End-to-end differentiable digital twin for the IOTA/FAST facility

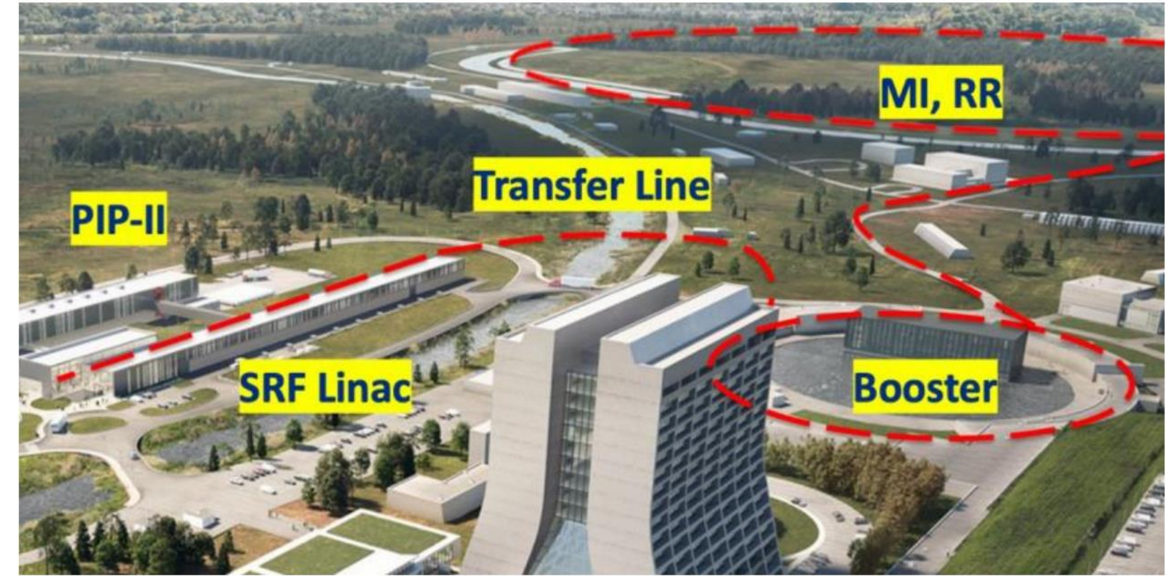
Nikita Kuklev, Nilanjan Banerjee, Michael Wallbank,
Jonathan Jarvis, Alexander Romanov, Dean Edstrom (FNAL)
John Wieland (MSU)

MaLAPA 2025

FNAL upgrades for LBNF/DUNE

Near future (~2030): **PIP-II**

- A new SRF linac of 800 MeV
- A new Beam Transfer Line
- Accelerator upgrades in the Booster, the Recycler Ring (RR) and Main Injector (MI)
- **1.2 MW on-target power**

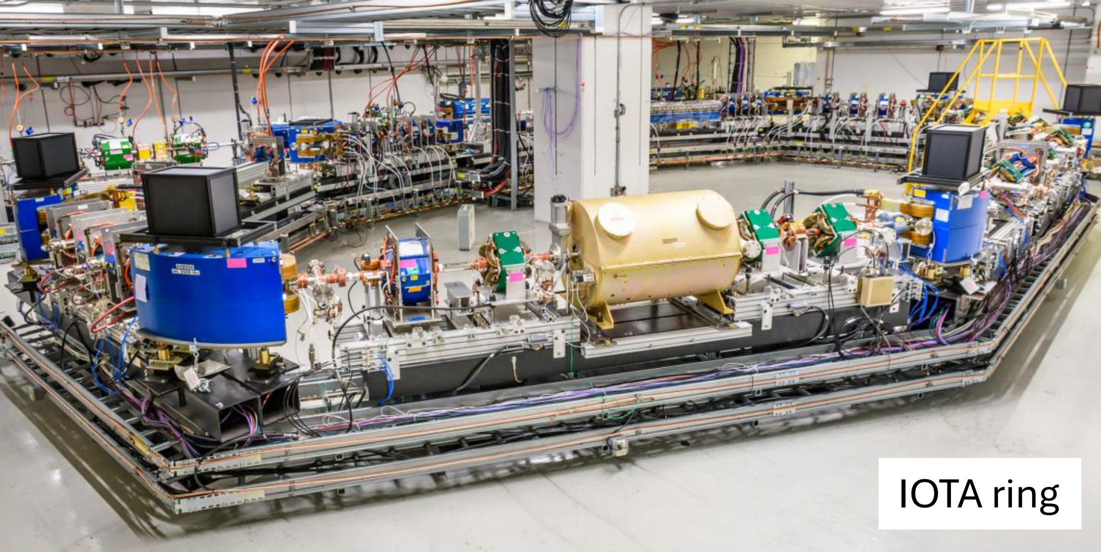


IOTA/FAST goal - develop useful ML methods for PIP-II commissioning and operation: optimization, simulation, and **digital twins**

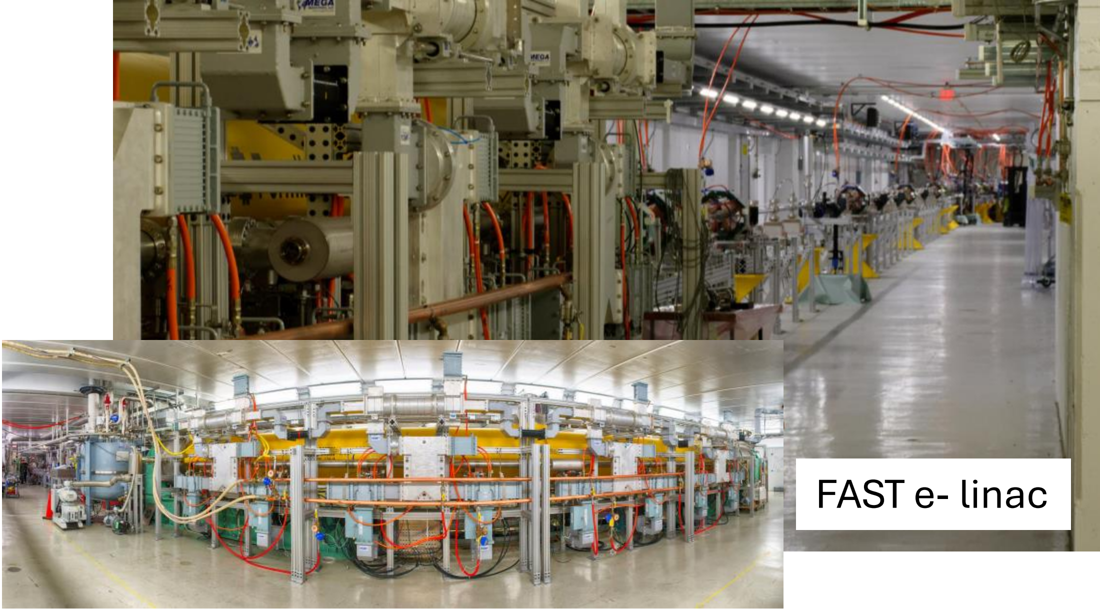
IOTA/FAST facility

A small research facility for demonstrating future accelerator designs and technologies

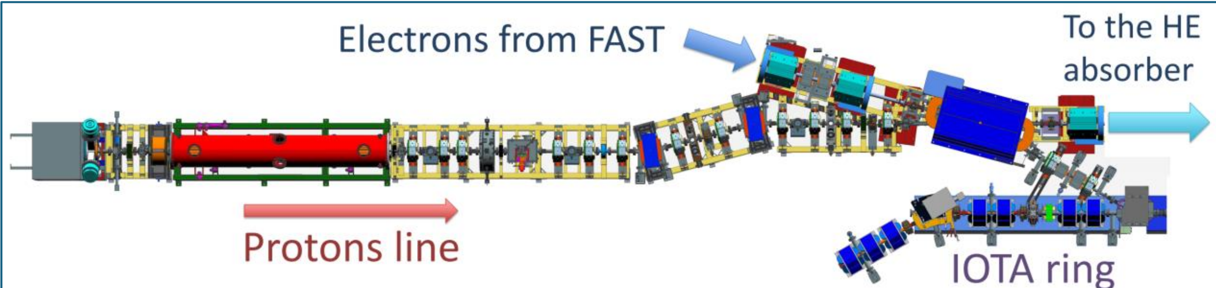
- Protons, electrons, SRF, normal RF, undulators, etc. - good testbed for ML applications



G. Stancari / Fermilab



Proton injector
(summer 2025 commissioning)



Long term goals for a “digital twin”

Use cases:

- Reference lattice implementation / source of truth
- Backend to debug ML and standard control applications
- Fast physical predictions (vs full simulation) with known uncertainty and gradient information
- Online model calibration to experimental data; beam/parameter inference

Unified lattice description and modelling synthesis

- A common physical accelerator description
- Synthesize down to specific codes

Device and control system interface (‘middle layer’)

- Use a common accelerator description and
- Synthesize down to specific codes

ML integration

- Surrogate model training/substitution
- Differentiable modelling and gradient export
- Model calibration and inference

Unified modelling

- Facility is reconfigured rapidly for various experiments
  - Undulators, nonlinear lenses, extra diagnostics, BxB feedback, ...
- A zoo of codes used by different projects
  - Even ‘official’ lattice ports often don’t match
  - Applying calibration/LOCO corrections hard
- It is **maintenance hell**
- Recent progress in the community:
  - OpenPMD for phase space
  - Xsuite (MAD-compatible) as common toolkit
  - PALS (lattice description standard, WIP)

Elegant – HPC DA/MA, OSC, general

PyOrbit – SC, nonlinear dynamics

MAD-X – lattice design

Synergia – SC, general

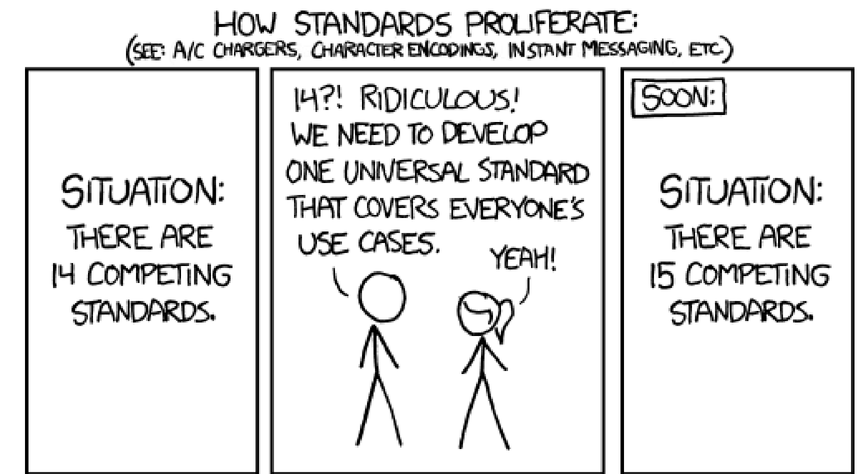
ImpactX – SC, general

Lifetrac – FMA

TraceWin – LEBT, RFQ

6Dsim - LOCO

Unified modelling



- Our approach: a custom shared **physical** description format loosely based on MAD-X/Xsuite

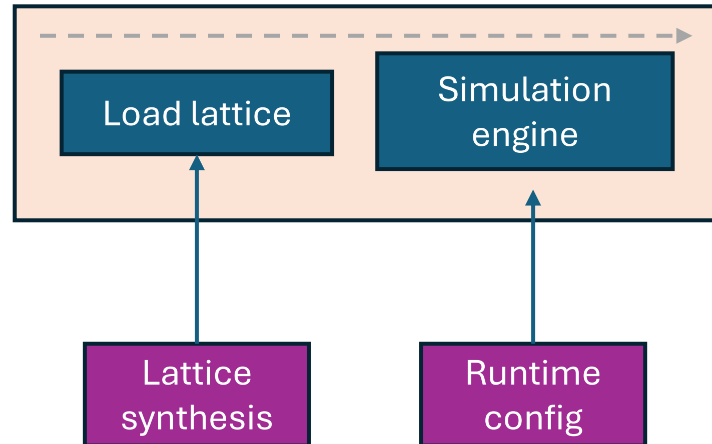
- Describes the hardware, not modelling
  - "rectangular magnet with gradient <k> / field map <f> at position <x/y/z>"

- Uses typical objects (elements, lines, and sequences)
  - Maintain compatibility as much as possible
  - If/when lattice standard gets done, hope to switch

Fermilab

Unified modelling

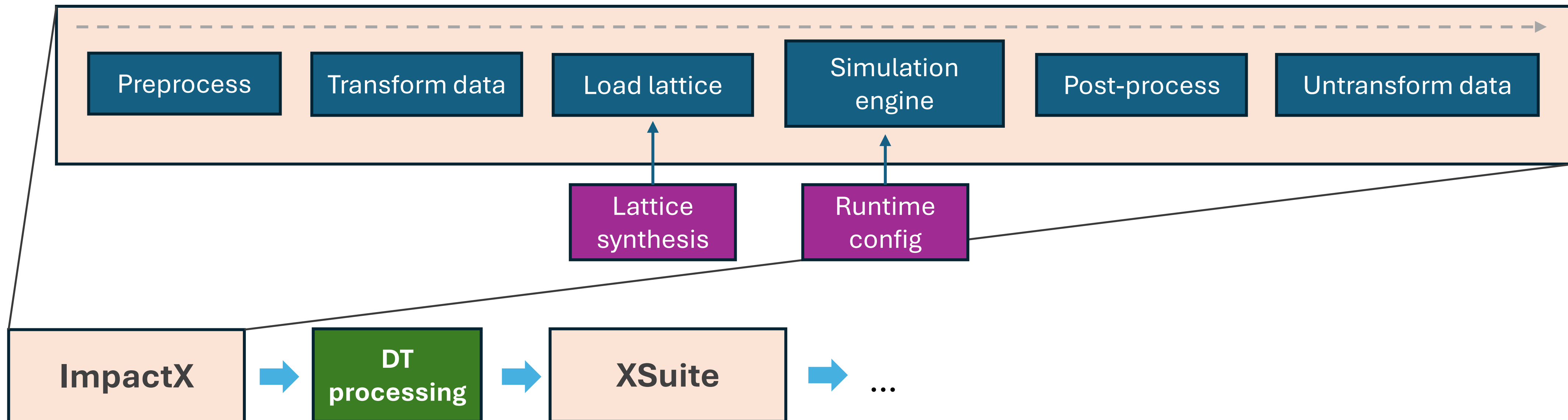
- A set of ‘synthesis’ produce final lattices and ‘engine’ modules run them
  - **No claim of universal support** – we only implement what we use (~all common magnet types + some exotic)



Fermilab

Simulation coupling

- Need to extract beam and device parameters
- Engines are responsible for converting from/into a set of standardized containers
  - Phase space (openPMD-ish)
  - Envelope (covariance matrix)
  - Twiss functions
  - Named scalar/vector parameters



Fermilab

Sim code selection

Criteria:

- Because of proton/electron species mix, require space-charge modelling
- Easy to add more elements
- Python bindings or native Python

Three codes met criteria:

BLAST-ImpactX / **impactx**

- C++ with Python bindings
- Most advanced SC solvers
- Lacks element physics (WIP)
- Can hook arbitrary Python functions for tracking

- Python, generates C kernels
- A bit less flexible for custom elements
- Easy transition from MAD-X
- Good lattice/distribution helper routines

ocelot-collab / **ocelot**

- Almost pure Python
- No advanced physics models
- Good for reference implementations

Core lattice implementation

- Implementation uses nested Pydantic objects – **fully serializable**
- Tried to pick sensible parameters from popular codes

```
class Bend(Element):
    angle: float = Field(0.0, description="angle of bend in [rad]")
    k1: float = Field(0.0, description="strength of quadrupole lens in [1/m^2]")
    k2: float = Field(0.0, description="strength of sextupole lens in [1/m^3]")
    e1: float = Field(0.0, description="the angle of inclination of the entrance face [rad]")
    e2: float = Field(0.0, description="the angle of inclination of the exit face [rad]")
    fint: NonNegativeFloat = Field(0.0, description="fringe field integral")
    fintx: Optional[NonNegativeFloat] = Field(
        None, description="allows (fintx > 0) to set fint at the element exit different from its entry value."
    )
    gap: NonNegativeFloat = Field(0.0, description="the magnet gap [m], NOTE in MAD and ELEGANT: HGAP = gap/2")
    h_pole1: float = Field(0.0, description="the curvature (1/r) of the entrance face")
    h_pole2: float = Field(0.0, description="the curvature (1/r) of the exit face")

    element_name: str = Field("Bend", frozen=True)

    @model_validator(mode="after")
    def validate_fint(self):
        if self.fintx is None:
            self.fintx = self.fint
        return self
```


```
{'id': '_ID_37566583_',
 'l': 0.3919,
 'tilt': 0.0,
 'dtilt': 0.0,
 'dx': 0.0,
 'dy': 0.0,
 'element_name': 'SBend',
 'angle': -0.5234923105139273,
 'k1': 0.0,
 'k2': 0.0,
 'e1': 0.0,
 'e2': 0.0,
 'fint': 0.0,
 'fintx': 0.0,
 'gap': 0.0,
 'h_pole1': 0.0,
 'h_pole2': 0.0,
 's_start': 10.829180200786354,
 's_mid': 11.025130200786354,
 's_end': 11.221080200786353},
```


Pydantic json dump

- Advanced features (i.e. unique fringe field models) supported as extra attributes per-code but not generalized

Example of a custom element

For digital twin performance and compatibility, need to add custom elements

Example: ImpactX memory-only BPM to avoid file I/O

```
class ImpactXWatchpoint(elements.Programmable):
    # std::function<void(ImpactXParticleContainer *, int, int)> m_push; //! hook for push of whole container (pc, step, period)
    def __init__(
        self,
        name=None,
        mode: Literal["coordinate", "parameter", "centroid"] = "coordinate",
        buffer_len: int = None,
        interval: int = 1,
    ):
        """
        Custom in-memory watchpoint element for ImpactX to store full or statistics of the particle container.
        """
        if name is None:
            name = f"wp_{get_random_name()}"
        self.mode = mode
        self.buffer_len = buffer_len
        self.interval = interval
        super().__init__(name=name, nslice=1, ds=0.0)
        self.push = self.pusher
        self.buffer = collections.deque(maxlen=self.buffer_len)
        self.enforce_unique_period = True
        self.last_period = -1

    def pusher(self, pc: ImpactXParticleContainer, step: int, period: int):
        # Push all particles relative to the reference particle
```


Fermilab

Lattice ‘synthesis’

Configuration + exporter for each code. Some are very simple, others complex (dipole edges, units, etc.).

```
1  lattice.sequence
```

```
[Drift(id='_ID_37089398_', l=0.15, tilt=0.0, dtilt=0.0, dx=0.0, dy=0.0, element_name='Drift', s_start=0.0, s_mid=0.075, s_end=0.15),
 Drift(id='_ID_81804225_', l=0.06, tilt=0.0, dtilt=0.0, dx=0.0, dy=0.0, element_name='Drift', s_start=0.15, s_mid=0.18, s_end=0.21),
```

```
1  lattice_config = ImpactXExporterConfig(n_slices=1, default_monitor_type="custom")
2  exporter = ImpactXExporter(config=lattice_config)
3  impactx_lattice = exporter.write_lattice(lattice)
4  impactx_lattice.sequence
```

```
[<impactx.elements.Drift, name=_ID_37089398_, ds=0.150000>,
 <impactx.elements.Drift, name=_ID_81804225_, ds=0.060000>,
```

Self-contained 'simulation' objects

To make a simulation pipeline, need building blocks

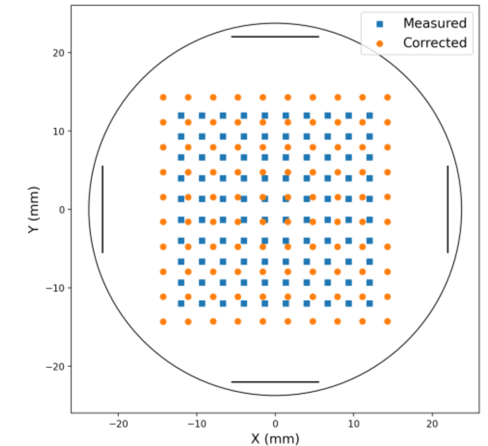
- Per-code container model
- Declarative mandatory inputs/outputs
  - Generic and special types (envelope, phase space, etc.)
  - Checked for type, coordinate system, etc.
- Shared state passed between simulations
  - i.e. global fidelity parameter

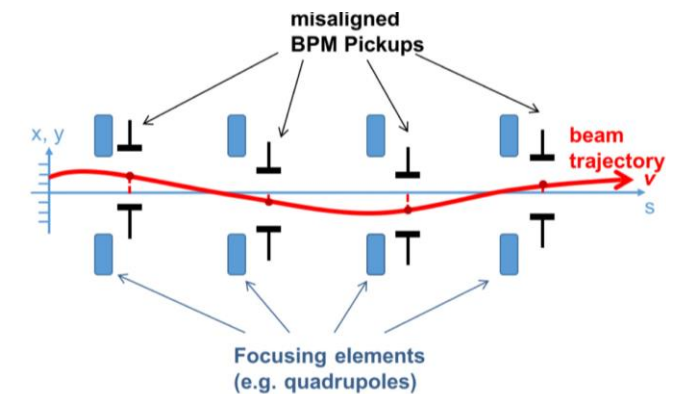
```python
1  sim = ImpactXTrackingModel(name='ipi_mebt_impactx', config_lattice=lattice_config, lattice=lattice, rp=rp)
2  sim.setup()
3  print(sim.model_dump_json(indent=2))
```


```
{
  "name": "ipi_mebt_impactx",
  "description": "",
  "version": 1,
  "extra_inputs_allowed": true,
  "extra_inputs_behavior": "passthrough",
  "inputs": {
    "fidelity": {
      "name": "fidelity",
      "type": "scalar",
      "dtype": "float64",
      "processors": [
        {
          "offset": 0.0,
          "multiply": 0.01,
          "name": "processor_offset_multiply"
        }
      ]
    }
  },
  "outputs": {
    "phase_space": {
      "name": "phase_space",
      "type": "coordinates",
      "dtype": "float64",
      "processors": []
    },
    "reference_particle": {
      "name": "reference_particle",
      "type": "reference_particle",
      "dtype": "float64",
      "processors": []
    },
    "walltime": {
      "name": "walltime",
      "type": "scalar",
      "dtype": "float64",
      "processors": []
    }
  },
  "config_runtime": {
    "space_charge": false,
    "particle_shape": 2,
    "diagnostics": false,
    "slice_step_diagnostics": false
  },
  "config_lattice": {
    "n_slices": 1,
    "default_monitor_type": "custom"
  },
```

Linking simulation with devices

- Complex device modelling is typically not part of simulation, requires bidirectional control:
  - Simulation outputs = device outputs
  - Device setpoints = simulation parameters
  - Real readbacks = device setpoints
- Many existing examples: middle layers, commissioning toolkits (APS, LBNL)
- Device example: BPM
  - Common effects: offset/gain/pincushion distortion/drift
  - More complex issues: fill pattern dependence (APS-U)
  - Several data streams (BxB, TBT, 10Hz orbit, 1Hz orbit, single pass)



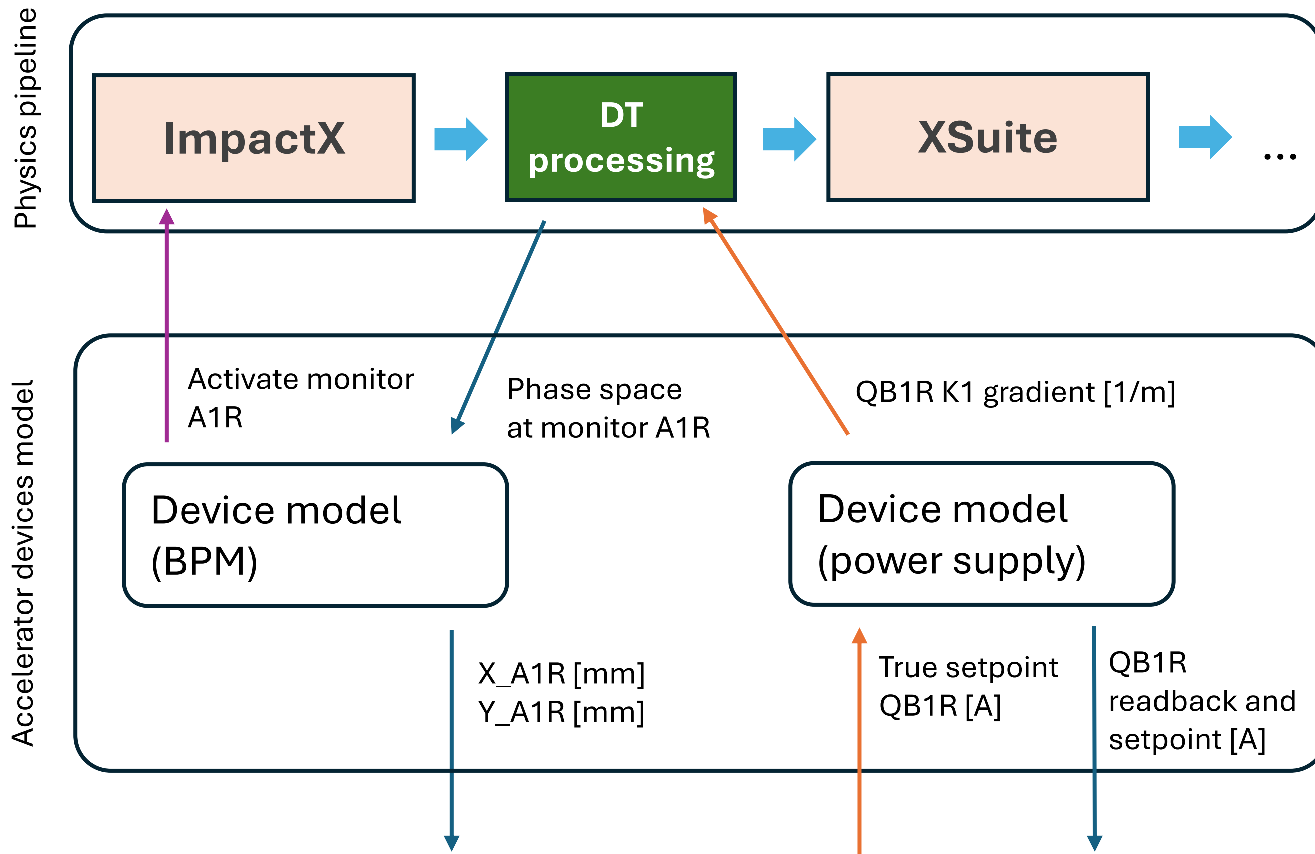


https://doi.org/10.1016/j.nima.2022.167898

Fermilab

Linking simulation with devices

- Device models are bound to a simulation segment and receive data after each pipeline stage



Fermilab

Linking simulation with devices

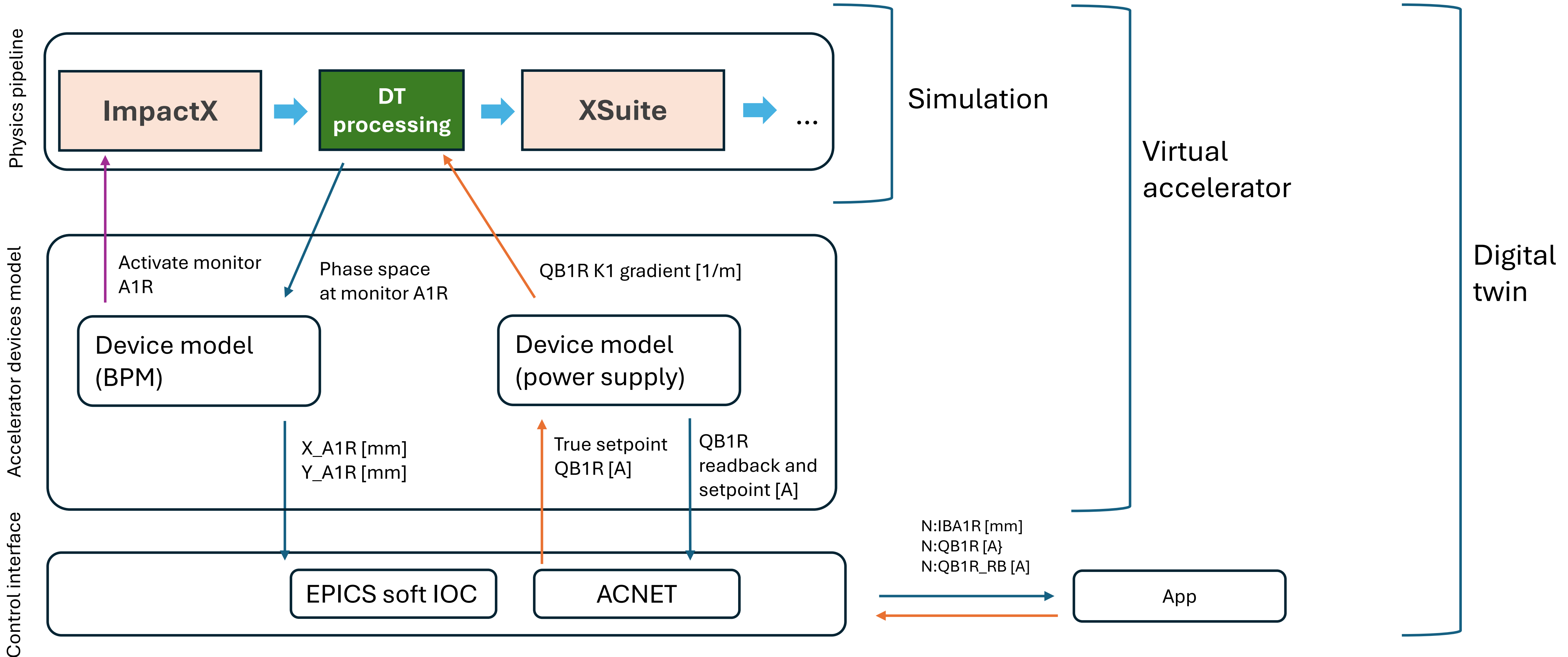
- So far have a couple generic models with noise/delay/etc., TBD what other features will be necessary
- Devices are also Pydantic models, as is the whole device layer

```
class GenericValueModel(VirtualDevice):
    value: float = None
    raw_value: float = None
    setpoint: float = None
    last_setpoint: float = None
    low: float = None
    high: float = None
    noise: NonNegativeFloat = None
    resolution: NonNegativeFloat = None
    model: Literal["instant", "exponential", "underdamped"] = "instant"

    pmodel_kwargs: dict = {}
    setpoint_update_rate: float = None
    readback_update_rate: float = None
    event_threshold: float = None

    time_last_update: float = None
    time_last_write: float = None
    time_last_read: float = None
    time_last_setpoint_event: float = None
    time_last_readback_event: float = None
    last_known_t: float = None
    seed: int = 42

    _rng: RandomState = None
    _time_last_call: float = None
```


Fermilab

Final layer – the control system

- Fermilab is unique in having to support two control systems. Not fun....



Fermilab

Final layer – the control system

- Use standard EPICS libraries (caproto, pcaspy) for soft IOC
  - Support monitoring, scan records, etc.
  - Advanced features like wildcard EPICS channels for debugging
- For ACNET, hard to replicate because it is a centralized system
  - Custom package ‘pacnet’ for Python control
  - Can intercept and send data to digital twin instead
  - Other languages not yet supported
- Long term, want full EPICS transition and/or use tools of ACORN modernization project

Object-based API

```
1  dpm = DPM(default_role='booster_studies')
2  toroid = DoubleDevice('B:TRMH1D.COMMON@E,1D,E,10')
✓ 0.0s
```

Full DRF2 parser

```
1  print(toroid.drf2.pretty_print())
✓ 0.0s

DiscreteRequest[B:TRMH1D.COMMON@E,1D,E,10]
 [self.device='B:TRMH1D']
 [self.property=<DRF_PROPERTY.READING: ':'>]
 [self.range=None]
 [self.field=<DRF_FIELD.COMMON: 5>]
 [self.event=<DRF_EVENT mode E: (E,1D,E,10)>]
 [self.extra=None]
```

Thread-based I/O, notebook friendly

```
1  toroid.read(adapter=dpm)
✓ 0.2s

5.245429365371785
```


Fermilab

Consistent event timing and ordering

- Typically, digital twins are one-shot
  - 1 simulation = 1 read-out
  - Does not match reality – we have 100Hz+ devices alongside <1Hz ones

- Time-based simulation tools are common in engineering (i.e. Simulink, LTspice, Qspice)
  - Because of ODEs/PDEs, they are using symplectic integrators too!

- To properly model real data streams, only need a small subset - a **timestep-based event loop**
  - Every device declares what next time step it wants
  - Simulation advances smallest step, and devices are asked for events.
  - Ordering is consistent and customizable (this can be a performance limitation)

```python
class OneHzTimer(VirtualDevice):
    def __init__(self, **kwargs):
        """ A test device that always returns 1s-quantized unix timestamp with 1Hz scan rate"""
        super().__init__(**kwargs)
        self._t_last = time.time()

    def query_next_event(self) -> float:
        """When the next possible update of this device will be required"""
        return self._t_last + 1.0
```

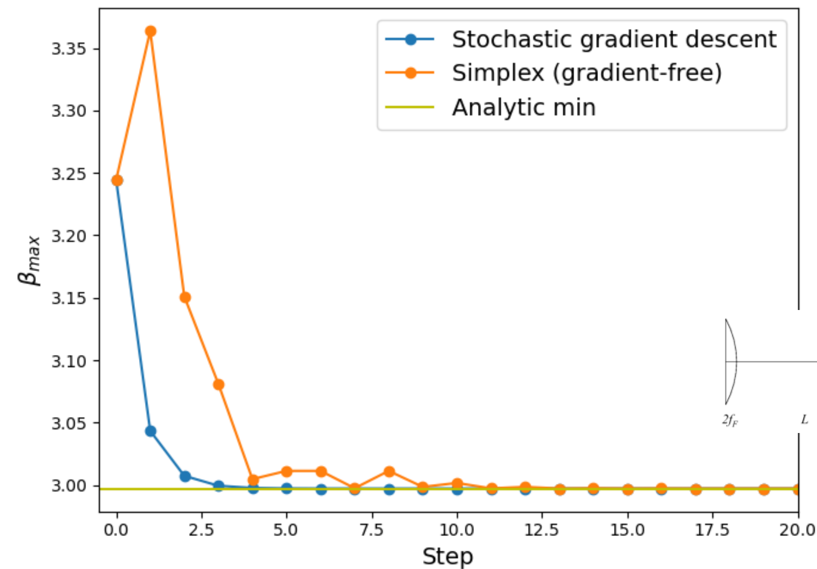

Fermilab

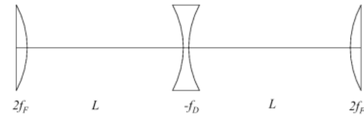
Differentiable modelling

Why differentiable?

Several talks already demonstrated that gradients are **very useful**

- BO priors
- Parameter inference
- Beam distribution prediction
- Most 'smooth' optimizations of the lattice

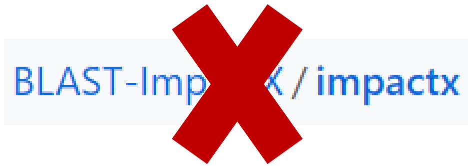




Use gradient information to quickly match lattice

Fermilab

Differentiable codes



Because of use cases in RL and other training, performance **is critical**

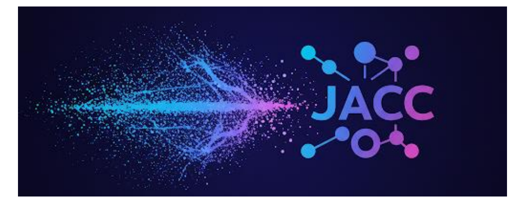
Two years back, started working on differentiable tracking via JAX (Google)

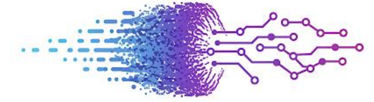
- (way) Harder to use than PyTorch
- Composable JIT compilation and vectorization
- In general, several times faster but not as much as it used to be
- Standard linear + symplectic order 2/4 drift-kick element set (~= Xsuite)

JACC (**J**ax for **ACC**elerators)

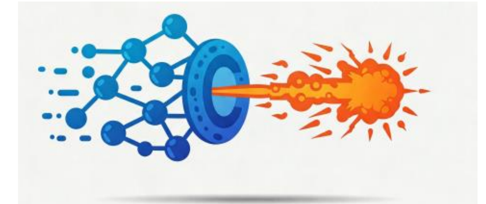
We are working on a benchmark for diff codes, stay tuned.

Logos courtesy of Google Gemini





JACC



Differentiable models are a cheat code

For RL, differentiable model converts model-free to model-based -> easier to train

- (or even supervised learning, if full reward gradient is known)

Even if using only as RL engine, several highly tuned, 'fully compiled' RL algorithms are available

PureJaxRL (End-to-End RL Training in Pure Jax)

PureJaxRL is a high-performance, end-to-end Jax Reinforcement Learning (RL) implementation. When running many agents in parallel on GPUs, our implementation is over 1000x faster than standard PyTorch RL implementations.

Stable-Baselines Jax (SBX)

Stable Baselines Jax (SBX) is a proof of concept version of Stable-Baselines3 in Jax, with recent algorithms like DroQ or CrossQ.

It provides a minimal number of features compared to SB3 but can be much faster (up to 20x times!): https://twitter.com/araffin2/status/1590714558628253698

(this note is in SB3 repo)

Equinox library provides PyTorch-ish interface

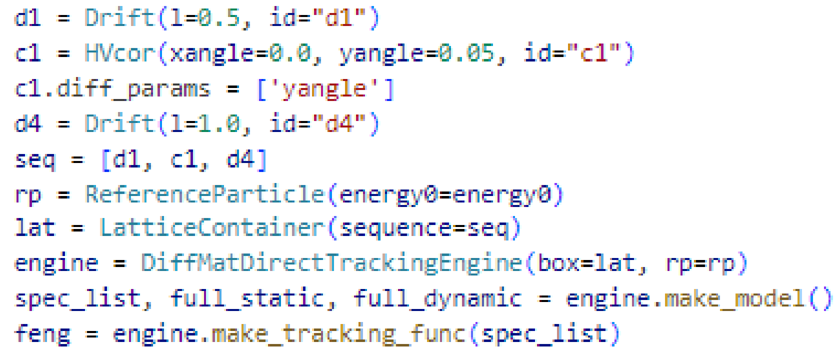
Equinox is your one-stop JAX library, for everything you need that isn't already in core JAX:

- neural networks (or more generally any model), with easy-to-use PyTorch-like syntax;
- filtered APIs for transformations;
- useful PyTree manipulation routines;
- advanced features like runtime errors;

Fermilab

Integration of differentiable codes with standard digital twin

Flagged user parameters are automatically configured



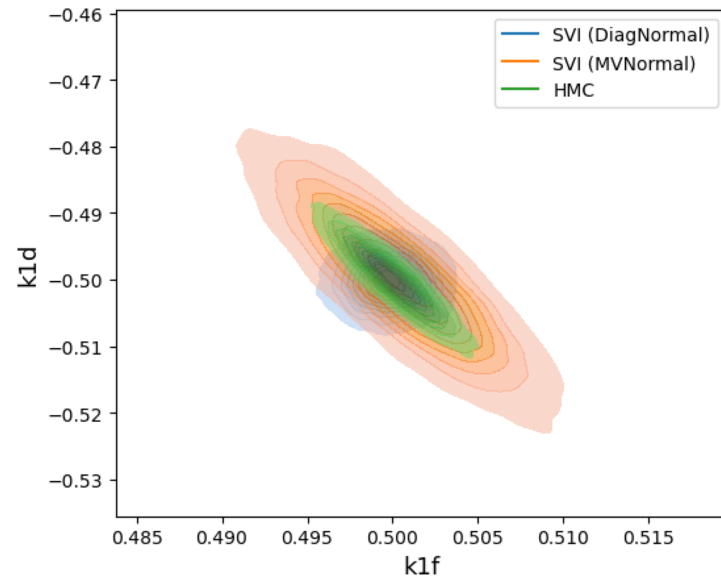
- All standard interfaces are supported – can swap in for a standard simulation
- If scalar loss function defined, gradients exported same as regular outputs

Still not particularly user-friendly for vector diff (Jvp, vJp)

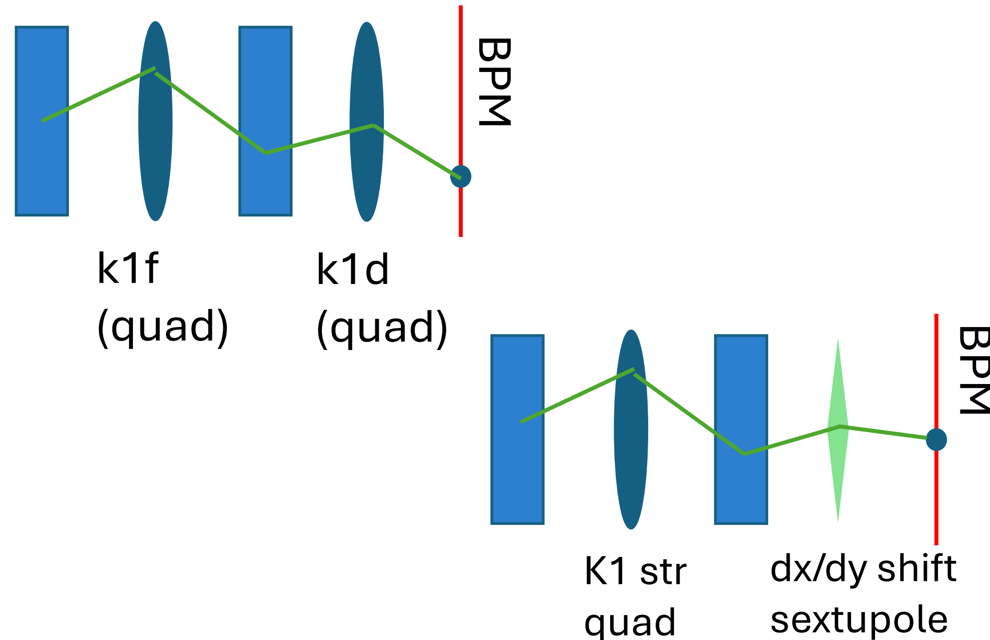
Differentiable models are a cheat code

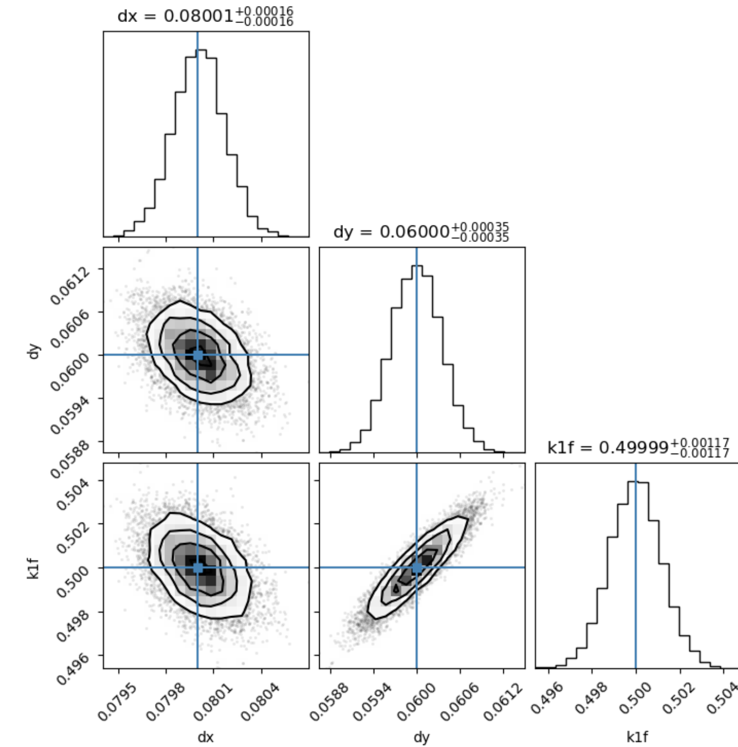
Initial focus with JACC was model calibration/parameter inference

- Fast enough to use ‘stats-based’ methods like Hamiltonian Monte Carlo (HMC)
- Current work:
  - Optics measurements (‘fast LOCO’)
  - Phase space reconstruction with nonlinear elements / poor diagnostics
  - Fast-enough ring tracking (how to slice, forward mode diff, etc.)



Expose correlations in uncertainty



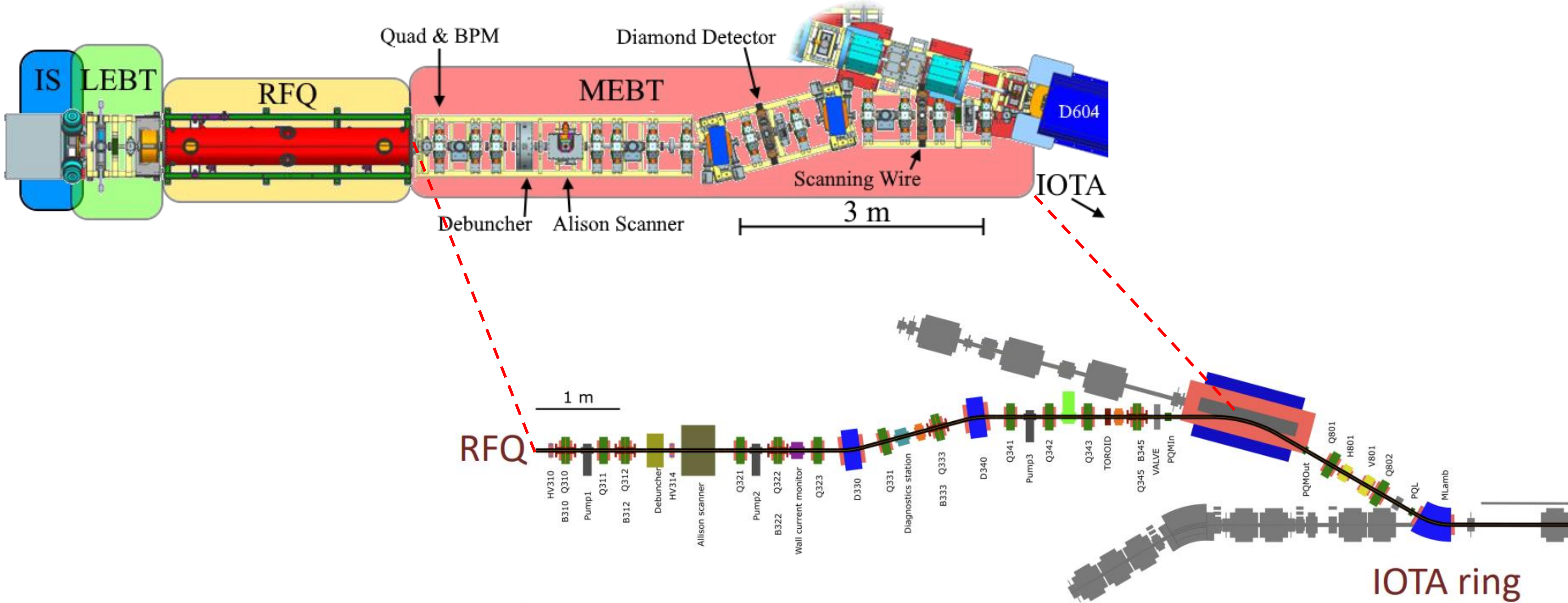


Fermilab

First application: IOTA/FAST IPI

Proton injection commissioning summer/fall 2025

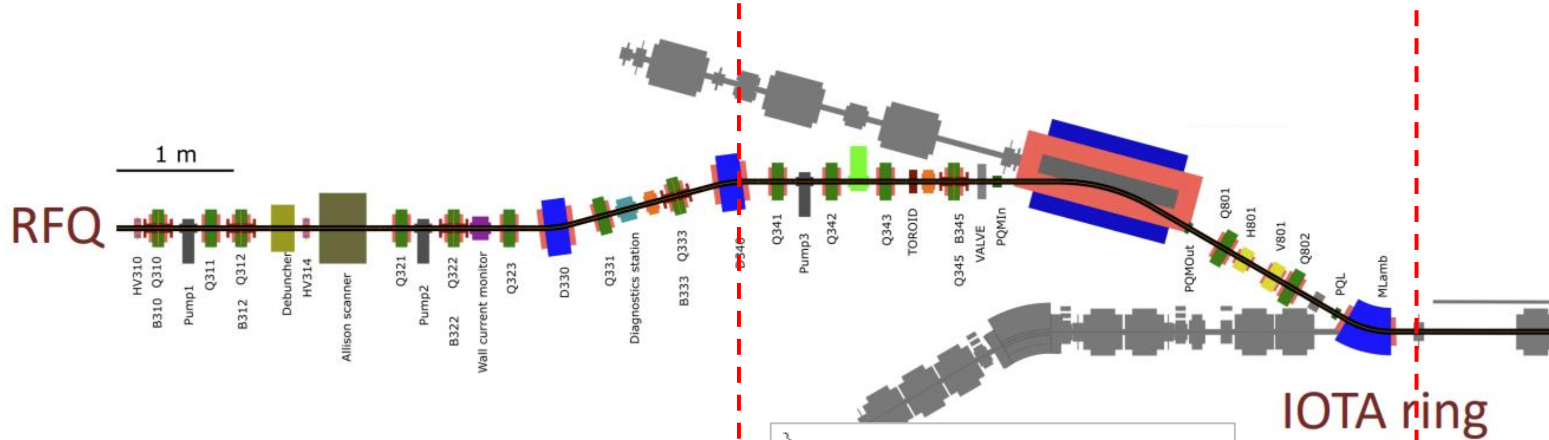
- Extreme space charge, lost of advanced diagnostics – good test of digital twins



Fermilab

First application: IOTA/FAST IPI

Demonstrated basic ‘pipeline functionality’ with classic, diff, and NN sections




```
{
  "models": {
    "0": {
      "name": "impactx",
      "description": "",
      "version": 1,
      "extra_inputs_allowed": true,
      "extra_inputs_behavior": "passthrough",
      "inputs": {},
      "outputs": {
        "phase_space": {
          "name": "phase_space",
          "type": "coordinates",
          "dtype": "float64",
          "processors": []
        },
```

```
},
"1": {
  "name": "jax",
  "description": "",
  "version": 1,
  "extra_inputs_allowed": true,
  "extra_inputs_behavior": "passthrough",
  "inputs": {},
  "outputs": {},
  "config_runtime": {
    "diff_mode": "reverse",
    "use_jit": true
  },
  "rp": {
    "charge": 1,
    "mass0": 938272089.4300001,
    "p0c": 68539116.18302412,
    "kin_energy0": 2500000.0,
    "energy0": 940772089.4300001,
    "gamma0": 1.002664472308367,
    "beta0": 0.07285411307700568
  }
}
```

```
"2": {
  "name": "eqx_jax_linear",
  "description": "",
  "version": 1,
  "extra_inputs_allowed": true,
  "extra_inputs_behavior": "passthrough",
  "inputs": {},
  "outputs": {},
  "config_runtime": {
    "diff_mode": "reverse",
    "use_jit": true,
    "eqx_pytree_file_path": "./models/eqx_pytree.eqx"
  },
```


Fermilab

Conclusions

We envision an ecosystem with **modular digital twin architecture** mixing conventional, differentiable, as well as (NN/GP/other) surrogate models.

Presented our attempt at implementation:

- Fully serializable and validated config through Pydantic
- Bidirectional data flows at physics model, device model, and control systems levels
- Natively supports Xsuite, ImpactX, and JAX for differentiable and surrogate modules

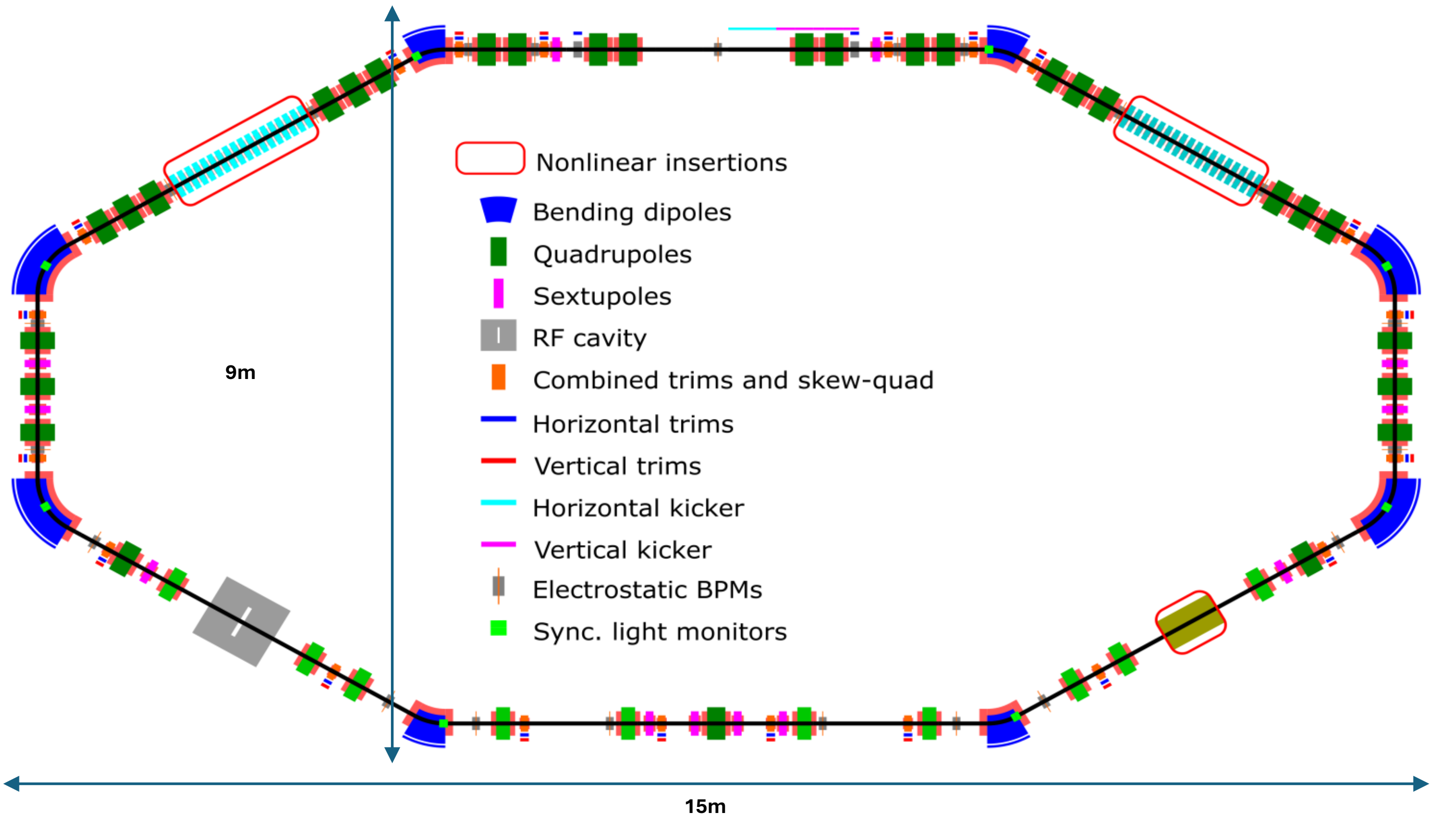
Accelerator Pipeline for Optimization and Low-Latency Operation
(APOLLO project)

Future work:

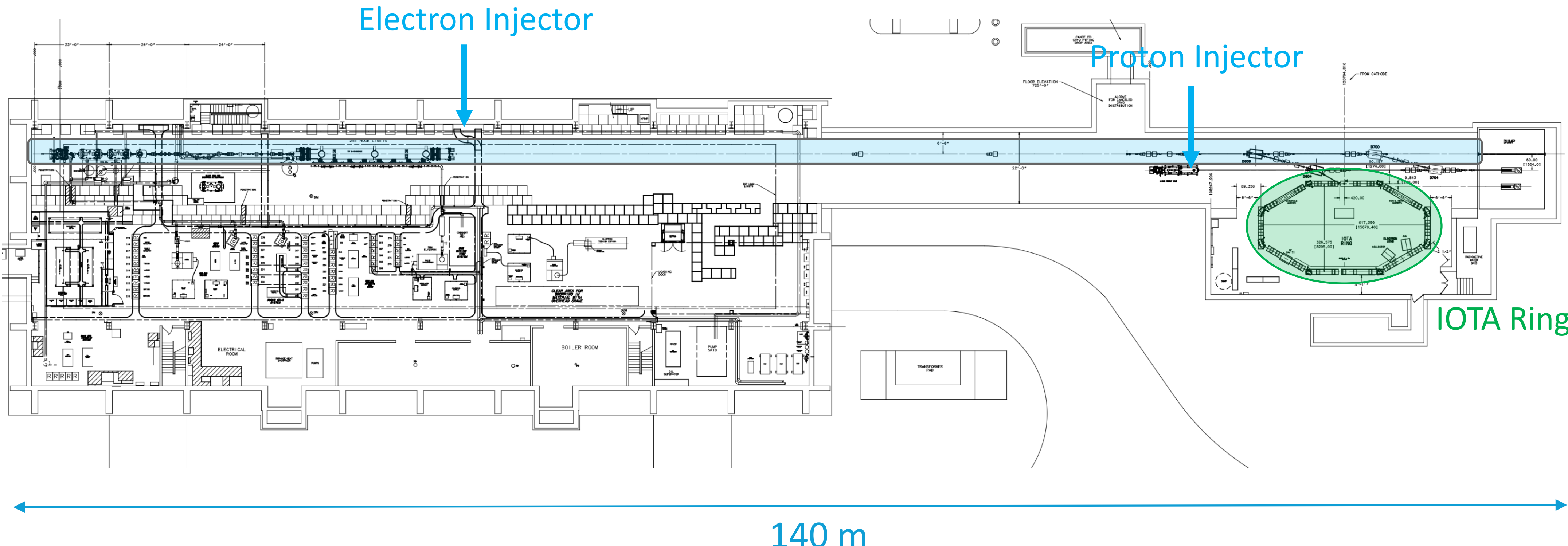
- Contribute elements to simulation codes (ImpactX) for better feature parity
- Grab space charge from Ocelot/Cheetah
- Investigate PyTorch JIT / Nvidia Warp / other new GPU kernel compilers
- Link to ACNET for IOTA/FAST IPI commissioning

BACKUPS

IOTA



IOTA/FAST facility



Fermilab