Deimos Controls LLC

James Logan

Deimos

Open-Source Scientific Data Acquisition & Laboratory Controls

DOI 10.5281/zenodo.15512326

2025-06-06

InterpN

N-Dimensional Interpolation

DOI 10.5281/zenodo.15512228

Flaw

Digital Filtering

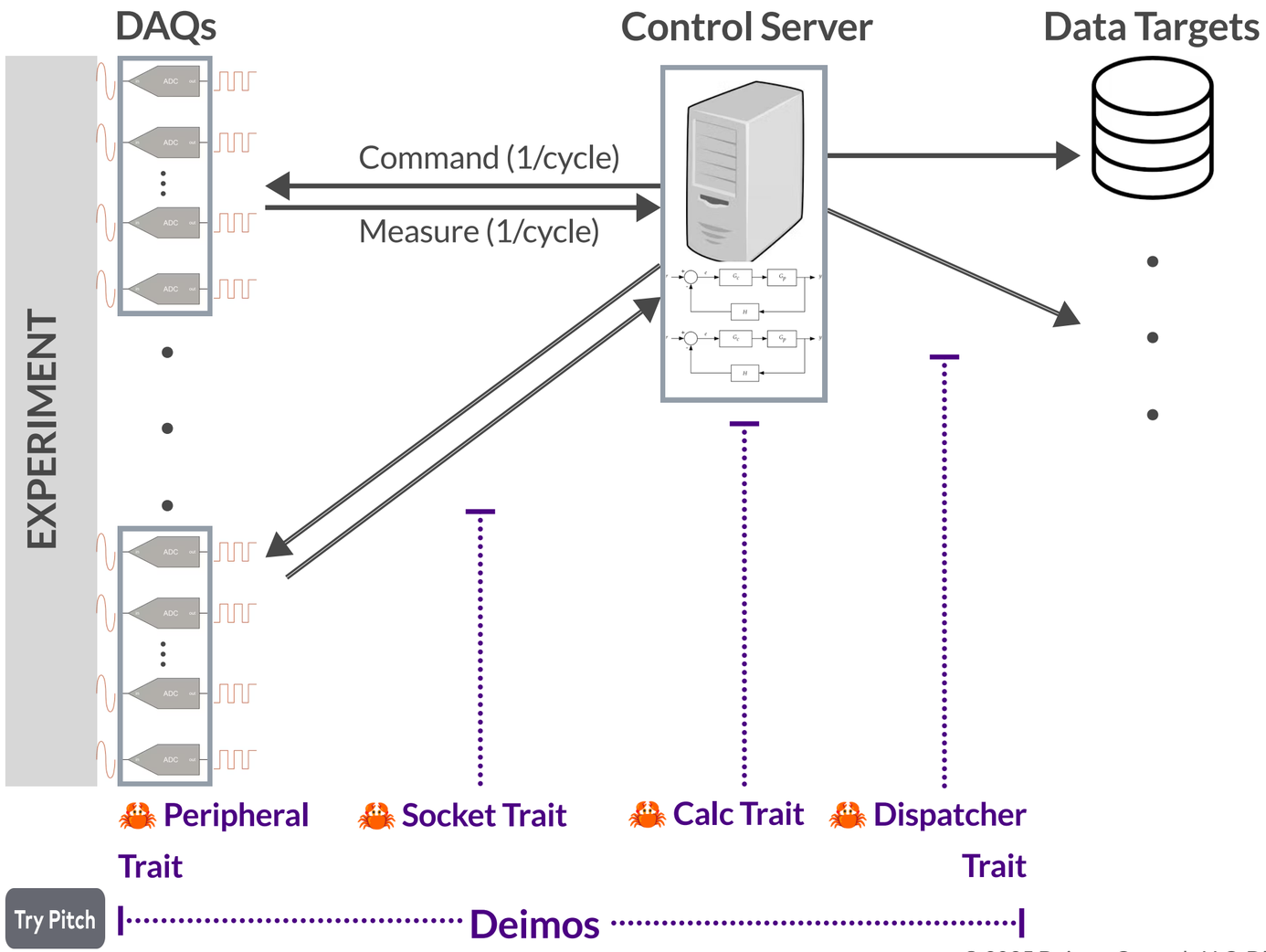
DOI 10.5281/zenodo.15512059

Try Pitch

support@deimoscontrols.com

Deimos Controls LLC

What's a DAQ?



We need data to validate computation & designs

📊 Scientific computing starts and ends with data

♻️ Experiments require control as well as measurement

but…

💲 Systems for collecting decision-quality data are expensive

🍝 Controls are often desynchronized from measurement

🔒 Uncertainties are opaque, proprietary, or incorrect

⛓️💥 Data pipelines are fragile and inflexible

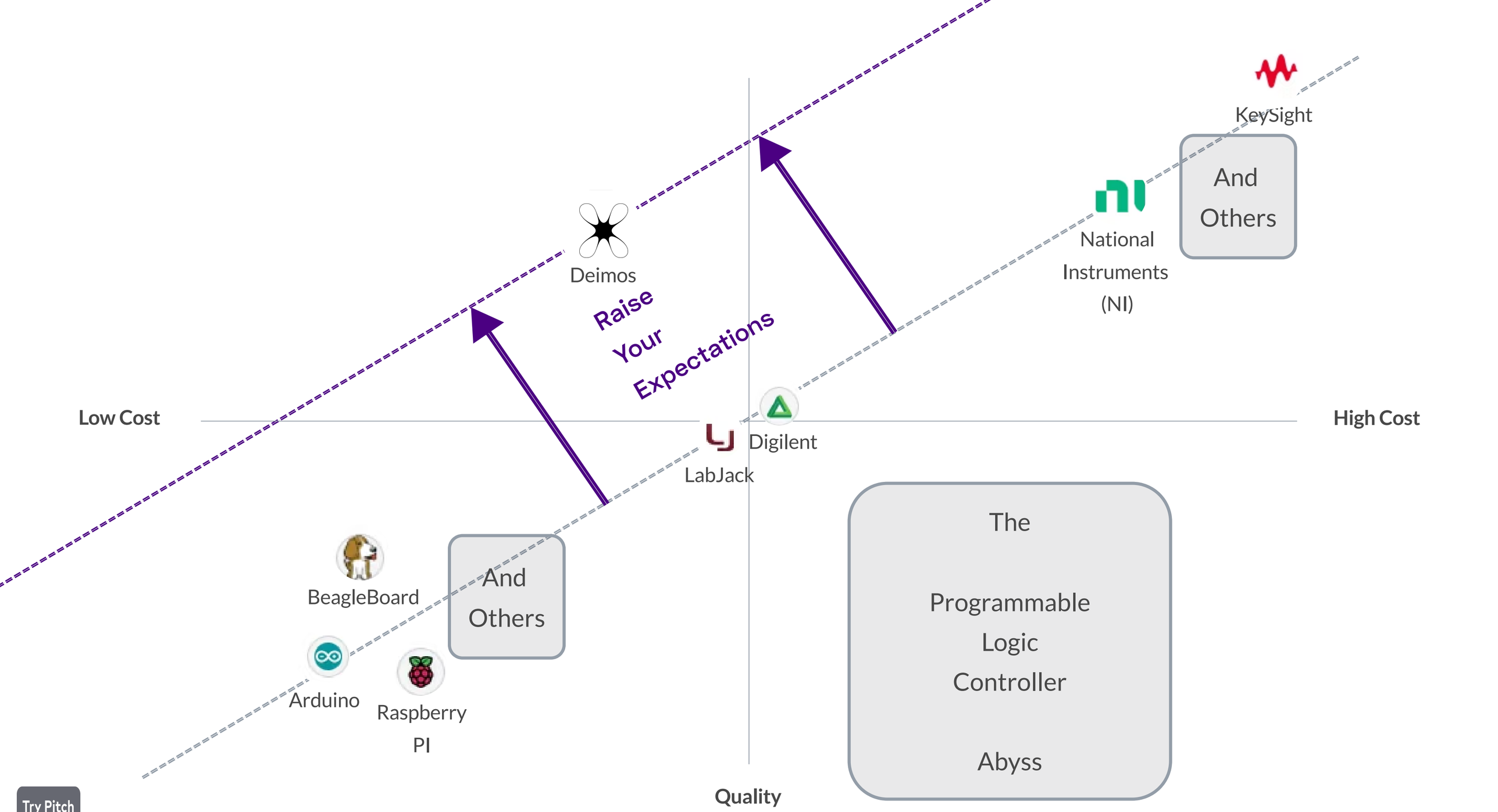
We can do better

📦 Open-source 'em all

🎯 Publish uncertainty analysis and filter dynamics

⏱️ Provide cross-domain sample time sync by default

🤝 Integrate sensor analog frontends & eliminate backplanes



Try Pitch

NO

NO

Calibrations

Use precision components and propagation of uncertainty!

Test to *confirm* accuracy for quality-control only - no need to carry unit-specific calibration coefficients.

NO

Configuration

Channels either exist or they don't. Users don't have to solve an integer linear program to figure out whether the system works for their application!

Onboard digital filters adapt to chosen data rate automatically.

NO

Subscriptions & Licenses

Purchase is ownership.

NO

Platform Restrictions

Operating Systems
Windows/Linux/Mac - if Rust has a standard library for it, it will run.

Networking & Control Hardware
Any network switch will do.
No need to buy a special control server.
No expensive PTP root clock.

JUST NO

Meet the Hardware

100% open-source hardware, firmware, software, and toolchain.

Development status late-stage prototyping (14th integrated prototype!), available soon

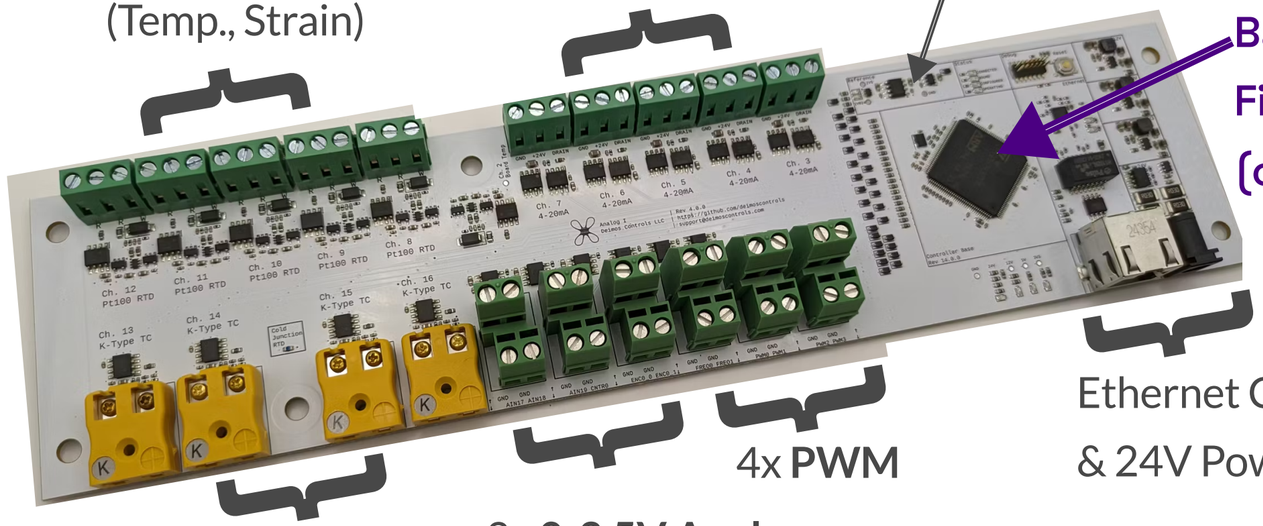
More hardware module varieties to come! Valve drivers, SPI/I2C interfaces, etc

5x **Resistance**
(Temp., Strain)

5x **4-20mA**
(Pressure, Angle, Etc)

0.02% Voltage Ref.

🦀 **100% Rust Bare-Metal Firmware (or BYO!)**

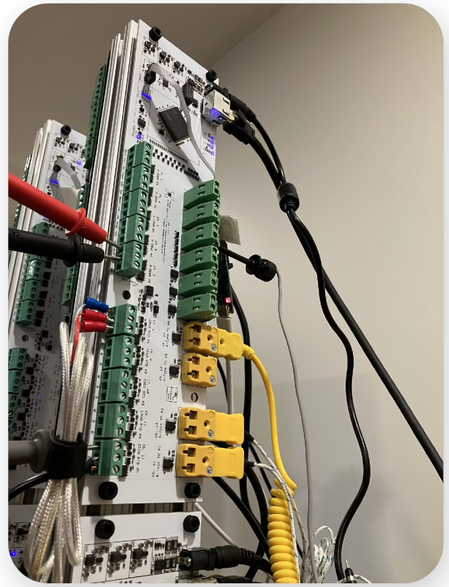


Ethernet Comms
& 24V Power

4x **PWM**

4x **Thermocouple**
(K-Type, Temp.)
w/ Cold-Junction
Compensation

3x **0-2.5V Analog**
2x **Frequency**
1x **Pulse Counter**
1x **Encoder**



8 units sync'd @ 5Hz-2kHz
(~200 raw channels!)

Deimos Controls LLC

Logic: Programmatic or Graphical - Take your Pick

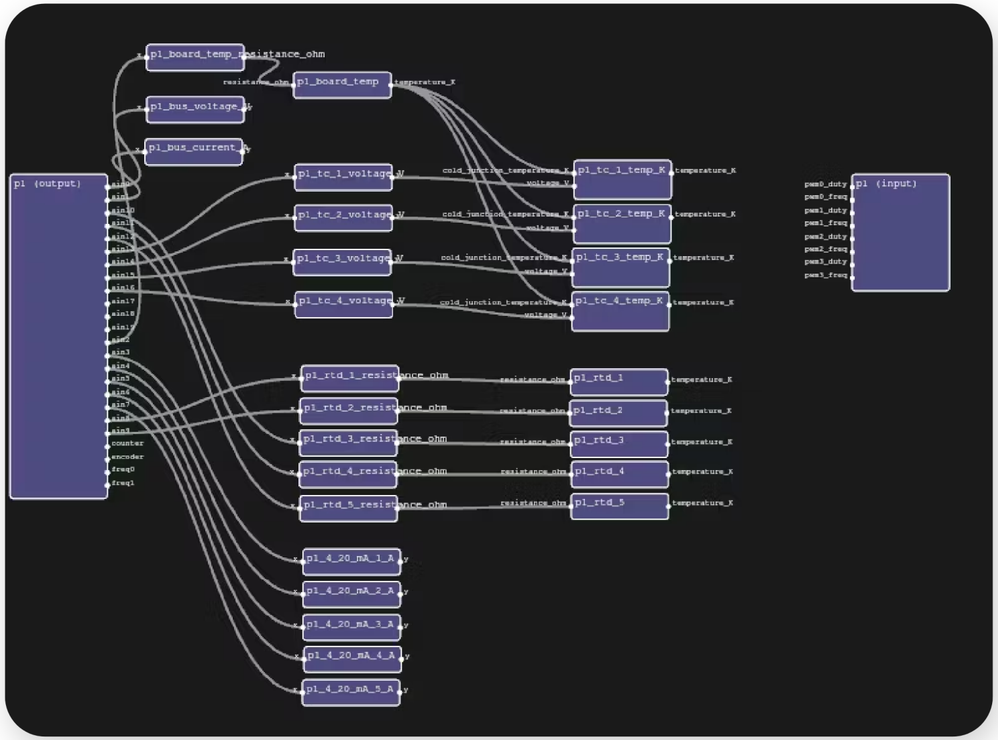
- 🦀 Control program written in Rust
  - Rich diff on version-controlled per-run configuration
  - First-class plugins for custom calculations and hardware
- 🦀🚧 Node-graph GUI will allow inspecting and configuring server-side computation
- 🐍🚧 Python bindings planned

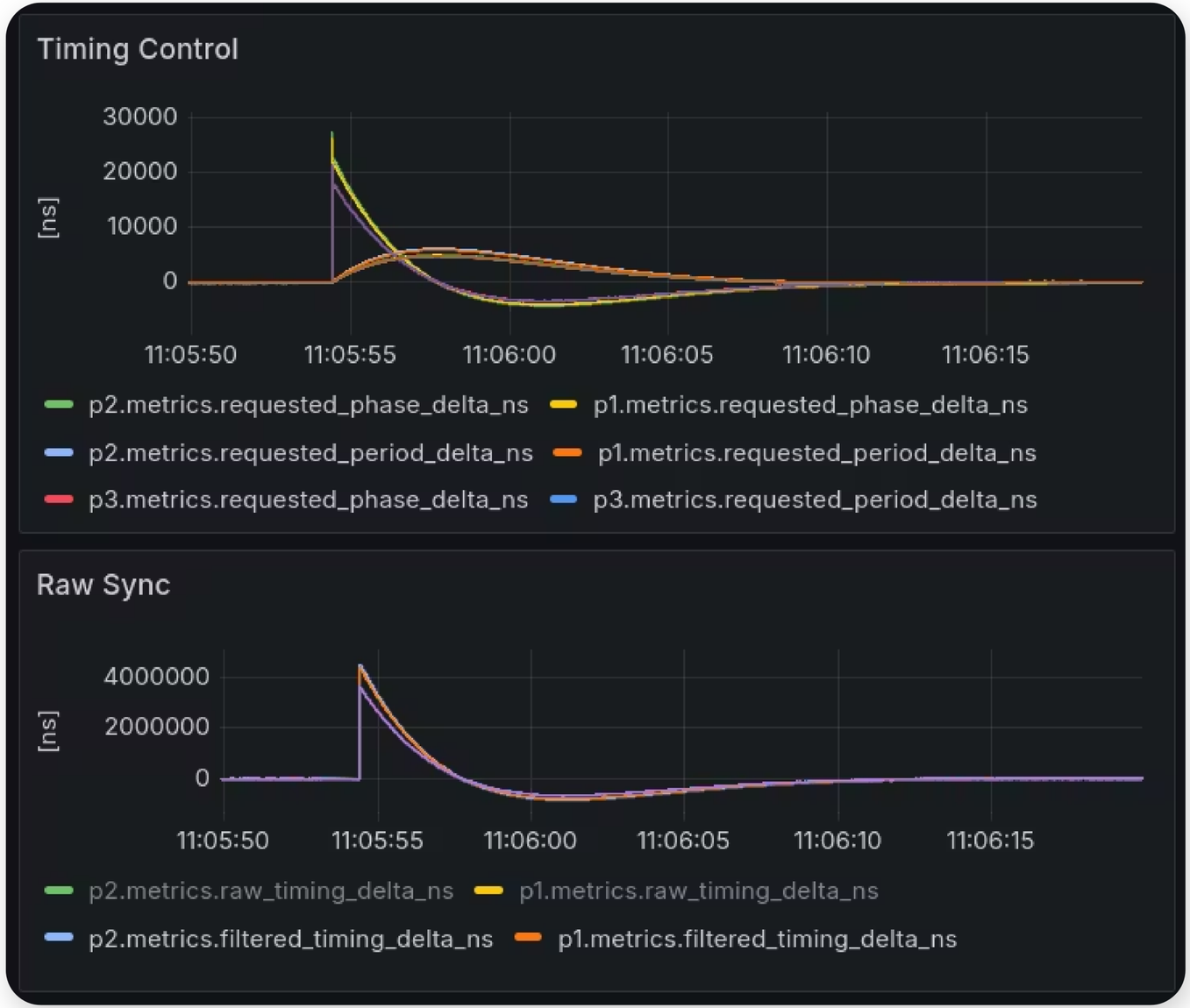
Code-First

```rust
// Define idle controller
let mut ctx: ControllerCtx = ControllerCtx::default();
ctx.op_name = "basic_example".into();
ctx.dt_ns = (1e9_f64 / 1000.0).ceil() as u32; // 1000 Hz
ctx.op_dir = "./software/deimos/examples".into();
let mut controller: Controller = Controller::new(ctx);

// Associate hardware peripheral(s)
controller.add_peripheral(name: "p1", p: Box::new(AnalogIRev4 { serial_number: 1 }));

// Set up data targets
let csv_dispatcher: Box<dyn Dispatcher> =
    Box::new(CsvDispatcher::new(
        chunk_size_megabytes: 50,
        overflow_behavior: dispatcher::Overflow::Wrap));
controller.add_dispatcher(csv_dispatcher);

// Serialize and deserialize the controller (for demonstration purposes)
let serialized_controller: String = serde_json::to_string_pretty(&controller).unwrap();
let _: Controller = serde_json::from_str(&serialized_controller).unwrap();
// std::fs::write("./basic_example.json", &serialized_controller).unwrap();

// Run control program
controller.run(plugins: &None).unwrap();
```

Try Pitch

GUI Under Construction





Software-Only

Distributed Time-Sync

TOTAL

Sub-microsecond time-sync between modules is achieved in just a few seconds **without specialized hardware.**

LOCAL

Peripheral time-synchronization is achieved by **closed-loop control in software** - hardware peripherals received requests to adjust their cycle timing to align with the control program.

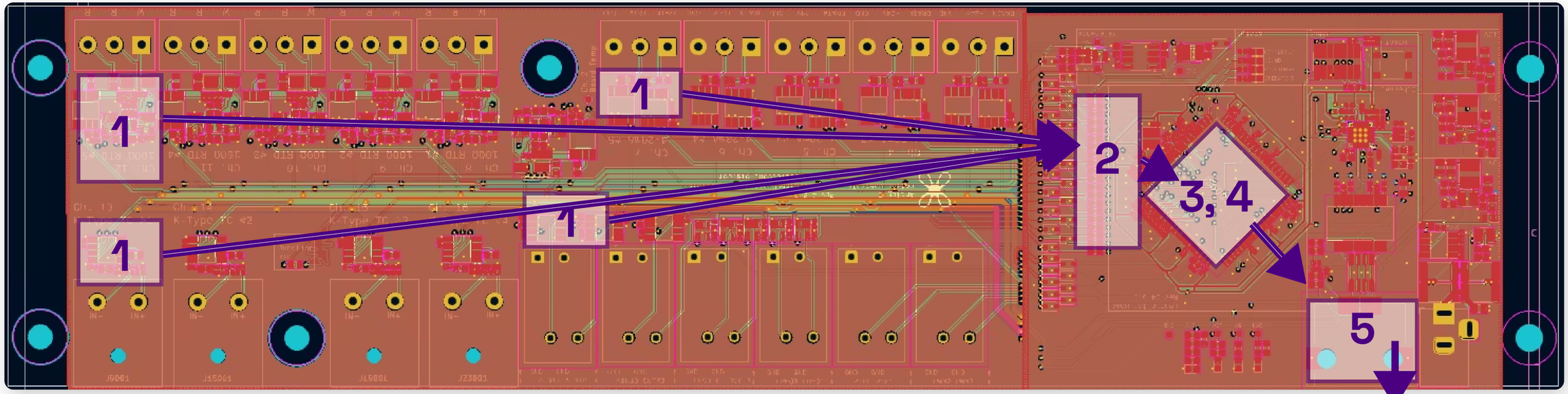
GLOBAL

Meanwhile, the control program uses **system monotonics**, which are updated by the operating system to gradually adjust to **align with network time via NTP**.

Try Pitch

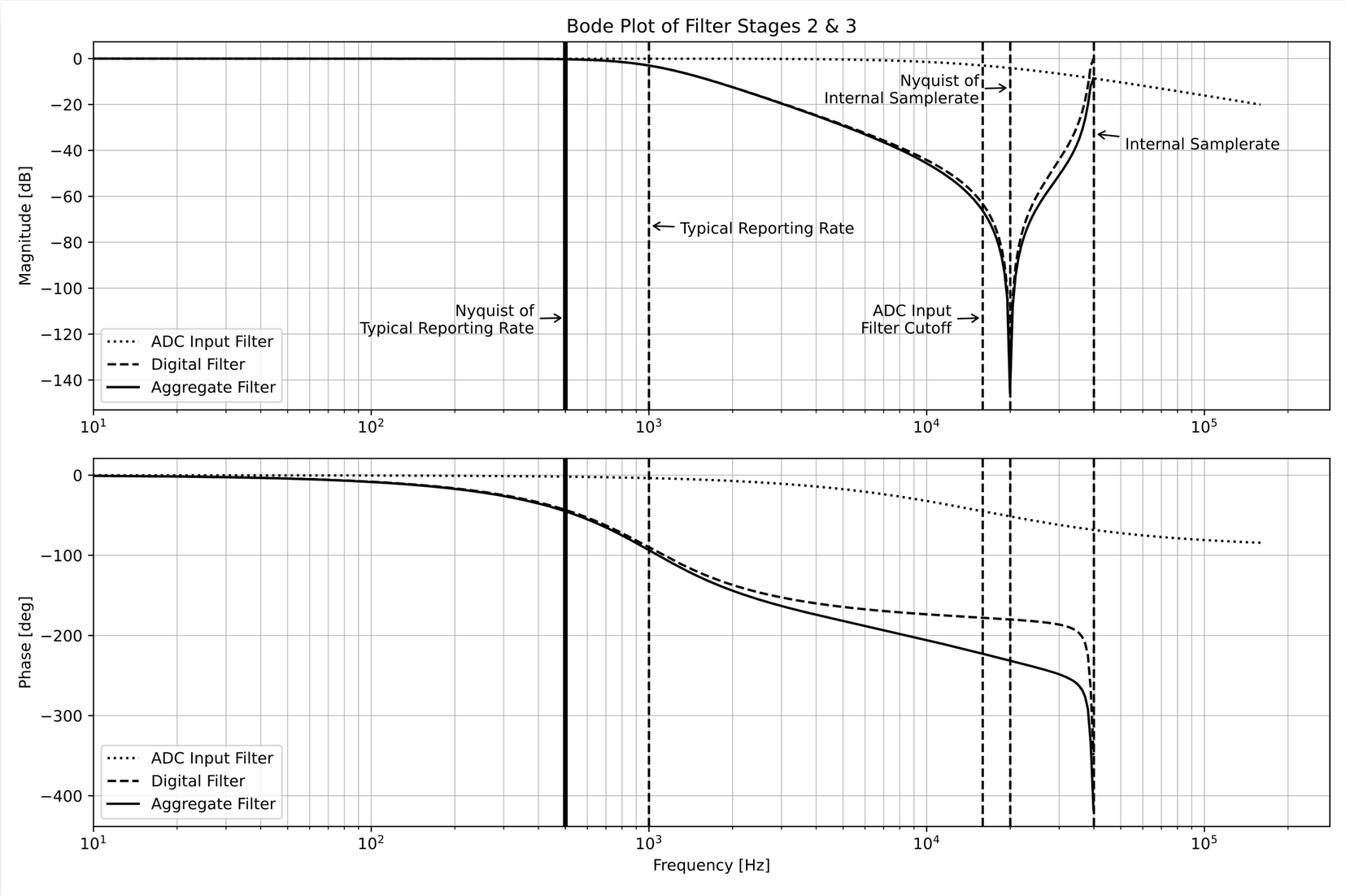
Filters 1/2

- 40 kHz ADC burst scanning @ 16 bit res; 800kS/s overall; 330ns sample-hold in 8 batches → 2.64us / scan (11%)
- Analog
  1. Case-specific analog filter at frontend (to reduce aliasing on RF noise)
  2. 16kHz analog filter at ADC inputs (to reduce pulldown & crosstalk as well as aliasing) (future: Bessel filters)
- Digital
  3. Future: Lagrange polynomial fractional-delay FIR filter for synthetic simultaneous sampling
  4. Adaptive IIR filter (Butterworth) initialized to match chosen reporting rate
  5. Decimation to reporting rate



Filters 2/2

- ADC input filter protects input drawdown
  - Subject to limits on peak amplifier current & output capacitance
  - Does not provide much actual filtering
  - First line of defense is analog frontend
- Digital filter does the heavy lifting
  - About 45deg phase lag @ Nyquist
  - Allows some aliasing in exchange for reduced phase distortion



Deimos
Controls
LLC

Rust

DOI 10.5281/zenodo.15512059

Flaw

Embedded digital filtering w/ lightweight adaptivity (no-std + no-alloc)

Try Pitch

Digital IIR Filtering in State-Space Form

- State-space representation in canonical form provides minimum number of nonzero filter coeffs
- No actual matrix operations needed - the update reduces to a couple vector products
- Normalize filters to samplerate - only one Pi group here

The system

Time-delays

Only one non-trivial row

$$A = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ -a_n & -a_{n-1} & -a_{n-2} & \dots & -a_1 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}, \quad C = \begin{bmatrix} c_n & c_{n-1} & c_{n-2} & \dots & c_1 \end{bmatrix}, \quad D = \begin{bmatrix} d_0 \end{bmatrix}$$

The state update

Vector dot & shift

Trivial

Filter internal state vector `x`

$$x_{k+1} = Ax_k + Bu_k$$

Measurement sample `u`

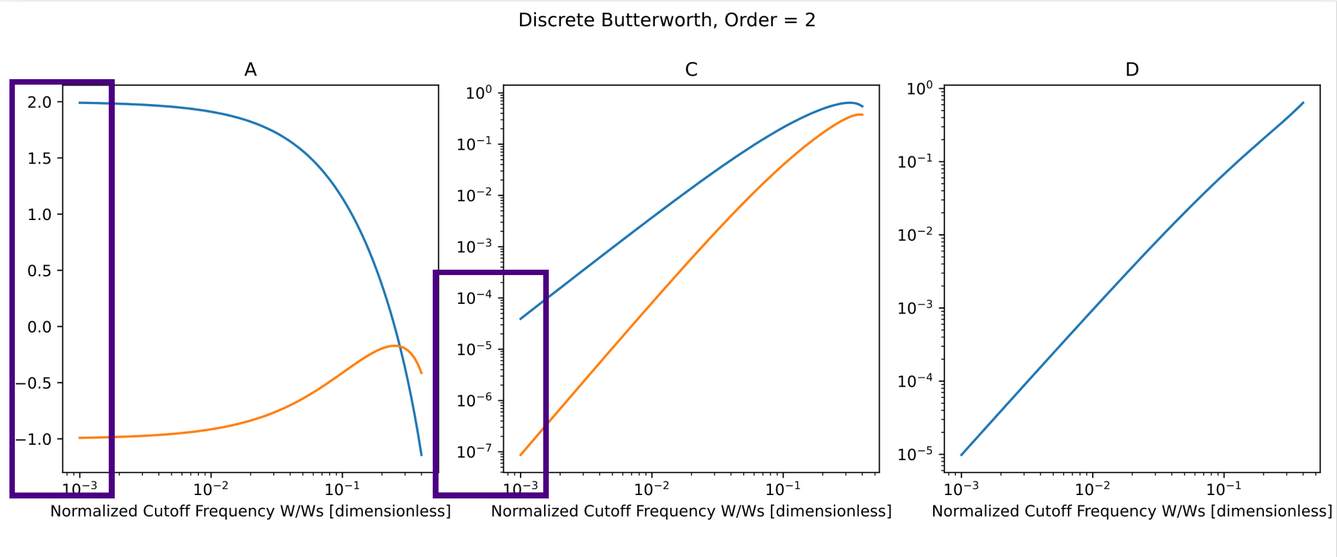
$$y_k = Cx_k + Du_k$$

Filtered measurement output

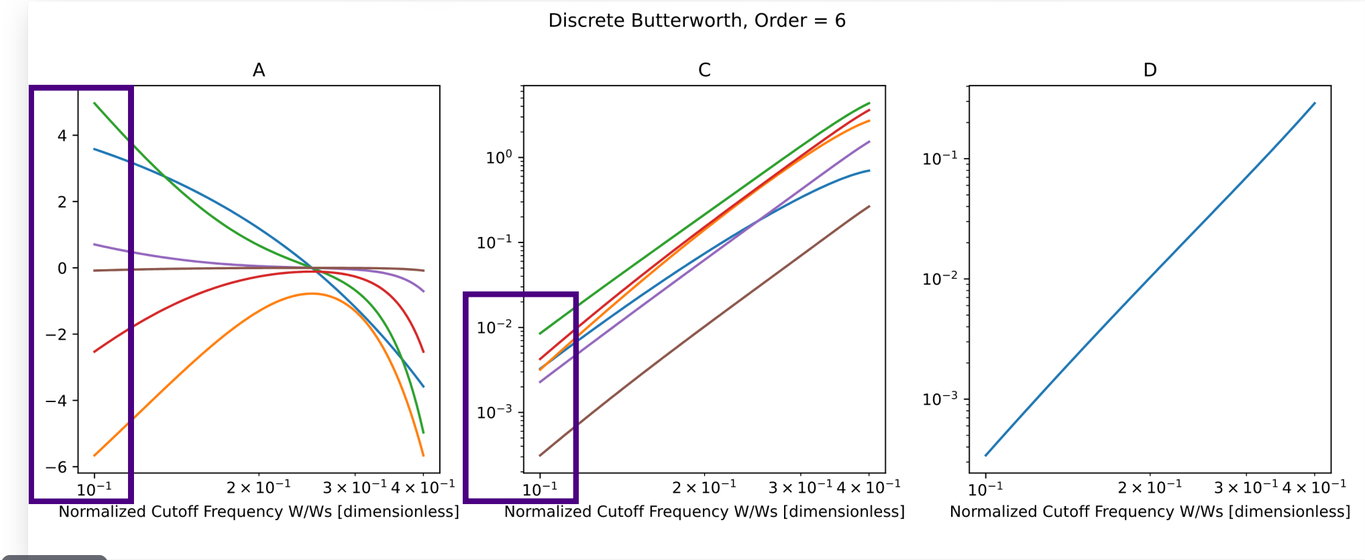
Vector dot

Trivial

Try Pitch



Interpolate on these to make a filter



Live Initialization &

32-Bit Floats

```rust
let mut out: T = start;
//    Sum the second contiguous segment first because it will have smaller values
//    for low-pass IIR filters, so this substantially improves float precision.
//    The tests don't pass without this as of 2025-05-14!
a_parts(&[T], &[T])
    .1 &[T]
    .iter() Iter<'_, T>
    .zip(x_parts.1.iter().rev()) impl Iterator<Item = (&T, &T)>
    .for_each(|(&aval: T, &xval: T)| out = out + aval * xval);
//    Sum the first contiguous segment
a_parts(&[T], &[T])
    .0 &[T]
    .iter() Iter<'_, T>
    .zip(x_parts.0.iter().rev()) impl Iterator<Item = (&T, &T)>
    .for_each(|(&aval: T, &xval: T)| out = out + aval * xval);
```

- Small number of nontrivial entries in filter
- Only one real independent input to filter construction
- Constructed live on micro via no-std interpolation
- Limited domain where sums on coeffs are representable
  - f32::EPSILON ≈ 1e-7
  - Stiffness increases for lower cutoff frequency
  - Nyquist limits upper cutoff
  - Result is a [lower, upper] band for a given filter kind

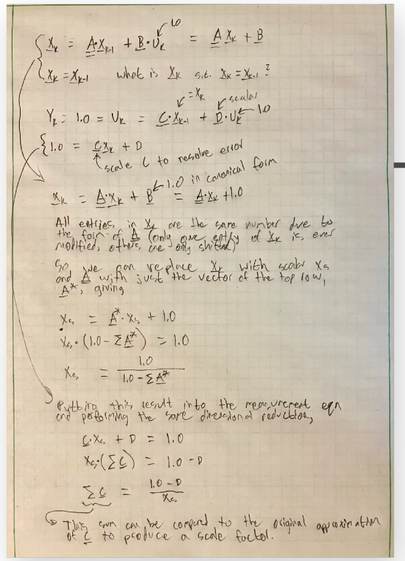
Try Pitch

Initialization & Error-Correction

- Interpolation introduces some error - we can correct the steady-state gain
- Summing dot products from **smallest to largest** reduces roundoff without fsum procedure
- Initializing filter internal state is possible without a matrix inverse due to particular structure of the system

Initialization

- **Find the steady internal state for a given input**
- **Set that as the current state**



M*TH

NOT EVEN ONCE

Measurement to initialize

$$x_s = \frac{u}{1.0 - \sum A}$$

Scalar steady-state value of all entries in `x` vector

```
pub fn initialize(&mut self, u: f32) {
    let asum: f32 = self.a.0.iter().sum();
    let xs: f32 = u / (1.0 - asum); // Scalar constant value for `X` entries
    self.x = Ring::new(xs);
}
```

Error-Correction

- **Find the steady-state gain**
- **Adjust coeffs to produce unity gain**

u = 1

$$x_s = \frac{1.0}{1.0 - \sum A}$$

D is a 1x1

$$\sum C = \frac{1.0 - d}{x_s}$$

Scale interpolated C to enforce this!

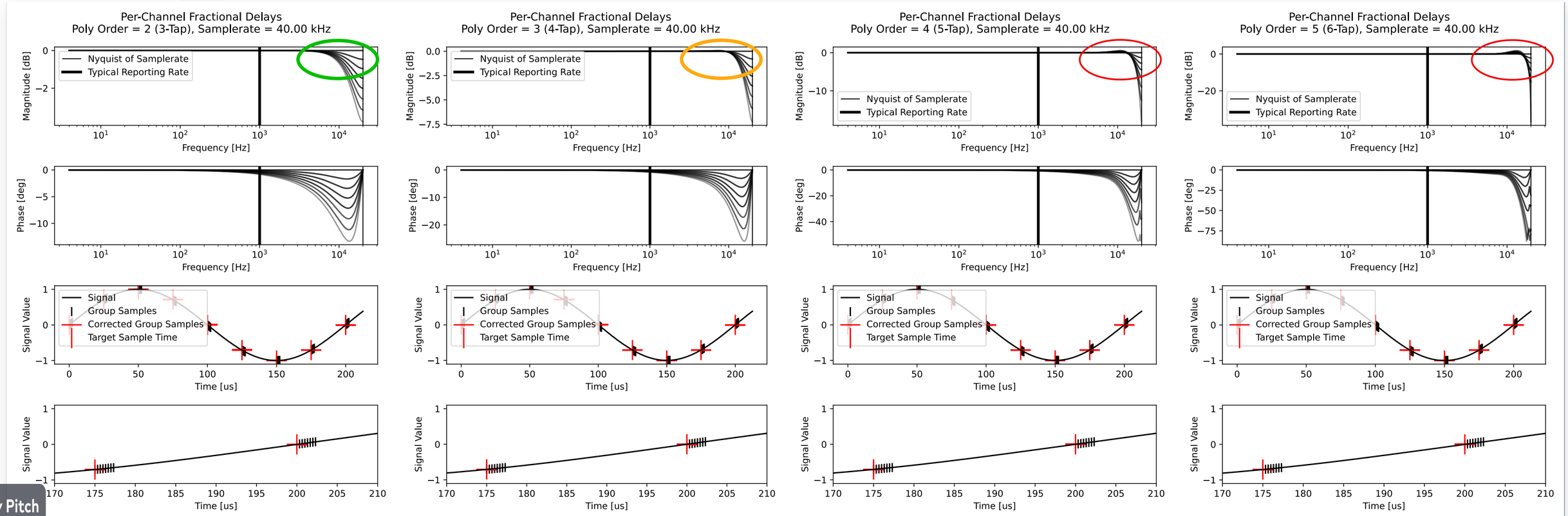
```
// Scale `C` to enforce unity gain at zero frequency, that is,
// that the step response should converge exactly.
//
// First, find scalar `xs` for every entry in filter state vector `x`
// such that x(k) == x(k-1).
let asum: f64 = a.iter().sum::<f32>() as f64;
let xs: f64 = 1.0 / (1.0 - asum);
// Then, find what the sum of `C` _should_ be to produce unity gain,
// and use that value to calculate a scale factor for the existing `C`.
let csum_desired: f64 = (1.0 - d as f64) / xs;
let csum: f64 = c.iter().sum::<f32>() as f64;
let scale_factor: f32 = (csum_desired / csum) as f32;
// Finally, scale `C` to, as closely as possible, produce unity gain.
c.iter_mut().for_each(|v: &mut f32| *v *= scale_factor);
```

Try Pitch

Fractional Delays: Polynomial Interpolation as an FIR Filter

- Minimum-order polynomial interpolation at a fixed location can be phrased as a dot product (Lagrange polys)
- An FIR filter is a dot product
- Tradeoff between phase linearity, overshoot, float precision, and compute cost with increasing order
- Constructed without allocation - general form involves matrix inverse, but special case of fixed samplerate is easy

✅ Higher-order not necessarily better

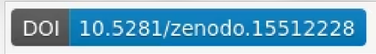


Try Pitch

Deimos
Controls
LLC

Python

Rust



InterpN

Fast, low-memory multi-dimensional interpolation (no-std+no-alloc compatible)

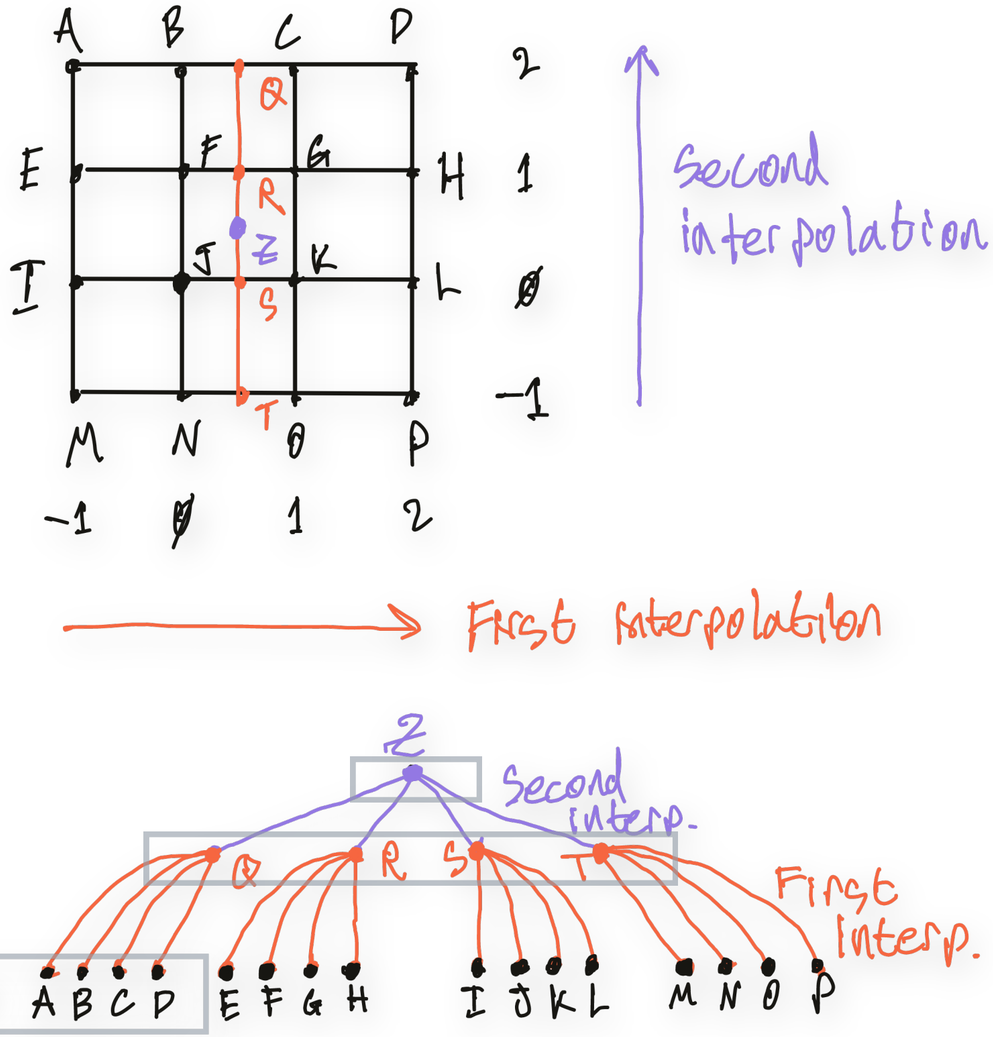
Try Pitch

Stack-Only

N-Dimensional Interpolation

Strategies

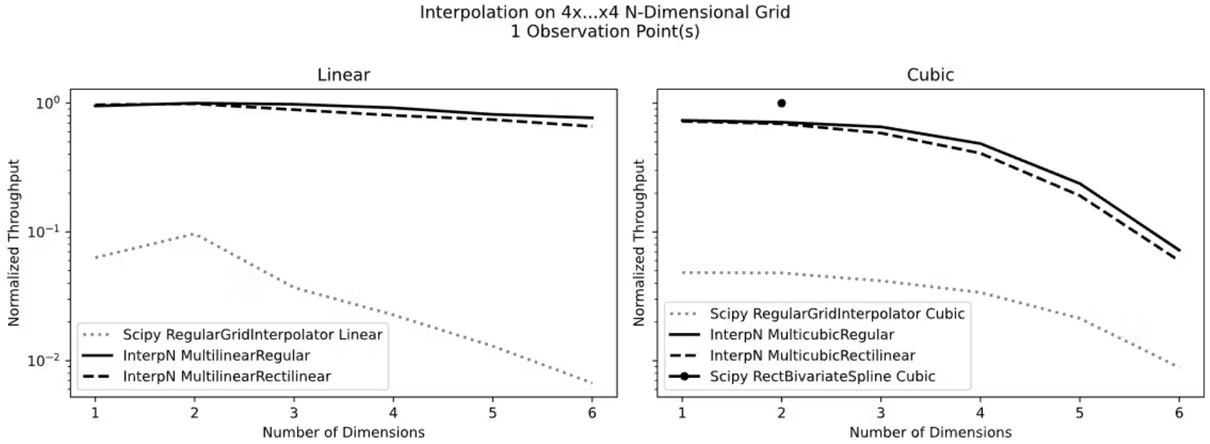
- ❌ Convolution
  - **Incorrect near the boundary** of the domain
- ❌ Hypercube
  - **Bad scaling** under extrapolation
- ✅ Recursive
  - Polynomial interpolation can be performed one axis at a time (de Casteljau algorithm, fitpack, etc)
  - Traditional methods require allocation during initialization and evaluation
  - We can do better
  - Complex, but it's easy to do complex things reliably in Rust
  - Would not attempt this in C

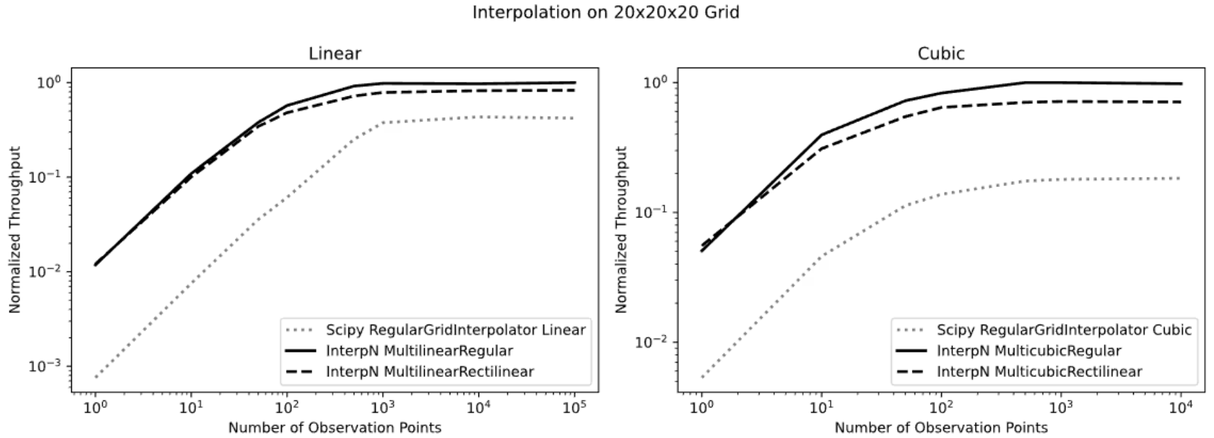


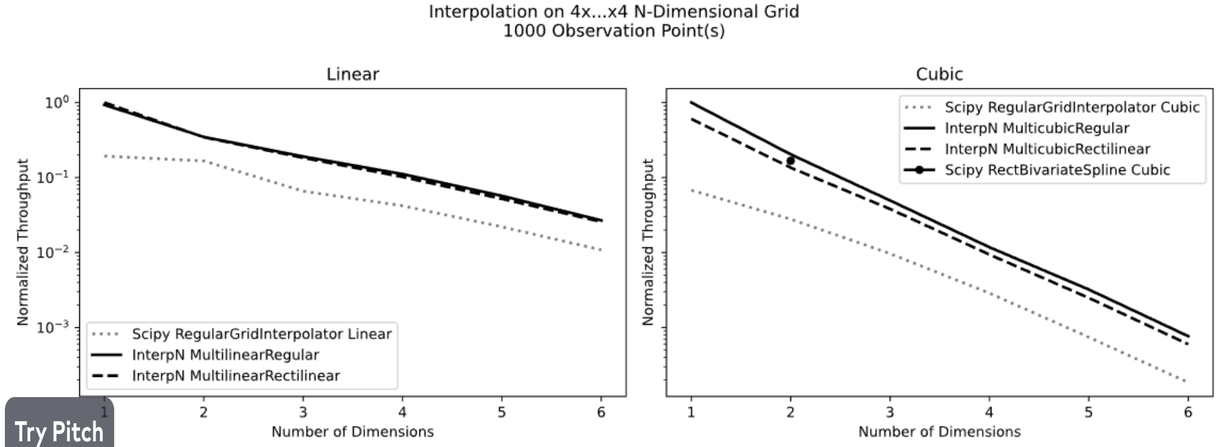
Try Pitch

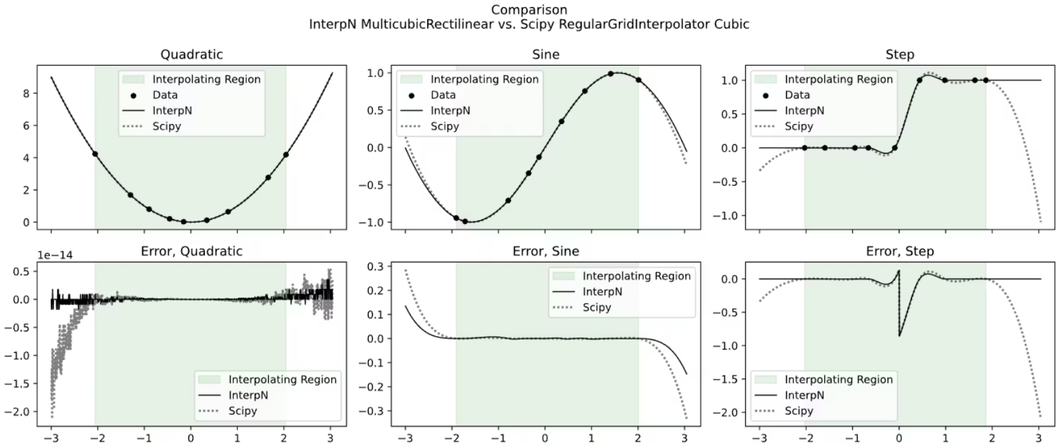
Interpolation Without Allocation - But at What Cost?

So there's a way - but what do we lose in performance when we give up the benefit of pre-computing part of the solution and allocating memory to store it? **Nothing!** In fact, with careful inlining & benchmarking, it's faster









Try Pitch

Deimos Controls LLC

Conclusion

Data collection is careful work, but not necessarily difficult or expensive.

- It is possible and not particularly hard to do analytically-defensible data collection
- This does not require or even benefit from an operating system (realtime or otherwise)

- It is possible and not particularly hard to do sub-microsecond time sync in software
- This does not require admin access, custom operating systems, or special clock hardware

- Rust makes it easy to do these things reliably
- Open-source is essential to reproducible science and engineering R&D (both hardware and software!)

James Logan
support@deimoscontrols.com

Pitch

Want to make a presentation like this one?

Start with a fully customizable template, create a beautiful deck in minutes, then easily share it with anyone.

Create a presentation (It's free)