
Waveform: Saw Tooth
Frequency: 1k
Amplitude: 1
Offset: 0
Initial Phase (Radians): 0
cmd
in0
in1
Add
out
in2
set_A
Multiply by Matrix
TPP: No Propagation
out0
out1
out2
in
Stream to Vector
Stream to Vector
Stream to Vector
iq0
iq1
iq2
ICA
Number of Inputs: 3
Output components: 3
Vector Length: 2.048k
ICA Torch
Number of Inputs: 3
Output components: 3
Vector Length: 2.048k
Torch Device: cuda
QT GUI Vector Sink
Vector Size: 2.048k
X-Axis Start Value: 0
X-Axis Step Value: 1
Ref Level: 0

ICA CPU Implementation

C++ implementation in GNU Radio using armadillo



Pitfalls

- Requires significant modification to handle *complex* signals
- They assume *a-priori* knowledge of the number of original sources.
- The output is ambiguous up to phase and order

Peraton

Adaptive Event Processing

A Practical BSS technique for Wireless Signals

Adaptive Event Processing

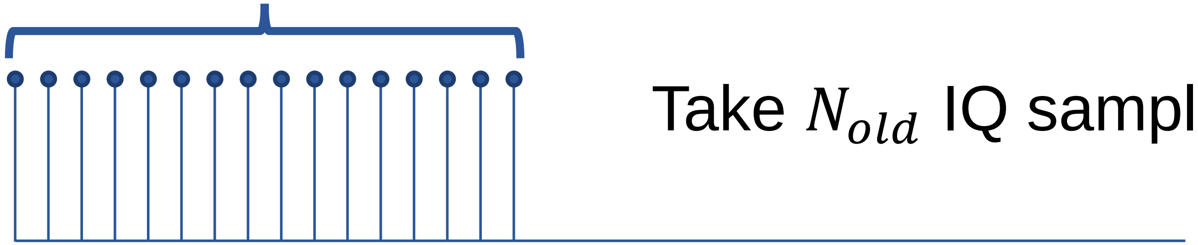
Intuition behind the algorithm

- Instead of assuming a relationship between sensor measurements, we assume a relationship between sensor measurements over time as signals appear and disappear

- The correlation between sensor measurements will give us information about how to separate out newly appearing signals from others

Adaptive Event Processing

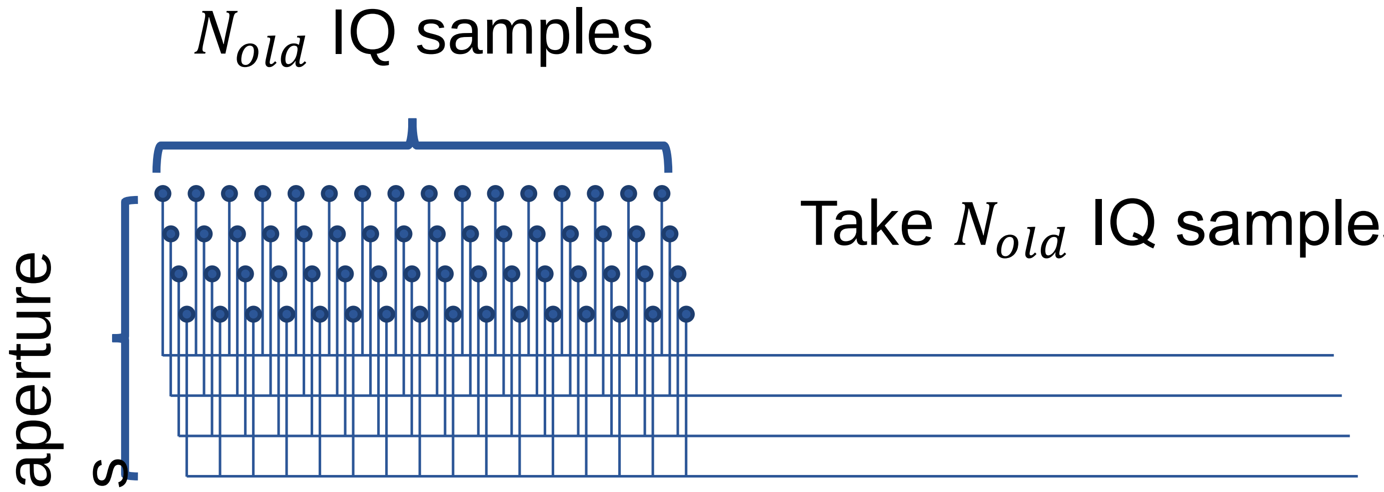
N_{old} IQ samples



Take N_{old} IQ samples

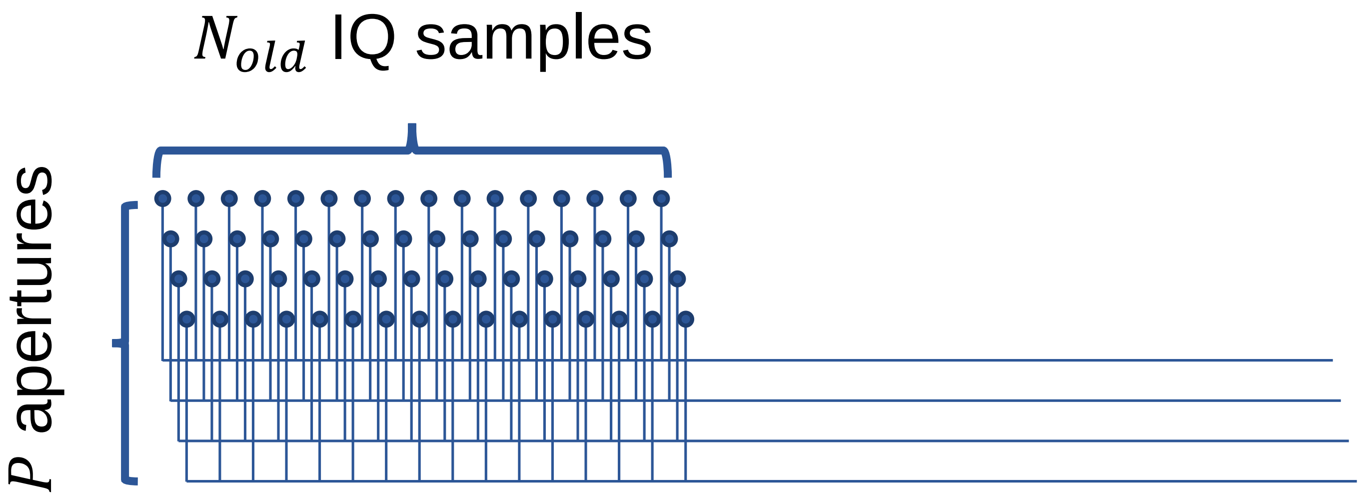
Adaptive Event Processing

Peraton



Take N_{old} IQ samples … across P apertures

Adaptive Event Processing

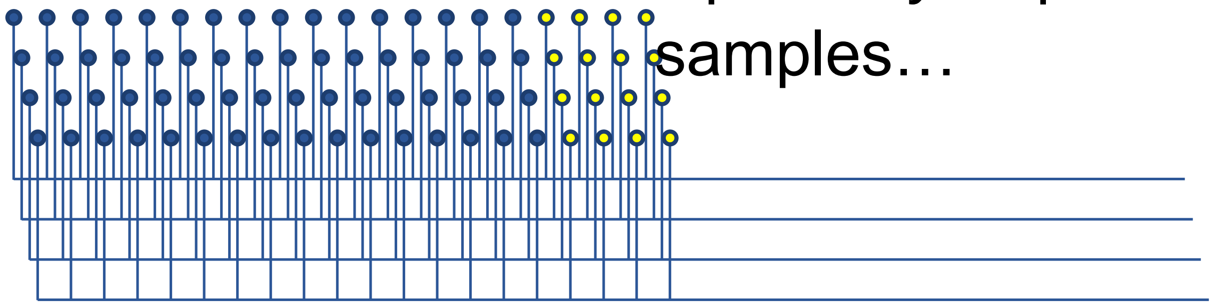


Calculate the covariance matrix across this segment

$$R_x^{old} = \left\{ \begin{matrix} 1 & \frac{E[(X_1-\mu_1)(X_2-\mu_2)]}{\sigma(X_1)\sigma(X_2)} & \cdots & \frac{E[(X_1-\mu_1)(X_P-\mu_P)]}{\sigma(X_1)\sigma(X_P)} \\ \frac{E[(X_2-\mu_2)(X_1-\mu_1)]}{\sigma(X_2)\sigma(X_1)} & 1 & \cdots & \frac{E[(X_1-\mu_1)(X_2-\mu_2)]}{\sigma(X_1)\sigma(X_2)} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{E[(X_P-\mu_P)(X_1-\mu_1)]}{\sigma(X_P)\sigma(X_1)} & \frac{E[(X_P-\mu_P)(X_2-\mu_2)]}{\sigma(X_P)\sigma(X_2)} & \cdots & 1 \end{matrix} \right\}$$

Adaptive Event Processing

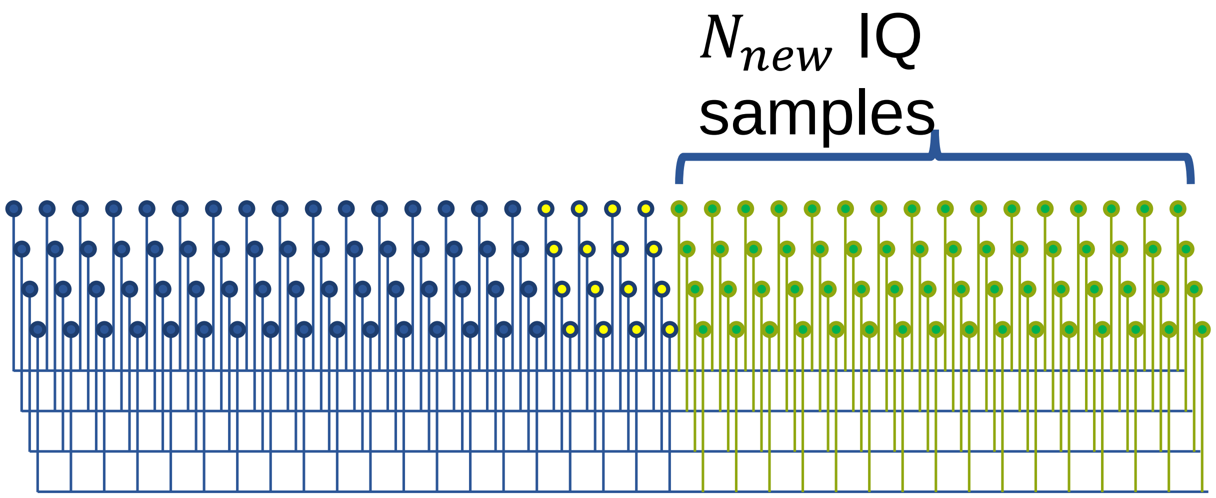
Peraton



Optionally skip over N_{gap} IQ samples…

Adaptive Event Processing

Peraton

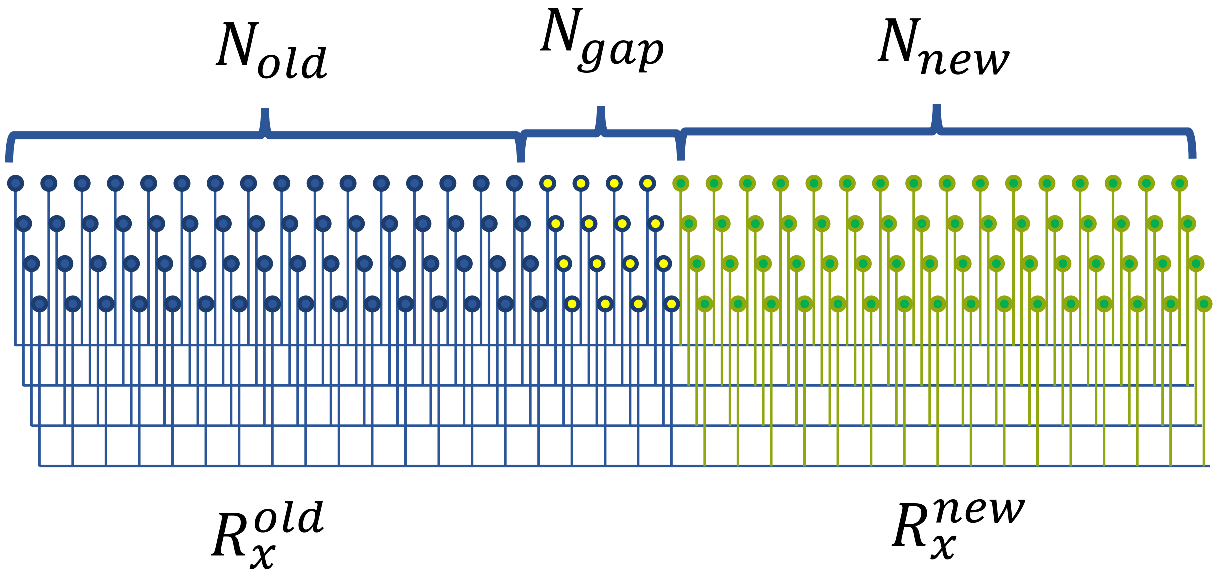


Take N_{new} samples and calculate R_x^{new}.

Now we have everything…

Adaptive Event Processing

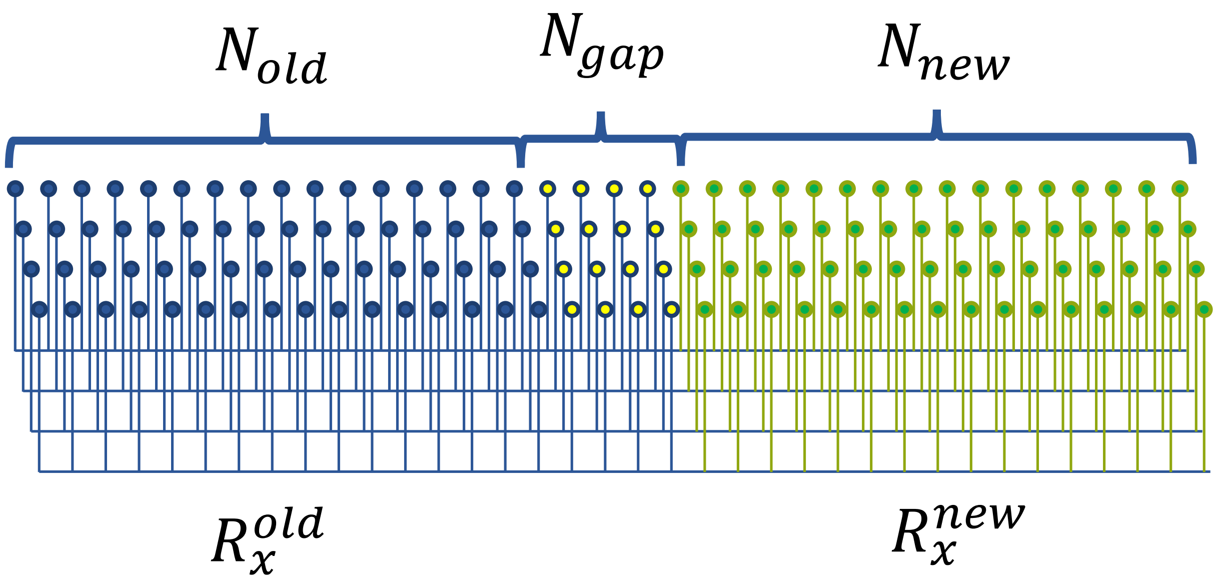
Peraton



One could refer to R_x^{old} and R_x^{new} as *spatial signatures* as these quantities capture the relationship of data between spatially separated elements.

Adaptive Event Processing

Peraton

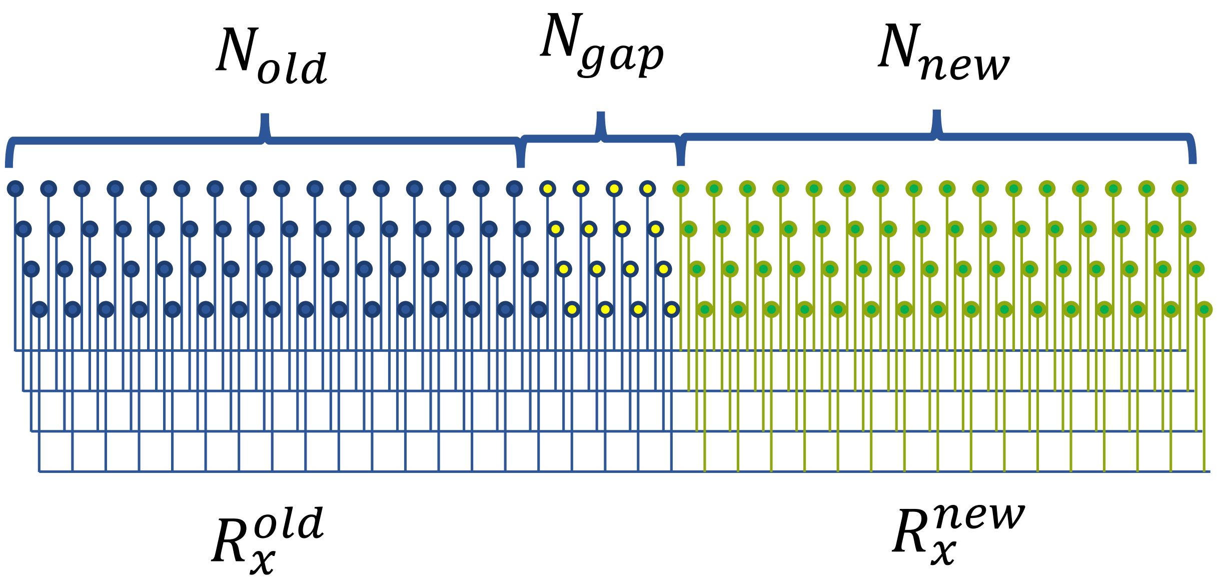


Now, to <u>detect *change*</u> between the time segment of N_{old} samples and N_{new} samples, we can look at:

$$A = {R_x^{old}}^{-1} R_x^{new}.$$

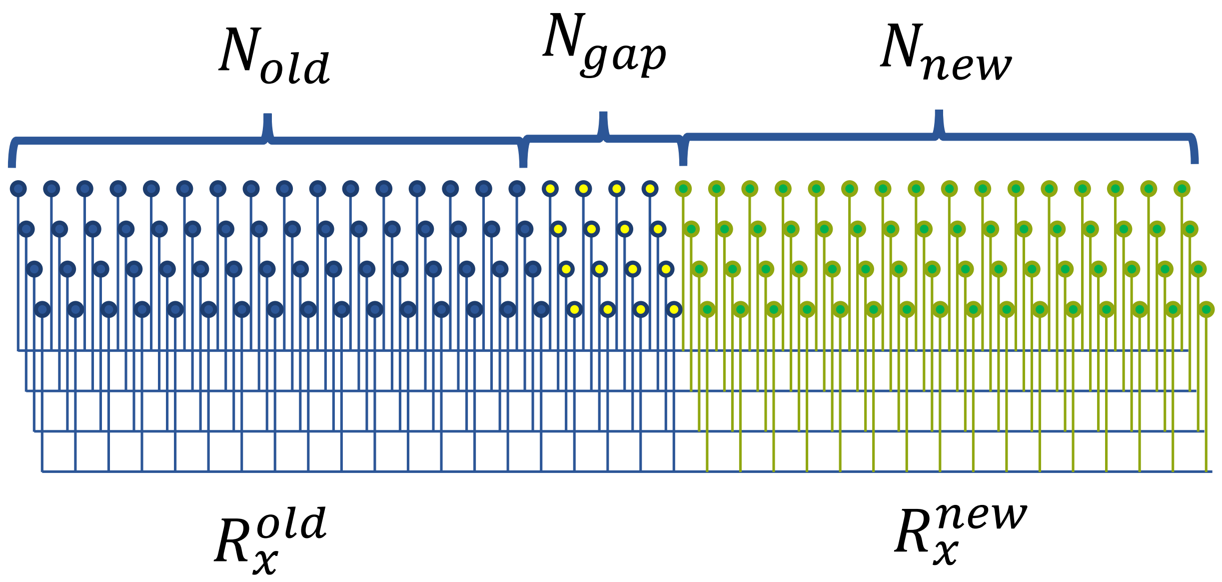
If the matrices R_x^{old} and R_x^{new} are nearly the same, A will be the identity matrix.

Adaptive Event Processing



Doing eigenvalue decomposition on A and taking the *largest* eigenvalue λ_{max}, we can build a *detector*.

Adaptive Event Processing



Taking the corresponding eigen-vector v_{max}, we can create a mixture that **highlights the signal that caused the change** by left-multiplying the incoming stream of data:

$\boldsymbol{X} = v_{max}^H \boldsymbol{Y}$, where v_{max}^H denotes conjugate-transpose.

Adaptive Event Processing

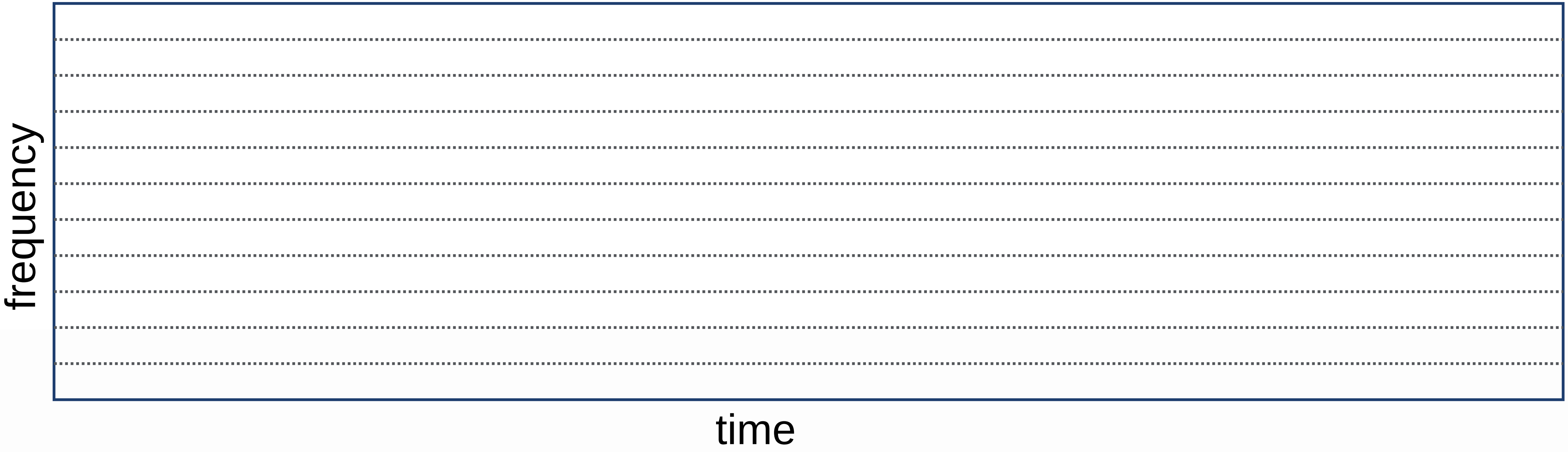
Peraton

This only works in a *narrowband* manner.

Adaptive Event Processing

Peraton

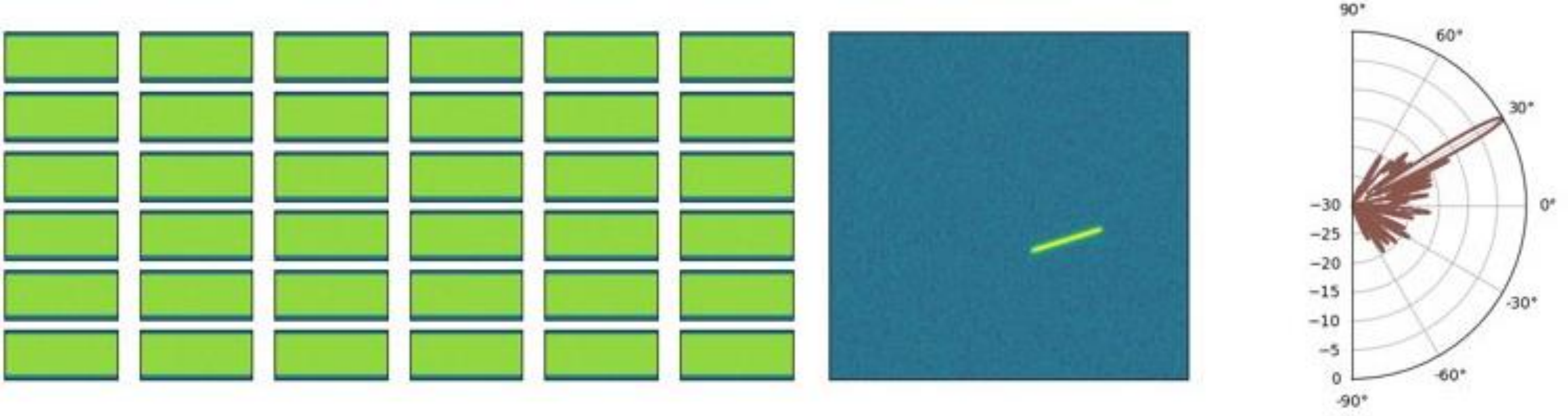
This only works in a *narrowband* manner.



We will have to split the spectrum into smaller frequency bands and detect.

Sub-channels can be created with a simple STFT

Adaptive Event Processing



AEP CPU Implementation

Implementing with armadillo

```
for (int block_idx = 0; block_idx < noutput_items; block_idx++) {
    // Detect Change
    unsigned int block_offset = block_idx * d_n_iqs * d_fft_size;
    memcpy(
        d_iq.memptr(), &in[block_offset], sizeof(gr_complex) * d_n_iqs * d_fft_size);


    d_R_new = d_iq * d_iq.t();
    d_R_new += d_eye;

    // Whiten by old R
    try {
        d_R_solve_max = solve(d_R_old, d_R_new, d_opts);
        float max_eig = abs(trace(d_R_solve_max));
        if (max_eig > d_threshold) {
            eig_gen(d_eig_vals, d_eig_vecs, d_R_solve_max);
            memcpy(d_weights, d_eig_vecs.colptr(0), sizeof(gr_complex) * d_n_iqs);
        }
    } catch (const std::runtime_error& error) {
        GR_LOG_WARN(d_debug_logger, error.what());
    }

    d_R_old = d_R_new;

    // Apply weights
    aep_beamform(out, block_idx, in);
}

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

Peraton

Implementation with libtorch

Alternative CPU and GPU implementation

Libtorch Integration

Using libtorch, the C++ interface to torch ML library

- Why?
  - To enable GPU and alternative CPU implementation
  - Integration with other devices that Torch works on
  - Use of multiple devices/cpu threads

- Certainly other ways to do it

- Plenty of room for runtime optimization

Peraton

Libtorch Integration Challenges

Using libtorch, the C++ interface to torch ML library

- libtorch uses different versions of common libraries than GR 3.10.x
  - CMake adjustments
  - .devcontainer files for reproducible environment

- API is different than armadillo
  - Search for alternatives

- Linear algebra operation options and are different
  - Different linear algebraic operations can work

Results

Using libtorch, the C++ interface to torch ML library

- ICA:
  - Armadillo: 21 MS/s
  - Torch CPU: 3.33 MS/s GPU: 6.66 MS/s

Results

Using libtorch, the C++ interface to torch ML library

- PCA:
  - Armadillo: CPU: 120 MS/s
  - Torch CPU: 75 MS/s GPU: 25 MS/s

Results

Using libtorch, the C++ interface to torch ML library

- AEP:
  - Armadillo: CPU: 125 MS/s
  - Torch CPU: 33 MS/s GPU: 66 MS/s

Peraton

Thank you

Code available at github.com/gvanhoy/gr-bss