Xsuite code

R. De Maria and G. Iadarola

With contributions from
A. Abramov, X. Buffat, P. Hermes, S. Kostoglou, A. Latina,
A. Oeftiger, M. Schwinzerl, G. Sterbini

https://xsuite.readthedocs.io

Outline

- **Introduction to Xsuite**
  - Motivation
  - Requirements
  - Design choices
  - Architecture
  - Development status
  - Documentation and developer's resources
- **Usage examples**
  - Single-particle tracking
  - Collective elements
  - Interface to other codes
- **Checks and first applications**
- **Summary**

Outline

- **Introduction to Xsuite**
  - Motivation
  - Requirements
  - Design choices
  - Architecture
  - Development status
  - Documentation and developer's resources
- **Usage examples**
  - Single-particle tracking
  - Collective elements
  - Interface to other codes
- **Checks and first applications**
- **Summary**

CERN-developed multiparticle codes

	Full lattice descrition	Dynamic effects (trims, noise)	Beam beam 4d (weak strong)	Beam beam 6d (weak strong)	e-cloud incoherent	Space charge frozen	Advanced collimation features	Impedances	Transverse feedbacks	Space charge PIC	e-cloud self-consistent	Beam beam 4d (strong strong)	Beam beam 6d (strong strong)	Synchrotron radiation	Beamstrahlung	Available on BOINC	Runs on GPU
MAD-X track	Available	Available	Available	Not available	Not available	Available	Not available	Not available	Not available	Not available	Not available	Not available	Not available	Available	Not available	Not available	Not available
Sixtrack	Available	Available	Available	Available	Not available	Not available	Available	Not available	Not available	Not available	Not available	Not available	Not available	Not available	Not available	Available	Not available
Sixtracklib	Available	Not available	Available	Available	Available	Available	Not available	Not available	Not available	Experimental	Not available	Not available	Not available	Not available	Not available	Not available	Available
PyHEADTAIL	Not available	Available	Available	Not available	Available	Available	Not available	Available	Available	Available	Available	Not available	Not available	Available	Not available	Not available	Experimental
COMBI	Not available	Available	Available	Available	Not available	Available	Not available	Available	Available	Not available	Not available	Available	Available	Available	Not available	Not available	Not available

Available | Not available | Experimental

We currently have at least **five CERN-developed multiparticle codes** that are **used in production** studies for CERN synchrotrons (+ need to use PyORBIT-PTC for Particle-In-Cell space charge studies)

This has multiple **drawbacks**:

- **Simulation capabilities are limited** (e.g. full-lattice + impedance is not possible)
- **Expensive** to maintain and further develop (duplicated efforts)
- **Long and very specific learning curve** for new-comers (know-how is not transferrable)
- Difficult to define a consistent strategy to tackle **future challenges**, FCC-ee, muon collider, PBC

CERN-developed multiparticle codes

	Full lattice descrition	Dynamic effects (trims, noise)	Beam beam 4d (weak strong)	Beam beam 6d (weak strong)	e-cloud incoherent	Space charge frozen	Advanced collimation features	Impedances	Transverse feedbacks	Space charge PIC	e-cloud self-consistent	Beam beam 4d (strong strong)	Beam beam 6d (strong strong)	Synchrotron radiation	Beamstrahlung	Available on BOINC	Runs on GPU
MAD-X track	Available	Available	Available	Not available	Not available	Available	Not available	Not available	Not available	Not available	Not available	Not available	Not available	Available	Not available	Not available	Not available
Sixtrack	Available	Available	Available	Available	Not available	Not available	Available	Not available	Not available	Not available	Not available	Not available	Not available	Not available	Not available	Available	Not available
Sixtracklib	Available	Not available	Available	Available	Available	Available	Not available	Not available	Not available	Experimental	Not available	Not available	Not available	Not available	Not available	Not available	Available
PyHEADTAIL	Not available	Available	Available	Not available	Available	Available	Not available	Available	Available	Available	Available	Not available	Not available	Available	Not available	Not available	Experimental
COMBI	Not available	Available	Available	Available	Not available	Available	Not available	Available	Available	Not available	Not available	Available	Available	Available	Not available	Not available	Not available

Available | Not available | Experimental

Adapting one of the existing codes to fulfil all the needs would be **very difficult**

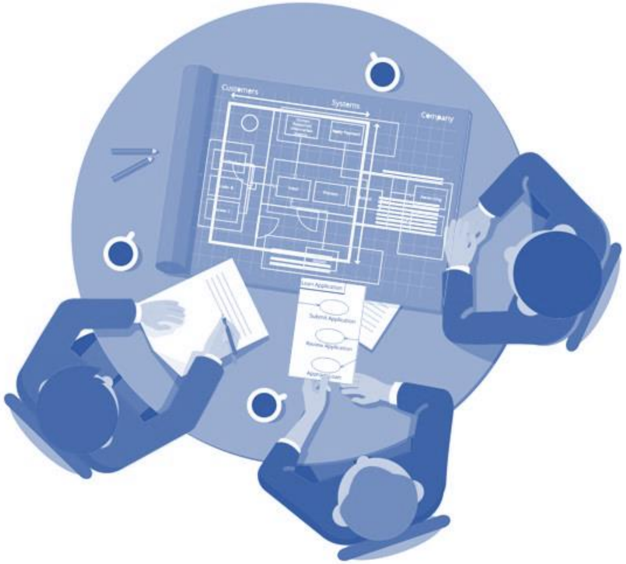
→ Opted to start a **new design (Xsuite) considering all requirements**

→ **No need to reinvent the wheel** → reused experience from existing codes, notably **sixtracklib** and **pyheadtail**

Xsuite: requirements

The following main **requirements** were identified :

- **Sustainability**: development/maintainance compatible with ABP's available manpower and knowhow
  - Favor **mainstream technologies** (e.g. python) to:
    - profit from existing knowhow in ABP
    - have a short learning curve for newcomers
    - "guarantee" sufficient long life of the code
  - **Code simple and slim:** introduction of new features should be “student friendly”



- Code should **easy and flexible to use** (scriptable)
- It should be **easy to interface** with many existing physics tools:
  - MAD-X via cpymad, PyHEADTAIL, pymask, COMBI/PyPLINE, FCC-EPFL framework
- **Speed** matters
  - Performance should stay in line with Sixtrack on CPU and with Sixtracklib on GPU
- Need to **run on CPUs and GPUs** from different vendors

Outline

- **Introduction to Xsuite**
  - Motivation
  - Requirements
  - Design choices
  - Architecture
  - Development status
  - Documentation and developer's resources
- **Usage examples**
  - Single-particle tracking
  - Collective elements
  - Interface to other codes
- **Checks and first applications**
- **Summary**

Design choice #1:

- The code is provided in the form of a set of **Python packages** (Xobjects, Xtrack, Xpart, …)

This has several **advantages**:

- **Profit of ABP know-how** and experience with python (OMC tools, pytimber, PyHEADTAIL, PyECLOUD, harpy, lumi modeling and followup tools, …)
- **Newcomers** typically have been already exposed to Python + **learning-curve is common many tools** used in ABP and at CERN for simulation, data analysis, operation…
- **Python can be used as glue** among Xsuite modules and with several CERN and general-purpose Python packages (plotting, fft, optimization, data storage, ML, …)
- Python is **easy to extend with C, C++ and FORTRAN** code for performance-critical parts

Xsuite – Support of GPUs

Support of **Graphics Processing Units (GPUs)** is a **necessary** requirement

→ applications like incoherent effects studies of space-charge or e-cloud are feasible only with GPUs

Market situation is somewhat **complicated**

→ there is no accepted standard for GPU programming

→ Different vendors have different languages, frameworks, etc.

→ Picture not expected to change on the short term

Design choice #2: same code should work on **multiple platforms**

- Usable on conventional CPUs (including multithreading support) and on GPUs from major vendors (NVIDIA, AMD, Intel)
- It is ready to be extended to new standards that are likely to come in the near future

Leveraged on available **open-source packages** for compiling/launching CPU and GPU code **through Python**

intel

NVIDIA

AMD

CuPy

PyOpenCL

CFFI

Xsuite is made of **five python modules**:

- One **low-level module (xobjects)** managing memory and code compilation at runtime on CPUs and GPUs
- Four **physics modules** which interact with the underlying computing platforms (CPU or GPU) through Xobjects

Physics modules

Xtrack
single particle
tracking engine

Xpart
generation of particles
distributions

Xfields
computation of EM fields
from particle ensembles

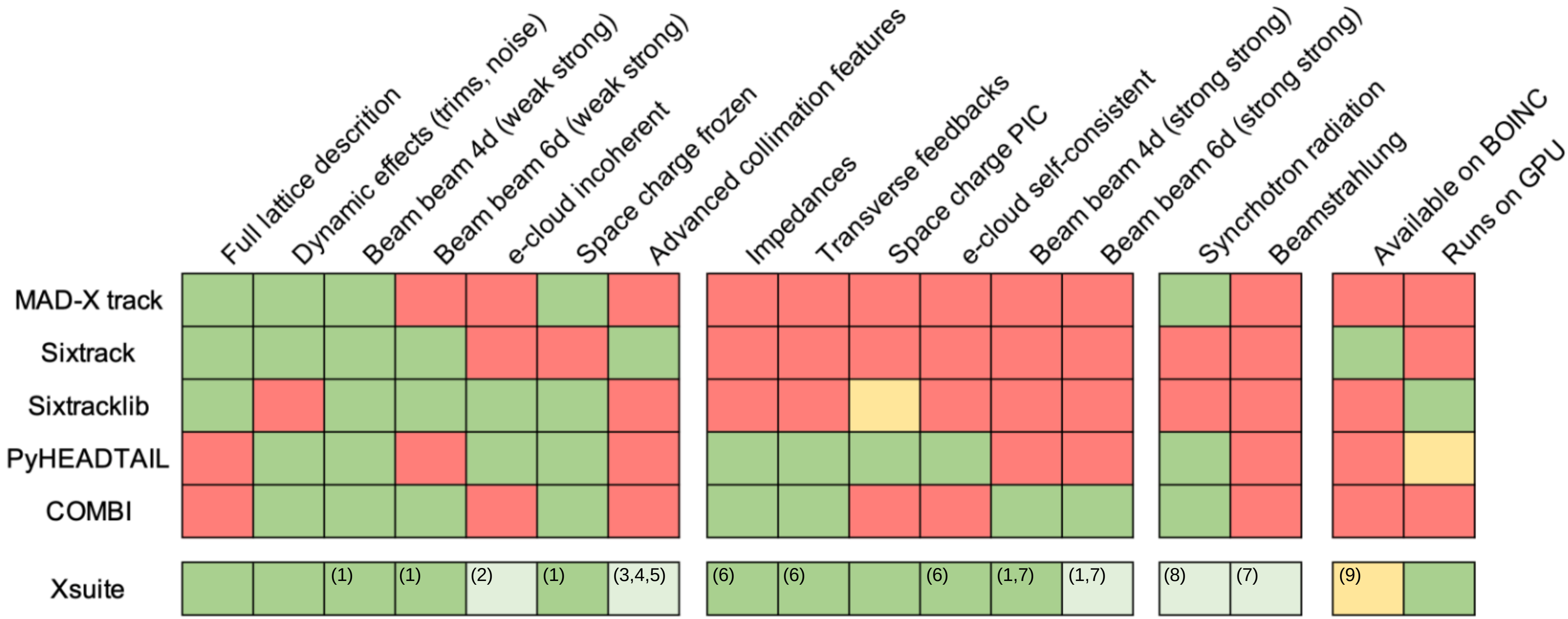
Xobjects
interface to different computing plaforms
(CPUs and GPUs of different vendors)

Outline

- **Introduction to Xsuite**
  - Motivation
  - Requirements
  - Design choices
  - Architecture
  - Development status
  - Documentation and developer's resources
- **Usage examples**
  - Single-particle tracking
  - Collective elements
  - Interface to other codes
- **Checks and first applications**
- **Summary**

Xsuite development status

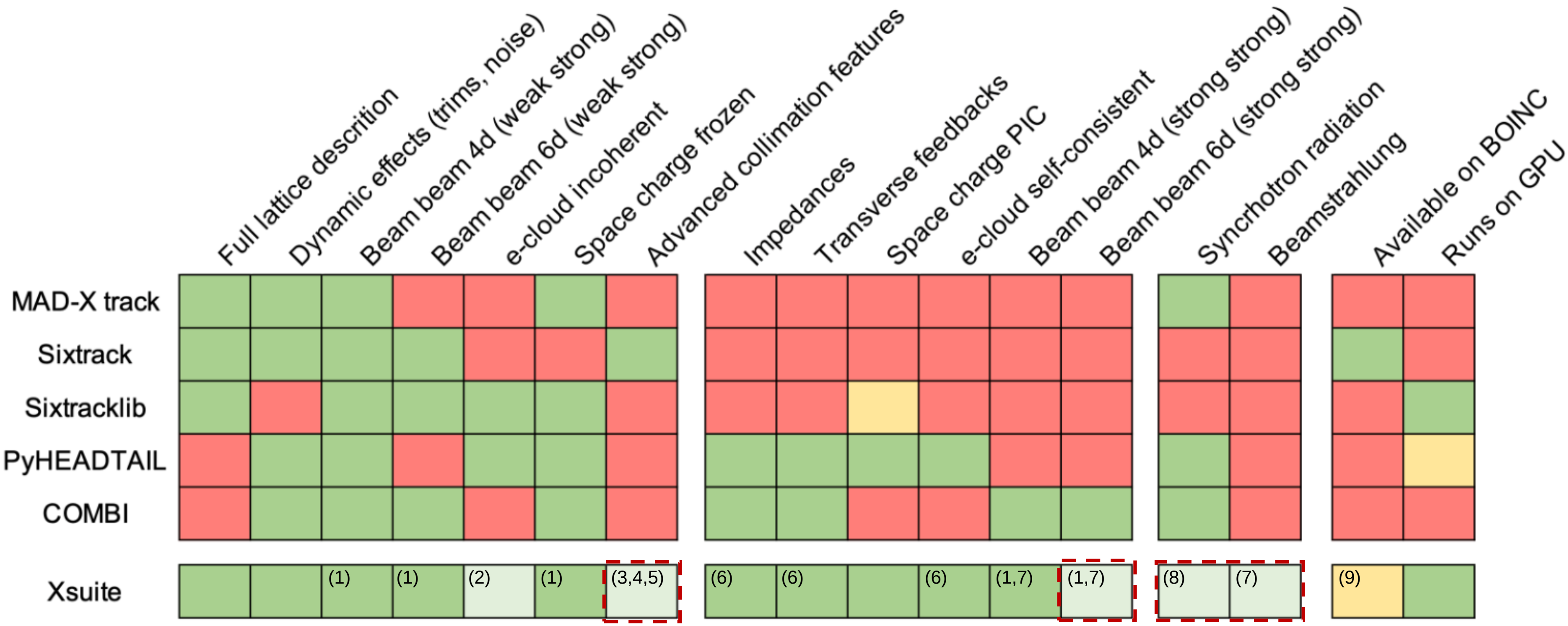
- Several **colleagues could already contribute** to the development (many thanks!)
  - → Demonstrated **short learning curve for developers**
  - → Greatly helped to achieve a **quick progress of the project** (Xsuite is now being used for first production studies)



(1) Uses optimized implementation of Faddeeva function providing x10 speedup on GPU (M.Schwinzerl)
(2) To be ported from Sixrtacklib (straightforward)
(3) Electron lens implemented (P. Hermes)
(4) Geant4 interface working (A. Abramov)
(5) Porting K2 scattering and Fluka coupling is under development (F. Van Der Veken, P. Hermes)
(6) Through PyHEADTAIL interface (X. Buffat) Only CPU for now
(7) Under development (P. Kicsiny, X. Buffat)
(8) Under development (A. Latina)
(9) Under study

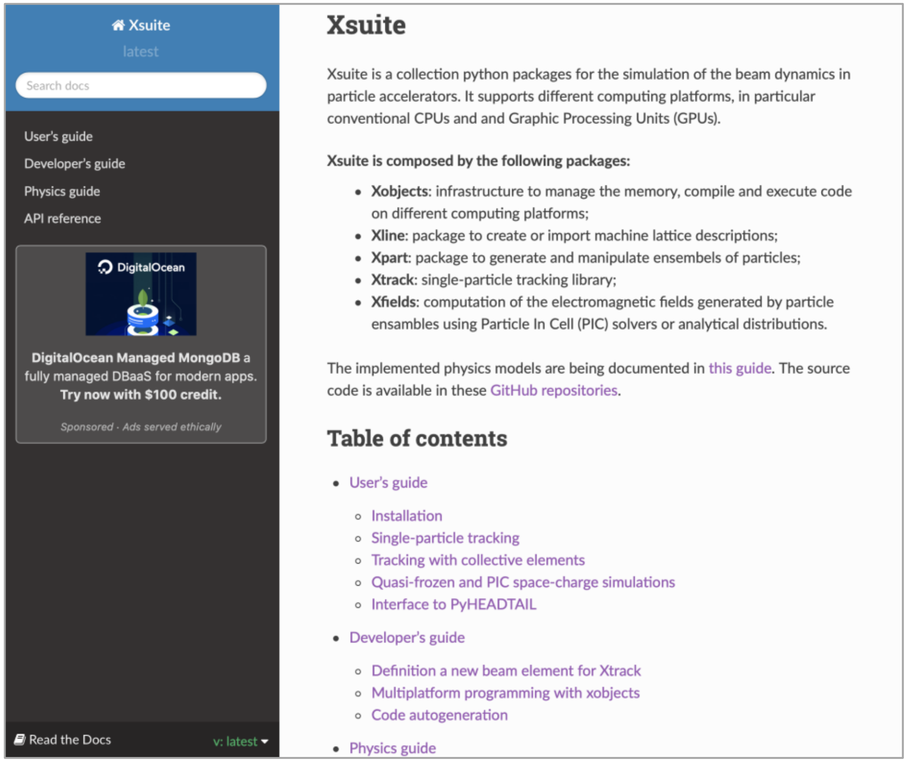
Xsuite development status

- Several **colleagues could already contribute** to the development (many thanks!)
  - → Demonstrated **short learning curve for developers**
  - → Greatly helped to achieve a **quick progress of the project** (Xsuite is now being used for first production studies)



→Include developments dedicated to **FCC-ee** (EPFL/Chart collaboration)

- Documentation pages available at https://xsuite.readthedocs.io and **integrated by sets of examples** available in the repository
  → So far **experience was very positive**: users with some python experience were able to get started with little or no tutoring
- Xsuite is intended as an **open-source community project**:
  - **User community** is **encouraged to contribute**
  - Documentation includes **developer's guide** on how to extend the code
  - Aiming at keeping learning curve for new developers as short as possible



Outline

- **Introduction to Xsuite**
  - Motivation
  - Requirements
  - Design choices
  - Architecture
  - Development status
  - Documentation and developer's resources
- **Usage examples**
  - Single-particle tracking
  - Collective elements
  - Interface to other codes
- **Checks and first applications**
- **Summary**

A basic example: single-particle tracking

Simulations are configured and launched with a **Python script** (or Jupyter notebook)

```
import xobjects as xo
import xtrack as xt
import xpart as xp

## Generate a simple beamline
line = xt.Line(
    elements=[xt.Drift(length=1.), xt.Multipole(knl=[0, 1.], ksl=[0,0]),
              xt.Drift(length=1.), xt.Multipole(knl=[0, -1.], ksl=[0,0])],
    element_names=['drift_0', 'quad_0', 'drift_1', 'quad_1'])

## Choose a context
context = xo.ContextCpu() # For CPU

## Transfer lattice on context and compile tracking code
tracker = xt.Tracker(_context=context, line=line)

## Build particle object on context
n_part = 200
import numpy as np
particles = xp.Particles(_context=context, p0c=6500e9,
    x=np.random.uniform(-1e-3, 1e-3, n_part),
    zeta=np.random.uniform(-1e-2, 1e-2, n_part),
    delta=np.random.uniform(-1e-4, 1e-4, n_part))

## Track (saving turn-by-turn data)
tracker.track(particles, num_turns=100, turn_by_turn_monitor=True)

## The particle is changed in place and turn-by-turn data is available at:
tracker.record_last_track.x, tracker.record_last_track.px # etc...
```

We import the Xsuite modules that we need

A basic example: single-particle tracking

Simulations are configured and launched with a Python script (or Jupyter notebook)

```
import xobjects as xo
import xtrack as xt
import xpart as xp

## Generate a simple beamline
line = xt.Line(
    elements=[xt.Drift(length=1.), xt.Multipole(knl=[0, 1.], ksl=[0,0]),
              xt.Drift(length=1.), xt.Multipole(knl=[0, -1.], ksl=[0,0])],
    element_names=['drift_0', 'quad_0', 'drift_1', 'quad_1'])

## Choose a context
context = xo.ContextCpu() # For CPU

## Transfer lattice on context and compile tracking code
tracker = xt.Tracker(_context=context, line=line)

## Build particle object on context
n_part = 200
import numpy as np
particles = xp.Particles(_context=context, p0c=6500e9,
    x=np.random.uniform(-1e-3, 1e-3, n_part),
    zeta=np.random.uniform(-1e-2, 1e-2, n_part),
    delta=np.random.uniform(-1e-4, 1e-4, n_part))

## Track (saving turn-by-turn data)
tracker.track(particles, num_turns=100, turn_by_turn_monitor=True)

## The particle is changed in place and turn-by-turn data is available at:
tracker.record_last_track.x, tracker.record_last_track.px # etc...
```

We use Xtrack to create a simple sequence (a FODO)
→ can import more complex lattice from MAD-X

A basic example: single-particle tracking

Simulations are configured and launched with a Python script (or Jupyter notebook)

```
import xobjects as xo
import xtrack as xt
import xpart as xp

## Generate a simple beamline
line = xt.Line(
    elements=[xt.Drift(length=1.), xt.Multipole(knl=[0, 1.], ksl=[0,0]),
              xt.Drift(length=1.), xt.Multipole(knl=[0, -1.], ksl=[0,0])],
    element_names=['drift_0', 'quad_0', 'drift_1', 'quad_1'])

## Choose a context
context = xo.ContextCpu() # For CPU

## Transfer lattice on context and compile tracking code
tracker = xt.Tracker(_context=context, line=line)

## Build particle object on context
n_part = 200
import numpy as np
particles = xp.Particles(_context=context, p0c=6500e9,
    x=np.random.uniform(-1e-3, 1e-3, n_part),
    zeta=np.random.uniform(-1e-2, 1e-2, n_part),
    delta=np.random.uniform(-1e-4, 1e-4, n_part))

## Track (saving turn-by-turn data)
tracker.track(particles, num_turns=100, turn_by_turn_monitor=True)

## The particle is changed in place and turn-by-turn data is available at:
tracker.record_last_track.x, tracker.record_last_track.px # etc...
```

We choose the computing platform on which we want to run (CPU or GPU)

A basic example: single-particle tracking

Simulations are configured and launched with a Python script (or Jupyter notebook)

```
import xobjects as xo
import xtrack as xt
import xpart as xp

## Generate a simple beamline
line = xt.Line(
    elements=[xt.Drift(length=1.), xt.Multipole(knl=[0, 1.], ksl=[0,0]),
              xt.Drift(length=1.), xt.Multipole(knl=[0, -1.], ksl=[0,0])],
    element_names=['drift_0', 'quad_0', 'drift_1', 'quad_1'])

## Choose a context
context = xo.ContextCpu() # For CPU

## Transfer lattice on context and compile tracking code
tracker = xt.Tracker(_context=context, line=line)

## Build particle object on context
n_part = 200
import numpy as np
particles = xp.Particles(_context=context, p0c=6500e9,
    x=np.random.uniform(-1e-3, 1e-3, n_part),
    zeta=np.random.uniform(-1e-2, 1e-2, n_part),
    delta=np.random.uniform(-1e-4, 1e-4, n_part))

## Track (saving turn-by-turn data)
tracker.track(particles, num_turns=100, turn_by_turn_monitor=True)

## The particle is changed in place and turn-by-turn data is available at:
tracker.record_last_track.x, tracker.record_last_track.px # etc...
```

We build a tracker object, which can track particles in our beam line on the chosen computing platform

A basic example: single-particle tracking

Simulations are configured and launched with a Python script (or Jupyter notebook)

```
import xobjects as xo
import xtrack as xt
import xpart as xp

## Generate a simple beamline
line = xt.Line(
    elements=[xt.Drift(length=1.), xt.Multipole(knl=[0, 1.], ksl=[0,0]),
              xt.Drift(length=1.), xt.Multipole(knl=[0, -1.], ksl=[0,0])],
    element_names=['drift_0', 'quad_0', 'drift_1', 'quad_1'])

## Choose a context
context = xo.ContextCpu() # For CPU

## Transfer lattice on context and compile tracking code
tracker = xt.Tracker(_context=context, line=line)

## Build particle object on context
n_part = 200
import numpy as np
particles = xp.Particles(_context=context, p0c=6500e9,
    x=np.random.uniform(-1e-3, 1e-3, n_part),
    zeta=np.random.uniform(-1e-2, 1e-2, n_part),
    delta=np.random.uniform(-1e-4, 1e-4, n_part))

## Track (saving turn-by-turn data)
tracker.track(particles, num_turns=100, turn_by_turn_monitor=True)

## The particle is changed in place and turn-by-turn data is available at:
tracker.record_last_track.x, tracker.record_last_track.px # etc...
```

We generate a set of particles (in this case using a standard python random generator)

A basic example: single-particle tracking

Simulations are configured and launched with a Python script (or Jupyter notebook)

```python
import xobjects as xo
import xtrack as xt
import xpart as xp

## Generate a simple beamline
line = xt.Line(
    elements=[xt.Drift(length=1.), xt.Multipole(knl=[0, 1.], ksl=[0,0]),
              xt.Drift(length=1.), xt.Multipole(knl=[0, -1.], ksl=[0,0])],
    element_names=['drift_0', 'quad_0', 'drift_1', 'quad_1'])

## Choose a context
context = xo.ContextCpu() # For CPU

## Transfer lattice on context and compile tracking code
tracker = xt.Tracker(_context=context, line=line)

## Build particle object on context
n_part = 200
import numpy as np
particles = xp.Particles(_context=context, p0c=6500e9,
    x=np.random.uniform(-1e-3, 1e-3, n_part),
    zeta=np.random.uniform(-1e-2, 1e-2, n_part),
    delta=np.random.uniform(-1e-4, 1e-4, n_part))

## Track (saving turn-by-turn data)
tracker.track(particles, num_turns=100, turn_by_turn_monitor=True)

## The particle is changed in place and turn-by-turn data is available at:
tracker.record_last_track.x, tracker.record_last_track.px # etc...
```

We launch the tracking (particles are updated as tracking progresses)

A basic example: single-particle tracking

Simulations are configured and launched with a Python script (or Jupyter notebook)

```
import xobjects as xo
import xtrack as xt
import xpart as xp

## Generate a simple beamline
line = xt.Line(
    elements=[xt.Drift(length=1.), xt.Multipole(knl=[0, 1.], ksl=[0,0]),
              xt.Drift(length=1.), xt.Multipole(knl=[0, -1.], ksl=[0,0])],
    element_names=['drift_0', 'quad_0', 'drift_1', 'quad_1'])

## Choose a context
context = xo.ContextCpu() # For CPU

## Transfer lattice on context and compile tracking code
tracker = xt.Tracker(_context=context, line=line)

## Build particle object on context
n_part = 200
import numpy as np
particles = xp.Particles(_context=context, p0c=6500e9,
    x=np.random.uniform(-1e-3, 1e-3, n_part),
    zeta=np.random.uniform(-1e-2, 1e-2, n_part),
    delta=np.random.uniform(-1e-4, 1e-4, n_part))

## Track (saving turn-by-turn data)
tracker.track(particles, num_turns=100, turn_by_turn_monitor=True)

## The particle is changed in place and turn-by-turn data is available at:
tracker.record_last_track.x, tracker.record_last_track.px # etc...
```

Access to the recorded particles coordinates

A basic example: single-particle tracking

Simulations are configured and launched with a Python script (or Jupyter notebook)

```
import xobjects as xo
import xtrack as xt
import xpart as xp

## Generate a simple beamline
line = xt.Line(
    elements=[xt.Drift(length=1.), xt.Multipole(knl=[0, 1.], ksl=[0,0]),
              xt.Drift(length=1.), xt.Multipole(knl=[0, -1.], ksl=[0,0])],
    element_names=['drift_0', 'quad_0', 'drift_1', 'quad_1'])

## Choose a context
context = xo.ContextCpu() # For CPU

## Transfer lattice on context and compile tracking code
tracker = xt.Tracker(_context=context, line=line)

## Build particle object on context
n_part = 200
import numpy as np
particles = xp.Particles(_context=context, p0c=6500e9,
    x=np.random.uniform(-1e-3, 1e-3, n_part),
    zeta=np.random.uniform(-1e-2, 1e-2, n_part),
    delta=np.random.uniform(-1e-4, 1e-4, n_part))

## Track (saving turn-by-turn data)
tracker.track(particles, num_turns=100, turn_by_turn_monitor=True)

## The particle is changed in place and turn-by-turn data is available at:
tracker.record_last_track.x, tracker.record_last_track.px # etc...
```

To run on GPU all we need to do is to change the context

A basic example: single-particle tracking

Simulations are configured and launched with a Python script (or Jupyter notebook)

```python
import xobjects as xo
import xtrack as xt
import xpart as xp

## Generate a simple beamline
line = xt.Line(
    elements=[xt.Drift(length=1.), xt.Multipole(knl=[0, 1.], ksl=[0,0]),
              xt.Drift(length=1.), xt.Multipole(knl=[0, -1.], ksl=[0,0])],
    element_names=['drift_0', 'quad_0', 'drift_1', 'quad_1'])

## Choose a context
context = xo.ContextCupy() # For NVIDIA GPUs

## Transfer lattice on context and compile tracking code
tracker = xt.Tracker(_context=context, line=line)

## Build particle object on context
n_part = 200
import numpy as np
particles = xp.Particles(_context=context, p0c=6500e9,
    x=np.random.uniform(-1e-3, 1e-3, n_part),
    zeta=np.random.uniform(-1e-2, 1e-2, n_part),
    delta=np.random.uniform(-1e-4, 1e-4, n_part))

## Track (saving turn-by-turn data)
tracker.track(particles, num_turns=100, turn_by_turn_monitor=True)

## The particle is changed in place and turn-by-turn data is available at:
tracker.record_last_track.x, tracker.record_last_track.px # etc...
```

To run on GPU all we need to do is to change the context

A basic example: single-particle tracking

Simulations are configured and launched with a Python script (or Jupyter notebook)

```
import xobjects as xo
import xtrack as xt
import xpart as xp

## Generate a simple beamline
line = xt.Line(
    elements=[xt.Drift(length=1.), xt.Multipole(knl=[0, 1.], ksl=[0,0]),
              xt.Drift(length=1.), xt.Multipole(knl=[0, -1.], ksl=[0,0])],
    element_names=['drift_0', 'quad_0', 'drift_1', 'quad_1'])

## Choose a context
context = xo.ContextPyopencl() # For AMD GPUs and other hardware

## Transfer lattice on context and compile tracking code
tracker = xt.Tracker(_context=context, line=line)

## Build particle object on context
n_part = 200
import numpy as np
particles = xp.Particles(_context=context, p0c=6500e9,
    x=np.random.uniform(-1e-3, 1e-3, n_part),
    zeta=np.random.uniform(-1e-2, 1e-2, n_part),
    delta=np.random.uniform(-1e-4, 1e-4, n_part))

## Track (saving turn-by-turn data)
tracker.track(particles, num_turns=100, turn_by_turn_monitor=True)

## The particle is changed in place and turn-by-turn data is available at:
tracker.record_last_track.x, tracker.record_last_track.px # etc...
```

To run on GPU all we need to do is to change the context

Outline

- **Introduction to Xsuite**
  - Motivation
  - Requirements
  - Design choices
  - Architecture
  - Development status
  - Documentation and developer's resources
- **Usage examples**
  - Single-particle tracking
  - Collective elements
  - Interface to other codes
- **Checks and first applications**
- **Summary**

Xsuite can handle **collective elements**, i.e. elements for which the action on a particle depends on the coordinates of other particles

→ it means that the **tracking of different particles cannot happen asynchronously**

No special action is required by the user. Collective elements are handled automatically by the the Xtrack tracker

```
# [Imports, contexts, particles as for single-particle simulations]

## Build a collective element (e.g. space-charge interaction)
import xfields as xf
spcharge = xf.SpaceCharge3D(_context=context, update_on_track=True,
    x_range=(-5e-3, 5e-3), y_range=(-4e-3, 4e-3), z_range=(-4e-3, 4e-3),
    length=1, nx=256, ny=256, nz=100, solver='FFTSolver2p5D')

## Build a simple beamline including the space-charge element
line = xt.Line(
    elements = [xt.Multipole(knl=[0, 1.]), xt.Drift(length=1.),
                spcharge,
                xt.Multipole(knl=[0, -1.]), xt.Drift(length=1.)]
    element_names = ['qf1', 'drift1', 'spcharge' 'qd1', 'drift2', '])


## Transfer lattice on context and compile tracking code
## as for single particle simulations
tracker = xt.Tracker(_context=context, line=line)
```

A PIC space-charge element is a collective element

Collective beam elements

Xsuite can handle **collective elements**, i.e. elements for which the action on a particle depends on the coordinates of other particles

→ it means that the **tracking of different particles cannot happen asynchronously**

No special action is required by the user. Collective elements are handled automatically by the the Xtrack tracker

```
# [Imports, contexts, particles as for single-particle simulations]

## Build a collective element (e.g. space-charge interaction)
import xfields as xf
spcharge = xf.SpaceCharge3D(_context=context, update_on_track=True,
    x_range=(-5e-3, 5e-3), y_range=(-4e-3, 4e-3), z_range=(-4e-3, 4e-3),
    length=1, nx=256, ny=256, nz=100, solver='FFTSolver2p5D')

## Build a simple beamline including the space-charge element
line = xt.Line(
    elements = [xt.Multipole(knl=[0, 1.]), xt.Drift(length=1.),
                spcharge,
                xt.Multipole(knl=[0, -1.]), xt.Drift(length=1.)]
    element_names = ['qf1', 'drift1', 'spcharge' 'qd1', 'drift2', '])


## Transfer lattice on context and compile tracking code
## as for single particle simulations
tracker = xt.Tracker(_context=context, line=line)
```

It can be included in a Xtrack line together with single-particle elements

Xsuite can handle **collective elements**, i.e. elements for which the action on a particle depends on the coordinates of other particles

→ it means that the **tracking of different particles cannot happen asynchronously**

No special action is required by the user. Collective elements are handled automatically by the the Xtrack tracker

```
# [Imports, contexts, particles as for single-particle simulations]

## Build a collective element (e.g. space-charge interaction)
import xfields as xf
spcharge = xf.SpaceCharge3D(_context=context, update_on_track=True,
    x_range=(-5e-3, 5e-3), y_range=(-4e-3, 4e-3), z_range=(-4e-3, 4e-3),
    length=1, nx=256, ny=256, nz=100, solver='FFTSolver2p5D')

## Build a simple beamline including the space-charge element
line = xt.Line(
    elements = [xt.Multipole(knl=[0, 1.]), xt.Drift(length=1.),
                spcharge,
                xt.Multipole(knl=[0, -1.]), xt.Drift(length=1.)]
    element_names = ['qf1', 'drift1', 'spcharge' 'qd1', 'drift2', '])

## Transfer lattice on context and compile tracking code
## as for single particle simulations
tracker = xt.Tracker(_context=context, line=line)
```

The tracker can be built as seen for single-particle simulations

The tracker takes care of **cutting the sequence** at the collective elements

- Tracking between the collective elements is performed asynchronously (better performance)
- Simulation of collective interactions is performed synchronously

Outline

- **Introduction to Xsuite**
  - Motivation
  - Requirements
  - Design choices
  - Architecture
  - Development status
  - Documentation and developer's resources
- **Usage examples**
  - Single-particle tracking
  - Collective elements
  - Interface to other codes
- **Checks and first applications**
- **Summary**

Xsuite is conceived to be interfaced to other Python modules

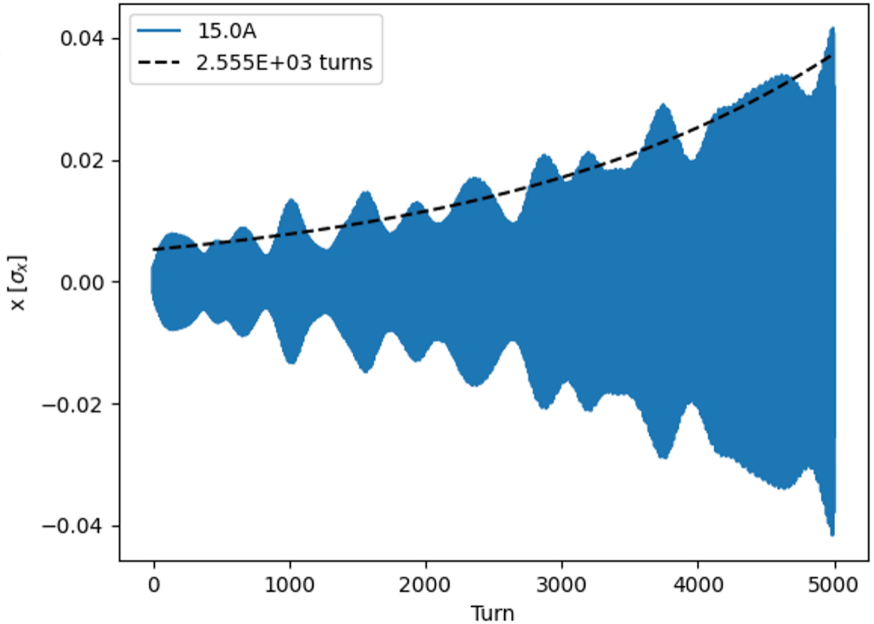
- Any **python object provideing a "el.track(particles)" method** can be insterted in a Xsuite lattice (assumes convention on particle coordinates naming and data structure)
- For example PyHEADTAIL can be used to intruduce **collective beam elements** (impedances, dampers, e-cloud) in Xsuite simulation
    - For this purpose we built a **"PyHEADTAIL-compatiblity mode"** in Xtrack as PyHEADTAIL uses a slightly different naming convention

```
import xtrack as xt
xt.enable_pyheadtail_interface()

## Create a PyHEADTAIL element
from PyHEADTAIL.feedback.transverse_damper import TransverseDamper
damper = TransverseDamper(dampingrate_x=10., dampingrate_y=15.)


## Build a simple sequence including the space-charge element
line = xt.Line(
    elements = [xt.Multipole(knl=[0, 1.]), xt.Drift(length=1.),
                damper,
                xt.Multipole(knl=[0, -1.]), xt.Drift(length=1.)]
    element_names = ['qf1', 'drift1', 'damper', 'qd1', 'drift2'])

## Transfer lattice on context and compile tracking code
## as for single particle simulations
tracker = xt.Tracker(_context=context, line=line)
```

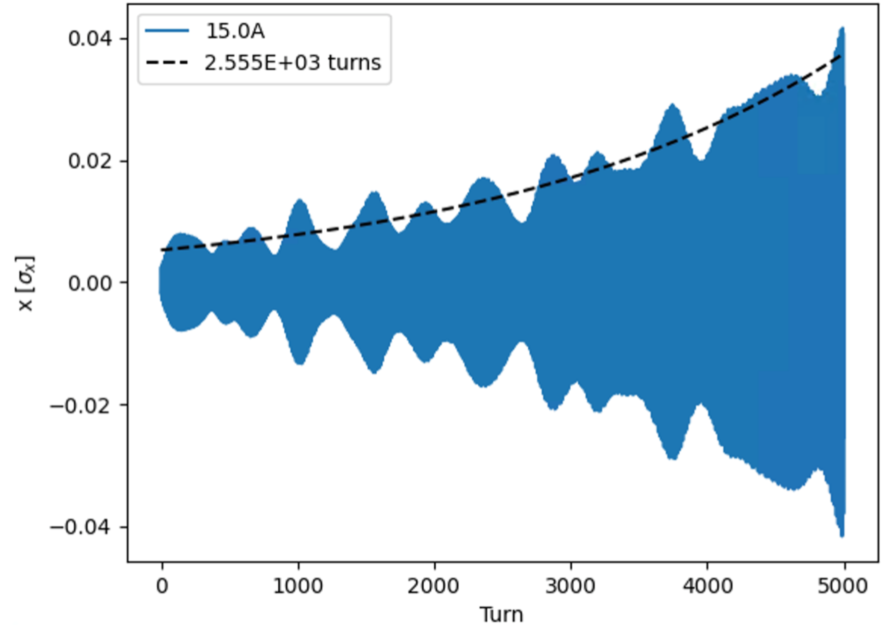
X. Buffat

Comparison

Tracking, impedance and damper in PyHEADTAIL



Tracking Xsuite impedance and in PyHEADTAIL



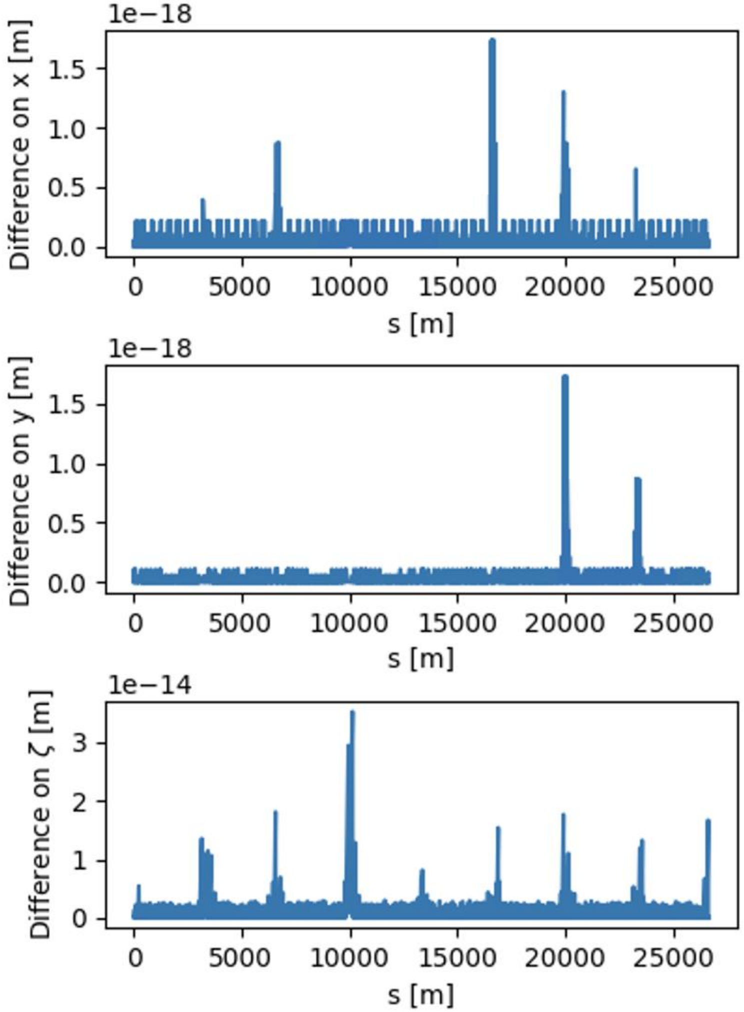

```
                 damper,
                 xt.Multipole(knl=[0, -1.]), xt.Drift(length=1.)]
     element_names = ['qf1', 'drift1', 'damper', 'qd1', 'drift2'])

## Transfer lattice on context and compile tracking code
## as for single particle simulations
tracker = xt.Tracker(_context=context, line=line)
```

Outline

- **Introduction to Xsuite**
  - Motivation
  - Requirements
  - Design choices
  - Architecture
  - Development status
  - Documentation and developer's resources
- **Usage examples**
  - Single-particle tracking
  - Collective elements
  - Interface to other codes
- **Checks and first applications**
- **Summary**

Single-particle tracking – benchmarks and performance

- Single-particle tracking has been **successfully benchmarked against SixTrack**
  → Checks performed for protons and ions
- **Computation time** very similar to Sixtrack on CPU and to sixtracklib on GPU



Platform	Computing time
CPU	190 (μs/part./turn)
GPU (Titan V, cupy)	0.80 (μs/part./turn)
GPU (Titan V, pyopencl)	0.85 (μs/part./turn)

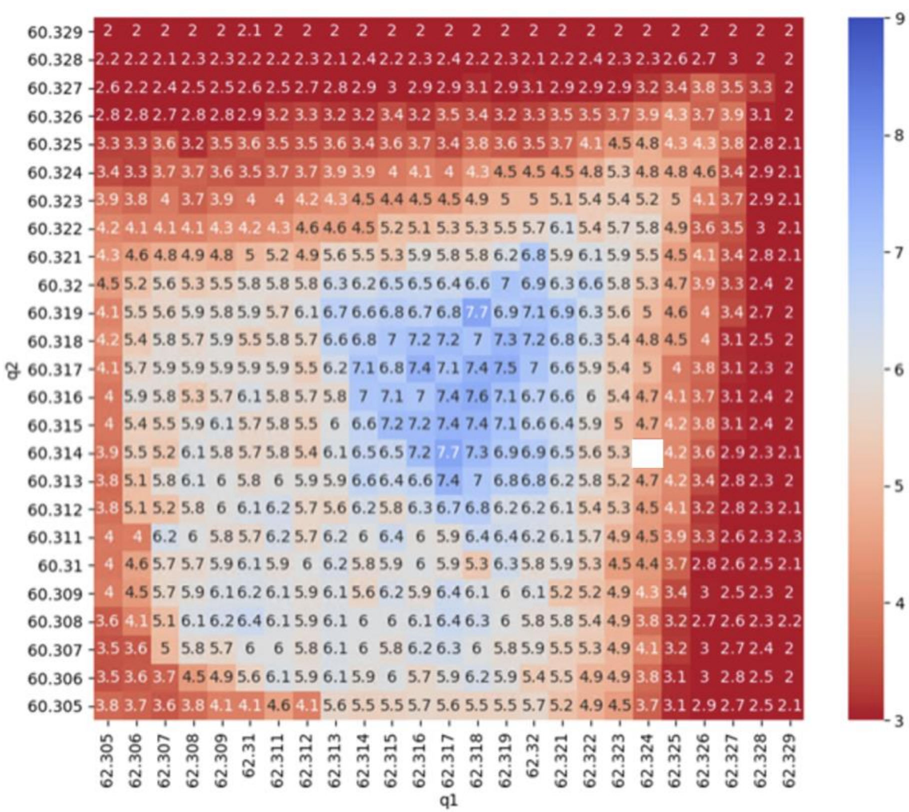
(*) tests made on ABP GPU server

HL-LHC Dynamic Aperture scan

G. Sterbini, K. Paraschou, S. Kostoglou

Example of integration with **other Pythonic tools** in a complex workflow

- **Pymask** used to prepare the machine configurations
- Generation of **matched particle distribution** using python module from pysixtrack
- **Job management** using a new **Python package (TreeMaker)**
- **Tracking** performed with **Xsuite** (parquet files used for data storage)
- **Dynamic Aperture computation** in Python using **Pandas**



Parameters of pilot study

Full HL-LHC lattice (20k elements)
Weak strong Beam-beam

N. tune configurations = 625
N. tracked particles/conf. = 1780
N. turns = 10^6

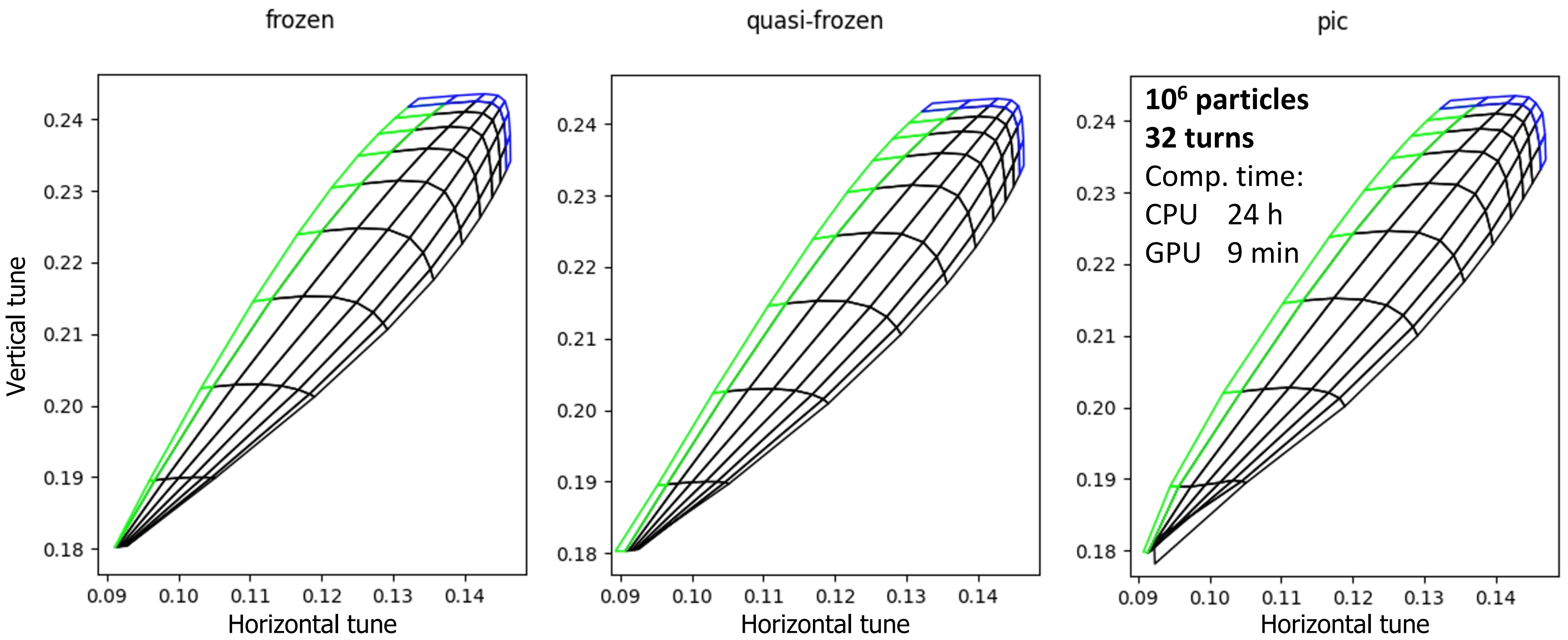
N. jobs = ~10'000

Comp. time ~48h on INFN- CNAF cluster

Space-charge – benchmarks and performance

Xsuite allows **different kinds of space-charge simulations** (frozen, quasi-frozen, Particle In Cell - switching from one to the other is straightforward)

- Tested in the realistic case of the full SPS lattice with **540 space-charge interactions**
- Example of application where **the usage of GPUs is practically mandatory**



X. Buffat

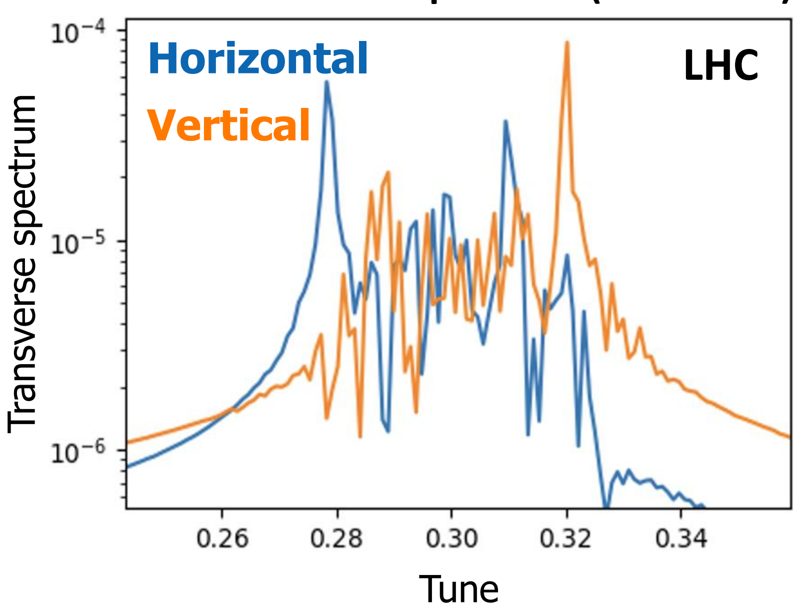
Xsuite used to simulate **strong-strong beam beam effects**

- Additional package (**PyPLINE**) is under development to provides multi-node parallelization and simulate many bunches

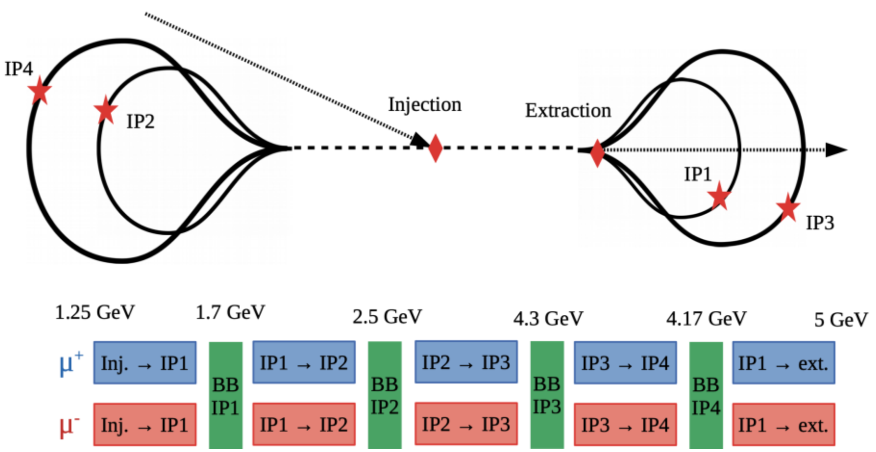
  → Provides **two-level parallelization** in combination with Xsuite multithreading

- **Tested and routinely used on CERN HPC cluster**

Coherent beam spectrum (2 bunches)



Xsuite used for first studies on **beam-beam effects** in recirculating linac for **muon collider**



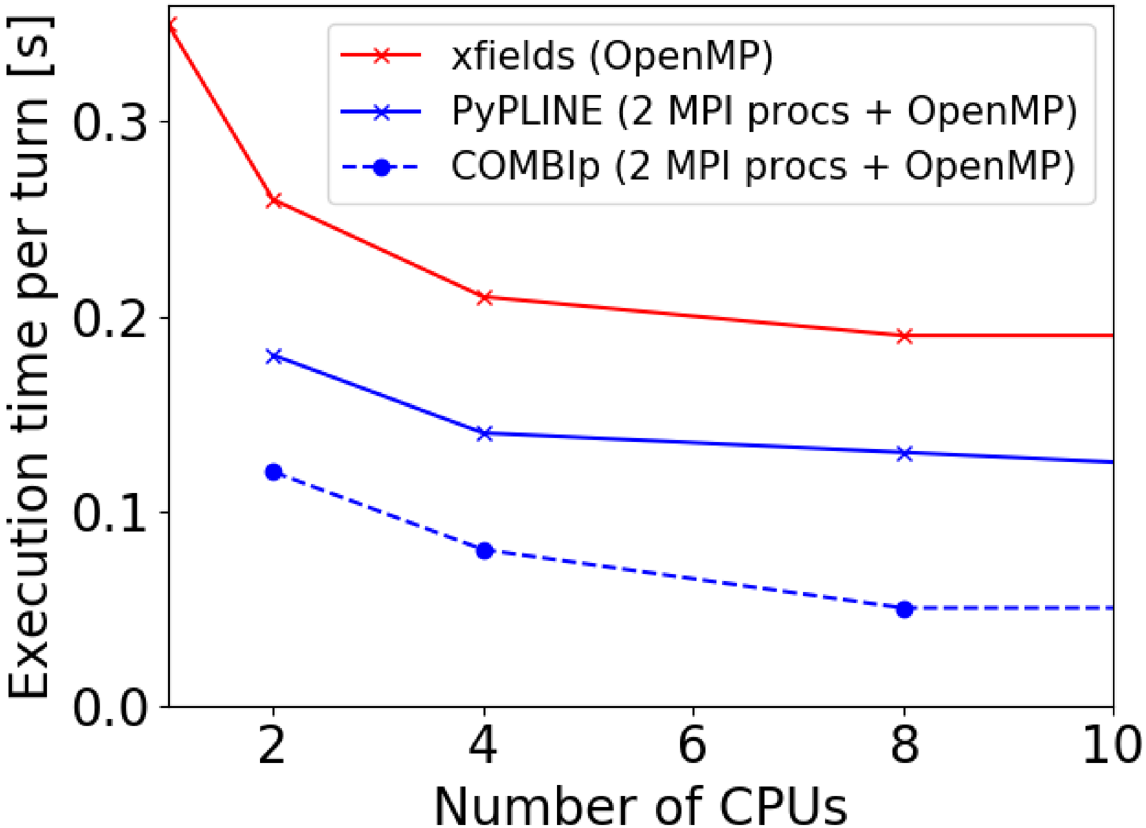
X. Buffat

Xsuite used to simulate **strong-strong beam beam effects**

- Additional package (**PyPLINE**) is under development to provides multi-node parallelization and simulate many bunches

  → Provides **two-level parallelization** in combination with Xsuite multithreading

- **Tested and routinely used on CERN HPC cluster**



Performance optimization is ongoing

- Speed getting close to COMBIp (heavily optimized in the past)

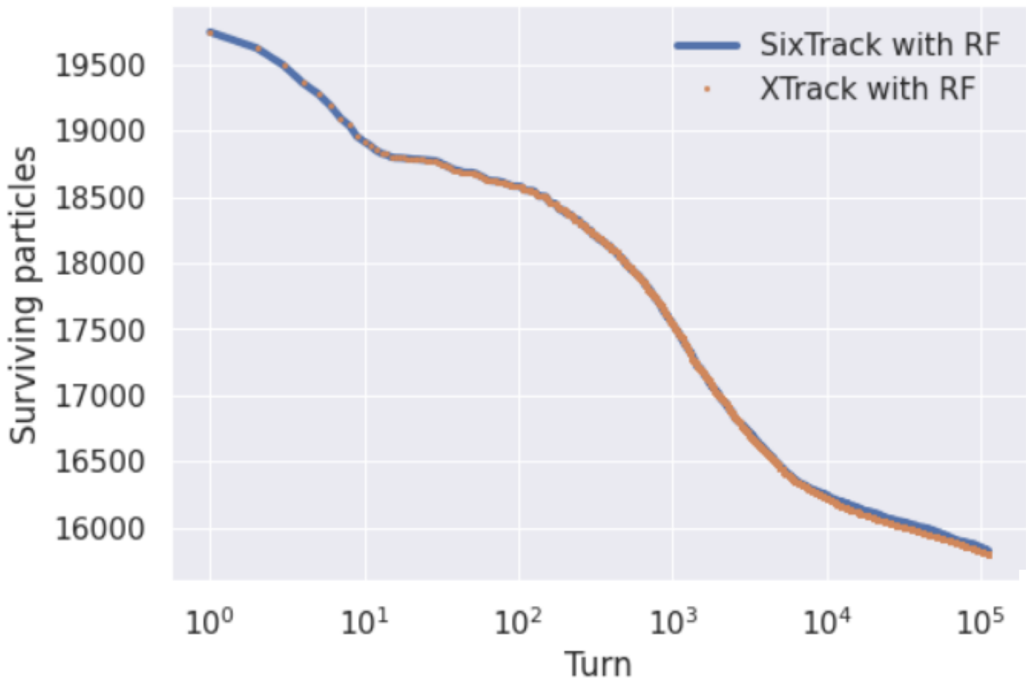
Hollow electron lens studies

P. Hermes

Xsuite is being used to study **halo depletion** with **hollow electron lenses** for HL-LHC

- Implemented hollow e-lens in Xtrack
- Benchmarked against Sixtrack
- Performed first realistic studies (parametric scans)

→ Showed **significant advantage of using GPUs**



Test run (10 turns)

Setup	Tracking time [s]
SixTrack without K2	40
XTrack without K2 (GPU)	0.3

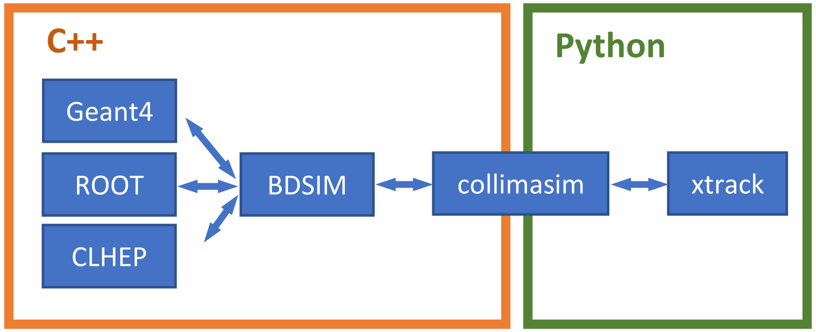
Realistic study (parametric scan)

	Simulated time interval	Number of jobs	Time needed*
Xtrack (GPU)	10s	400	~ 24 h
SixTrack	1s	40000	~ 7 days

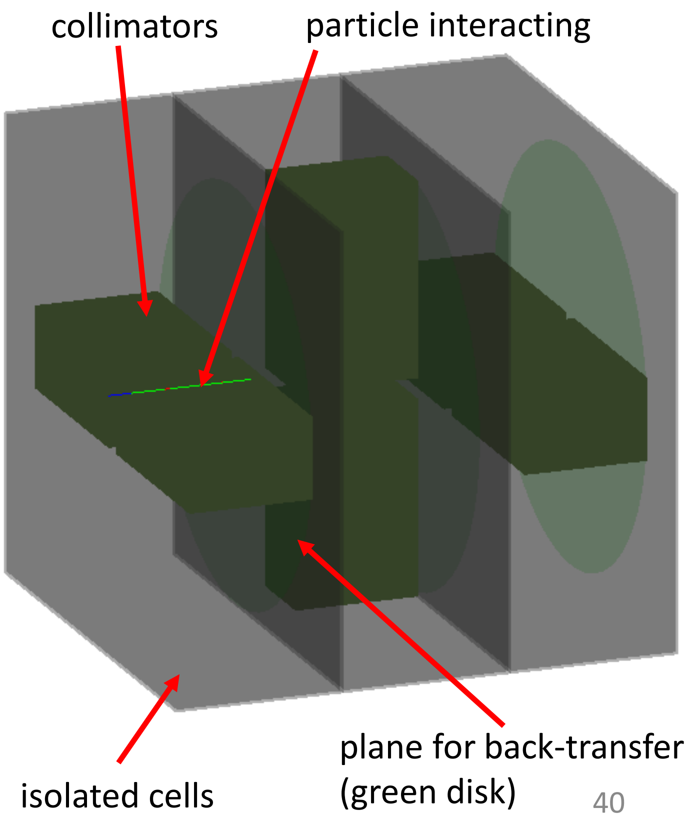
* CPUs and GPUs in HTCondor

More details at https://indico.cern.ch/event/1068125/contributions/4491551

A. Abramov



- Needed for **FCC-ee collimation studies**
- Using a C++ framework based on BDSIM (L. Nevay):
  - Geant4 radiation transport model with collimators in individual cells
  - Particles exchanged between the tracking code and the Geant4 model
  - Similar mechanism to the SixTrack-FLUKA coupling
- Dedicated C++ - Python interface implemented (collimasim)
- The first integration with Xtrack is available:
  - Supports collimator definition, beamline integration, and particle transfer
  - Tests ongoing



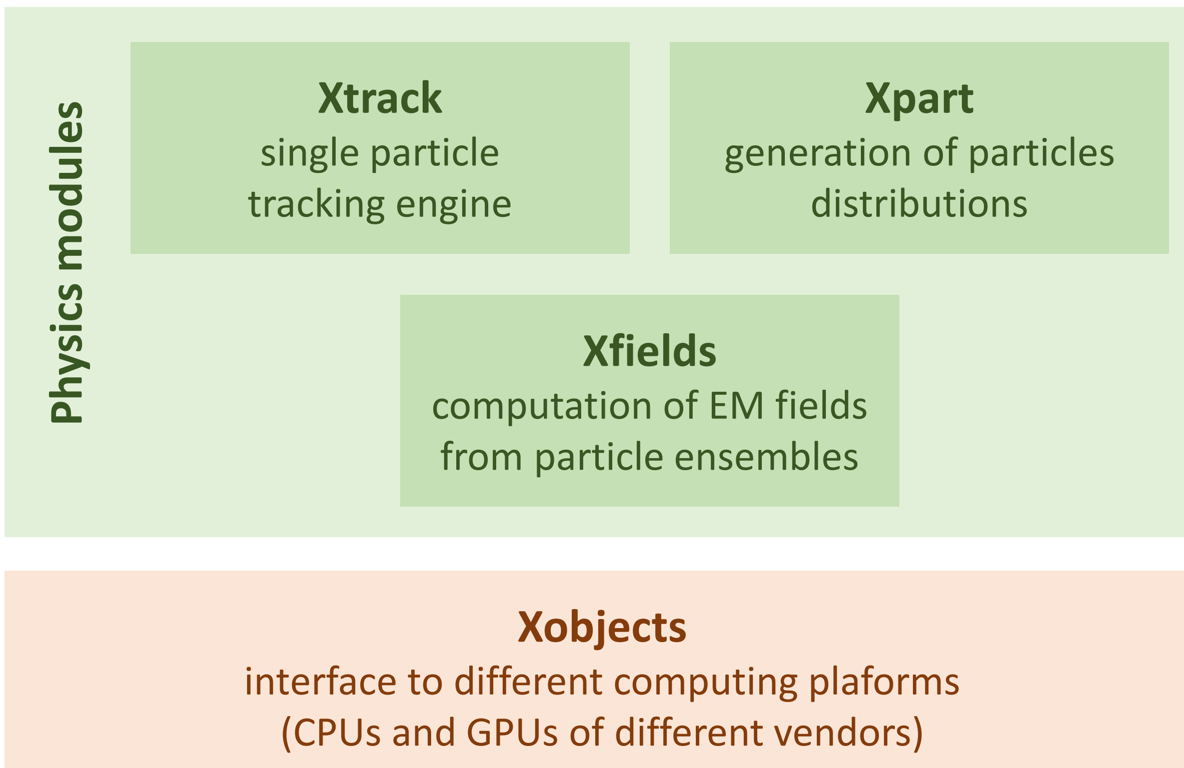
Outline

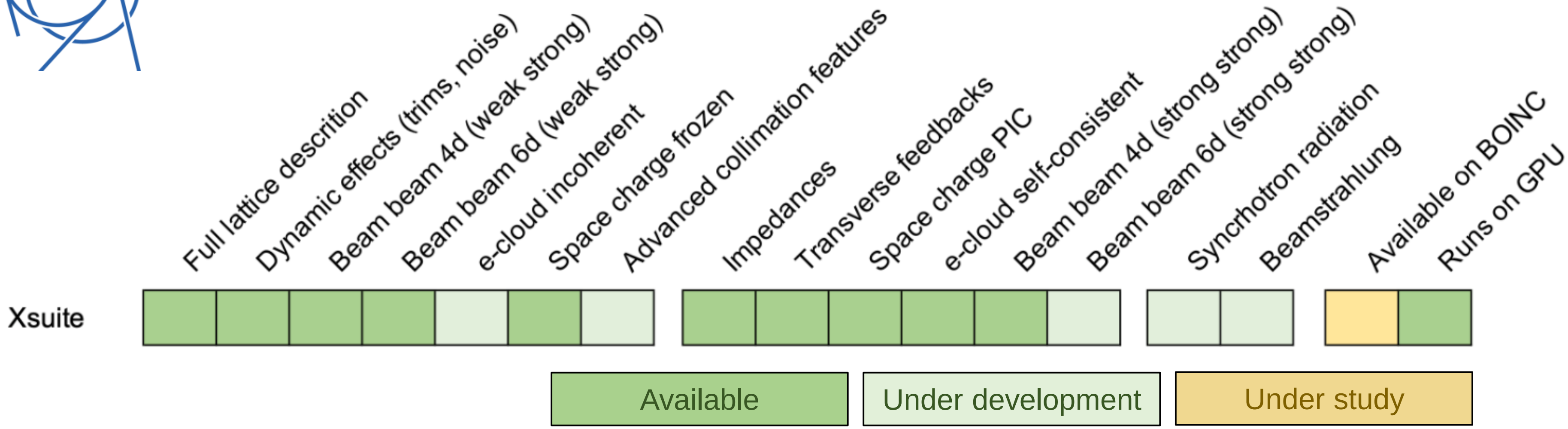
- **Introduction to Xsuite**
  - Motivation
  - Requirements
  - Design choices
  - Architecture
  - Development status
  - Documentation and developer's resources
- **Usage examples**
  - Single-particle tracking
  - Collective elements
  - Interface to other codes
- **Checks and first applications**
- **Final remarks and summary**

And the ecosystem is growing…

Already a few **spin-offs** from the community (some at early stages):

- **PyPLINE**: multilevel parallelization for strong-strong beam beam simulations)
- **Xdeps**: equivalent of MAD-X deferred expressions in python
- **Xsequence**: sequence manager for different codes (including knobs via xdeps), smart slicing, etc. (driven by EPFL collaborators for FCC-ee dev. efforts)
- **Xcollimation:** setup and post-processing of collimation simulations





Xsuite development **experience so far**:

- Shows **feasibility of integrated modular code** covering the application of our interest
- Demonstrates a **convenient approach to handle multiple computing platform** while keeping compact and readable physics code
- Already **being used for production runs** → gradually becoming our workhorse for tracking simulations
- **Very positive response from external collaborators** (EPFL team working on FCC-ee software, Gamma factory collaboration, GSI, SEEIIST)

You are very welcome to give it a try, give us feedback and contribute more features!

Thanks for your attention!

Test suite

- To verify that new modifications don't affect the functionality and correctness of existing features, **test suites are implemented for all modules**
  - Notably they include check of tracking results for LHC, HL-LHC and SPS
- Before releasing new versions of the code, the **tests are run on different computing platforms** (CPU and GPU)

```
=================================== test session starts ===================================
platform linux -- Python 3.8.5, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /home/giadarol/Desktop/20210303_xfields_gpudev/xobjects
plugins: cov-2.12.1
collected 58 items

tests/test_align.py .                                                              [  1%]
tests/test_array.py ............                                                   [ 22%]
tests/test_buffer.py ...........                                                   [ 41%]
tests/test_capi.py ......                                                          [ 51%]
tests/test_chunk.py .                                                              [ 53%]
tests/test_kernel.py ..                                                            [ 56%]
tests/test_nplike_arrays.py ...                                                    [ 62%]
tests/test_ref.py ..                                                               [ 65%]
tests/test_scalars.py ..                                                           [ 68%]
tests/test_strides.py ..                                                           [ 72%]
tests/test_string.py .....                                                         [ 81%]
tests/test_struct.py ........                                                      [ 94%]
tests/test_unionref.py ...                                                         [100%]

=================================== 58 passed in 7.71s ===================================
```

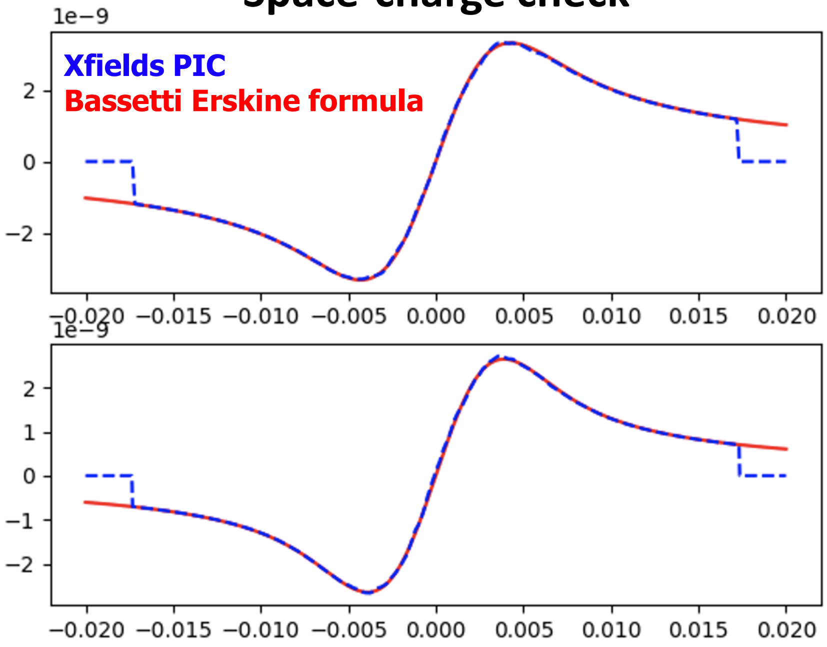
```
=================================== test session starts ===================================
platform linux -- Python 3.8.5, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /home/giadarol/Desktop/20210303_xfields_gpudev/xfields
plugins: cov-2.12.1
collected 7 items

tests/test_beambeam.py .                                                           [ 14%]
tests/test_cerrf.py ..                                                             [ 42%]
tests/test_mean_std.py .                                                           [ 57%]
tests/test_profiles.py .                                                           [ 71%]
tests/test_spacecharge.py ..                                                       [100%]

============================== 7 passed in 82.77s (0:01:22) ==============================
```

```
=================================== test session starts ===================================
platform linux -- Python 3.8.5, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /home/giadarol/Desktop/20210303_xfields_gpudev/xtrack
plugins: cov-2.12.1
collected 11 items

tests/test_aperture_turn_ele_and_monitor.py ..                                     [ 18%]
tests/test_collective_tracker.py .                                                 [ 27%]
tests/test_collimation_infrastructure.py .                                         [ 36%]
tests/test_dress.py ..                                                             [ 54%]
tests/test_elements.py ..                                                          [ 72%]
tests/test_full_rings.py .                                                         [ 81%]
tests/test_pyht_interface.py .                                                     [ 90%]
tests/test_random_gen.py .                                                         [100%]

============================ 11 passed in 273.03s (0:04:33) ============================
```

Space-charge – benchmarks and performance

- Different methods crosschecked against each other
- Particular care in optimizing performance on GPU

Space-charge check



Platform	Computing time
CPU	5.5 s
GPU (Titan V, cupy)	20 ms
GPU (Titan V, via pyopencl)	38 ms

(*) tests made on ABP GPU server for typical SPS space-charge interaction (PIC)

We did not start from scratch, instead we could **learn and inherit features** from the following existing tools:

sixtraklib-pysixtrack

- Clean, tested and documented implementation of **machine elements** (basically reused without changes, physics from SixTrack)
- **Particle description** with redundant energy variables for better precision and speed (from sixtrack experience)
- Experience with **multiplatform code** (CPU/GPU)
- Tools for **importing machine model** from MAD-X or sixtrack input

PyHEADTAIL

- Driving a multiparticle simulation through **Python**
- Usage of vectorization through **numpy** to speed up parts of the simulation directly in Python

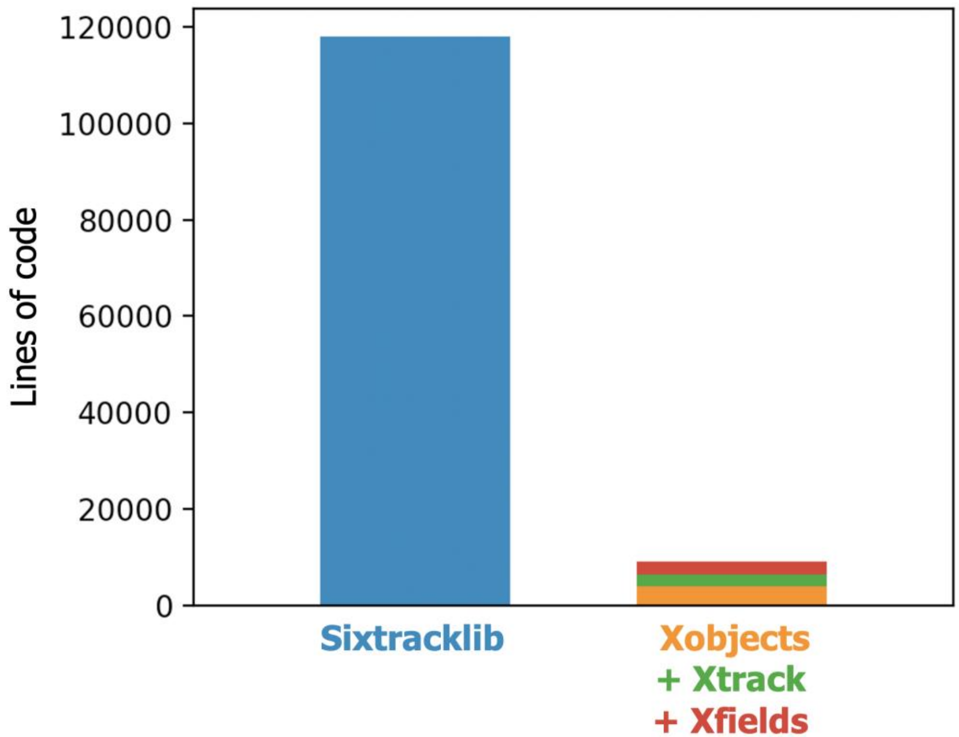
PyPIC

- **2D and 3D FFT Particle In Cell** with integrated Green functions
- Experience with **CPU and GPU**

Xsuite: code complexity

- Xsuite leverages **Python's flexibility** (introspection) and **massive code autogeneration** to **minimize code complexity**
  - Code is **compact and readable** (significant step forward w.r.t. Sixtracklib, where we had achieved multiplatform compatibility using pre-compiler macros)
  - A developer who knows the basics of Python and C can **easily contribute code** (e.g. introduce new beam elements)

→ **Fundamental** to guarantee future development and maintenance with **available manpower**!



- **Introduction to Xsuite**
  - Motivation
  - Requirements
  - Design choices
  - Architecture
  - Development status
  - Documentation and developer's resources
- **Usage examples**
  - Single-particle tracking
  - Import an existing lattice
  - Collective elements
  - PyHEADTAIL interface
- **Checks and first applications**
- **A look under the hood** (optional)
  - Multiplatform programming with Xobjects
- **Summary**

Xobjects – data manipulation in python

The main features of Xobjects can be illustrated with a simple **example** (Xsuite physics packages are largely based on the features illustrated here)

A **Xobjects Class** can be defined as follows:

```
import xobjects as xo

class DataStructure(xo.Struct):
    a = xo.Float64[:] # Array
    b = xo.Float64[:] # Array
    c = xo.Float64[:] # Array
    s = xo.Float64    # Scalar
```

An **instance of our class** can be instantiated on CPU or GPU by passing the appropriate context

```
# ctx = xo.ContextCpu()
ctx = xo.ContextCupy() # for NVIDIA GPUs

obj = DataStructure(_context=ctx,
                    a=[1,2,3], b=[4,5,6],
                    c=[0,0,0], s=0)
```

Independently on the context, the **object is accessible in read/write directly from Python**. For example:

```
print(obj.a[2]) # gives: 3
obj.a[2] = 10
print(obj.a[2]) # gives: 10
```

```python
# From before
class DataStructure(xo.Struct):
    a = xo.Float64[:]
    b = xo.Float64[:]
    c = xo.Float64[:]
    s = xo.Float64

# ctx = xo.ContextCpu() # CPU
ctx = xo.ContextCupy()  # GPU

obj = DataStructure(_context=ctx,
                    a=[1,2,3], b=[4,5,6],
                    c=[0,0,0], s=0)
```

The definition of a Xobject class in Python, **automatically triggers the generation of a set of functions (C-API)** that can be used in C code to access the data.

They can be inspected by:

```python
print(DataStructure._gen_c_decl(conf={}))
```

which gives (without the comments):

```c
// ...

// Get the length of the array DataStructure.a
int64_t DataStructure_len_a(DataStructure obj);

// Get a pointer to the array DataStructure.a
ArrNFloat64 DataStructure_getp_a(DataStructure obj);

// Get an element of the array DataStructure.a
double DataStructure_get_a(const DataStructure obj, int64_t i0);

// Set an element of the array DataStructure.a
void DataStructure_set_a(DataStructure obj, int64_t i0, double value);

// get a pointer to an element of the array DataStructure.a
double DataStructure_getp1_a(const DataStructure obj, int64_t i0);

// ... similarly for b, c and s
```

Xobjects – writing cross-platform C code

A **C function that can be parallelized when running** on GPU is called "Kernel".

Example: C function that computes obj.c = obj.a * obj.b

```
src = '''
/*gpukern*/
void myprod(DataStructure ob, int nelem){
   for (int ii=0; ii<nelem; ii++){ //vectorize_over ii nelem
      double a_ii = DataStructure_get_a(ob, ii);
      double b_ii = DataStructure_get_b(ob, ii);
      double c_ii = a_ii * b_ii;
      DataStructure_set_c(ob, ii, c_ii);
   }//end_vectorize
}
'''
```

```
# From before
class DataStructure(xo.Struct):
    a = xo.Float64[:]
    b = xo.Float64[:]
    c = xo.Float64[:]
    s = xo.Float64

# ctx = xo.ContextCpu() # CPU
ctx = xo.ContextCupy()  # GPU

obj = DataStructure(_context=ctx,
            a=[1,2,3], b=[4,5,6],
            c=[0,0,0], s=0)
```

Xobjects – writing cross-platform C code

A **C function that can be parallelized when running** on GPU is called "Kernel".

Example: C function that computes obj.c = obj.a * obj.b

```
src = '''
/*gpukern*/
void myprod(DataStructure ob, int nelem){
   for (int ii=0; ii<nelem; ii++){ //vectorize_over ii nelem
      double a_ii = DataStructure_get_a(ob, ii);
      double b_ii = DataStructure_get_b(ob, ii);
      double c_ii = a_ii * b_ii;
      DataStructure_set_c(ob, ii, c_ii);
   }//end_vectorize
}
'''
```

```
# From before
class DataStructure(xo.Struct):
    a = xo.Float64[:]
    b = xo.Float64[:]
    c = xo.Float64[:]
    s = xo.Float64

# ctx = xo.ContextCpu() # CPU
ctx = xo.ContextCupy()  # GPU

obj = DataStructure(_context=ctx,
            a=[1,2,3], b=[4,5,6],
            c=[0,0,0], s=0)
```

(Comments in red are Xobjects annotation, defining how to parallelize the code on GPU)

The Xobjects context compiles the function from python:

```
ctx.add_kernels(
    sources=[src],
    kernels={'myprod': xo.Kernel(
                args = [xo.Arg(DataStructure, name='ob'),
                        xo.Arg(xo.Int32, name='nelem')],
                n_threads='nelem')
            } )
```

The kernel can be easily called from Python and is executed on CPU or GPU based on the context:

```
# obj.a contains [3., 4., 5.] , obj.b contains [4., 5., 6.]
ctx.kernels.myprod(ob=obj, nelem=len(obj.a))
# obj.c contains [12., 20., 30.]
```

Before compiling, Xobjects **specializes the code** for the chosen computing platform.

- Specialization and compilation of the C code are **done at runtime** through Python, right before starting the simulation→ gives a lot of flexibility

Code written by the user

```
/*gpukern*/ void myprod(DataStructure ob, int nelem){

  for (int ii=0; ii<nelem; ii++){ //vectorize_over ii nelem

      double a_ii = DataStructure_get_a(ob, ii);
      double b_ii = DataStructure_get_b(ob, ii);
      double c_ii = a_ii * b_ii;
      DataStructure_set_c(ob, ii, c_ii);

  }//end_vectorize
}
```

Code specialized for CPU

```
void myprod(DataStructure ob, int nelem){

  for (int ii=0; ii<nelem; ii++){ //autovectorized

      double a_ii = DataStructure_get_a(ob, ii);
      double b_ii = DataStructure_get_b(ob, ii);
      double c_ii = a_ii * b_ii;
      DataStructure_set_c(ob, ii, c_ii);

  }//end autovectorized
}
```

Code specialized for GPU (OpenCL)

```
__kernel void myprod(DataStructure ob, int nelem){

  int ii; //autovectorized
  ii=get_global_id(0); //autovectorized

      double a_ii = DataStructure_get_a(ob, ii);
      double b_ii = DataStructure_get_b(ob, ii);
      double c_ii = a_ii * b_ii;
      DataStructure_set_c(ob, ii, c_ii);

  //end autovectorized
}
```

Xobjects – code specialization

Before compiling, Xobjects **specializes the code** for the chosen computing platform.

- Specialization and compilation of the C code are **done at runtime** through Python, right before starting the simulation→ gives a lot of flexibility

Code written by the user

```
/*gpukern*/ void myprod(DataStructure ob, int nelem){

  for (int ii=0; ii<nelem; ii++){ //vectorize_over ii nelem

      double a_ii = DataStructure_get_a(ob, ii);
      double b_ii = DataStructure_get_b(ob, ii);
      double c_ii = a_ii * b_ii;
      DataStructure_set_c(ob, ii, c_ii);

  }//end_vectorize
}
```

Code specialized for CPU

```
void myprod(DataStructure ob, int nelem){

  for (int ii=0; ii<nelem; ii++){ //autovectorized

     double a_ii = DataStructure_get_a(ob, ii);
     double b_ii = DataStructure_get_b(ob, ii);
     double c_ii = a_ii * b_ii;
     DataStructure_set_c(ob, ii, c_ii);

  }//end autovectorized
}
```

Code specialized for GPU (Cuda)

```
__global__ void myprod(DataStructure ob, int nelem){
    int ii; //autovectorized
    ii=blockDim.x * blockIdx.x + threadIdx.x; //au
    if (ii<nelem){

      double a_ii = DataStructure_get_a(ob, ii);
      double b_ii = DataStructure_get_b(ob, ii);
      double c_ii = a_ii * b_ii;
      DataStructure_set_c(ob, ii, c_ii);

    }//end autovectorized
}
```

- **Introduction to Xsuite**
  - Motivation
  - Requirements
  - Design choices
  - Architecture
  - Development status
  - Documentation and developer's resources
- **Usage examples**
  - Single-particle tracking
  - Import an existing lattice
  - Collective elements
  - PyHEADTAIL interface
- **Checks and first applications**
- **A look under the hood** (optional)
  - Multiplatform programming with Xobjects
- **Summary**