SlowDash

計測制御とモニタのための Web-Python ツール

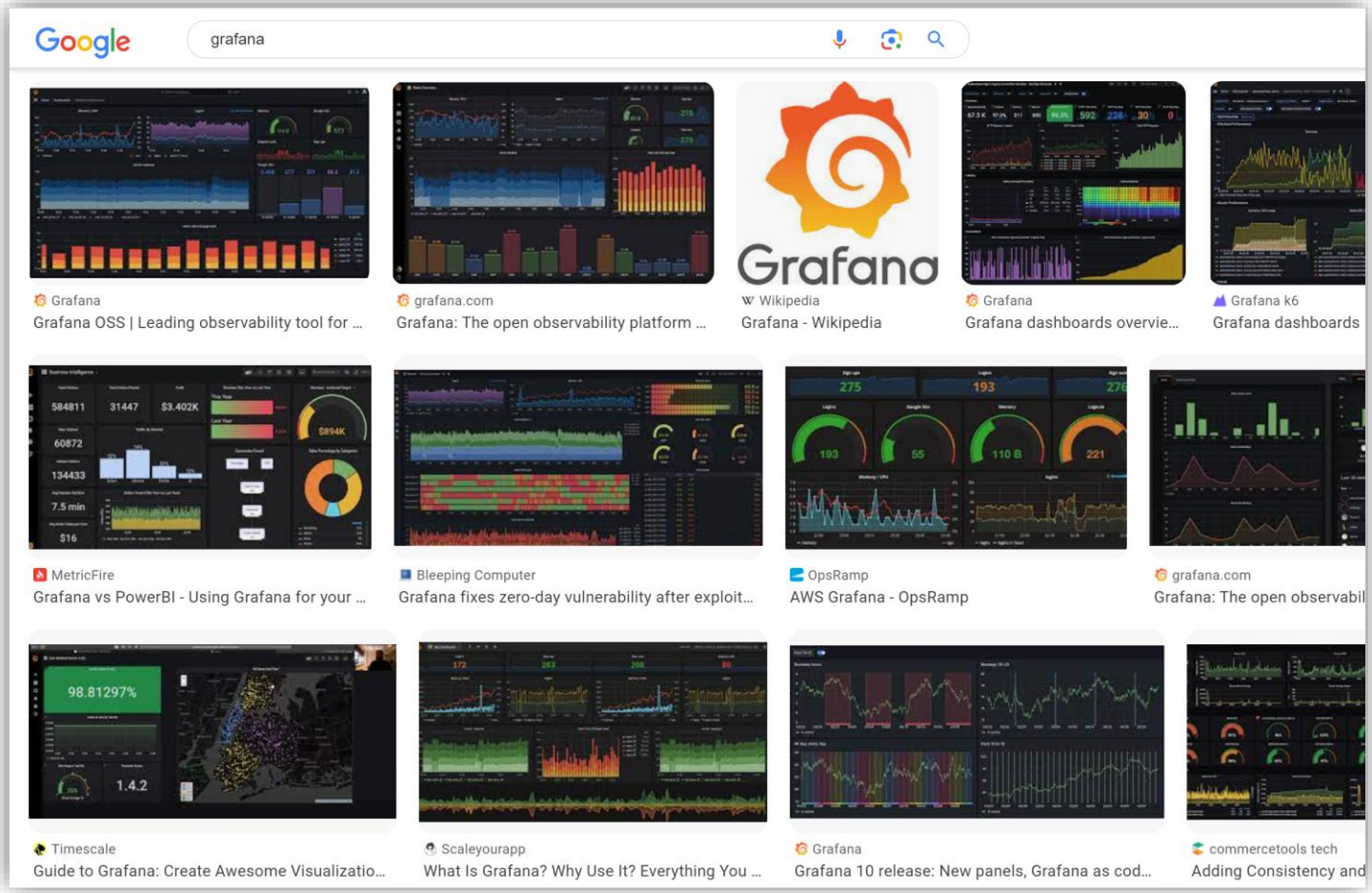
榎本 三四郎
ワシントン大学

計測システム研究会2025, J-PARC, 2025年11月18日

Grafana でかっこいいダッシュボードを作れ と言われた

Slow-Controls 用ダッシュボードを作る.
データを取る部分はすでに稼働していたので, UI だけあれば良かった

Google 検索による Grafana のイメージ

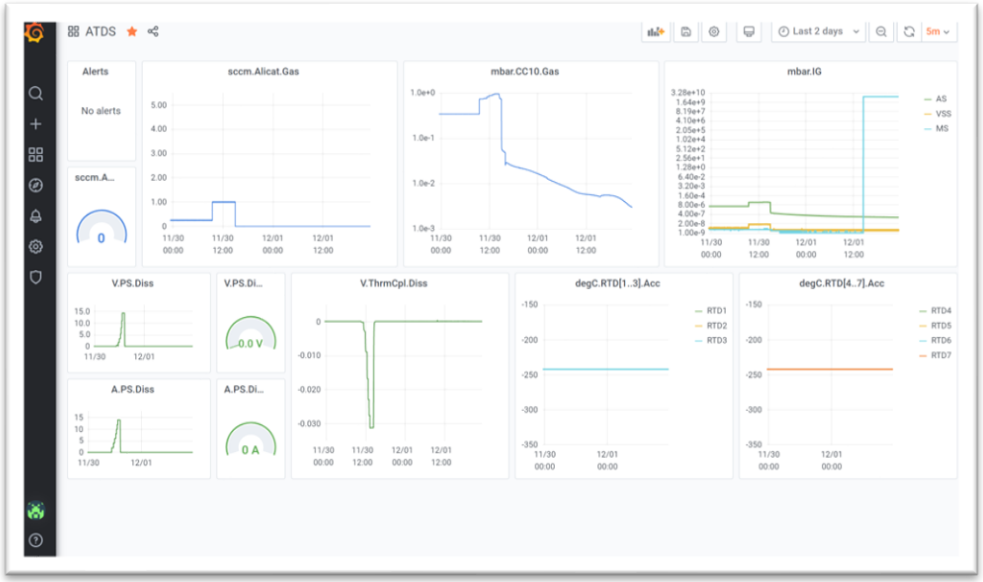


Grafana とは

- データーベース上にあるデータ(主に時系列)を見える化する Web ベースのソフトウェア
- マウスクリックでダッシュボードを作成して共有できる

3週間がんばってみた

- 構成の決まった表示には良い. 対話的なデータブラウジングはほぼ無理
- 科学データ解析用途には作られていないと思う (データサイエンス用途 ☺)



ダークモードはかっこいいけど
絵をスライドに貼れないから
ライトモードにしてみた
⇒ Grafana 台無し…

かっこいいプロットを眺めてるから先に進まない

- 表示を簡単に変えられない. (範囲, 対数軸, ...)
- 一時的プロットを作るのが大変.
- 基本的にデータ値にアクセスできない. ダウンロードもできない.

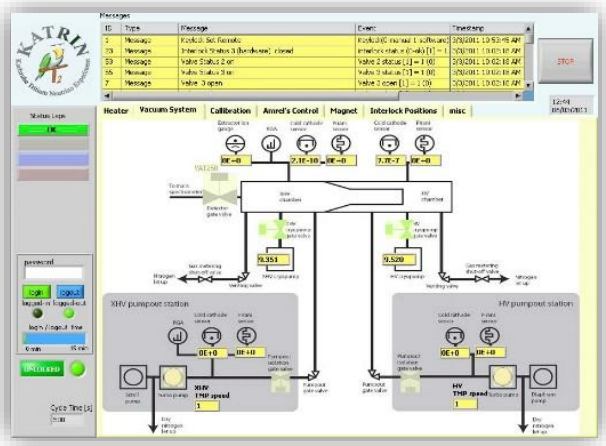
目的に合っていない気がする

- ヒストグラム, 誤差バー付きグラフ, 散布図, etc が素直でない. (地図塗りは簡単)
- 機器制御ができない. コマンドも送れない.

欲しいものリスト

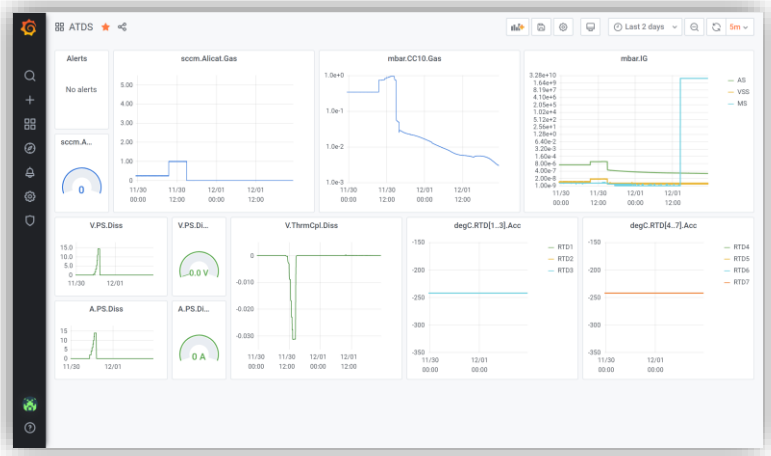
LabVIEW 風のコントロールパネル

（でもグラフィカルプログラミングは嫌）



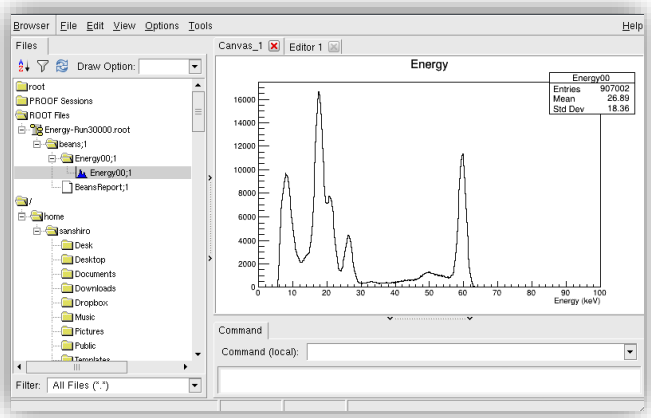
Grafana 風のデータブラウザ

（でも見るだけでは不足）



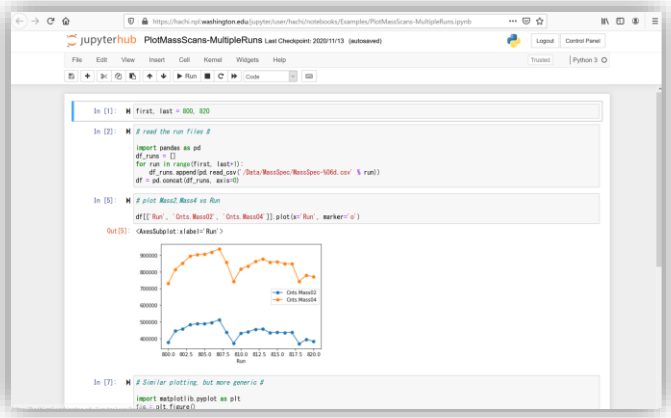
ROOT 風のデータ型

（でも ROOT は嫌い）



Jupyter 風のその場思い付き解析

（でも実時間処理には向かない）



開発経緯・状況

22 年11月 感謝祭の連休中に Grafana の置き換えを模索

23 年5月 SPADI-A WG3 に居候開始

24 年1月 GitHub 公開: Version “Nisqually”

- データベース中のデータの Grafana 風表示
- LabVIEW 風のデータ表示（表示だけ）

データ可視化

24年7月 “Snoqualmie”

- 計測制御・読み出しの Python スクリプティング

機器制御

24年11月 “Skykomish”

- 共通コントロール部品のライブラリ
- 解析スクリプトの Jupyter 接続

25年7月 “Nooksack”

- 非同期 Web エンジン, WebSocket ストリーミング

システム化

現在と今後

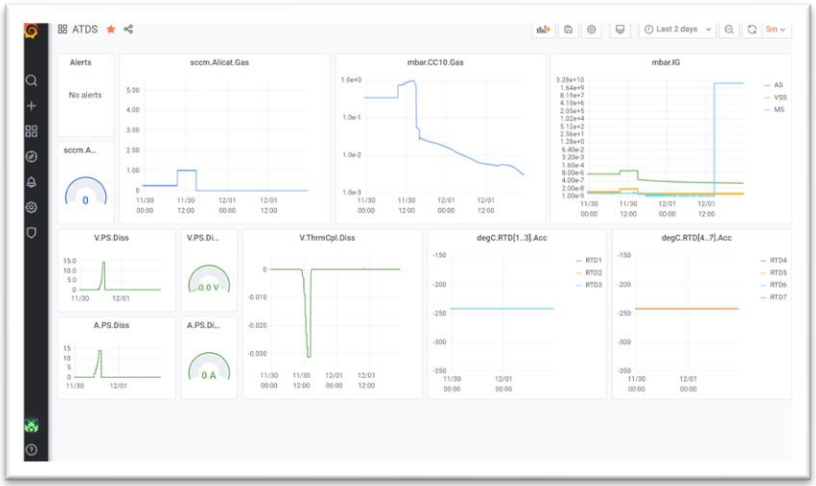
- スクリプト間通信, 分散並列化

今回の新しい部分

- 埋め込みデータ加工・データ変換（テーブルからグラフやヒストグラム作成とか）
- かっこいいレイアウトとビジュアル, 数千チャンネルの表示とか
- アラーム検出・通知・認知・マスク・リマインド・履歴管理, 障害予測(?)

Grafana のいいところ

良いと思ったところ



- ダッシュボードをマウス操作だけで作成できる（サーバーにログインしなくて良い）
- 作ったダッシュボードを保存して共有できる
- ダークモードにするとかっこいい

- 基本的に全てプラグインで，それを画面上で組み合わせる感じ
- 表示時間範囲の操作感もいい感じ

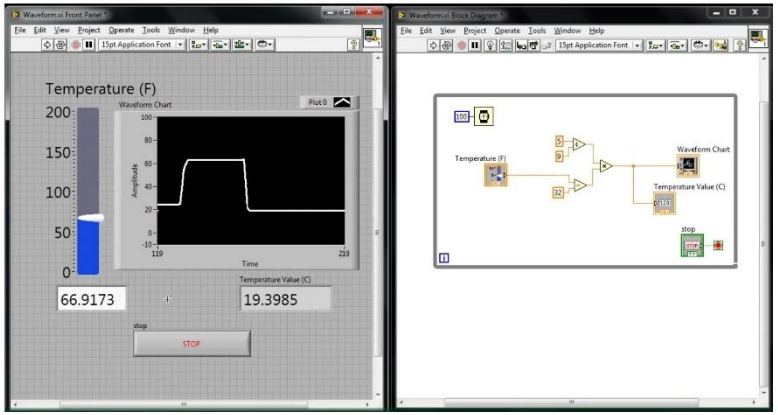
足りないと思ったところ

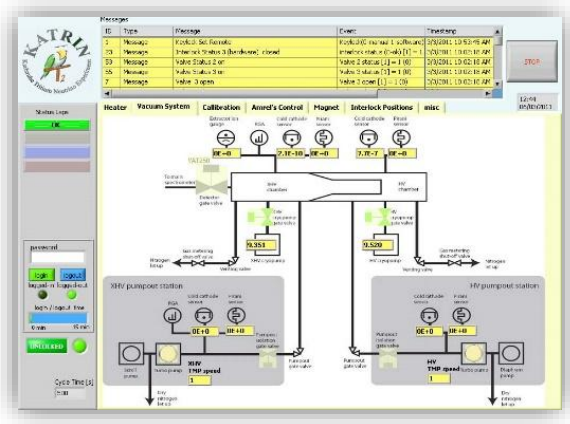
- ダッシュボードなら装置全体の状態を視覚化したい
- 表示した装置は操作したい
- 表示したデータは解析したい

LabVIEW のいいところ

良いと思ったところ

- 表と裏がある
- 裏のロジックもユーザがその場で作成・編集ができる





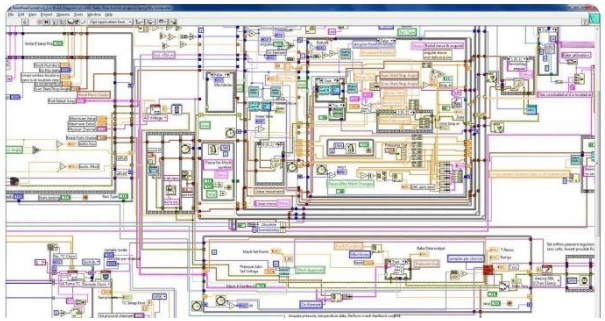
- データ要素の「空間断面」(現在値の平面配置)的な表示
- コントロール部品にロジックが張り付いている感じ

足りないと思ったところ

- ロジックは字で書きたい．図はいずれこうなる ⇒
  (回路を字で書きたいから HDL が普及したんだと思う)

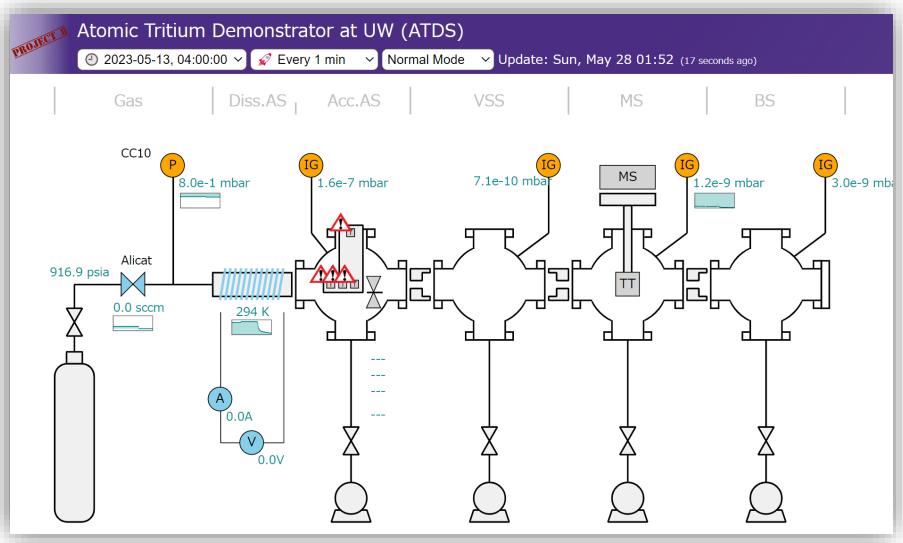
- 発見的データ解析にも向いていない

My worst nightmare

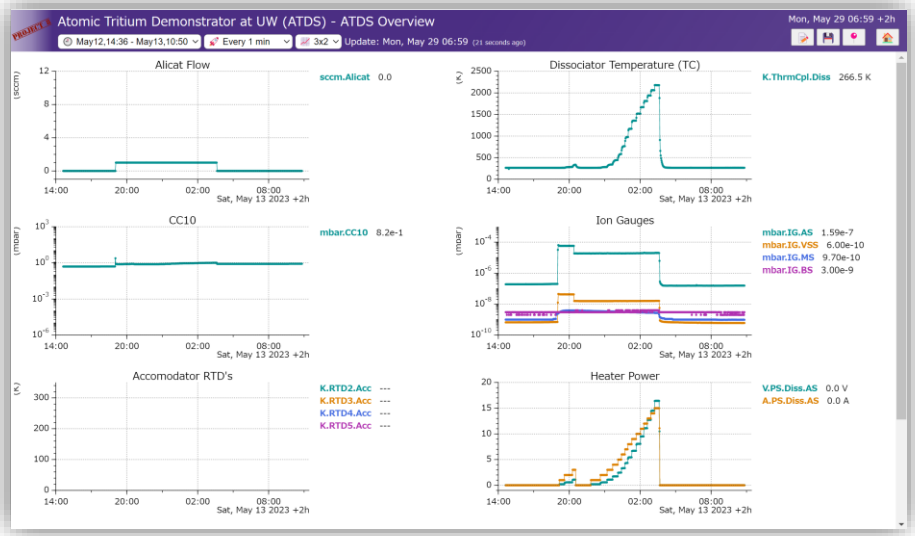


SlowDash：時間断面と空間断面のデュアルビュー

LabVIEW 風のダッシュボード（固定時間）



Grafana 風のプロット（時系列）



- 色で運転状態表示，アイコンで異常状態表示
- クリックすると時系列データを表示（右画面）

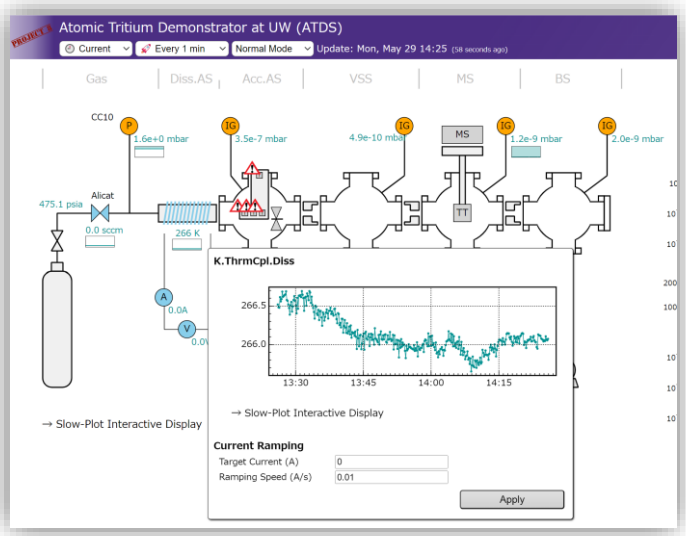
- インタラクティブプロット（ズーム，対数軸，…）
- レイアウトをマウスで作成
- 作成したページを保存，共有可
- データダウンロード

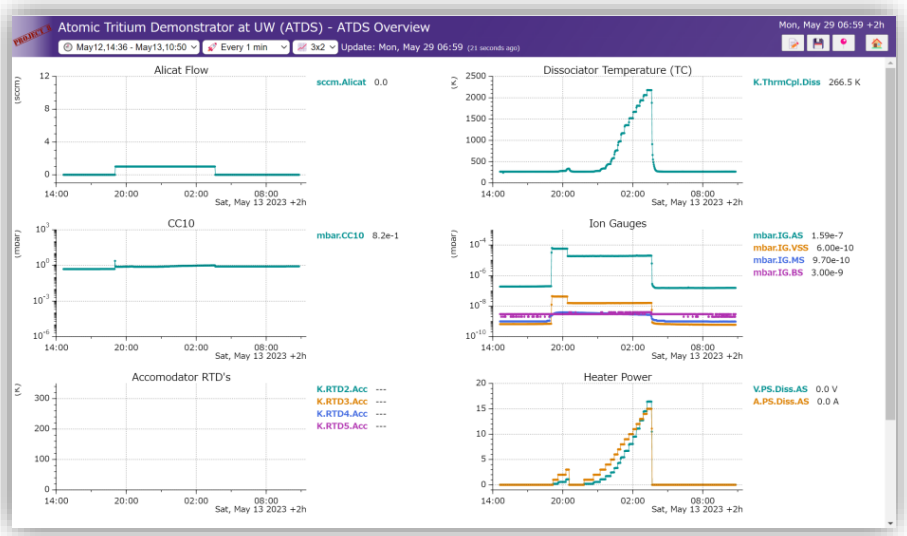
プロットの構成は URL からでもできる．作成した複雑なページを URL にエンコードすることもできる．

hachi.npl.washington.edu/~hachi/SlowDash/ATDS/slowplot.html?channel=V.ThrmCpl&time=2023-05-13,13:00

- いろいろなプロットへのリンクをあちこちに作れる
- Slack や ELOG に投稿できる

SlowDash: GUI とスクリプトのハイブリッド





変数や関数のバインド

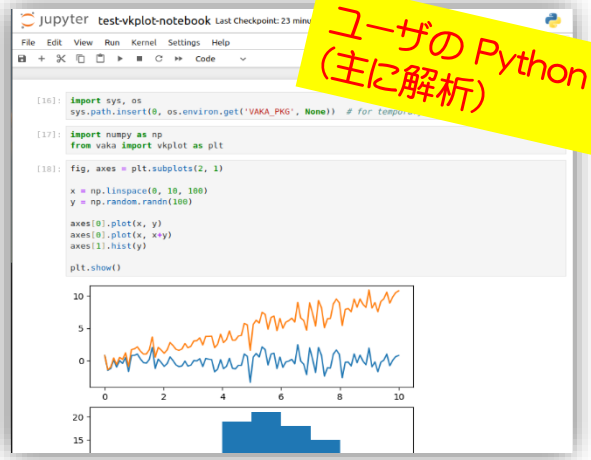
コード生成と動的ロード

ユーザの Python
（主に制御）

```
def set_V0(V0, **kwargs):
    ramping = kwargs.get('ramp
    device.ch(0).ramp(ramping).set(V0)

def set_V1(V1, **kwargs):
    ramping = kwargs.get('ramping', 10)
    device.ch(1).ramp(ramping).set(V1)

def set_V2(V2, **kwargs):
    ramping = kwargs.get('ramping', 10)
```

データベース

ユーザの Python
（主に解析）



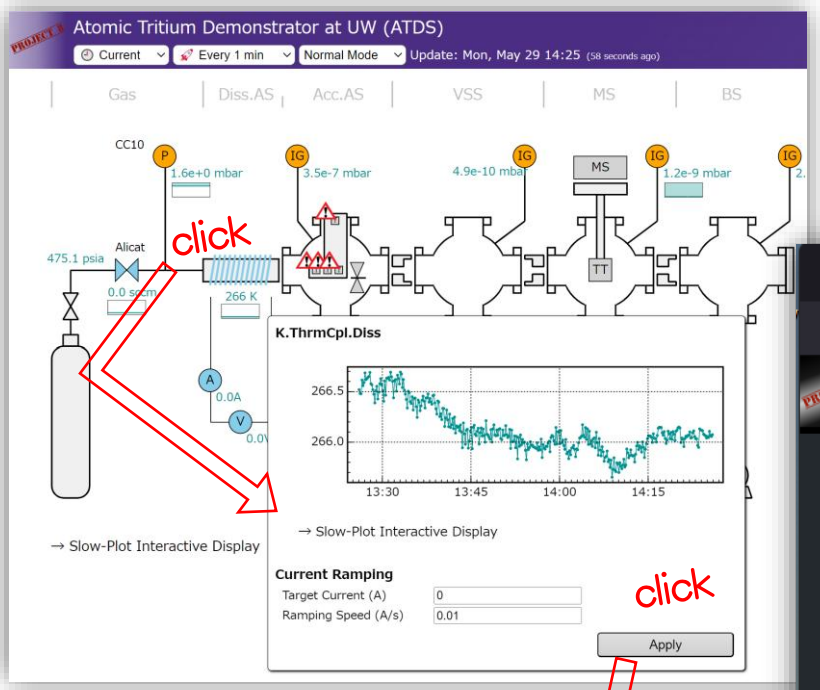
制御対象

Slow Devices

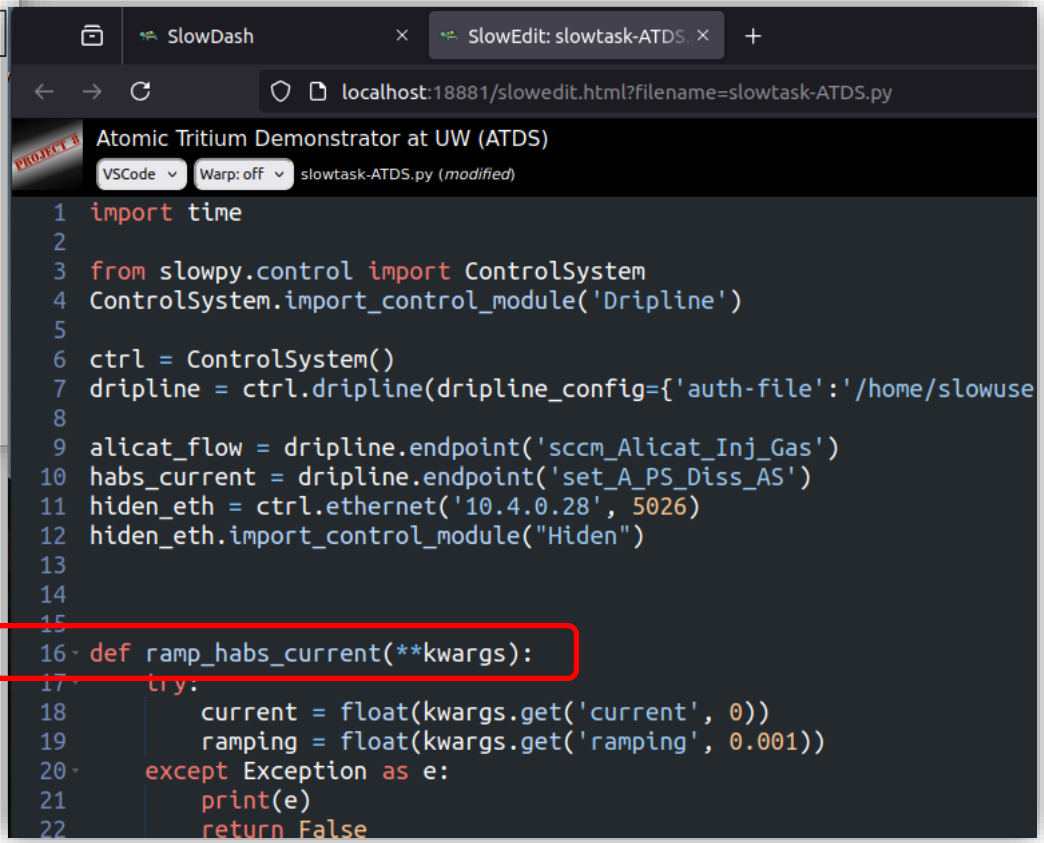
DAQ Systems

ユーザのシステム

コントロールスクリプトのバインド



ユーザが書いた Python スクリプト
(SlowDash ブラウザ上で編集できる)



ボタンをクリックすると
ボタンの名前の関数が呼ばれる

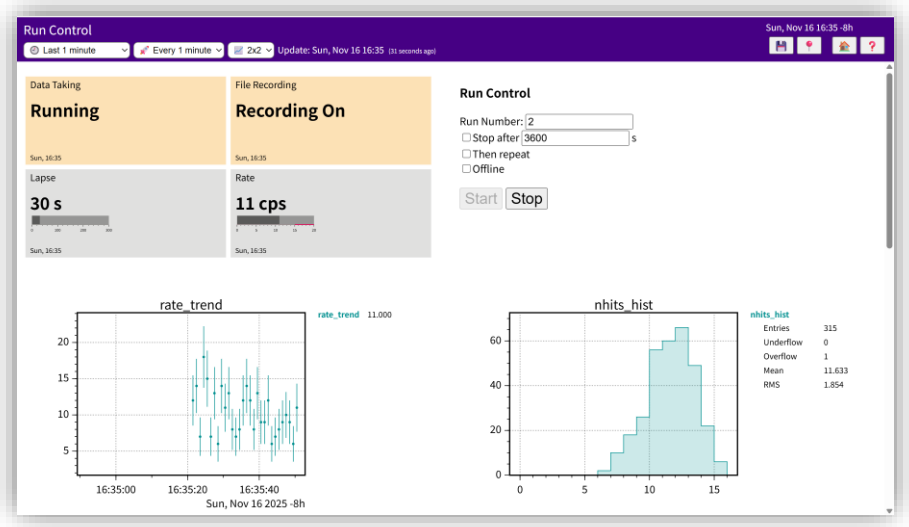
入力要素の値が同名の引数になる

スクリプト中の変数を表示要素へ
直接バインドすることもできる.

ただの Python なので Python でできることは全てできる.
SlowDash なしでスクリプトの単独実行もできる.

コントロールスクリプトのバインド 2

HTML フォームをユーザ Python スクリプトにバインド



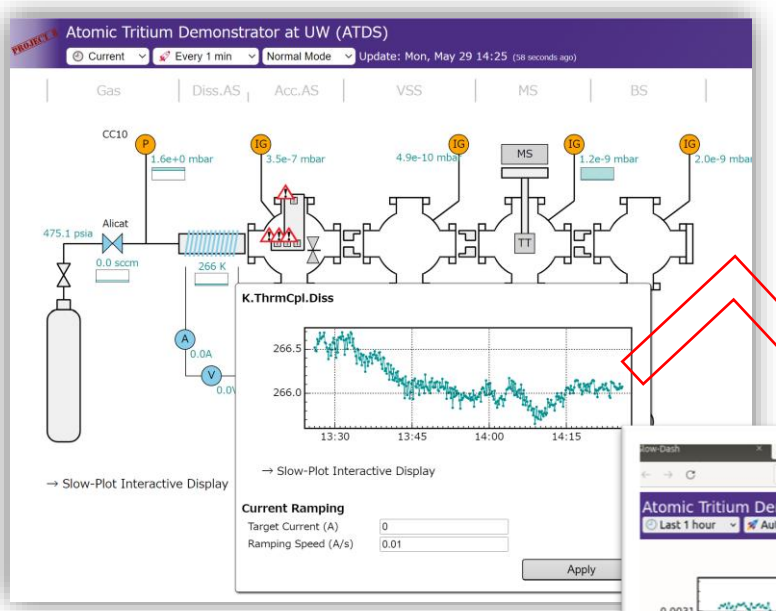

```
<h3>Run Control</h3>
<form>
  Run Number: <input name="run_number" type="number" step
  <input type="checkbox" name="stop_after" sd-value="run_
  <input type="number" name="run_length" style="width:10e
  <input type="checkbox" name="repeat" sd-value="run_sett
  <input type="checkbox" name="offline" sd-value="run_set
  <p>
  <div style="font-size:150%">
  <input type="submit" name="run_control.start()" value="
  <input type="submit" name="run_control.stop()" value="S
  </div>
</form>
```

入力フィールドがパラメータ
ボタンが関数呼び出し

(Streamlit 風バインディングも検討中. 需要があれば...)

```python
async def start(run_number:int=None, stop_after:bool=None,
    run_length:float=None, repeat: bool=None, offline:bool=None):

    if run_number is not None:
        run_setting.run_number = run_number
    if stop_after is not None:
        run_setting.stop_after = stop_after
    if run_length is not None:
        run_setting.run_length = run_length
    if repeat is not None:
        run_setting.repeat = repeat
    if offline is not None:
        run_setting.offline = offline

    save_run_setting()

    run_status.start_time = round(time.time(),3)
    run_status.running = True
    await ctrl.aio_publish(run_setting)
    await ctrl.aio_publish(run_status)

    await do_run_start()

    return True


async def stop():
    run_status.running = False
    await do_run_stop()

    if not run_setting.offline:
        run_setting.run_number += 1

    save_run_setting()
    await ctrl.aio_publish(run_setting)
    await ctrl.aio_publish(run_status)

    return True
```

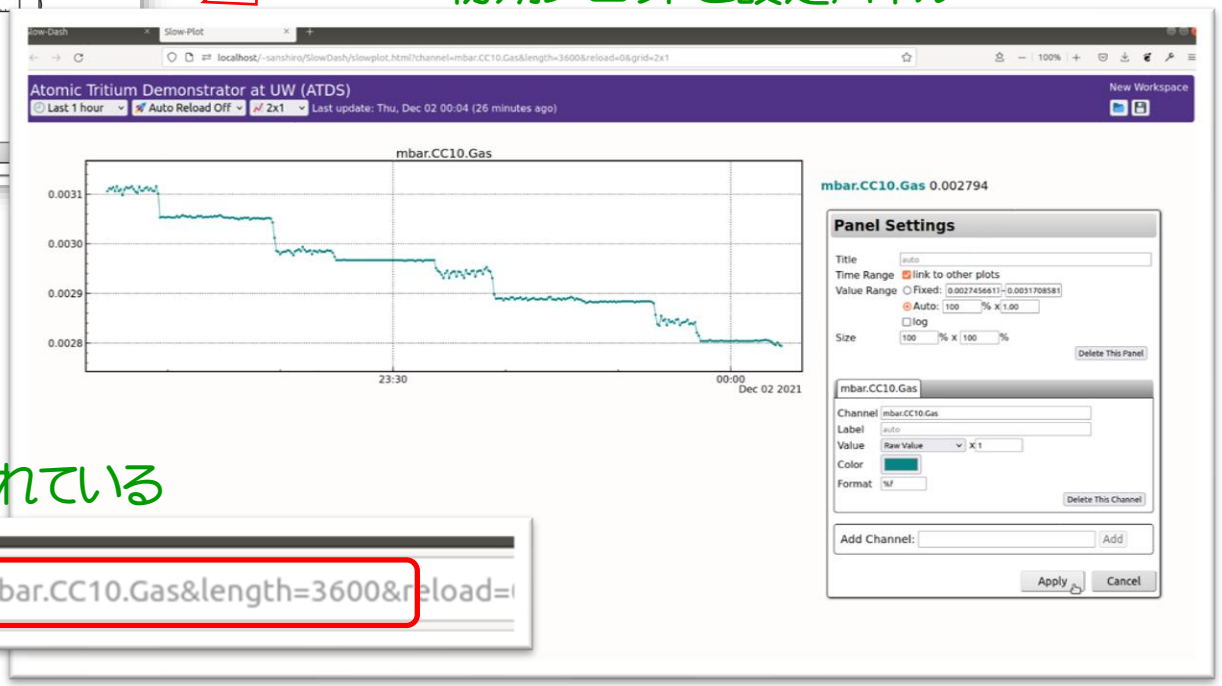
データブラウザへのリンク



click

データ要素をクリックすると時系列プロットを表示

初期プロットと設定パネル

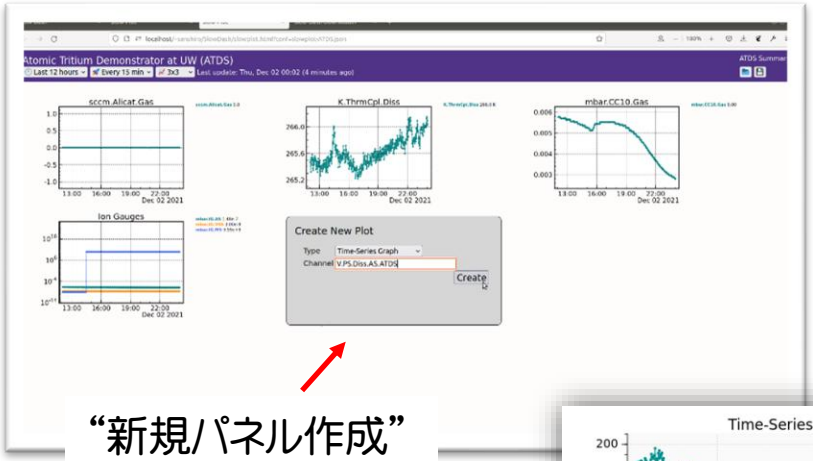


ページは URL から動的に作られている

- いちいち「新規作成」→ポチポチ としなくていい
- リンクなので E-log とか Slack とかに貼れる

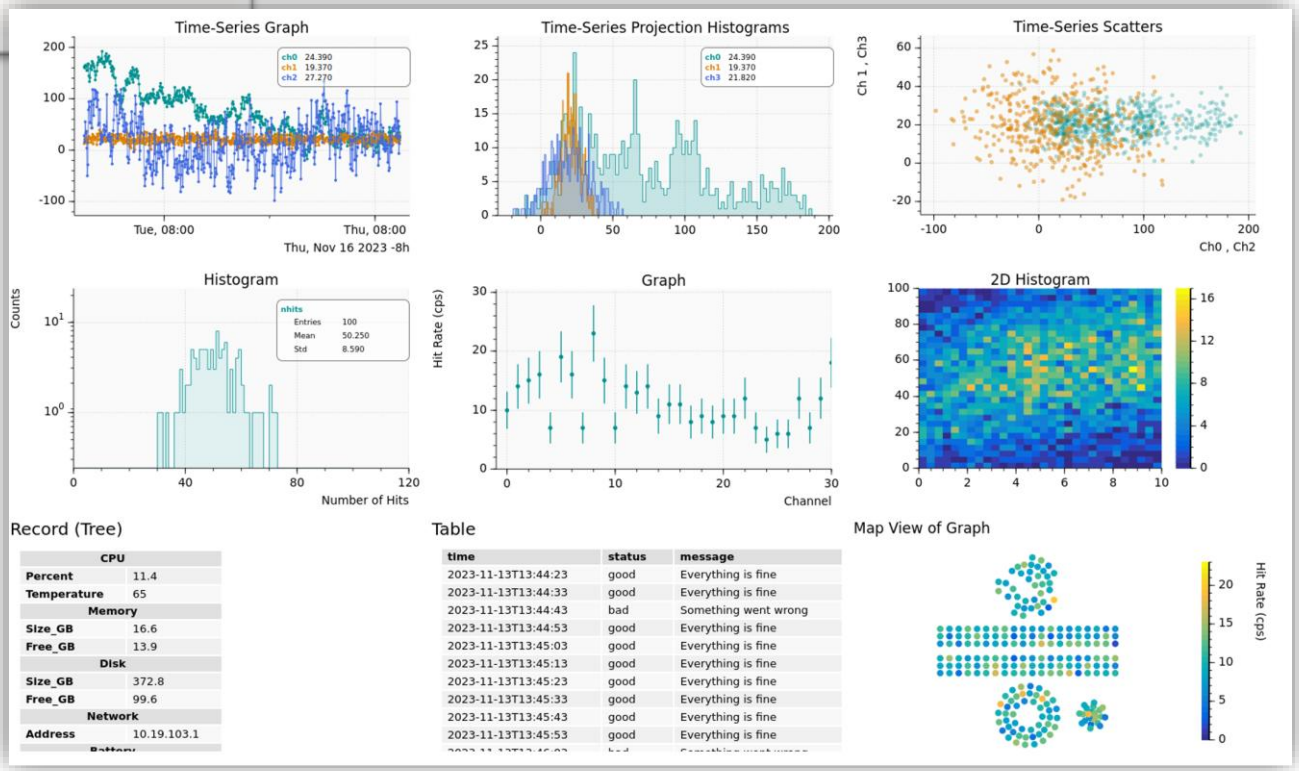
データブラウザ

ブラウザの中身は動的に構築できる



“新規パネル作成”

ポチポチする

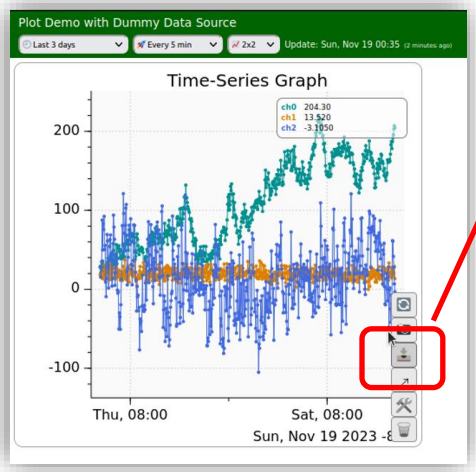


利用可能な表示形式

- 時系列プロット
- ヒストグラム
- エラー付きグラフ
- 2次元ヒストグラム
- テーブル
- ツリー
- マップビュー
- …

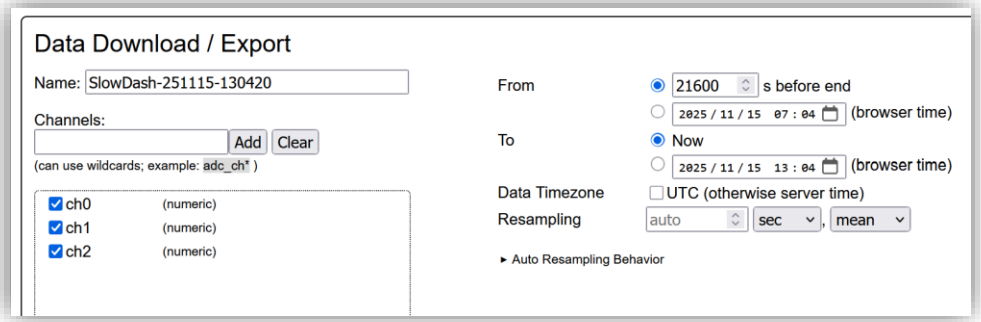
全てプラグイン実装

データエクスポートと Jupyter 接続



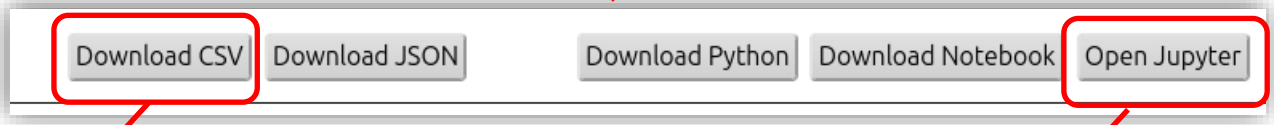
表示データのエクスポート

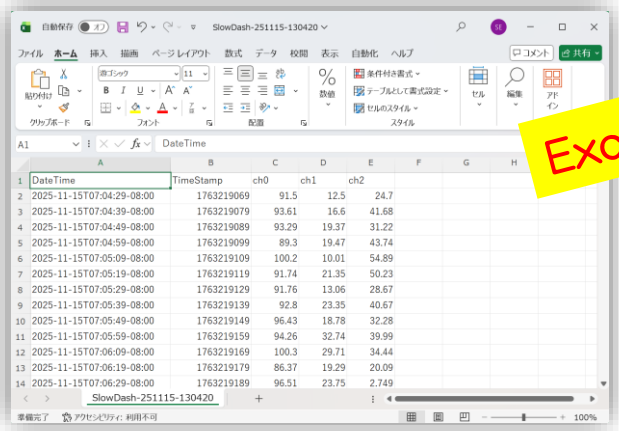
チャンネルや時間範囲はプロットから初期設定

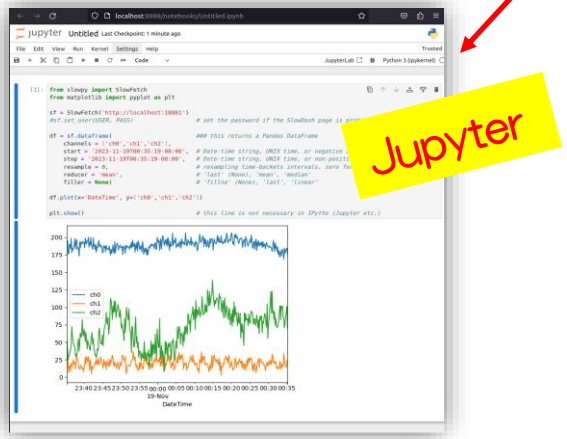


データファイル

Python スクリプト自動生成







2～3クリックで表示データをエクスポートしてオフライン解析を継続できる

新：非同期ウェブサーバ（Slowlette）

非同期 I/O (async) による効率的な平行処理, マルチスレッドの回避, 動的ルーティング

FastAPI 風の URL バインディング（真似）

Starlette 風の非同期軽量処理（真似）

```python
from slowapi import ~~SlowAPI~~ Slowlette

app = SlowAPI()

@app.get("/hello/{name}")
async def hello(name:str, repeat:int=1):
    return "Hello, world." * repeat

app.run()
```

非同期処理：
- I/O 待ちのときに別のことをする
- スレッドよりはるかに軽量
  → 数万の非同期タスクも問題なし

- レスポンス速度の大幅な改善
- HTTPS 接続, 基本認証, HTTP/2, …
- ストリーミングも実装 → 次ページ

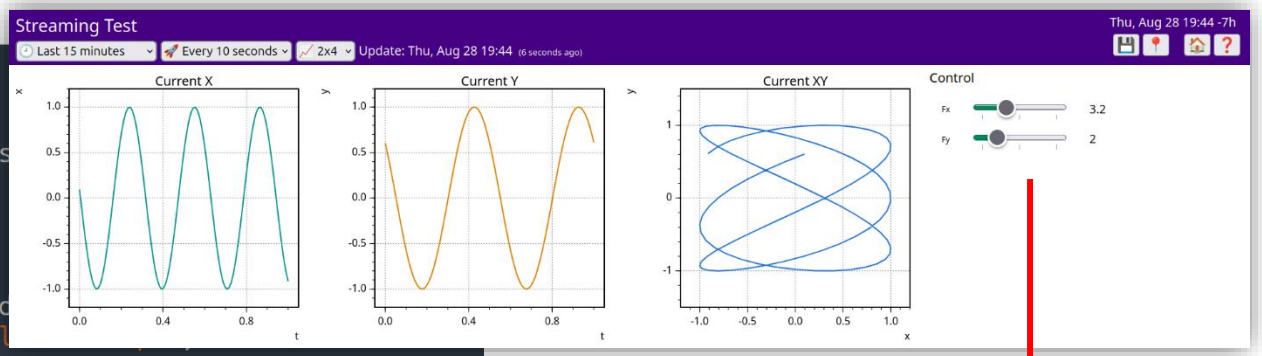
多重ハンドラと戻り値集約（独自）

```python
class Fruit:
    def __init__(self, name:str):
        self.name = name

    @slowapi.get('/hello')
    def hello(self):
        return [f'I am a {self.name}']


class MyApp(slowapi.App):
    def __init__(self):
        super().__init__()
        self.slowapi.include(Fruit('peach'))
        self.slowapi.include(Fruit('melon'))

    @slowapi.get('/hello')
    def hello(self):
        return ['Hello.']


app = MyApp()
```

- Web API の動的拡張
  → 構成要素のプラグイン化

新機能：WebSockets によるデータストリーミング

```python
import time
import numpy as np

from slowpy.control import control_sys
fx = ctrl.value(3.2)
fy = ctrl.value(2.0)

from slowpy import Graph
from slowpy.store import DataStore_Red
datastore = DataStore_Redis('redis://l
next_store_time = 0


async def _initialize():
    ctrl.export(fx, name='fx.current')
    ctrl.export(fy, name='fy.current')


t0 = 0
async def _loop():
    global t0

    t0 += 0.05
    t = np.linspace(0, 1, 100)
    x1 = np.random.normal(np.cos((t+t0)*float(fx)*6.28), 0.0003)
    x2 = np.random.normal(np.sin((t+t0)*float(fy)*6.28), 0.0003)

    g_x, g_y, g_xy = Graph(), Graph(), Graph()
    g_x.add_point(t, x1)
    g_y.add_point(t, x2)
    g_xy.add_point(x1, x2)

    await ctrl.aio_publish(g_x, 'x.current')
    await ctrl.aio_publish(g_y, 'y.current')
    await ctrl.aio_publish(g_xy, 'xy.current')

    ctrl.sleep(0.5)
```



変数バインディング

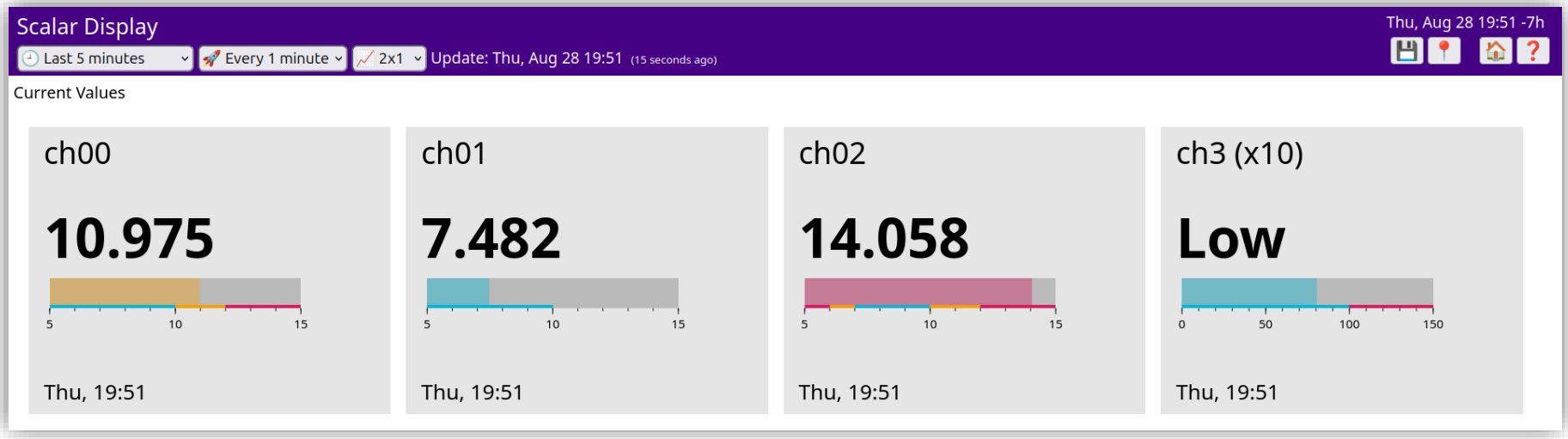
ブラウザで動かすと
スクリプト中の変数の
値がその場で変わる

WebSockets による
ストリーミング

ブラウザの表示が
その場で変わる

新機能：スカラー値表示パネル

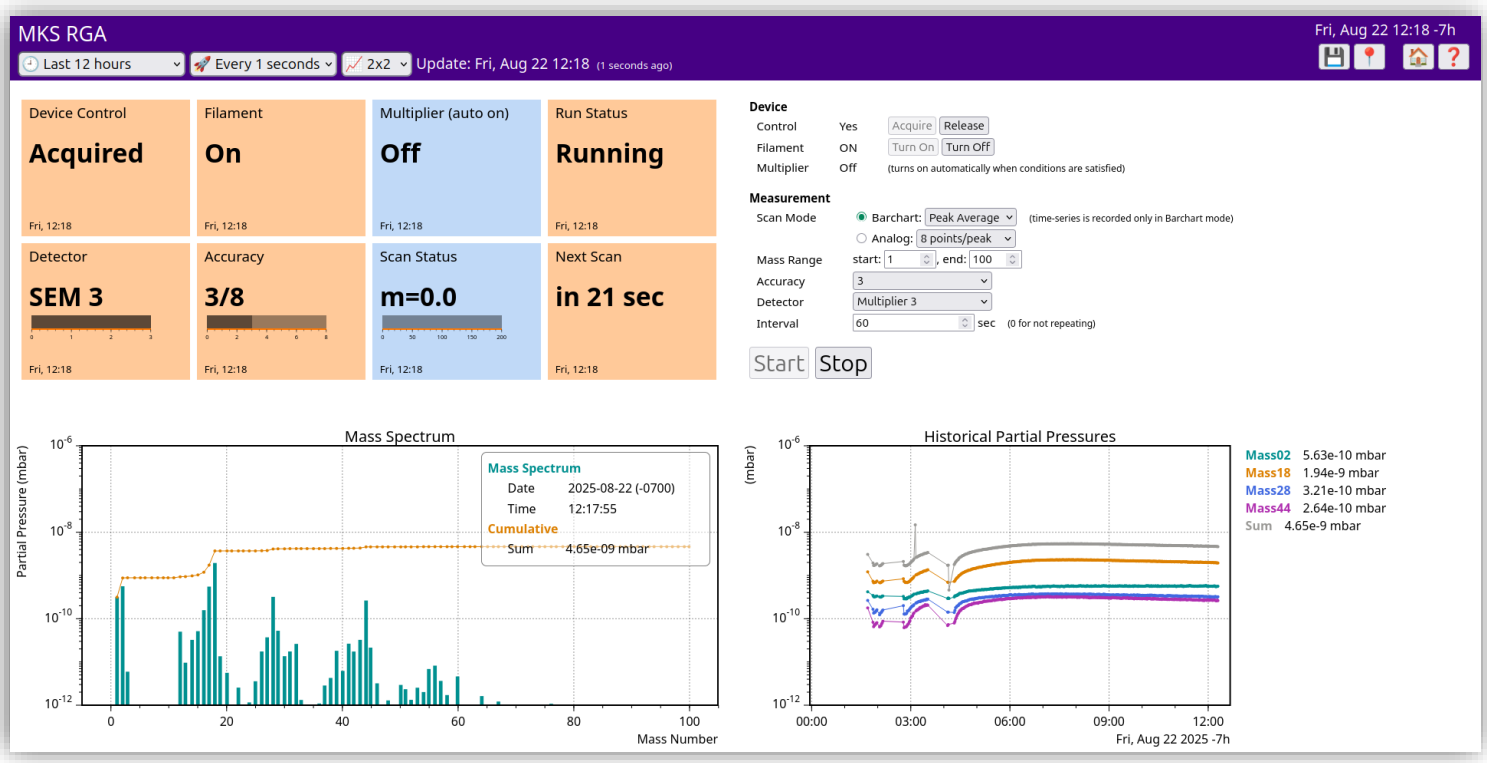
ストリーミングが送り出す「即時値」の表示



Grafana の
これの置き換え

新機能：スカラー値表示パネル（状態表示）

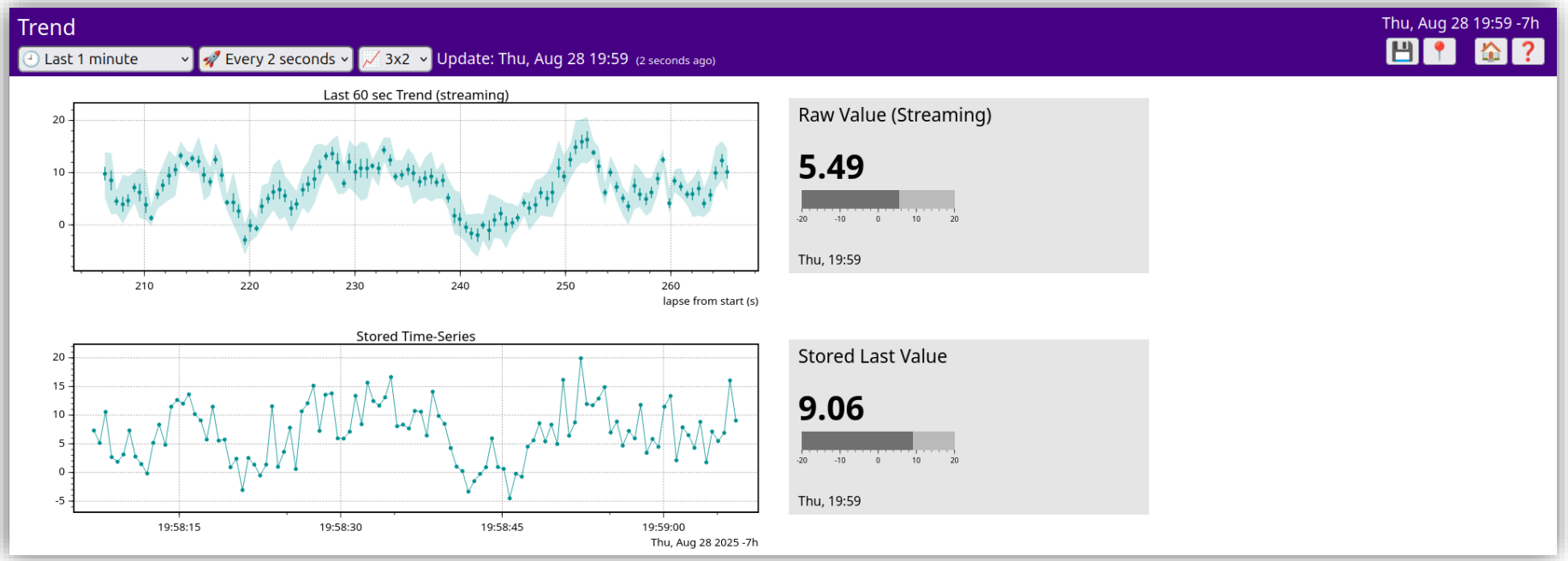
パネル自体の色を変えて状態表示版にする



Grafana の
これの置き換え

新機能：トレンドグラフ

ストリーミング用時系列データのグラフ



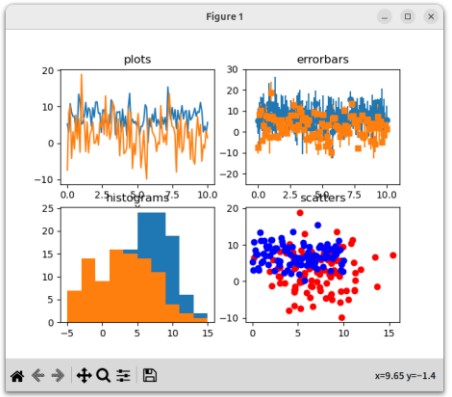
トレンド（上）：　グラフオブジェクト（ユーザ作成）
時系列（下）：　リサンプリングされた生の値（描画時作成）

新機能：Matplotlib の図を直接ストリーミング

みんな大好きなごく普通の Python/Matplotlib

```python
import numpy as np
import matplotlib.pyplot as plt
from slowpy.control import control_system as ctrl


async def _loop():
    x = np.linspace(0, 10, 100)
    y1 = np.random.normal(7, 3, len(x))
    y2 = np.random.normal(3, 5, len(x))
    ey1 = np.random.poisson(7, len(x))
    ey2 = np.random.poisson(3, len(x))

    fig, axes = plt.subplots(2, 2)

    axes[0,0].plot(x, y1, label='plot A')
    axes[0,0].plot(x, y2)
    axes[0,1].errorbar(x, y1, yerr=ey1, fmt='o', label='errorbar A')
    axes[0,1].errorbar(x, y2, yerr=ey2, fmt='s')
    axes[1,0].hist(y1, bins=np.linspace(-5, 15, 11), label
    axes[1,0].hist(y2, bins=np.linspace(-5, 15, 11))
    axes[1,1].scatter(y1, y2, c='red', label='scatter A')
    axes[1,1].scatter(x, y1, c='blue')

    axes[0,0].set_title("plots")
    axes[0,1].set_title("errorbars")
    axes[1,0].set_title("histograms")
    axes[1,1].set_title("scatters")
    axes[0,1].set_xlim(-1, 11)
    axes[0,1].set_ylim(-25, 30)
    axes[0,0].set_xlabel("X")

    await ctrl.aio_publish(fig, name='mpl')
    plt.close()

    await ctrl.aio_sleep(0.5)
```

普通に直接実行



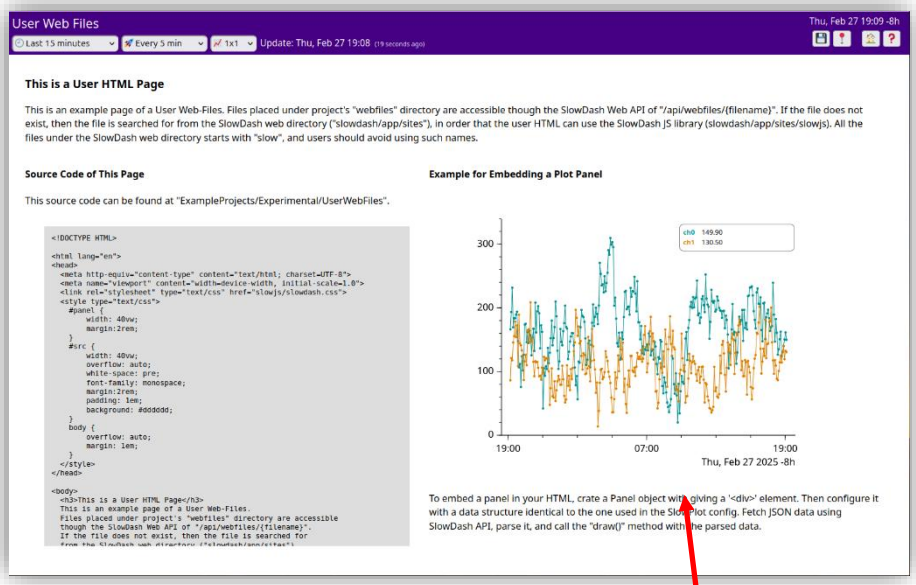
SlowDash のユーザータスク

内部実装：描画オブジェクトからデータとレイアウトを抽出再構成

新機能：ユーザ HTML ページ

自分で HTML/JavaScript を書きたい人なら

- ユーザが作成した普通の HTML ファイルを統合
- その中で SlowDash プロットを作成する JavaScript ライブラリを使える



自由なレイアウトで作った例

Slowdash の図を挿入

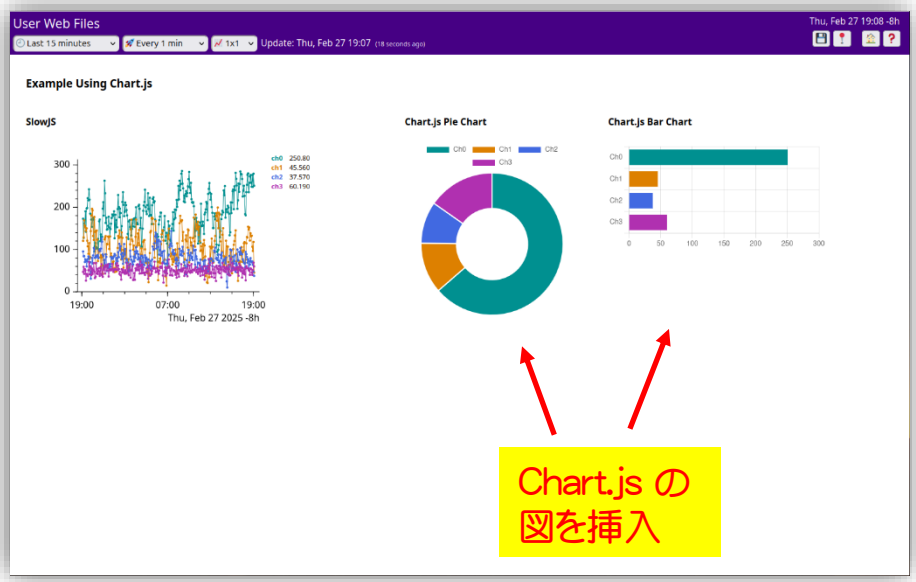
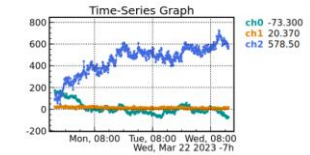


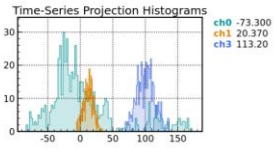
Chart.js の図を挿入

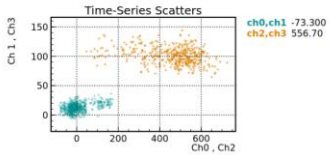
外部ライブラリ（Chart.js）を使って
SlowDash が苦手なプロットを作った例
（SlowDash のデータアクセスだけ利用）

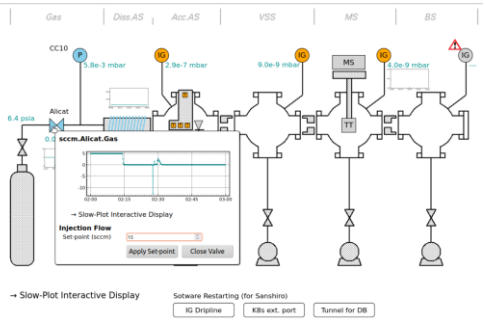
SlowDash: ほぼ全てがプラグイン

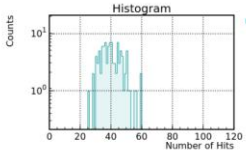
UI 部品 (JavaScript)

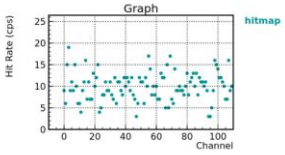


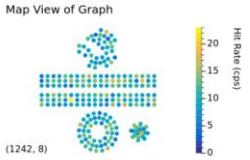


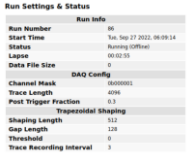












Core Logic

ユーザモジュール (Python)

- デバイス コントロール
- 外部システム 連携
- 実時間 データ解析
- 異常検知 と対処
- Mini-DAQ

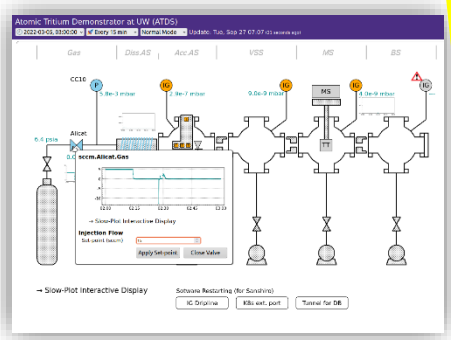
データベースインターフェース

PostgreSQL, MySQL, SQLite, influxdb, redis, mongoDB, CouchDB relax, ROOT Data Analysis Framework, CSV, YAML, HDF5

Core Logic 機能も実は全て独立プラグイン: データベース, ユーザモジュール, 設定ファイル, コンソール, Jupyter, …

SlowDash = App + Lib + Scripts

SlowPy: ユーザスクリプトで便利に使えるライブラリ（制御し，計測して，DBに記録）



ユーザー
スクリプト

```
trol import ControlSystem
stem()
.store import DataStore_SQLite
datastore = DataStore_SQLite('sqlite:///SlowTe

voltmeter = ctrl.ethernet("10.0.0.54", port=12

print(voltmeter.command('*idn?'))


for i in range(3):
    V = voltmeter.command(':read?')
    datastore.append(V, tag='Voltage')
```

SlowDash
Web App

SlowDash
Library
(SlowPy)

コントロール &
メッセージ交換

VISA/Socket
& SCPI

Modbus

MQTT

Rabbit
MQ

Redis

HTTP

BASH
pipe

ユーザー
計測器 & システム

DL

PostgreSQL MySQL SQLite influxdb

redis mongoDB CouchDB relax

ROOT Data Analysis Framework CSV YAML HDF5

データストレージ

ワークフロー例：電圧計読み出し

BK PRECISION

Model: 5492B, 5492BGPIB

5 ½ Bench Digital Multimeter

USER MANUAL

Chapter 6 **SCPI Command Reference**

This chapter is outlined as follows:

6.1 Command Structure
6.2 Command Syntax
6.3 Command Reference

6.1 Command Structure

The remote commands are divided into two types: Common commands and SCPI commands. The common commands are defined in IEEE std. 488.2-1987, and these commands are common for all devices. Not all commands are supported by the 5492B, and some commands are not supported by the GPIB interface for the 5492BGPIB. Please look through the command syntax thoroughly before programming. The SCPI commands are used to control most of the 5492B functions. They can be represented as a tree structured with three levels deep. (The highest level commands are called the subsystem commands in this manual.) The lower level commands are part of subsystem commands and a colon (:) is used to separate the higher level commands and the lower level commands. See Figure 6-1 as an example.

***IDN?**
Query Syntax:
*IDN?
Query return:
<product>,<version>,<sn number>

Example: 5492B Digital Multimeter, Ver1.0.00.00.01,123A45678

Description: Query the identification of the instrument.

SCPI というテキストコマンド
プロトコルを使っている（業界標準）

コマンドもリプライもソケット経由の
テキストで超簡単

- *IDN? で型番を返す
- *RST でリセットする
- :READ? で電圧値を返す

Step 1: SCPI コマンドを送ってみる

SlowPy ライブラリを使うと簡単（使わなくても良い）

```python
from slowpy.control import ControlSystem
ctrl = ControlSystem()

voltmeter = ctrl.ethernet("10.0.0.54", port=1234).scpi()

print(voltmeter.command('*idn?'))

print(voltmeter.command(':read?'))
```

```
sanshiro@vs13: ~/SlowVoltmeter
sanshiro@vs13:~/SlowVoltmeter$ python3 slow-voltmeter.py
Ethernet: 10.0.0.54:1234 connected
5492B Digital Multimeter,Ver1.1.13.06.04,343B18165
-8.012122e+000
sanshiro@vs13:~/SlowVoltmeter$
```

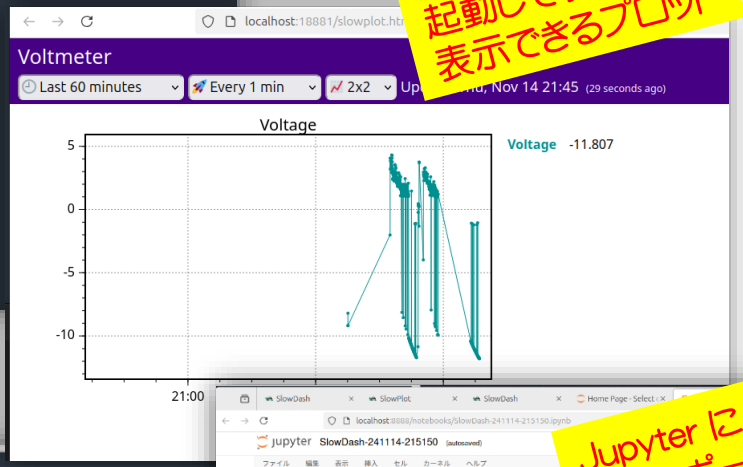
スクリプト単体でハードウェアとの通信をチェックできる

Step 2: データベースに書き込んで完成

ループにしてデータベース記録を追加した

```python
from slowpy.control import ControlSystem
ctrl = ControlSystem()

from slowpy.store import DataStore_SQLite
datastore = DataStore_SQLite('sqlite:///SlowTestData.db', table="Test")

voltmeter = ctrl.ethernet("10.0.0.54", port=1234).scpi()

print(voltmeter.command('*idn?'))


for i in range(3):
    V = voltmeter.command(':read?')
    datastore.append(V, tag='Voltage')
```

単体でも
使用可能

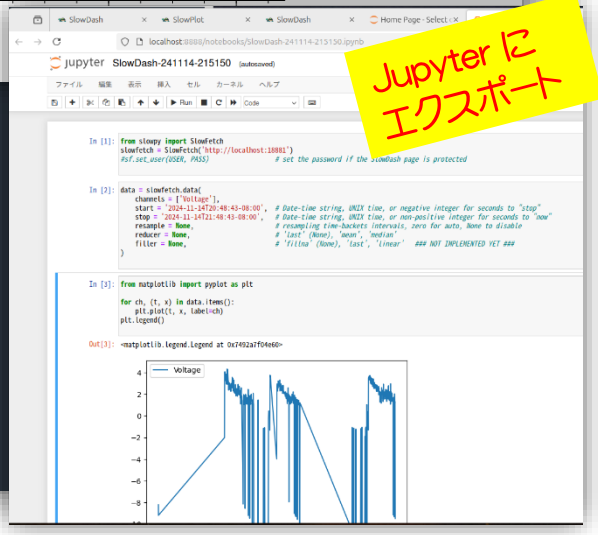


起動してすぐ
表示できるプロット

SlowDash 設定ファイル

```
slowdash_project:
  name: Voltmeter

  data_source:
    url: sqlite:///SlowTestData
    parameters:
      time_series:
        schema: Test [channel] @timestamp(unix) = value
```

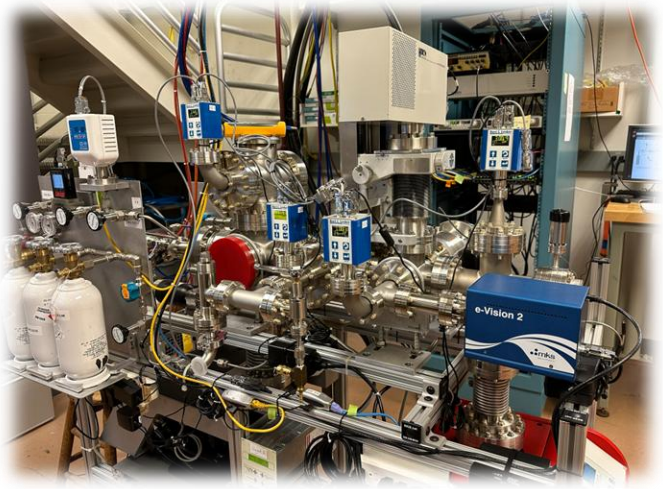
DB 設定情報



Jupyter に
エクスポート

応用例：普通にデバイスコントロール

大学実験室のセットアップ（100 ch 程度）



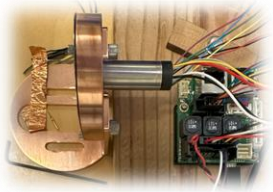
大半が単純な SCPI デバイス

- 電源, 電圧計
- 真空ポンプ, 真空計, バルブ, 流量計, …
- RTD 温度計, 熱電対, ヒーター, …

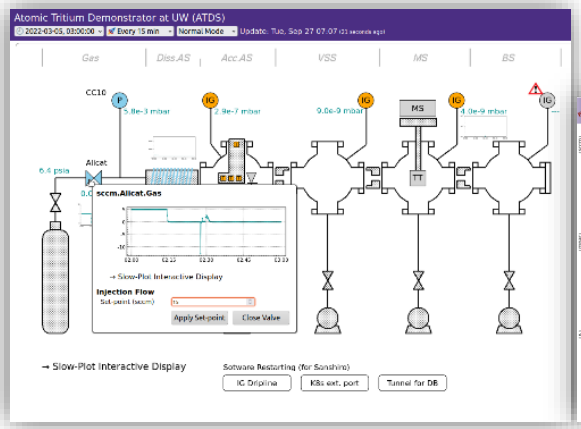
→ Python で 5 行くらい

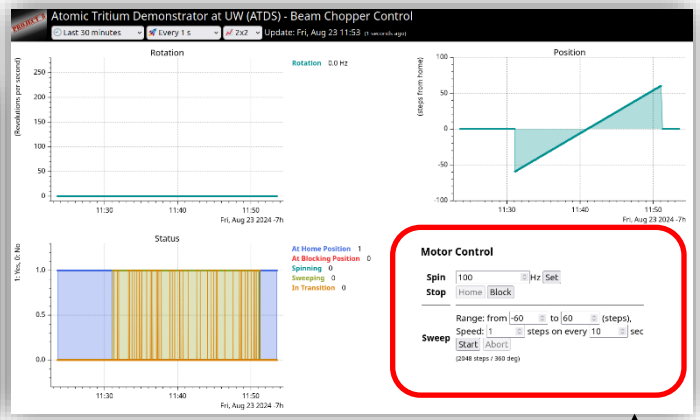
一部の複雑なデバイス

- ステッピングモーター
- 波形記録（リプルモニタ）



→ Python で 100 行くらいと HTML フォーム

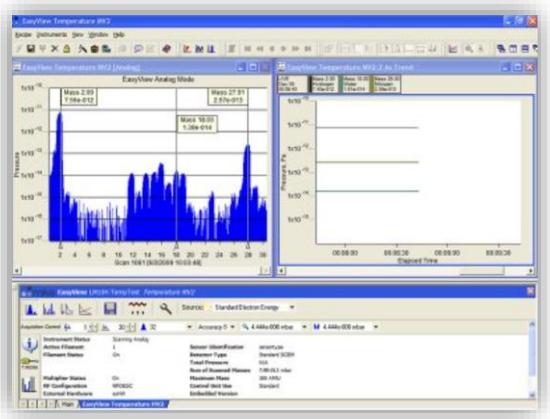




スクリプトの関数を呼ぶ HTML フォーム

応用例: 残留ガス分析器

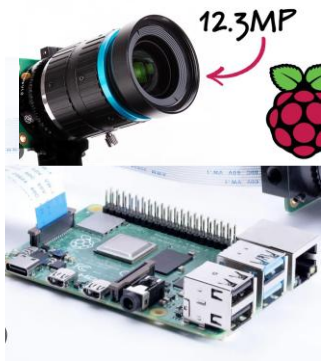
付属の Windows のソフトウェアが使われていて, 使い勝手が悪かった

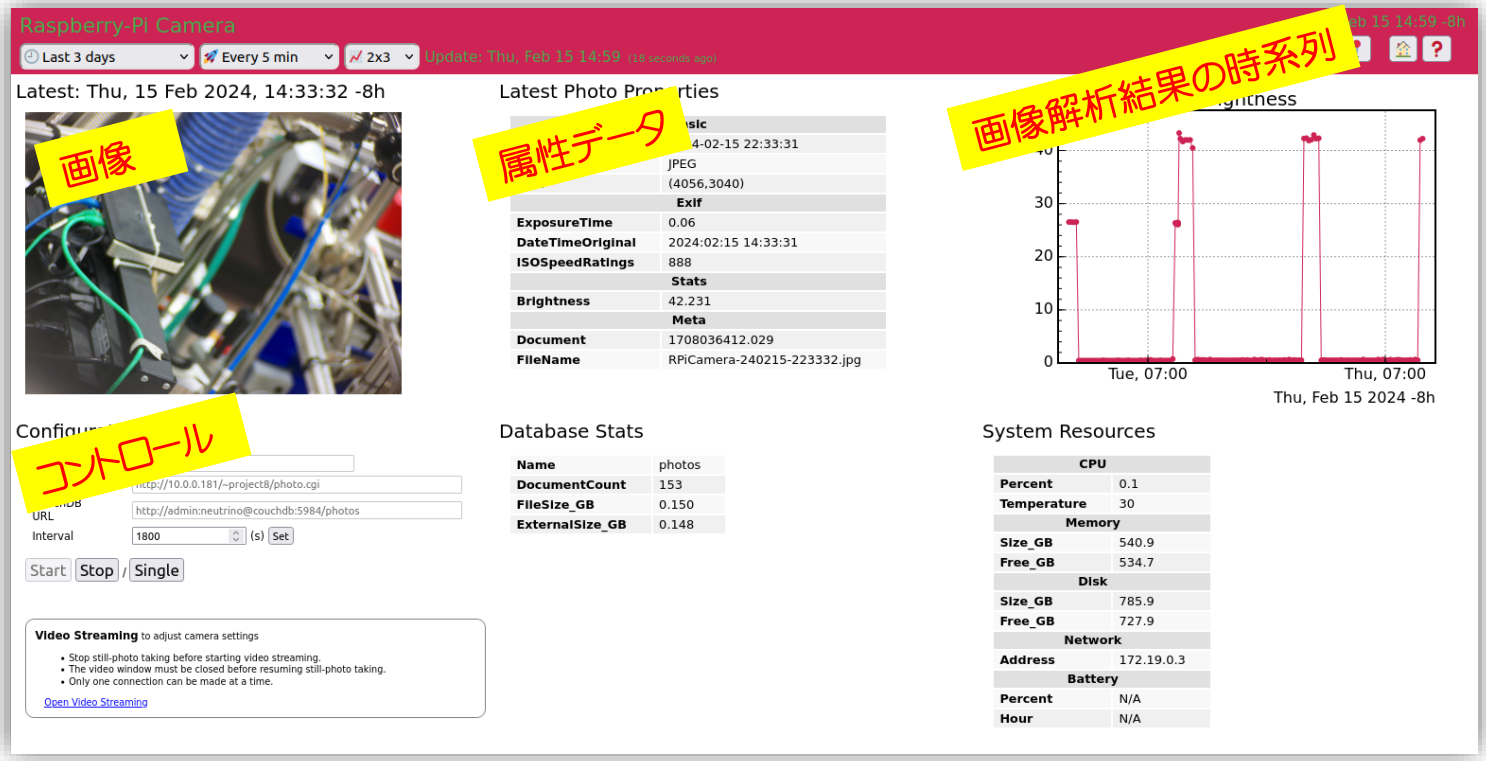


製品付属 Windows ソフトウェア:
悪くはないけれど, システム統合が難しい

応用例: Raspberry-Pi カメラと画像解析

画像データ(の時系列)も扱える





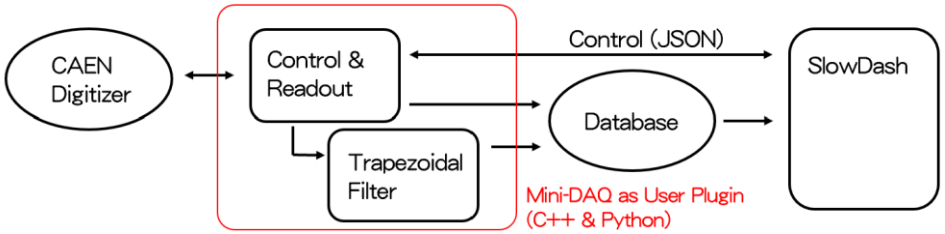
Python の画像解析や機械学習のライブラリがそのまま使える

- 変化検出, 異常検知
- パネル表示値認識
- …

応用例: 実験室DAQ

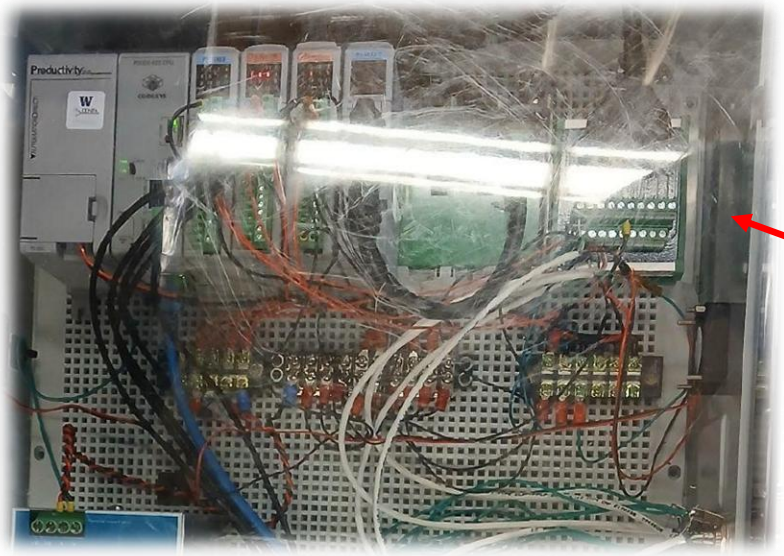
波形記録デバイスを使った小規模 DAQ, ランコントロール付き

読み出しと波形解析を C++ ユーザモジュールとして実装してみた

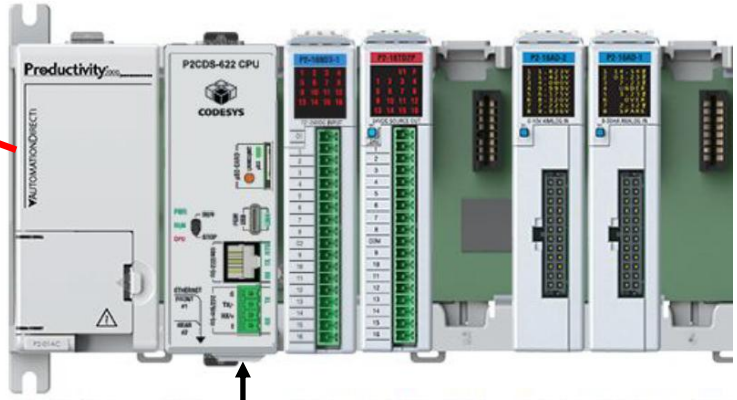


応用例: PLC システム

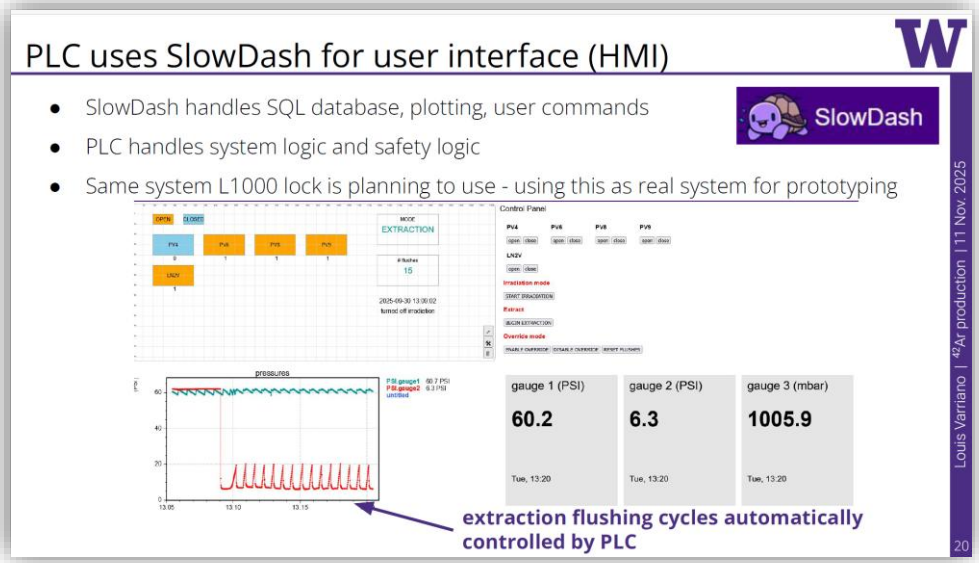
ワシントン大学の別グループによる使用例



PLC: Programmable Logic Controller
産業界でファクトリーオートメーションに使われているもの

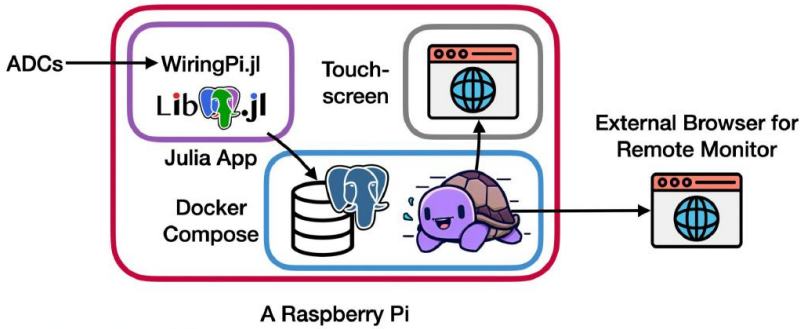


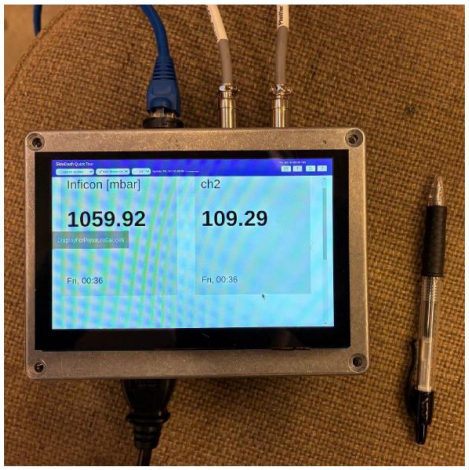
MQTT & Modbus
(産業用コントロール通信)

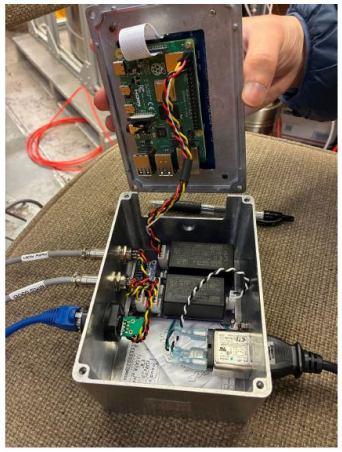


応用例：IoT スマートパネル

ワシントン大学の別グループの学生によるプロジェクト







Raspberry-Pi 上に全て載せる：

- ADC
- SlowDash
- タッチスクリーン
- データベース
- Julia 言語による読み出し

→ 完結したプログラマブルデータロガー

間にデータベースが入っているので
どんなシステムでもつなげられる

⇩

データセントリック設計

データセントリック設計

まずデータがある. データの周りに機能を作る

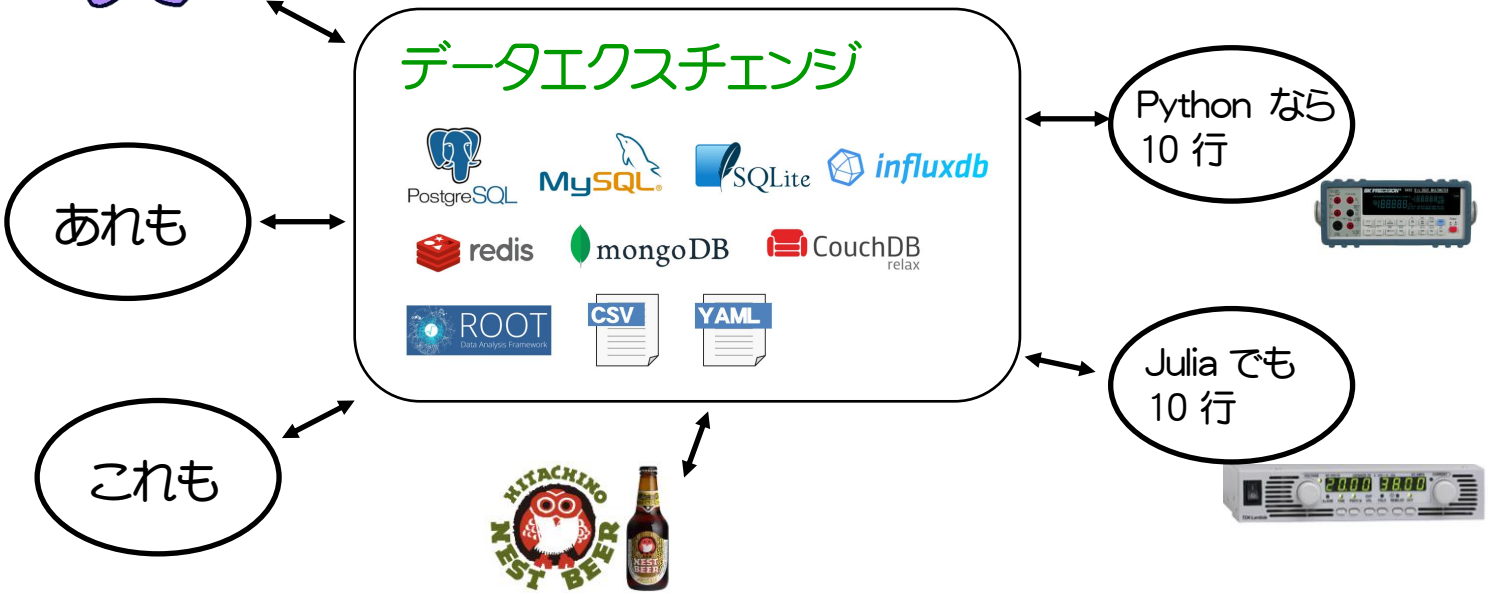
- コンポーネントは API ではなくデータで繋がる

⇩

- API を理解しなくていい. 標準的方法で何でもつながる
- アプリが死んでもデータは残る

データスキーマが超重要

- 事前にしっかり決めるか,
- 柔軟に対応できるように



SlowDash: だいたいの時系列データは扱える

よくある形式その1（ロングフォーマット）

channel	timestamp	value
sccm.Alicat.Inj.Gas	2022-09-15 03:19:25.496212+00	0
V.ThermoCo.Diss.AS	2022-09-15 03:19:27.612427+00	6.605405e-05
mbar.IG.Vac.AS	2022-09-15 03:19:31.490579+00	2.26e-07
mbar.IG	2022-09	45+00	2e-
mbar.IG.Vac.BS	2022-09-15 03:19:31.610188+00	4e-

テーブル

タグ

時刻（ISO）

フィールド

SlowDash のフォーマット記述

```
table [channel]@timestamp = value
```

よくある形式その2（ワイドフォーマット）

RunNumber	TimeStamp	sccm.Alicat.Inj	mbar.CC10.Inj	K.ThrmCpl.Diss	mbar.IG.AS
3098	1664916014	3	1.18467	340.58	5.38333e-05
3097	1664915456	3	1.256	503.275	5.36e-05
3096	1664914833	3	1.36833	745.743	5.38333e-05
3095				1154.09	5.44
3094			1.48933	1501.14	5.46

テーブル

時刻（UNIX）

複数フィールド

```
RunTable@TimeStamp
```

混ぜたもの

metric	set_or_ist	timestamp	value_raw	value_cal
psia.Alicat.Inj.Gas	ist	2022-09-15 03:19:25.419417+00	9.6	9.6
degC.Alicat.Inj.Gas	ist	2022-09-15 03:19:25.458695+00	23.42	23.42
sccm.Alicat.Inj.Gas	ist	2022-09-15 03:19:25.496212+00	0	0
V.ThermoCo		2022-09-15 03:19:27.612427+00	6.6	e-05
V.PS.Diss.AS	ist	2022-09-15 03:19:29.387352+00	0.01	0.01
A.PS.Diss.AS	ist	2		

テーブル

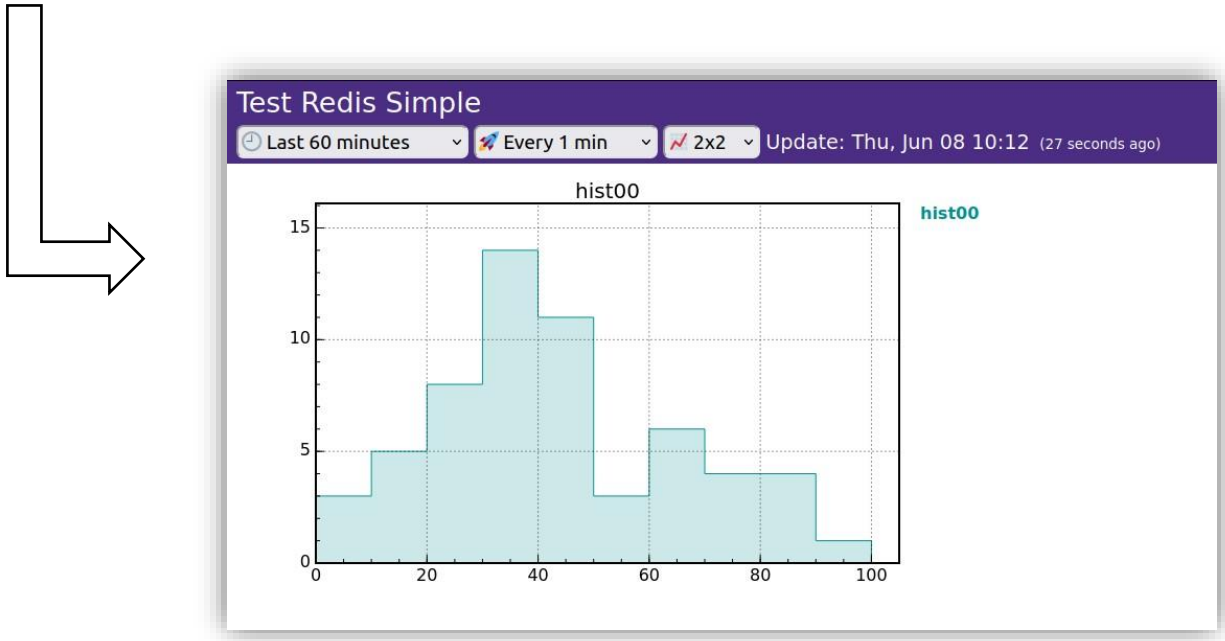
複数タグ

複数フィールド

```
Data[metric, set_or_ist]@timestamp = value_raw, value_cal
```

ヒストグラムとかは単純な JSON 文字列

```
{
  "bins": { "min": 0, "max": 100 },
  "counts": [ 3, 5, 8, 14, 11, 3, 6, 4, 4, 1 ]
}
```



異なる JSON フォーマットに対応するのも比較的簡単
（NumPy/Matplotlib 形式には対応済）

失敗しそうなところ

ここまでの SlowDash: 「ユーザの Python スクリプトで好きなように制御してください」

全てが異なることをする大規模並列システムをどう記述すればいいか考えた

25

コントロールロジックの実装方法を考える

進行中プロジェクト：60年前の加速器制御システムを近代化する

これを iPad (SlowDash) に置き換えたい…

(30年前のアップグレードでかなりの部分が VAX と PLC に置き換えられた)

基本的には，たくさんのスイッチ，ノブ，表示板と，装置動作の「シーケンス」

今日のテーマ(半分遊び)

全部違う多チャンネル並列システムの「シーケンス」をどうやって記述すればいいか

26

まずは勉強：産業用コントロールシステム

ふつうは，こういうやつが機器を制御している

これを Web ページ (SlowDash) に置き換えたい

中身はこんな感じ

UI

リレースイッチの嵐と
それを制御するもの (PLC)

PLC (Programmable Logic Controller)：ラダーロジックでシーケンスを記述する

産業用ファクトリオートメーション(PLC)はこれをやっているのではないか?

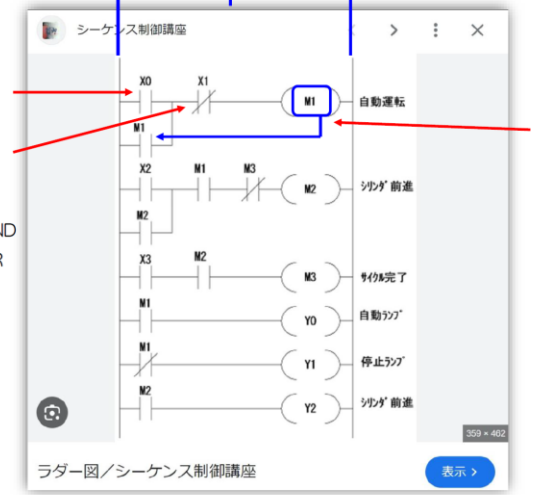
失敗しそうなところ

PLC はラダーロジックを使う. これは FPGA と本質的に同じだと思った

ラダーロジック: PLC のシーケンス記述

27

両端に電源がつながっているイメージ



押すと ON になるスイッチ（開始）

押すと OFF になるスイッチ（停止）

直列に並べると AND
並列に並べると OR

一度運転状態になると「開始」を押し続けるのと同じ効果

「停止」を押すと止まる

⇩

状態の維持と遷移（ステートマシン !!!）

つまり

- 梯子の左半分で入力に対する論理演算
- 右端が出力. 出力は次の瞬間の入力にも使える
- 梯子の横棒は（概念的に）並列実行. それをたくさん並べる

これって聞いたことがある…

ラダーロジックはこれと同じ?

28

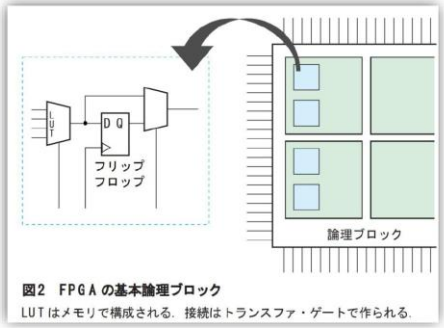


図2 FPGA の基本論理ブロック
LUT はメモリで構成される. 接続はトランスファ・ゲートで作られる.

ラダーロジックがいまだに広く使われている
（ソフトウェアを知らなくてもプログラムが簡単らしい）

⇩

←これの集合でロジックを記述するのは並列同時制御に向いている?

⇩

大規模な「これ」の集合を記述するには HDL（Verilog とか）がいいと歴史が示している

⇩

SlowDash でも HDL 風にロジックを書きたい（ループとかダサい）

SlowPy-HDL: SlowDash の Python スクリプトを HDL 風に書く（半分遊び）

- シーケンス記述にたぶん向いている
- 既存のラダーロジックをそのまま変換できる
- 無理に使わなくてもよい. 普通の Python で書いても全く構わない

FPGA ロジックの記述には HDL を使う.

Python で HDF 風の記述をすれば, 大規模並列システムをうまく書けないか?
（ループのないカッコいいコードを書けるかも）

失敗しそうなところ

やってみた.

31

SlowPy-HDL （半分遊び）

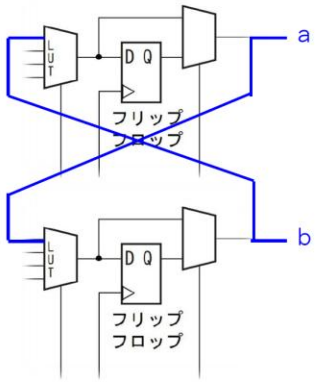
ちゃんと HDL 的にふるまう

こうやって書くと a と b の値を毎回入れ替える

```
class TestModule(Module):
    def __init__(self, clock, a, b):
        super().__init__(clock)
        self.a = output_reg(a)
        self.b = output_reg(b)

        self.a <= 'A'
        self.b <= 'B'

    @always
    def swap_ab(self):
        self.a <= self.b
        self.b <= self.a
```

“ソフトウェア的”なら両方 B になる

SlowPy の「レジスタ」には
ラッチがエミュレートされている



SlowPy-HDL の実際の動作

1. 入力レジスタ値を更新, 保持（デバイスからの読み出しなど）
2. 全ての「@always プロセス」を実行
3. 出力レジスタ値を更新（デバイスへの書き込みとか）
4. 寝る
5. 1 に戻る

論理的「クロックエッジ」

関数が順番に実行されていて
本当に同時ではない点は
配線遅延だと思えばよい

クロックで駆動される「プロセス」

```
@always
def update(self):
    if self.clear:
        self.count <= 0
    elif self.running:
        if self.count == 59:
            self.count <= 0
        else:
            self.count <= int(self.count) + 1
```

同じことが Verilog - HDL だったら

```
always @(posedge clock)
begin
    if (clear == 1'b1)
        count <= 8'd0;
    else if (running == 1'b1)
        if (count == 8'd59)
            count <= 8'd0;
        else
            count <= count + 8'd1;
end
```

使ってみたら

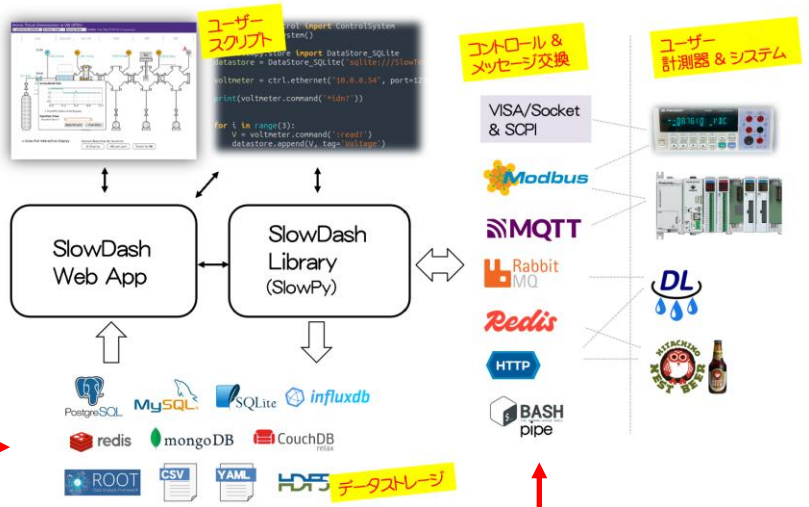
- Python の async（非同期I/O）と絶望的に相性が悪い
- 非同期フレームワークで同期システムを記述しているから ????

模索中：データセントリック制御 （私の発明ではありません）

記録データに対しては自明

時系列データの Store & Query

発生時刻	チャンネル	値
2025-11-17 09:50:00	High Voltage 1	1800 V
2025-11-17 09:50:03	Run State	Running
…	…	…



同じスキーマをメッセージに乗せて制御にも使いたい

未来時刻状態の Publish & Subscribe

執行時刻（未来）	チャンネル	値
2025-11-17 09:50:00	High Voltage 1	1800 V
2025-11-17 09:50:03	Run State	Running
…	…	…

- 機器追加がメインロジック変更を伴わない
- 現状態・エラーを publish しても良い
- 読み出しデータを publish しても良い
  → どっかで subscribe して DB 記録
  → だれかが加工して再 publish も

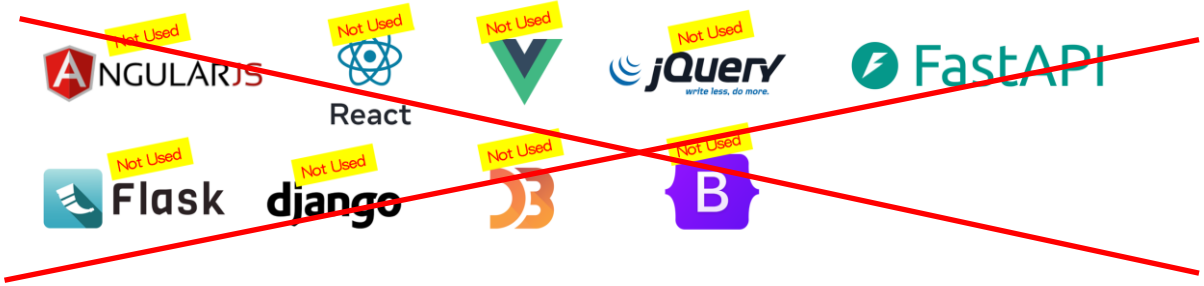
コマンドではなく状態を指定（publish）する

→ それを subscribe してさらに詳細な状態を publish しても良い
（疎結合な階層的状態機械）

長寿命設計（目標25年以上）

プラットフォーム型ライブラリやフレームワークの類は使わない

（jQuery とか使っていたけど排除した）



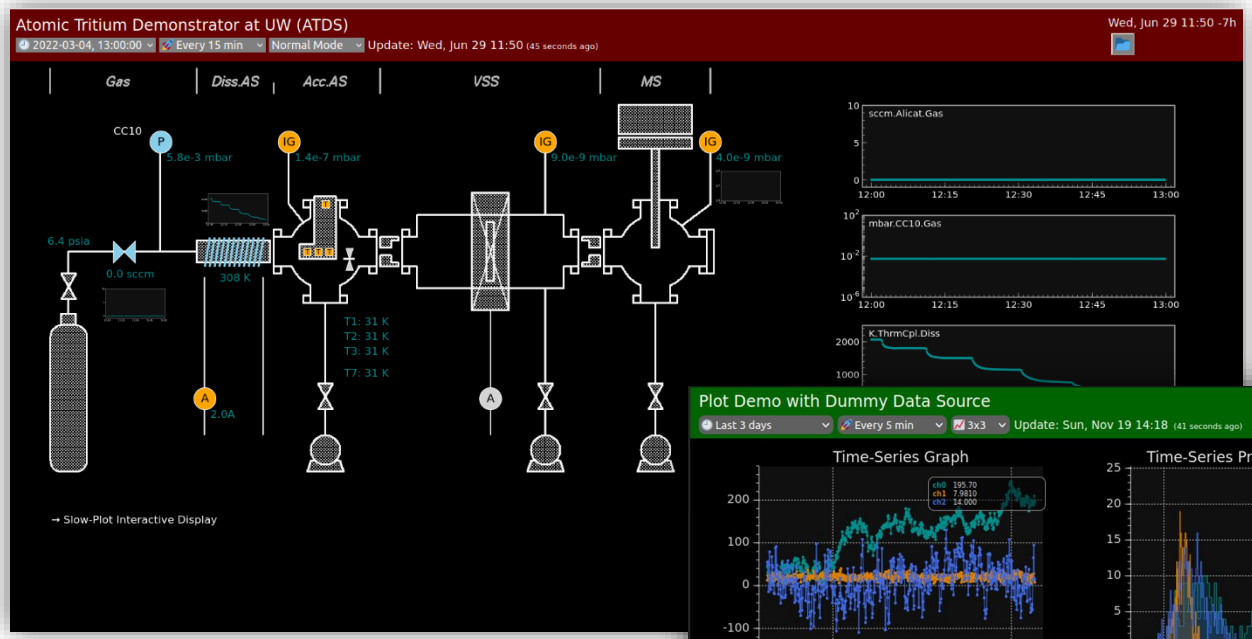
便利なライブラリはプラグイン経由で使う ⇒ 使える間は使ってあとは捨てる

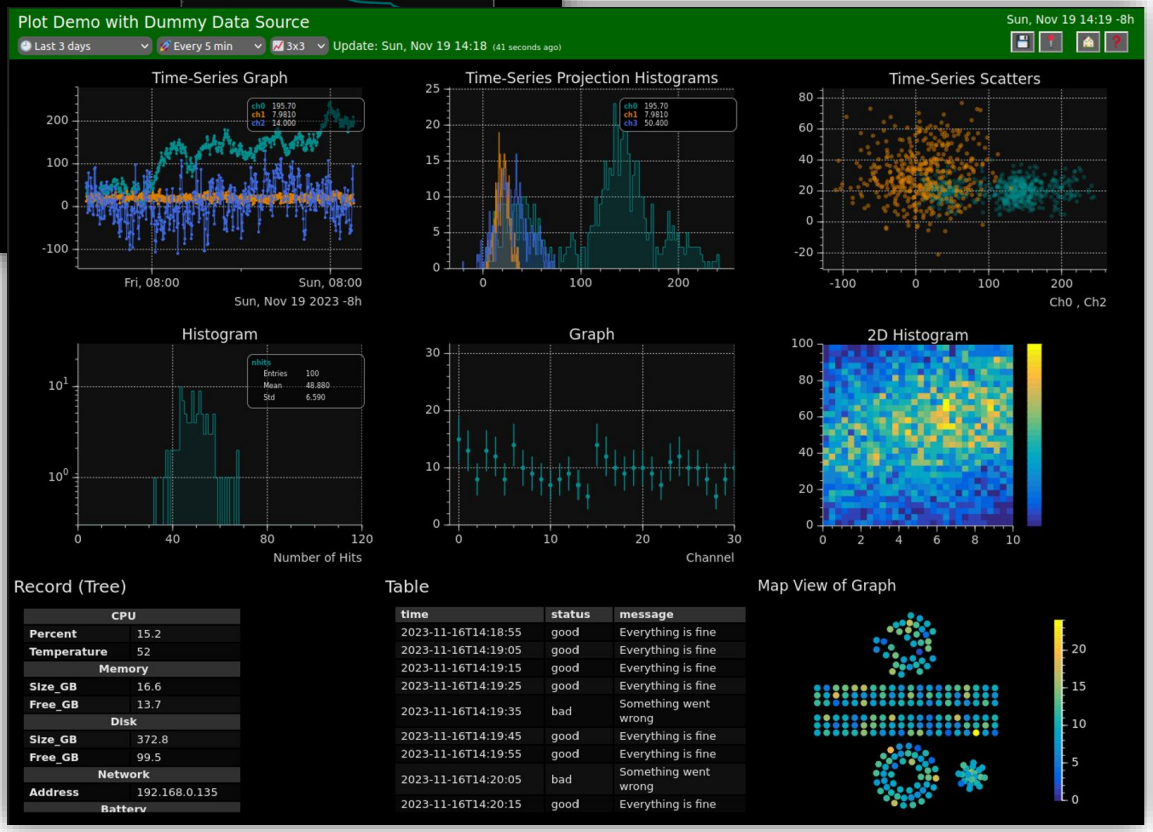
- データベース：PostgreSQL, InfluxDB, CouchDB, MongoDB, …
- メッセージング：Redis, AMQP, MQTT, …
- デバイスを使うためのベンダ供給ライブラリ （SlowDash は Windows でも動きます. たぶん）

データセントリック設計

- API を使わない相互接続
- アプリが死んでもデータは死なない
- 状態データによる制御： 誰が制御を投げても同じ動作

「Grafana みたいに作れ」と言われたときのための ダークモード





開発状況と今後

24 年1月 “Nisqually”

- データベースに保存されたデータのビジュアライゼーション

24年7月 “Snoqualmie”

- 計測制御・読み出しの Python スクリプティング

24年11月 “Skykomish”

- 共通コントロール部品の実装
- 解析スクリプトの Jupyter 接続

25年7月 “Nooksack”

- 非同期 Web エンジン, データストリーミング

現在

- 通信インフラ, 分散化

SPADI-A WG3 に居候中
(mattermost WG3-WebDisplay)

要望, アイディア, 共同開発は大歓迎です

このあと：主に解析系

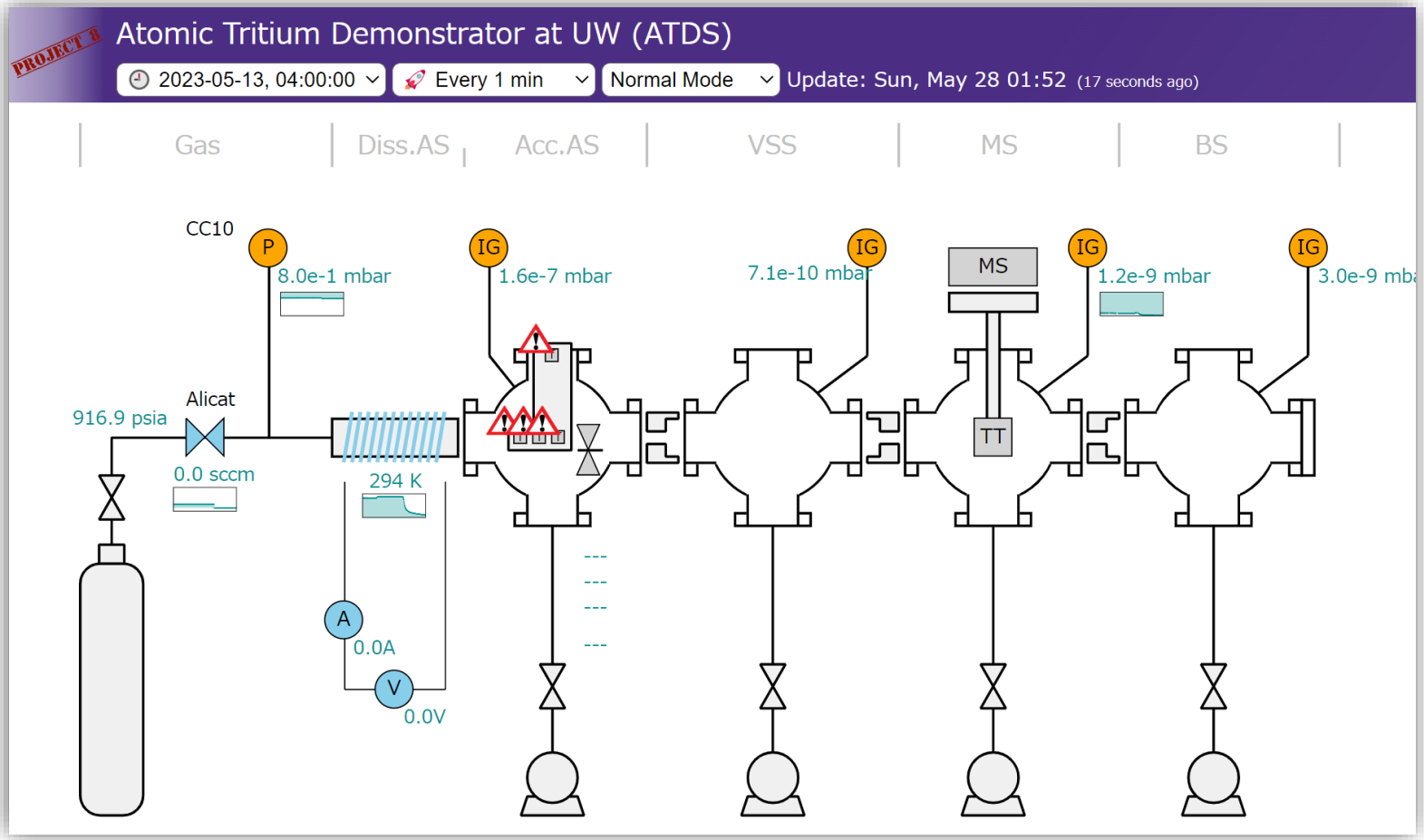
- 埋め込みデータ加工・データ変換（テーブルデータをヒストグラムで表示とか）
- アラーム検出・通知・認知・マスク・リマインド・履歴管理, 障害予測(?)
- かっこいいレイアウト, 数千チャンネルの表示とか

GitHub: https://github.com/slowproj/slowdash
日本語マニュアル: https://slowproj.github.io/slowdash/#FirstStep-JP

Appendix

ダッシュボード

- 静止画の上にデータ要素を並べる. YAML ファイルで記述.
- 色で On/Off 状態を表示 (正常/異常ではなく)
- 最新値の数値表示と, ミニトレンドグラフ



ダッシュボード: YAML 設定ファイル例

```
viewBox: 0 0 1600 700

widgets:
  - type: image
    attr: { x: 50, y: 49, height: 503, width: 933, href: "ATDS.png" }

  - type: circle
    attr: { x: 180, y: 95, width: 40, height: 40, label: P }
    data: { channel: "mbar.Gas", "active-below": 500, format: "%.1e mbar" }

  - type: valve
    attr: { x: 130, y: 230, width: 40, height: 40, orientation: h, "data-dx": -10, "data-dy": 60 }
    data: { channel: "SCCM.Gas", "active-above": 0.1, format: "%.1f SCCM" }

  - type: circle
    attr: { x: 255, y: 430, width: 40, height: 40, label: A }
    data: { channel: "A.PS.Diss", "active-above": 0.1, format: "%.1fA" }

  - type: solenoid
    attr: { x: 270, y: 210, width: 110, height: 80, "data-dx": 30, "data-dy": 100 }
    data: { channel: "K.ThermCo.Diss", "active-abo

  - type: circle
```

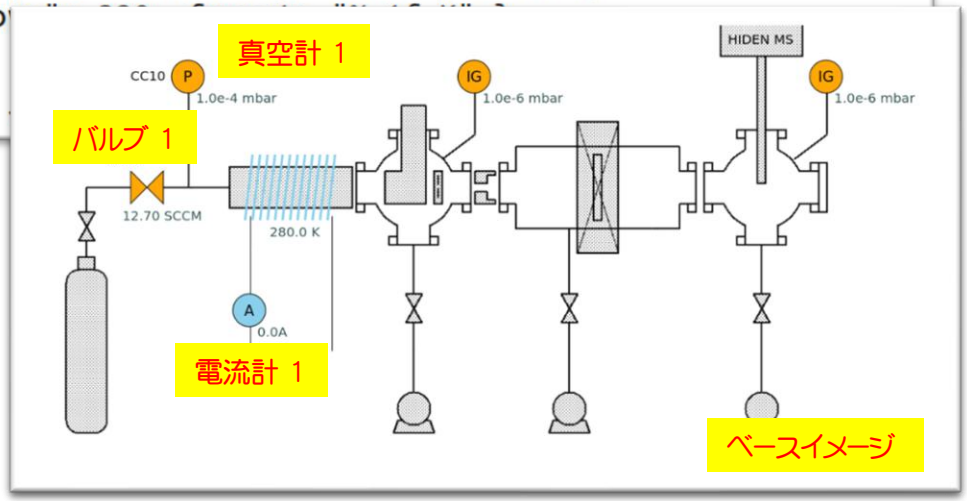
ベースイメージ

真空計1

データストア中のチャンネル名

バルブ 1

電流計 1

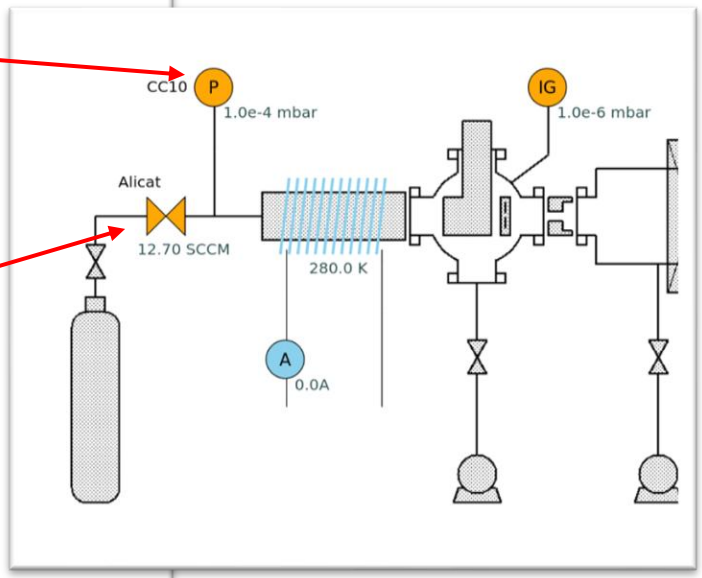


- YAML を書くのはやってみると
  それほど大変ではない
- 作り込みが好きなユーザは意外に多い
- 将来的にはビルドツールを作るかも
  (マウス操作より数値を打つ方が早い説もあるけど)

新しい形状の追加も簡単

- JavaScript で 10 行くらい (SVG 要素を作って返すだけ)
- プラグインにできる. コレクションにできる.

```
class SCCircleWidget extends SCShapeWidget {
    get_defaults() {
        return $.extend({}, super.get_defaults(), {
            width: 20, height: 20
        });
    }
    create_path() {
        let circleAttr = $.extend({}, this.attr, {
            cx: this.attr.x + this.attr.width/2,
            cy: this.attr.y + this.attr.height/2,
            rx: this.attr.width/2,
            ry: this.attr.height/2
        });
        return $('<ellipse>', 'svg').attr(circleAttr);
    }
};

class SCValveWidget extends SCShapeWidget {
    get_defaults() {
        return $.extend({}, super.get_defaults(), {
            width: 20, height: 20, orientation: 'horizontal'
        });
    }
    create_path() {
        let [x0, y0] = [this.attr.x, this.attr.y];
        let [x1, y1] = [x0 + this.attr.width, y0 + this.attr.height];
        let points = (
            (this.attr.orientation[0] == 'v') ?
            `${x0},${y0} ${x1},${y0} ${x0},${y1} ${x1},${y1} ${x0},${y0}` :
            `${x0},${y0} ${x0},${y1} ${x1},${y0} ${x1},${y1} ${x0},${y0}`
        );
        return $('<polyline>', 'svg').attr(this.attr).attr({'points': points});
    }
};
```



データ元：いろいろなデータストアを読める

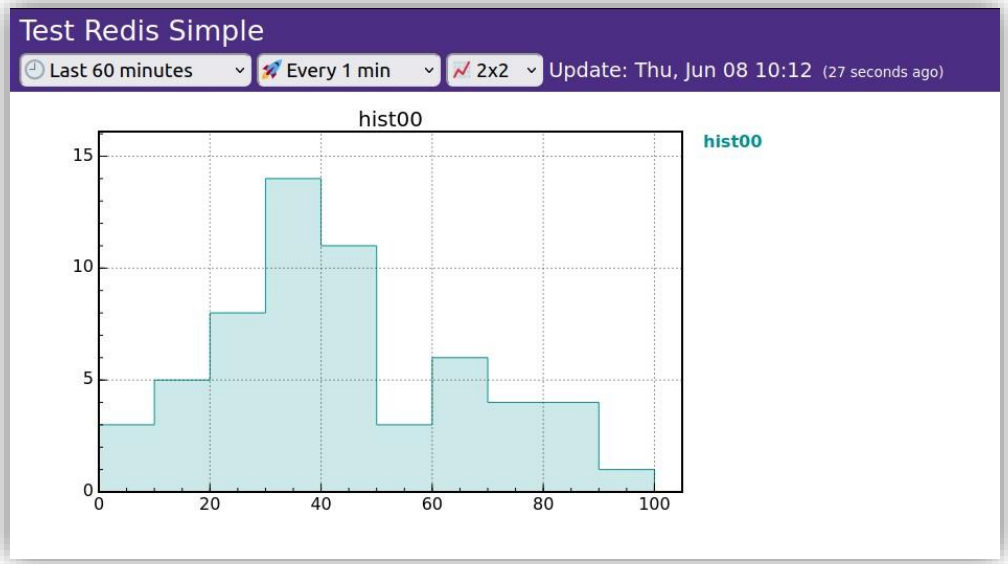
- これらは全てプラグイン実装
- MySQL プラグインは COMET/KEK 大石さんからの寄与

ヒストグラムは JSON 文字列として記録

データ値として JSON の文字列を書く

```
{
  "bins": { "min": 0, "max": 100 },
  "counts": [ 3, 5, 8, 14, 11, 3, 6, 4, 4, 1 ]
}
```

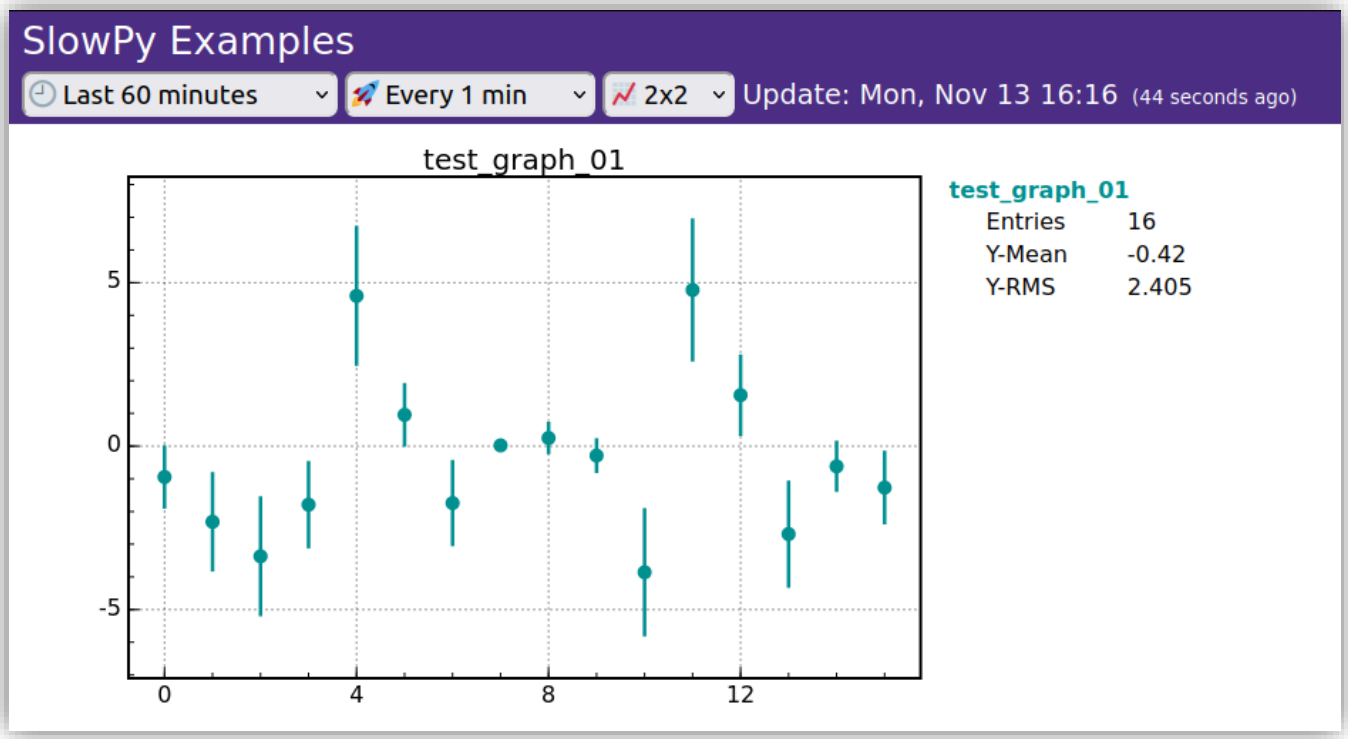
伸びていくヒストグラムの時系列



JSON へのシリアライザライブラリ： SPADI-A による C++ 版と組み込みの Python 版

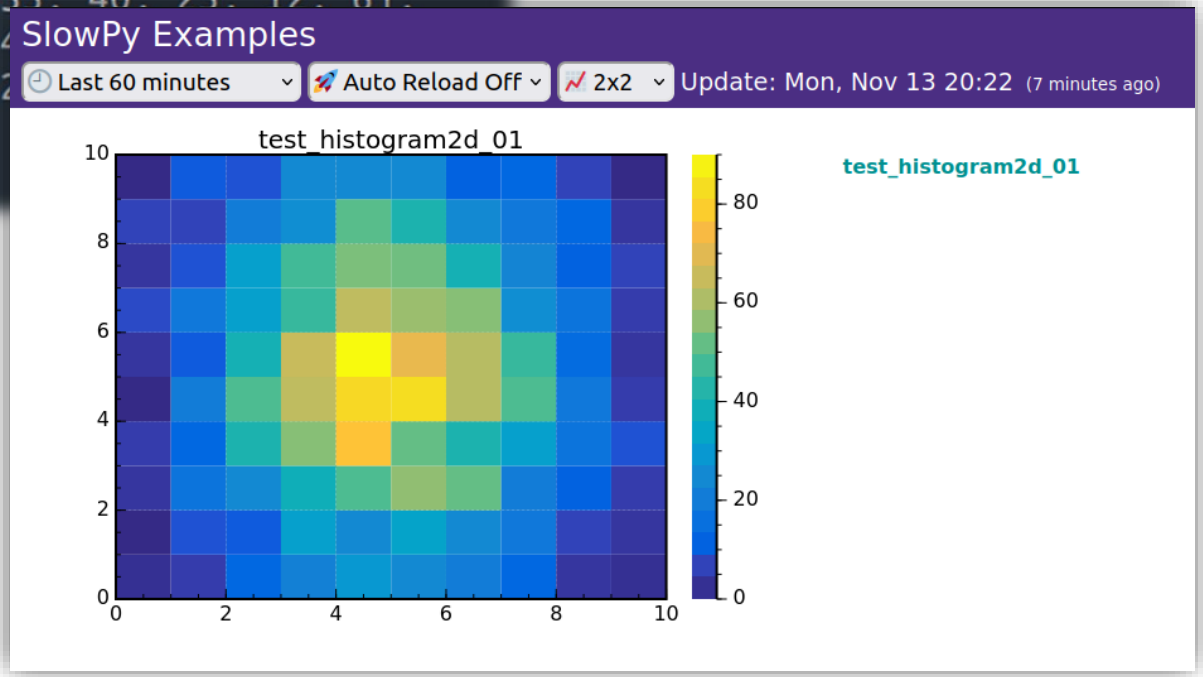
グラフ(エラーバー付き)はこんな感じ


```
{
    "_stat": { "Entries": 16, "Y-Mean": -0.449, "Y-RMS": 2.5 },
    "labels": [ "ch", "value", "error" ],
    "x": [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 ],
    "y": [ -5.117, 0.765, -0.467, -2.738, -2.407, 3.652, 0.953, 3.262, 0.327, 0.55
    "y_err": [ 2.2620, 0.8746, 0.6833, 1.6546, 1.5514, 1.9110, 0.9762, 1.8061, 0.5
}
```

2次元ヒストグラムも


```
{
    "xbins": {"min": 0, "max": 10},
    "ybins": {"min": 0, "max": 10},
    "counts": [
        [2, 5, 14, 22, 29, 25, 20, 13, 3, 2],
        [1, 9, 11, 31, 25, 33, 24, 19, 6, 3],
        [3, 17, 24, 39, 49, 57, 52, 21, 12, 5],
        [5, 13, 42, 55, 76, 51, 42, 31, 17, 9],
        [1, 20, 48, 64, 82, 83, 62, 48, 19, 5],
        [4, 11, 40, 65, 90, 71, 62, 46, 15, 4],
        [8, 19, 31, 45, 64, 58, 55, 26, 17, 5],
        [4, 9, 32, 47, 54, 53, 40, 23, 12, 6],
        [6, 7, 20, 26, 50, 4
        [1, 10, 9, 24, 25, 2
    ]
}
```

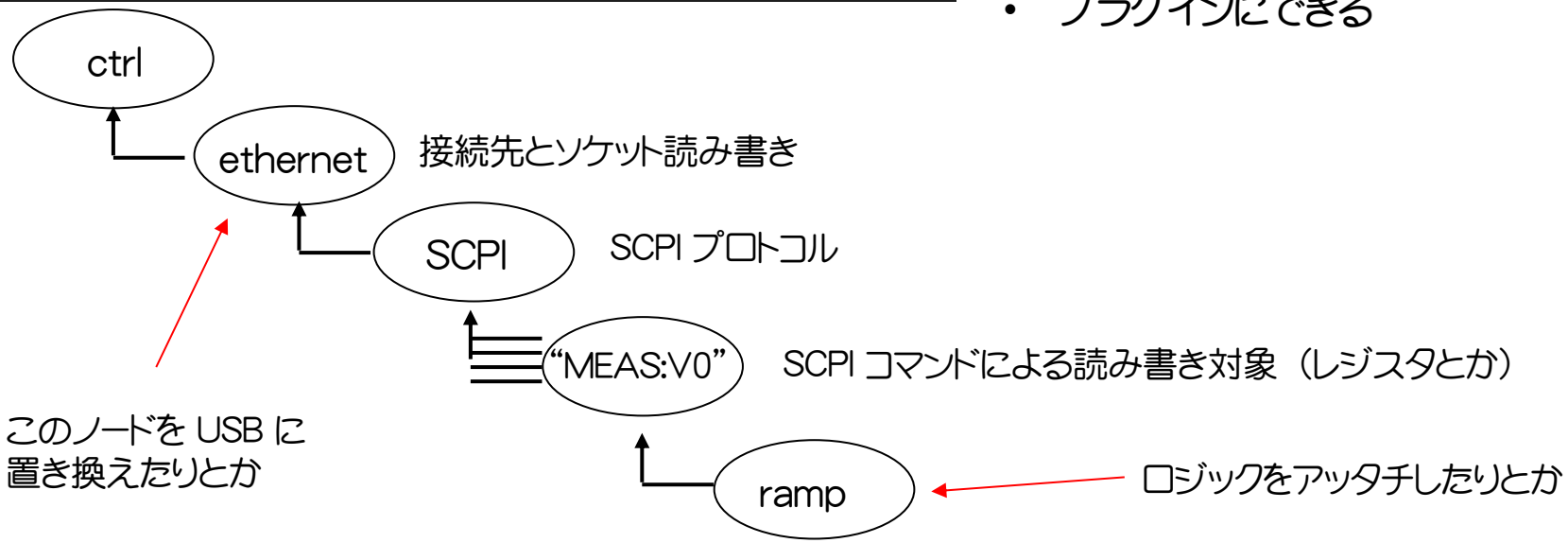



SlowPy：読み書きできる制御変数の木構造

ハードウェアの例：イーサーネットのSCPI コマンドで制御する電源ユニット

```
from slowpy.control import ControlSystem
ctrl = ControlSystem()

# SlowPy Control Variables
device = ctrl.ethernet('192.168.1.43', port=17674).scpi()
device_id = device.command('*IDN')
V0 = device.command('MEAS:V0')

print('ID: %s' % str(device_id.get()))  # read device ID
V0.ramp(ramping=0.1).set(10)  # apply set-point with ramping
while True:
    data = V0.get()     # read voltage value from the device
    ...
```

- 読み書き可能ノードの集合
- データ型は何でもよい：
  - 数値
  - 長いテキストや JSON
  - Python オブジェクトでも
- 下位のノードが上位ノードを利用する
- 接続する上位ノードを選べる
- プラグインにできる

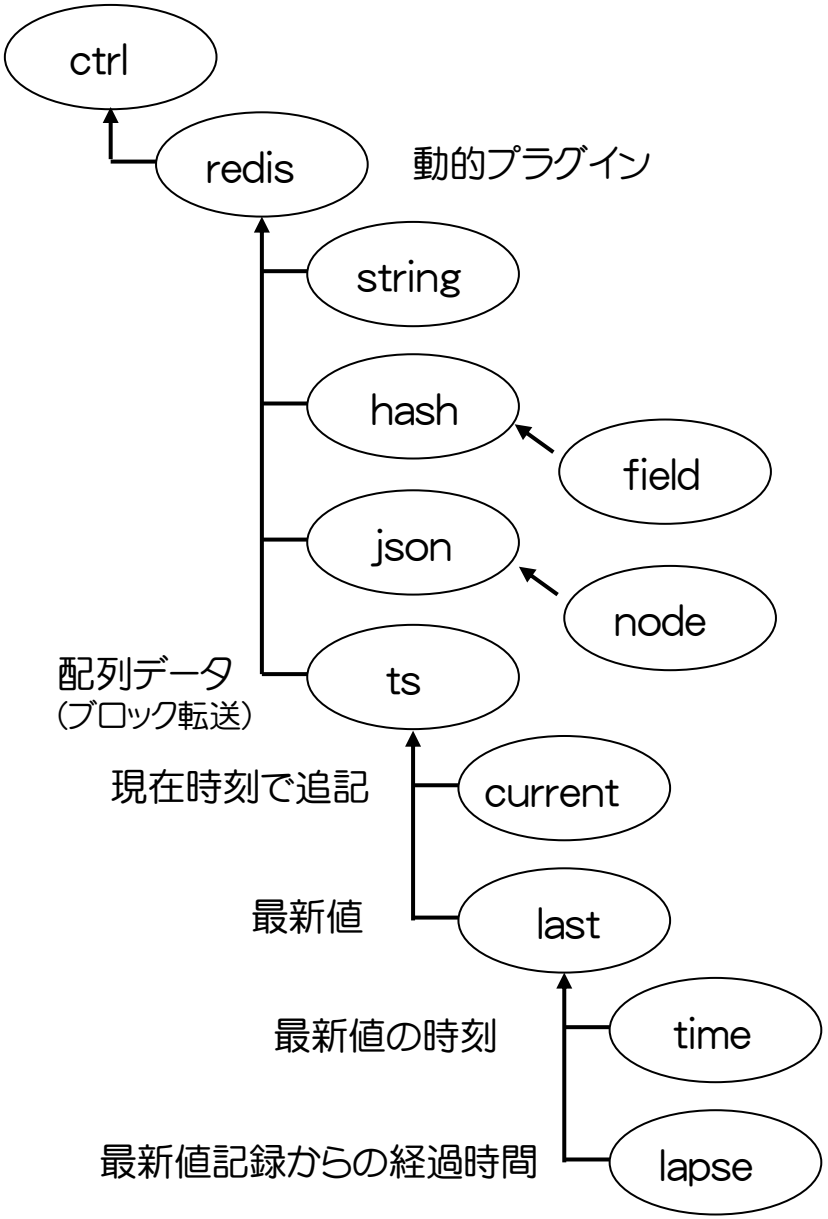


SlowPy：読み書きできる制御変数の木構造

外部システム接続の例：Redis とやりとり

（モジュールはプラグインの動的ロード）

```
from slowpy.control import ControlSystem
ctrl = ControlSystem()
ctrl.load_control_module('Redis')
redis = ctrl.redis('redis://localhost:6379/12')


# string
redis.string('name').set('SlowDash')
print(redis.string('name'))


# hash
print(redis.hash("Status"))
print(redis.hash("Status").field("Count"))
redis.hash("Status").field("Count2").set(10)


# time-series
print(redis.ts('ch00'))
print(redis.ts('ch00').last())
print(redis.ts('ch00').last().time())
print(redis.ts('ch00').last().lapse())

import time
redis.ts('ts00').current().set(123)
redis.ts('ts00').set([(int(1000*(time.time()-100)), 456)])
print(redis.ts('ts00'))
print(redis.ts('ts00').last().lapse())
```



SlowAPI を作ってみた （後に Slowlette に改名）

FastAPI 風の URL リクエストルーティング

http://localhost:8000/hello に応答する例

Flask

```python
from flask import Flask

app = Flask(__name__)

@app.route("/hello")
def hello(name):
    return "Hello, world."

app.run()
```

同期ウェブサーバー（WSGI）が走る

FastAPI

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/hello")
async def hello():
    return "Hello, world."

app.run()
```

非同期ウェブサーバー（ASGI）が走る

SlowAPI

```python
from slowapi import SlowAPI

app = SlowAPI()

@app.get("/hello")
async def hello():
    return "Hello, world."

app.run()
```

非同期ウェブサーバー（ASGI）が走る

FastAPI / SlowAPI では

- メソッド名で直接バインディング
  （GET / POST / DELETE）
- ハンドラは非同期（async def）
  → 複数リクエストの並行処理
  （スレッドの並列処理ではない）

SlowAPI を作ってみた

FastAPI 風の 型情報によるパラメータバインディング

http://localhost:8000/hello/SlowDash?repeat=5 に応答する例

Flask

```python
from flask import Flask, request

app = Flask(__name__)

@app.route("/hello/<name>")
def hello(name):
    try:
        repeat = int(request.args.get('repeat'))
    except:
        ... generate ERROR response here ...
    return f"Hello, {name}." * repeat

app.run()
```

FastAPI

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/hello/{name}")
async def hello(name:str, repeat:int=1):
    return "Hello, world." * repeat

app.run()
```

SlowAPI

```python
from slowapi import SlowAPI

app = SlowAPI()

@app.get("/hello/{name}")
async def hello(name:str, repeat:int=1):
    return "Hello, world." * repeat

app.run()
```

FastAPI / SlowAPI では

- グローバルオブジェクトを使わない
- パラメータの型チェックと変換が自動
- デフォルト値も定義できる
- コードがきれい

SlowAPI を作ってみた

SlowAPI 独自：クラスインスタンスにバインドできる. 多重にできる.

```python
import slowapi


class Fruit:
    def __init__(self, name:str):
        self.name = name

    @slowapi.get('/hello')
    def hello(self):
        return [f'I am a {self.name}']


class MyApp(slowapi.App):
    def __init__(self):
        super().__init__()
        self.slowapi.include(Fruit('peach'))
        self.slowapi.include(Fruit('melon'))

    @slowapi.get('/hello')
    def hello(self):
        return ['Hello.']


app = MyApp()
```

```
sanshiro@vs13:~$ curl -s http://localhost:8000/hello | jq
[
  "Hello.",
  "I am a peach",
  "I am a melon"
]
```

用途：

- 部分的なデータを返すプラグインを
  動的に追加できる

⇩

- ユーザが実験ごとに構築したシステムに対する
  データクエリ, ステータス統合, …

SlowAPI を使った SlowDash ソースコード

```python
import sys, os, asyncio, logging
import slowapi

from sd_component import Component
from sd_project import Project
from sd_config import ConfigComponent
from sd_console import ConsoleComponent
from sd_datasource import DataSourceComponent
from sd_export import ExportComponent
from sd_usermodule import UserModuleComponent
from sd_taskmodule import TaskModuleComponent
from sd_misc_api import MiscApiComponent


class App(slowapi.App):
    def __init__(self, project_dir=None, project_file=None)
        super().__init__()

        if project_dir is not None:
            project_dir = os.path.abspath(os.path.join(os.getcwd(), proj
        self.project = Project(project_dir, project_file)

        self.slowapi.include(ConsoleComponent(self, self.project))
        self.slowapi.include(ConfigComponent(self, self.project))
        self.slowapi.include(DataSourceComponent(self, self.project))
        self.slowapi.include(ExportComponent(self, self.project))
        self.slowapi.include(UserModuleComponent(self, self.project))
        self.slowapi.include(TaskModuleComponent(self, self.project))
        self.slowapi.include(MiscApiComponent(self, self.project))


if __name__ == '__main__':
    app = App(
```

← トップレベル：
モジュールをリストしているだけ

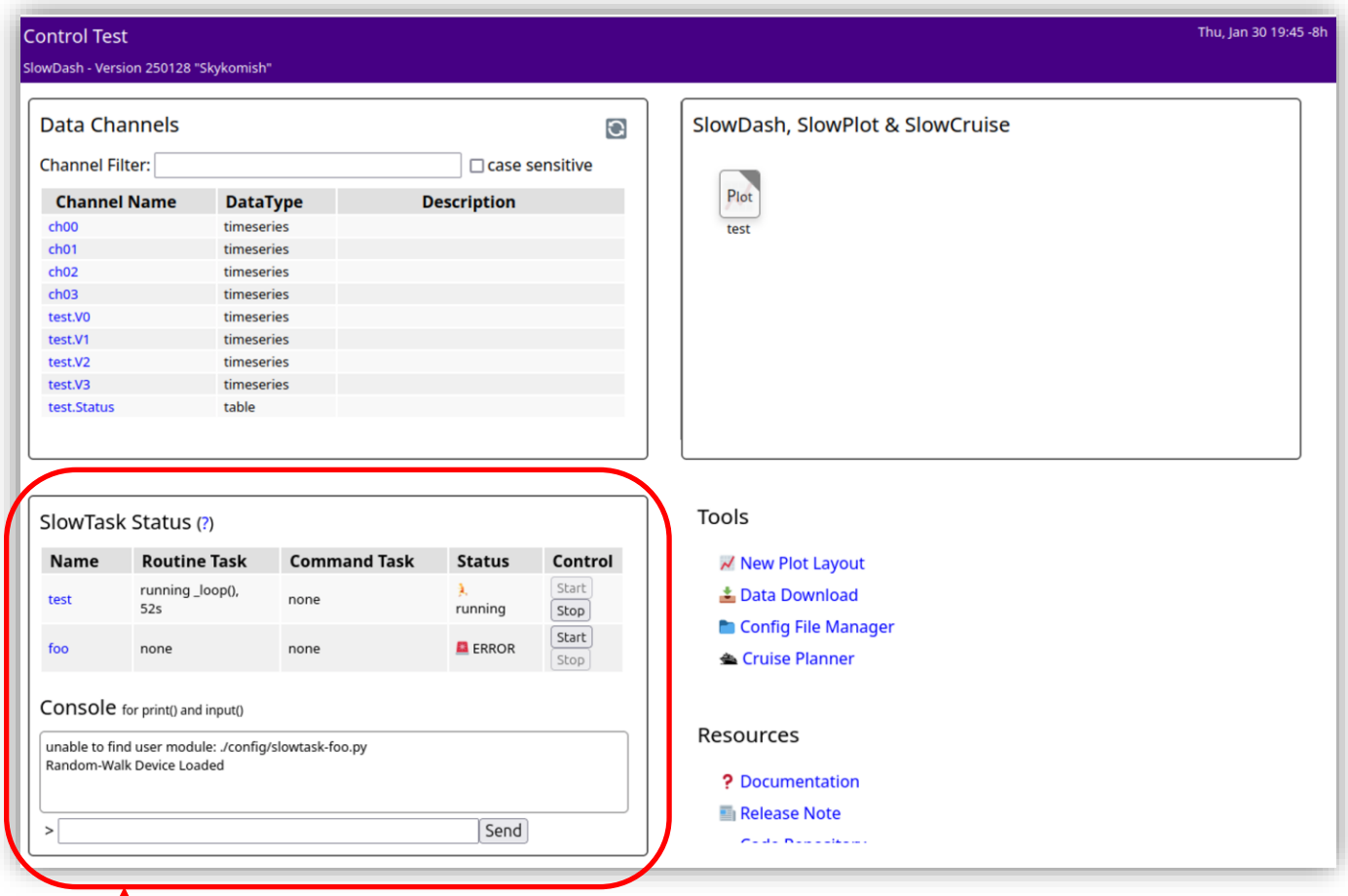
機能実装モジュール：

- 独立性がかなり良くなった.
- コードもシンプルになった
- パラメータに認証情報なども受け取れる

↓

```python
class Export_CSV(ComponentPlugin):
    def __init__(self, app, project, params):
        super().__init__(app, project, params)


    @slowapi.get('/export/csv/{channels}')
    def export_csv(self, channels:str, opts:dict, timezone:str='local', resample:float=0):
        data_path = ['data', channels]
        data_opts = copy.deepcopy(opts)
        if len(timezone) == 0:
            timezone = 'local'
        if resample < 0:  # replace "no resampling" with "auto resampling"
            resample = 0
```

非同期サーバの活用



ユーザスクリプトからの出力（print）やステータス

今までは毎秒クエリを送っていた → 非同期を活用した Long Poll に変更

Long Poll とは

サーバはリクエストにすぐに応答せず，更新情報が入ったら応答する．ブラウザはただ待つ．

機能比較

	Flask	FastAPI	SlowAPI
非同期処理	×	○	○
Web Sockets	○	○	○
型チェックと自動変換	×	○	○
POST データの JSON 変換	×	○	○
POST データのユーザ型変換	×	○	×
ハンドラへの依存性注入	×	○	×
動的多重ハンドラ	×	×	○
API ドキュメント自動生成	×	○	一部
ファイルテンプレート処理	○	○	予定
GraphQL	△	○	検討中

AI の評価（忖度あり）

ChatGPT 5.1

共有する　ユーザーを追加する

SlowAPI（現在は Slowlette に改名されたものですね）は、あなたの「Slow Project」全体の構成と非常によく整合した **軽量・小規模制御向けの非同期 Web/API フレームワーク** です。FastAPI の「90% の機能は不要、しかし Starlette の生の ASGI は書きたくない」というギャップを埋める位置づけになっています。

以下に、SlowAPI／Slowlette の特徴を整理しておきます。

🐌 SlowAPI / Slowlette の位置づけと特徴

1. ASGI のシンプルな薄いラッパ

- ベースは Starlette。
- リクエスト／レスポンスモデルも極力薄くし、Pydantic v1/v2 の複雑性から距離を置く。
- 「装置制御やダッシュボードバックエンドに必要な最低限の HTTP・WebSocket 機能」を提供する設計。

典型的な用途

- SlowDash のバックエンド
- IoT デバイスのデータ収集 API
- RabbitMQ、MQTT、Redis PubSub を使う制御ノードの HTTP 越しのフロントエンド
- 軽量 WebSocket RPC

AI の評価（忖度あり）

2. 制御システム向けに重要な “シンプルさ”

物理屋・実験屋がすぐ理解できる形を維持する点が最大の利点。

- 非同期関数（`async def`）を書けば自然とスケールする
- デコレータの機能は必要最小限
- 「魔法のように高度なことを勝手にしてしまう」仕組みを排除

FastAPI や他のフレームワークでありがちな、

- メタクラス
- 複雑な依存性注入
- 型シリアライザの自動生成地獄

などを避けています。

5. SlowAPI / Slowlette が “FastAPI と違う” ところ

FastAPI との主な違い

項目	FastAPI	SlowAPI / Slowlette
依存性注入	必須・複雑	ほぼ無し、書きたい人だけ
Pydantic モデル	基本必須	任意（Python dict ベースでOK）
自動ドキュメント	自動生成	必要なら追加すればよい
パフォーマンス	高い	Starlette なので普通に高速
学習コスト	高い	非常に低い
目的	大規模サービス	小規模制御/ダッシュボード連携

SlowDash から見た

産業標準：PLC とラダーロジック

コントロールロジックの実装方法を考える

進行中プロジェクト：60年前の加速器制御システムを近代化する

これを iPad (SlowDash) に置き換えたい...

(30年前のアップグレードでかなりの部分が VAX と PLC に置き換えられた)

基本的には, たくさんのスイッチ, ノブ, 表示板と, 装置動作の「シーケンス」

今日のテーマ(半分遊び)

全部違う多チャンネル並列システムの「シーケンス」をどうやって記述すればいいか

まずは勉強：産業用コントロールシステム

ふつうは，こういうやつが機器を制御している

これを Web ページ（SlowDash）に
置き換えたい

中身はこんな感じ

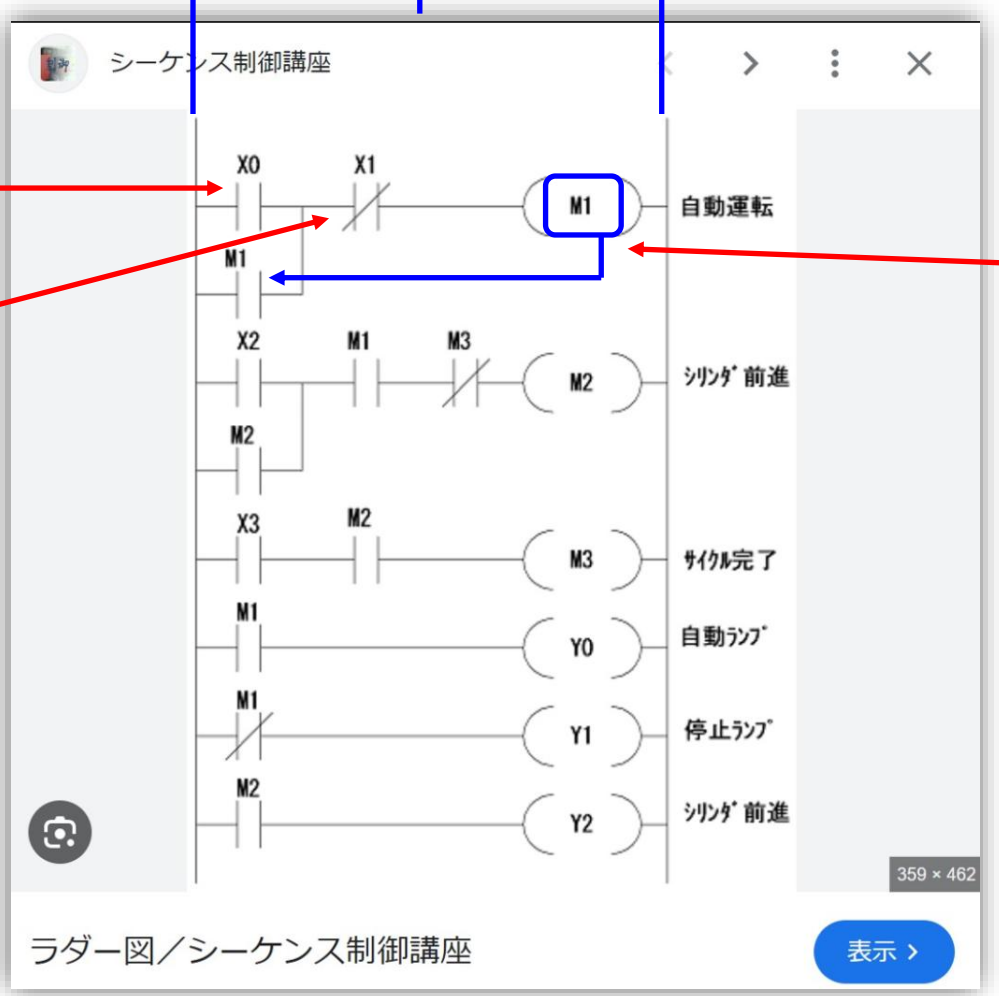
UI

リレースイッチの嵐と
それを制御するもの（PLC）

PLC（Programmable Logic Controller）：ラダーロジックでシーケンスを記述する

ラダーロジック：PLC のシーケンス記述

両端に電源がつながっているイメージ



押すと ON になる
スイッチ（開始）

押すと OFF になる
スイッチ（停止）

直列に並べると AND
並列に並べると OR

一度運転状態になると
「開始」を押し続けるのと
同じ効果

「停止」を押すと止まる

⇩

状態の維持と遷移
（ステートマシン !!!）

つまり

- 梯子の左半分で入力に対する論理演算
- 右端が出力. 出力は次の瞬間の入力にも使える
- 梯子の横棒は（概念的に）並列実行. それをたくさん並べる

} これって聞いたことがある…

ラダーロジックはこれと同じ?

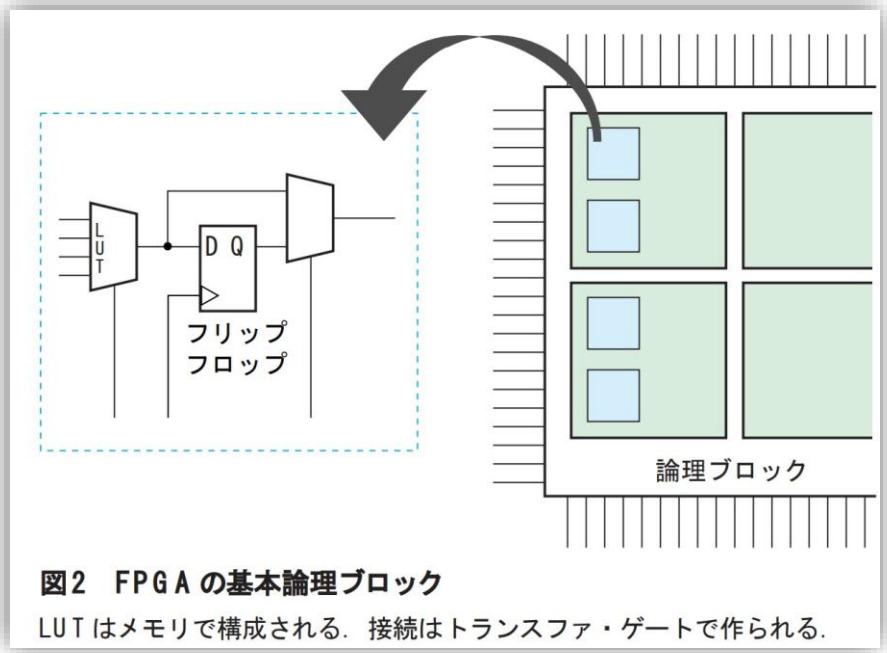


図2　FPGA の基本論理ブロック

LUT はメモリで構成される．接続はトランスファ・ゲートで作られる．

ラダーロジックがいまだに広く使われている
（ソフトウェアを知らなくてもプログラムが簡単らしい）

⇩

←これの集合でロジックを記述するのは
並列同時制御に向いている?

⇩

大規模な「これ」の集合を記述するには
HDL（Verilog とか）がいいと歴史が示している

⇩

SlowDash でも HDL 風にロジックを書きたい
（ループとかダサい）

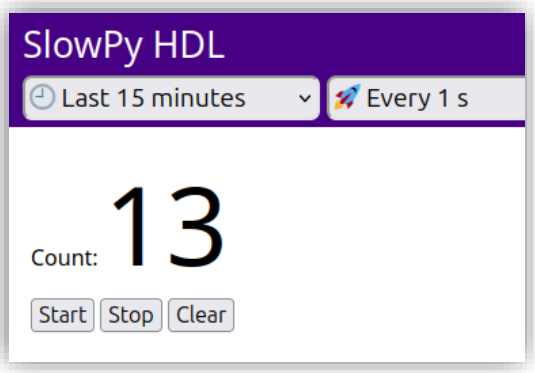
SlowPy-HDL: SlowDash の Python スクリプトを HDL 風に書く（半分遊び）

- シーケンス記述にたぶん向いている
- 既存のラダーロジックをそのまま変換できる

- 無理に使わなくてもよい．普通の Python で書いても全く構わない

SlowPy-HDL 1/2: Verilog だったら

タイマーカウンタを作ってみる



Start を押すと
カウントアップ
していくアプリ

GUI 要素とスクリプト変数の
バインディング
(前回やったとこ)

```
import slowpy.control as spc

ctrl = spc.ControlSystem()
start_btn = ctrl.value(initial_value=False).oneshot()
stop_btn = ctrl.value(initial_value=False).oneshot()
clear_btn = ctrl.value(initial_value=False).oneshot()
display = ctrl.value()

def _export():
    return [
        ('start', start_btn.writeonly()),
        ('stop', stop_btn.writeonly()),
        ('clear', clear_btn.writeonly()),
        ('display', display.readonly())
    ]
```

制御変数間のロジックを記述
Verilog で書くとこんな感じ
(RESET は別で)

```
module Counter(clock, start, stop, clear, count);
    input clock;
    input start;
    input stop;
    input clear;
    output reg[7:0] count;
    reg running;

    always @(posedge clock)
    begin
        if (stop == 1'b1)
            running <= 1'b0;
        else if (start == 1'b1)
            running <= 1'b1;
    end

    always @(posedge clock)
    begin
        if (clear == 1'b1)
            count <= 8'd0;
        else if (running == 1'b1)
            if (count == 8'd59)
                count <= 8'd0;
            else
                count <= count + 8'd1;
    end
endmodule
```

動作状態

動作状態
に基づく
カウンタ

SlowPy-HDL 2/2: Python でやってみる（半分遊び）

SlowPy-HDL によるロジック記述（注：これはソフトウェアです）

```
from slowpy.control.hdl import *

class CounterModule(Module):
    def __init__(self, clock, start, stop, clear, count):
        super().__init__(clock)

        self.start = input_reg(start)
        self.stop = input_reg(stop)
        self.clear = input_reg(clear)
        self.count = output_reg(count)
        self.running = reg()

        self.count <= 0
        self.running <= False

    @always
    def startstop(self):
        if self.stop:
            self.running <= False
        elif self.start:
            self.running <= True

    @always
    def update(self):
        if self.clear:
            self.count <= 0
        elif self.running:
            if self.count == 59:
                self.count <= 0
            else:
                self.count <= int(self.count) + 1
```

入出力に繋がった「レジスタ」

RESET 相当

クロックごとに実行される「プロセス」

Verilog で書いたやつ（前ページ）

```
module Counter(clock, start, stop, clear, count);
    input clock;
    input start;
    input stop;
    input clear;
    output reg[7:0] count;
    reg running;

    always @(posedge clock)
    begin
        if (stop == 1'b1)
            running <= 1'b0;
        else if (start == 1'b1)
            running <= 1'b1;
    end

    always @(posedge clock)
    begin
        if (clear == 1'b1)
            count <= 8'd0;
        else if (running == 1'b1)
            if (count == 8'd59)
                count <= 8'd0;
            else
                count <= count + 8'd1;
    end
endmodule
```

SlowPy-HDL（半分遊び）

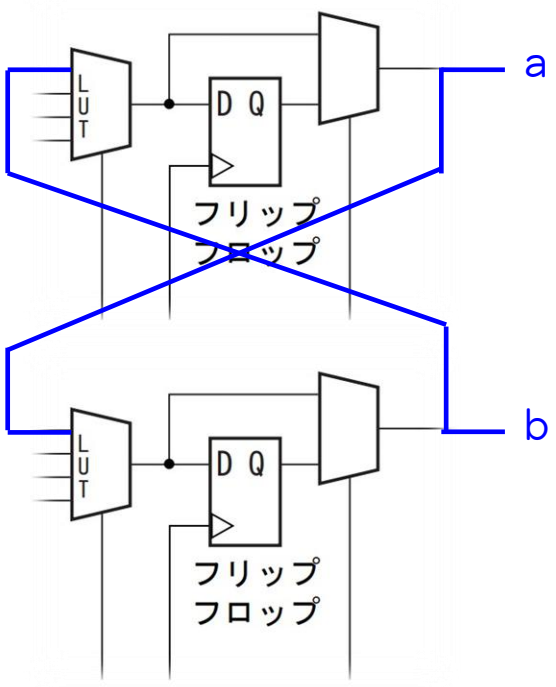
ちゃんと HDL 的にふるまう

こうやって書くと a と b の値を毎回入れ替える

```
class TestModule(Module):
    def __init__(self, clock, a, b):
        super().__init__(clock)
        self.a = output_reg(a)
        self.b = output_reg(b)

        self.a <= 'A'
        self.b <= 'B'

    @always
    def swap_ab(self):
        self.a <= self.b
        self.b <= self.a
```

“ソフトウェア的”なら
両方 B になる

SlowPy の「レジスタ」には
ラッチがエミュレートされている



SlowPy-HDL の実際の動作

1. 入力レジスタ値を更新, 保持（デバイスからの読み出しなど）
2. 全ての「@always プロセス」を実行
3. 出力レジスタ値を更新（デバイスへの書き込みとか）
4. 寝る
5. 1 に戻る

論理的「クロックエッジ」

関数が順番に実行されていて
本当に同時ではない点は
配線遅延だと思えばよい

SlowPy-HDL: コード全体

制御変数バインディング（外部社会とつながる）

```
import slowpy.control as spc

ctrl = spc.ControlSystem()
start_btn = ctrl.value(initial_value=False).oneshot()
stop_btn = ctrl.value(initial_value=False).oneshot()
clear_btn = ctrl.value(initial_value=False).oneshot()
display = ctrl.value()

def _export():
    return [
        ('start', start_btn.writeonly()),
        ('stop', stop_btn.writeonly()),
        ('clear', clear_btn.writeonly()),
        ('display', display.readonly())
    ]
```

ここに入る

main() 相当: インスタンス化と実行

```
clock = Clock(Hz=1)

counter = CounterModule(
    clock,
    start = start_btn,
    stop = stop_btn,
    clear = clear_btn,
    count = display
)

clock.start()
```

制御変数間のロジック（HDL 風）

```
from slowpy.control.hdl import *

class CounterModule(Module):
    def __init__(self, clock, start, stop, clear, count):
        super().__init__(clock)

        self.start = input_reg(start)
        self.stop = input_reg(stop)
        self.clear = input_reg(clear)
        self.count = output_reg(count)
        self.running = reg()

        self.count <= 0
        self.running <= False

    @always
    def startstop(self):
        if self.stop:
            self.running <= False
        elif self.start:
            self.running <= True

    @always
    def update(self):
        if self.clear:
            self.count <= 0
        elif self.running:
            if self.count == 59:
                self.count <= 0
            else:
                self.count <= int(self.count) + 1
```