Xsuite

R. De Maria and G. Iadarola

Contributions to development and testing by:
A. Abramov, F. Asvesta, H. Bartosik, P. Belanger, X. Buffat, M. Boucard, F. Carlier, D. Demetriadou, D. Di Croce, P. Hermes, S. Kostoglou, A. Fornara, P. Kicsini, E. Lamb, A. Latina, C. E. Montanari, K. Paraschou, A. Poyet, T. Pugnat, V. Rodin, M. Schwinzerl, G. Simon, G. Sterbini, F. Van der Veken, M. Zampetakis

Many thanks to:
M. Giovannozzi, T. Persson, T. Pieloni, G. Rumolo, Y. Papaphilippou, S. Redaelli, R. Tomas

https://xsuite.readthedocs.io

Outline

Outline

Multiparticle codes developed in BE-ABP

	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	🟩	🟩	🟩	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥
Sixtrack	🟩	🟩	🟩	🟩	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥
Sixtracklib	🟩	🟥	🟩	🟩	🟩	🟩	🟥	🟥	🟥	🟨	🟥	🟥	🟥	🟥	🟥	🟥	🟩
PyHEADTAIL	🟥	🟩	🟩	🟥	🟩	🟩	🟥	🟩	🟩	🟩	🟩	🟥	🟥	🟩	🟥	🟥	🟨
COMBI	🟥	🟩	🟩	🟩	🟥	🟩	🟥	🟩	🟩	🟥	🟥	🟩	🟩	🟩	🟥	🟥	🟥

🟩 Available 🟥 Not available 🟨 Experimental

In BE-ABP we have at least **five internally developed 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

Multiparticle codes developed in BE-ABP

	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	🟩	🟩	🟩	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥
Sixtrack	🟩	🟩	🟩	🟩	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥
Sixtracklib	🟩	🟥	🟩	🟩	🟩	🟩	🟥	🟥	🟥	🟨	🟥	🟥	🟥	🟥	🟥	🟥	🟩
PyHEADTAIL	🟥	🟩	🟩	🟥	🟩	🟩	🟥	🟩	🟩	🟩	🟩	🟥	🟥	🟩	🟥	🟥	🟨
COMBI	🟥	🟩	🟩	🟩	🟥	🟩	🟥	🟩	🟩	🟥	🟥	🟩	🟩	🟩	🟥	🟥	🟥

🟩 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 advanced features
- **Summary**

Xsuite – Python packages

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 BE-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**

Xsuite – architecture

Modular structure (Python packages)

Physics modules

Xpart
generation of particles distributions

Xtrack
single particle tracking engine

Xfields
computation of EM fields from particle ensembles

Xdeps
Dependency manager, deferred expressions

Xcoll
Collimation methods

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

CFFI PyOpenCL CuPy

Lower level libraries
(external, open source)

intel AMD NVIDIA

Hardware

Outline

Xsuite development status

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

🟩	Available
⬜	In development
🟥	Not available
🟨	Experimental

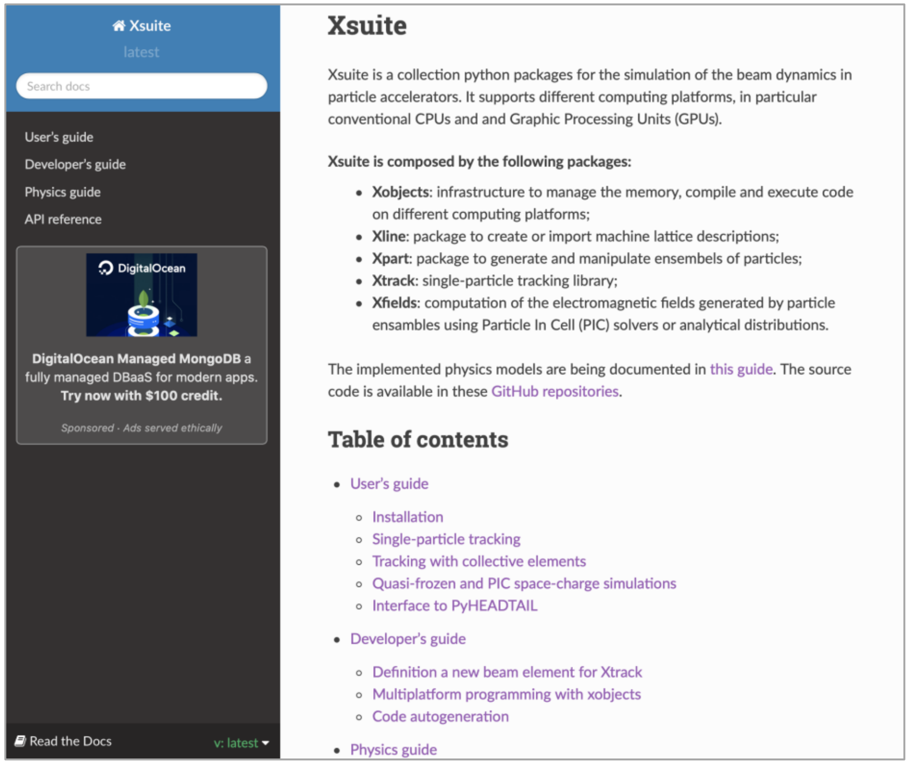
	Full lattice descrition	Dynamic effects (trims, noise)	Beam beam 4d (weak strong)	Beam beam 6d (weak strong)	e-cloud incoherent	Space charge frozen	Advanced collimation features	Impedances	Transverse feedbacks	Space charge PIC	e-cloud self-consistent	Beam beam 4d (strong strong)	Beam beam 6d (strong strong)	Synchrotron radiation	Beamstrahlung	Available on BOINC	Runs on GPU
MAD-X track	🟩	🟩	🟩	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥
Sixtrack	🟩	🟩	🟩	🟩	🟥	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟥
Sixtracklib	🟩	🟥	🟩	🟩	🟩	🟩	🟥	🟥	🟥	🟨	🟥	🟥	🟥	🟥	🟥	🟥	🟩
PyHEADTAIL	🟥	🟩	🟩	🟥	🟩	🟩	🟥	🟩	🟩	🟩	🟩	🟥	🟥	🟩	🟥	🟥	🟨
COMBI	🟥	🟩	🟩	🟩	🟥	🟩	🟥	🟩	🟩	🟥	🟥	🟩	🟩	🟩	🟥	🟥	🟥
Xsuite	🟩	🟩	🟩	🟩	🟩	🟩	🟩/⬜	🟩	🟩	🟩	🟩	🟩	⬜	🟩	⬜	⬜	🟩

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 advanced features
- **Summary**

User's guide

- 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


Xsuite
latest
Search docs
User's guide
Developer's guide
Physics guide
API reference
DigitalOcean
DigitalOcean Managed MongoDB a fully managed DBaaS for modern apps. Try now with $100 credit.
Sponsored · Ads served ethically
Read the Docs
v: latest

Xsuite

Xsuite is a collection python packages for the simulation of the beam dynamics in particle accelerators. It supports different computing platforms, in particular conventional CPUs and and Graphic Processing Units (GPUs).

Xsuite is composed by the following packages:

- **Xobjects**: infrastructure to manage the memory, compile and execute code on different computing platforms;
- **Xline**: package to create or import machine lattice descriptions;
- **Xpart**: package to generate and manipulate ensembels of particles;
- **Xtrack**: single-particle tracking library;
- **Xfields**: computation of the electromagnetic fields generated by particle ensembles using Particle In Cell (PIC) solvers or analytical distributions.

The implemented physics models are being documented in this guide. The source code is available in these GitHub repositories.

Table of contents

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 advanced features
- **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)

```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 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)

```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 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)

```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...
```

Access to the recorded particles coordinates

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...
```

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)

```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.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

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)
```

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

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)
```

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 advanced features
- **Summary**

Interface to other codes

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

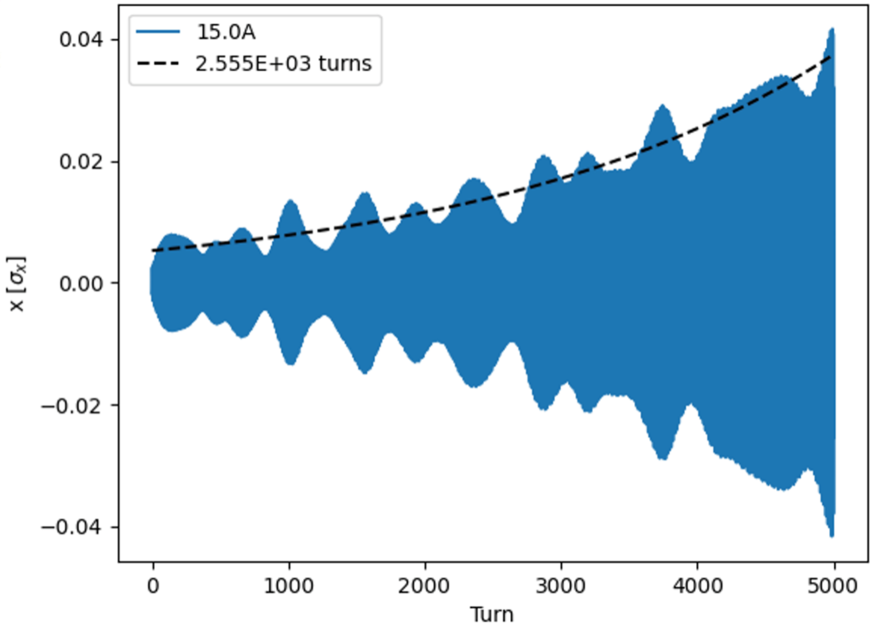
```
import xtrack as xt
xt.enable_pyheadtail_interface()

## 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)
```

Interface to other codes

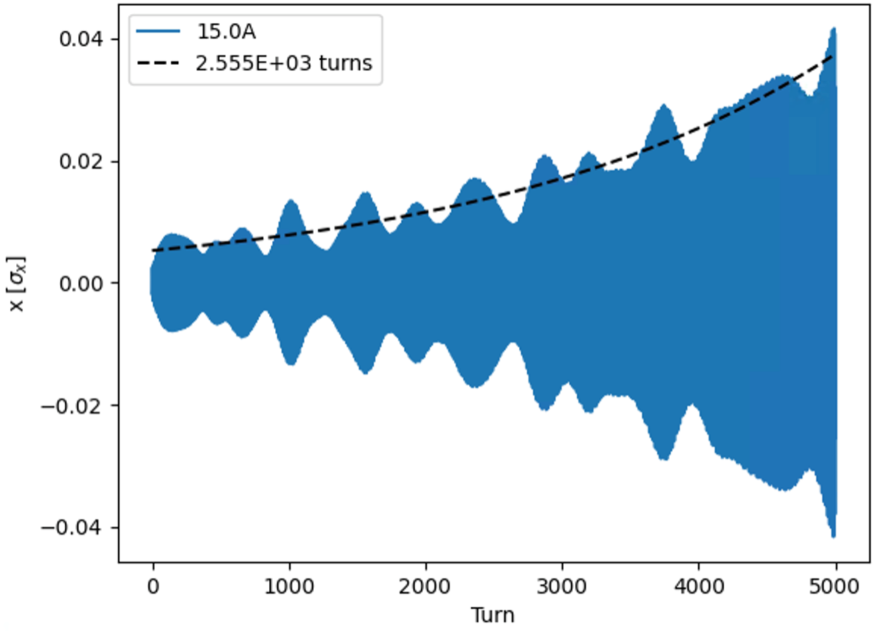
X. Buffat

Comparison

Tracking, impedance and damper in PyHEADTAIL



Tracking Xsuite impedance and in PyHEADTAIL

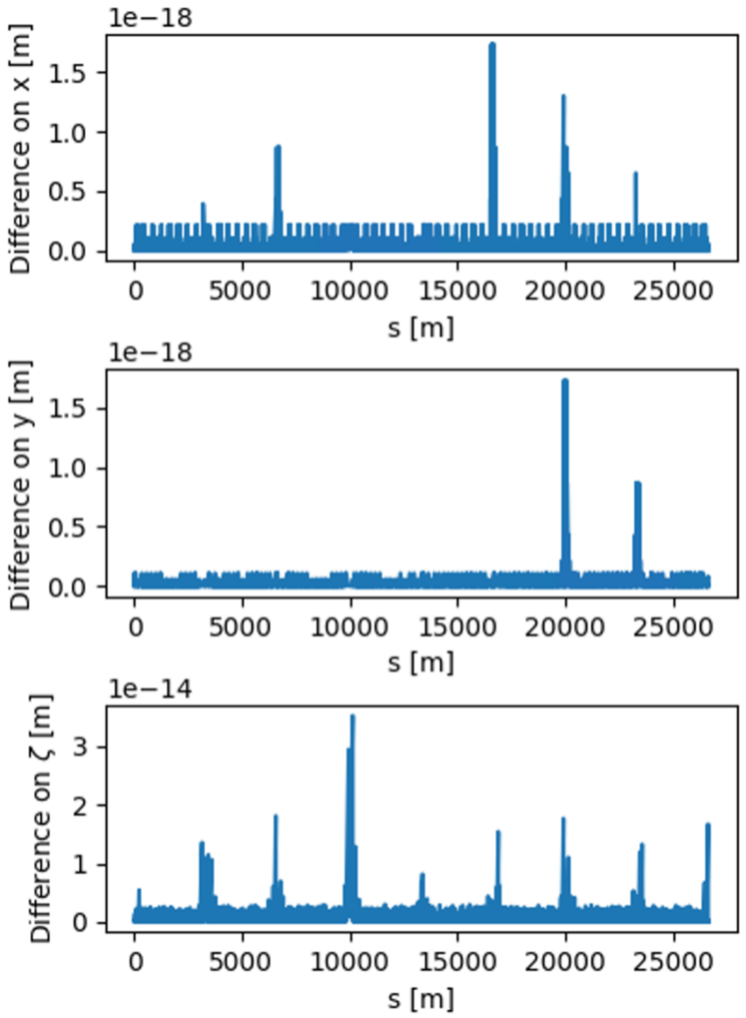


```
                damper,
                xt.Multipole(knl=[0, -1.]), xt.Drift(length=1.)]
    element_names = ['qf1', 'drift1', 'damper', 'qd1', 'drift2'])

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

Outline

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
  - ○ Still something to gain for very ideal lattices (no errors) and with beam-beam



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

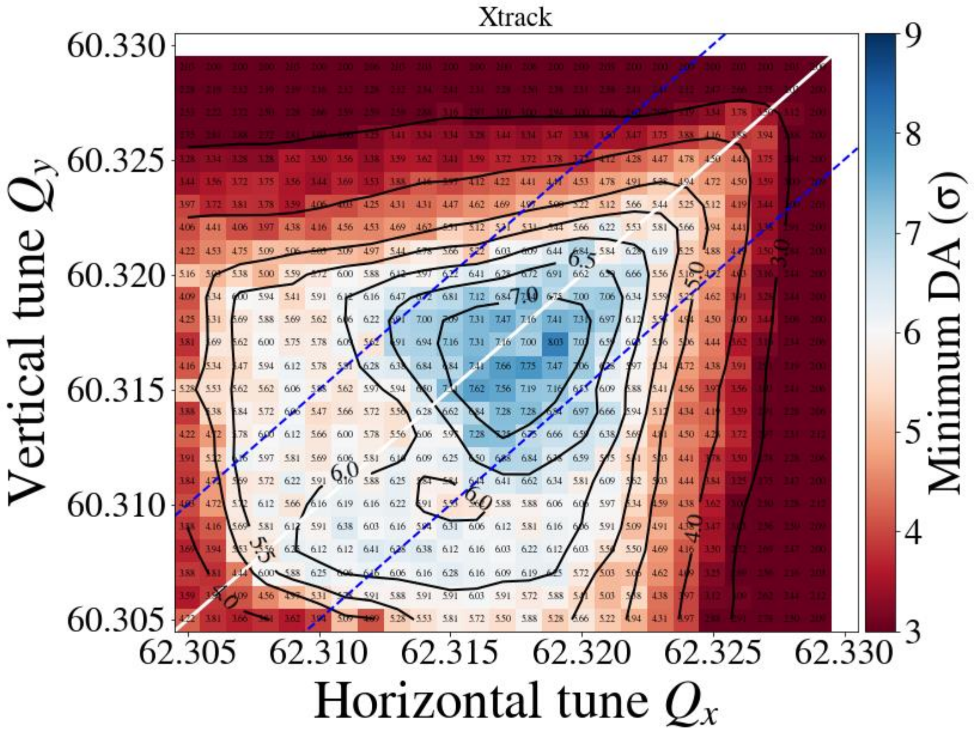
Based on past work from M. Schwinzerl

HL-LHC Dynamic Aperture scan

G. Sterbini, S. Kostoglou

First DA studies with Xsuite. Package used in combination with **other Pythonic tools:**

- **Pymask** used to prepare the machine configurations
- **Job management** using a new **Python package (TreeMaker)**
- **Dynamic Aperture computation** in Python using **Pandas**



Parameters of pilot study

N. jobs = ~10'000

Comp. time ~48h on INFN- CNAF cluster

Full HL-LHC lattice (20k elements)

Weak strong Beam-beam

N. tune configurations = 625

N. tracked particles/conf. = 1780

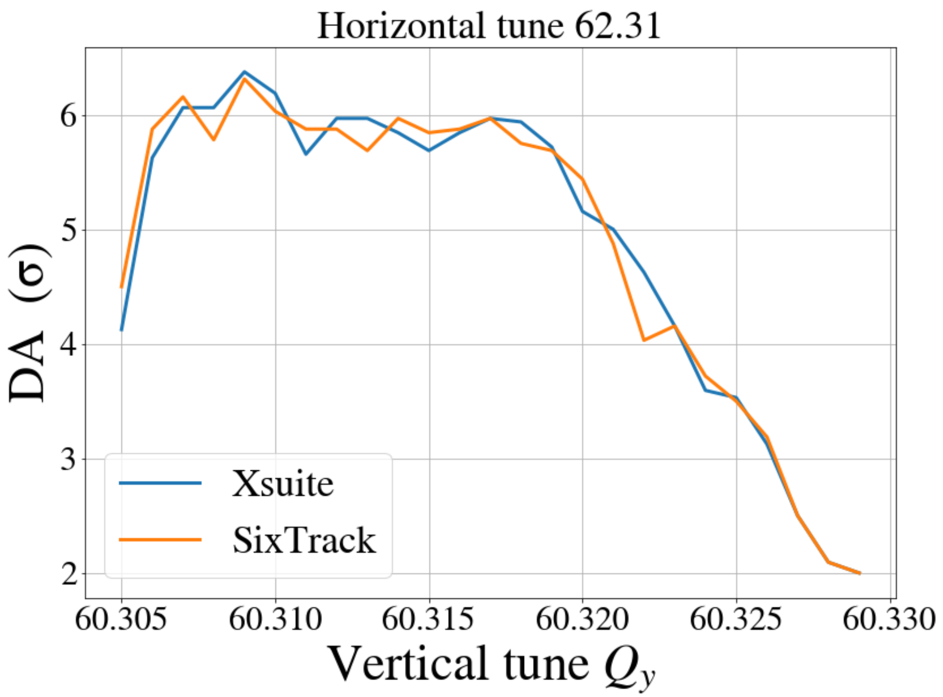
N. turns = 10^6

HL-LHC Dynamic Aperture scan

G. Sterbini, S. Kostoglou

First DA studies with Xsuite. Package used in combination with **other Pythonic tools:**

- **Pymask** used to prepare the machine configurations
- **Job management** using a new **Python package (TreeMaker)**
- **Dynamic Aperture computation** in Python using **Pandas**



Parameters of pilot study

N. jobs = ~10'000

Comp. time ~48h on INFN- CNAF cluster

Full HL-LHC lattice (20k elements)

Weak strong Beam-beam

N. tune configurations = 625

N. tracked particles/conf. = 1780

N. turns = 10^6

R&D studies on Dynamic Aperture

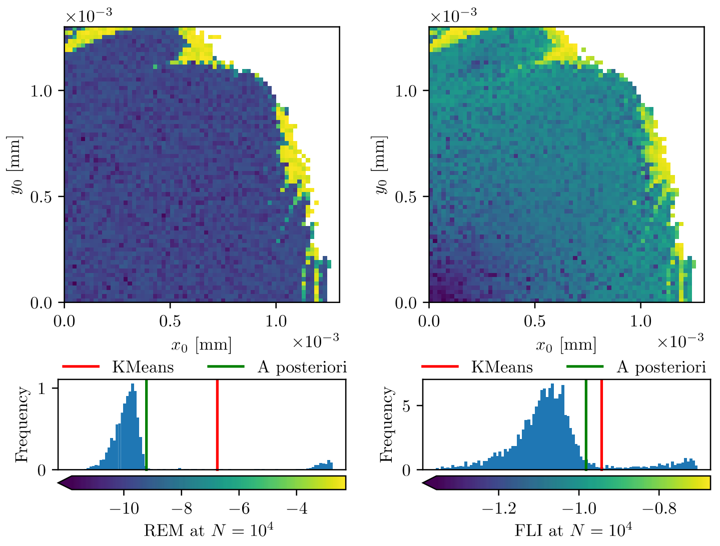
C. E. Montanari, F. Van der Veken, A. Bazzani, M. Giovannozzi, G. Turchetti

- Testing predictive capacity on chaotic behaviors with **different dynamic indicators**:
  - Classical indicators from accelerator community and more exotic indicators used in cosmology and astrophysics
- Use of **machine learning for smart sampling** of the phase space

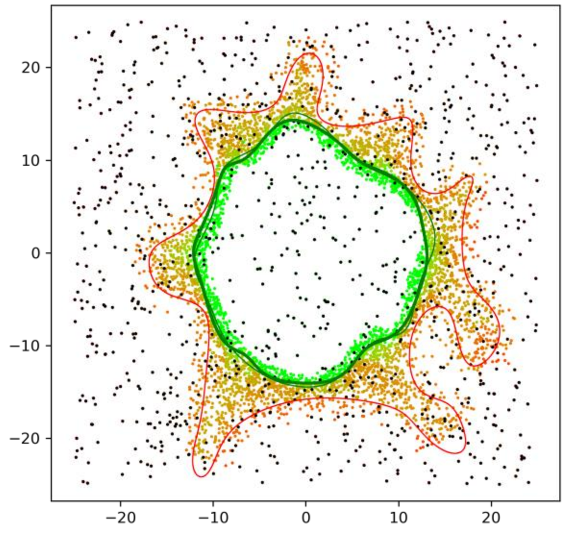
Xsuite instrumental for these studies since it provides:

- Easy configuration of "ghost particles" used as probes for the dynamic indicators
- Easy implementation of unphysical actions, e.g. displacement renormalization every n turn
- Fast parallel tracking on large amounts of particles on GPU

Comparison of dynamic indicators (HL-LHC)



Smart sampling with ML



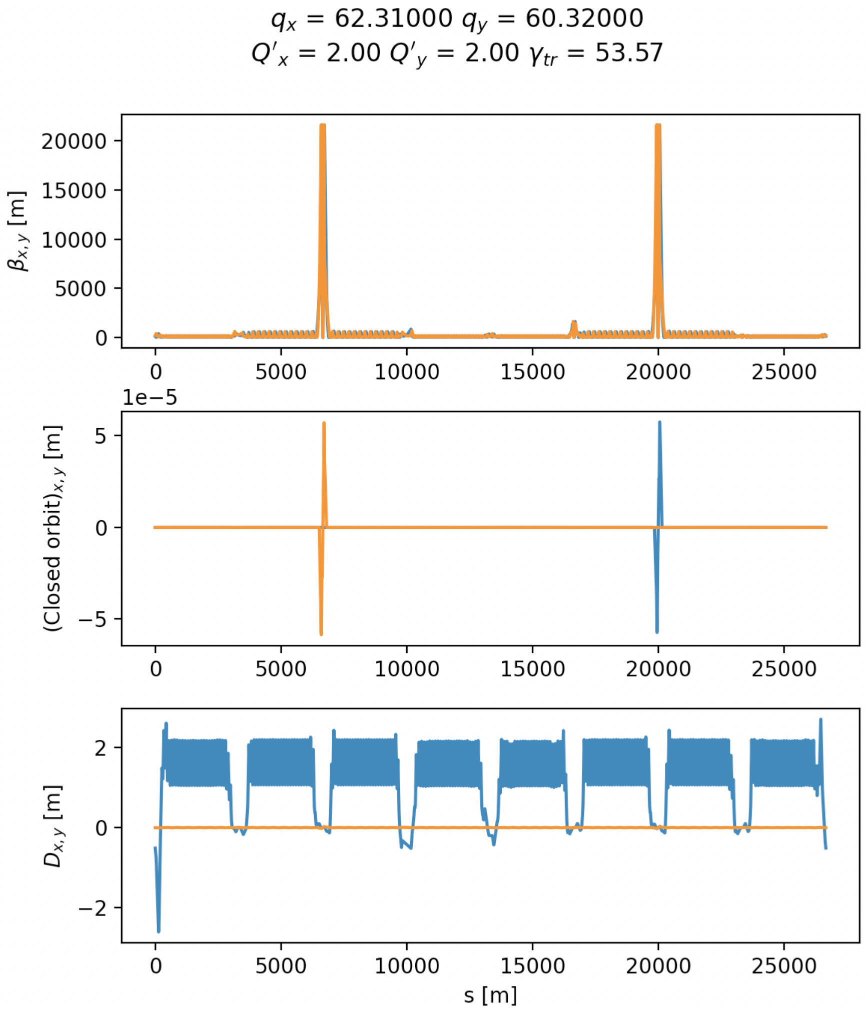
Twiss

A **twiss method** was introduced in Xtrack to compute **closed orbit**, linear optics and other parameters.

How it works:

- Searches **6D closed orbit** (using scipy fsolve)
- Computes **first order one-turn matrix** using **finite differences** around the closed orbit
- Diagonalization, identification and sorting of the **eigenmodes**
- **Track particles associated to the eigenvectors** around the machine
- Compute **twiss parameters** from the eigenvalues/eigenvectors
- Repeat off momentum for chromaticity

Tested for HL-LHC, PSB, ELENA, Elettra, CLIC-DR, FCC-ee → found to be very accurate

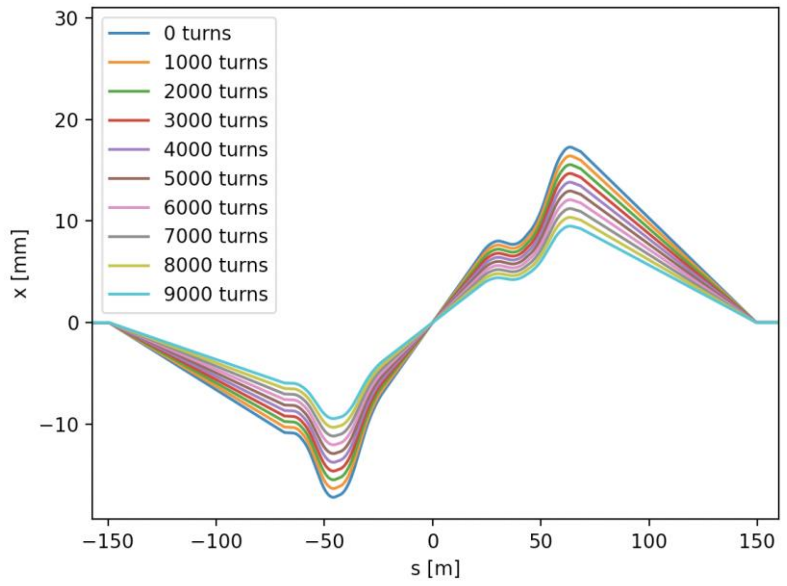


Xdeps

The **Xdeps package** can be used to **import deferred expression from MAD-X model** to Xsuite simulations.

- Knobs imported from MAD-X can be **easily changed before the simulations** (configure the machine) **or during the simulation** to model transients, ripples, noise

```
# Simulate change of crossing angle from 300 urad
# to 25 urad in 10000 turns

num_turns = 10000
phi_table = np.linspace(150, 300, 10000)

for i_turn in range(num_turns):
     tracker.vars['on_x1'] = phi_table[i_turn]
     tracker.vars['on_x5'] = phi_table[i_turn]
     tracker.track(particles)
```



Xdeps

The package provides tools to **analyze dependencies introduced by deferred expressions**:

```
# For example we can che how the dipole corrector 'mcbyv.4r1.b1' is controlled:
print(tracker.element_refs['mcbxfah.3r1'].knl[0]._expr)
# ---> returns "(-vars['acbxh3.r1'])"

# We can see that the variable controlling the corrector is in turn controlled
# by an expression involving several other variables:
print(tracker.vars['acbxh3.r1']._expr)
# ---> returns
#         (((((((-3.529000650090648e-07*vars['on_x1hs'])
#         -(1.349958221397232e-07*vars['on_x1hl']))
#          +(1.154711348310621e-05*vars['on_sep1h']))
#          +(1.535247516521591e-05*vars['on_o1h']))
#          -(9.919546388675102e-07*vars['on_a1h']))
#          +(3.769003853335184e-05*vars['on_ccpr1h']))
#           +(1.197587664190056e-05*vars['on_ccmr1h']))

# The list of variables cotrolling the selected variable can be found by:
print(tracker.vars['acbxh3.r1']._expr._get_dependencies())
# ---> returns {vars['on_ccpr1h'], vars['on_x1hs'], vars['on_x1hl'],
#               vars['on_ccmr1h'], vars['on_sep1h'], vars['on_o1h'],
#               vars['on_a1h']}

# It is possible to get the list of all entities controlled by a given
# variable by using the method `_find_dependant_targets`:
tracker.vars['on_x1']._find_dependant_targets()
# ---> returns
#         [vars['on_x1'],
#          vars['on_x1hl'],
#          vars['on_dx1hl'],
#          vars['on_x1hs'],
#          vars['acbxh3.l1'],
#          element_refs['mcbxfah.3l1'],
#          element_refs['mcbxfah.3l1'].knl[0],
#          element_refs['mcbxfah.3l1'].knl,
#            ...............
```

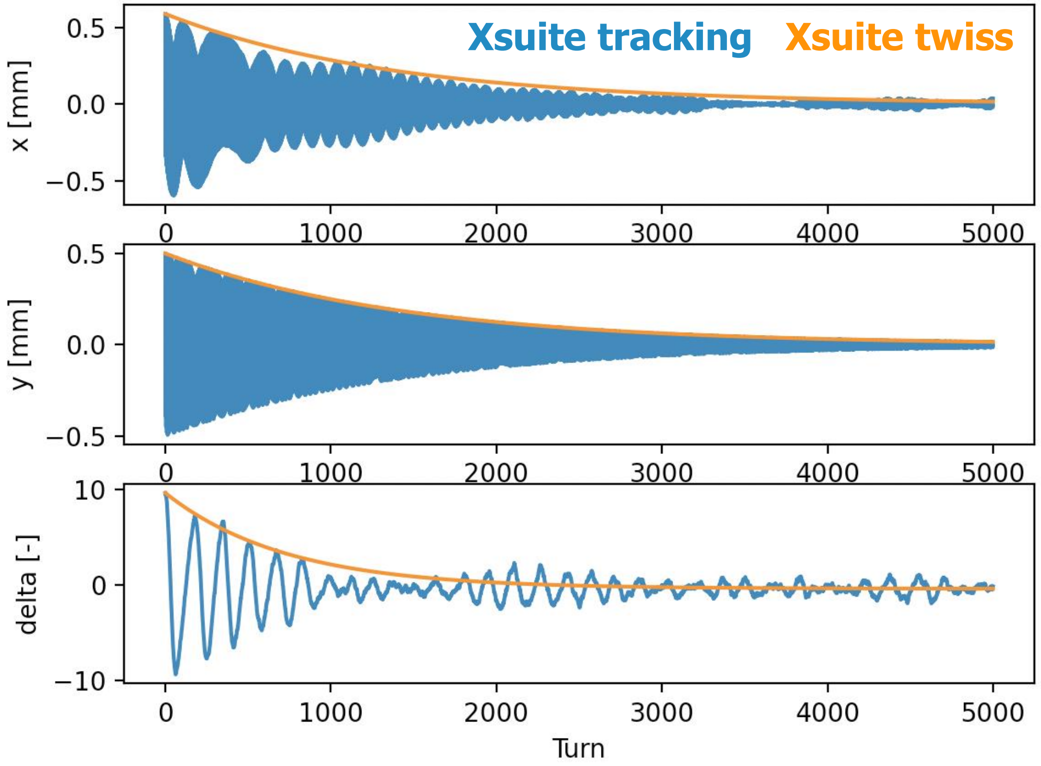
Synchrotron radiation

A. Abramov, A. Latina, A. Poyet, G. Simon, M. Zampetakis

Synchrotron radiation emission introduced for **thin magnetic elements** (no solenoid yet), largely based on MAD-X and PLACET implementations. User can choose between:

- **Mean power emission** → only damping
- **Quantum description** (emission of individual photons) → damping + excitation

Xtrack twiss computes energy loss and **damping constants** (from one-turn matrix eigenvalues) → checked against MAD-X



Method	Damping constant $\alpha_t[1/s]$
MAD-X EMIT Thick	196.3
MAD-X TRACK Thick	196.3
MAD-X TRACK Thin (after fix)	198.4
MAD-X Twiss thin using $D = \frac{\oint k_0 D_x (k_1 + k_0^2)\, ds}{\oint k_0^2\, ds}$ $\alpha_t = \frac{W_0}{2E_0T_0}(2+D)$	198.2
Xtrack	**198.2**

Study supported by CHART

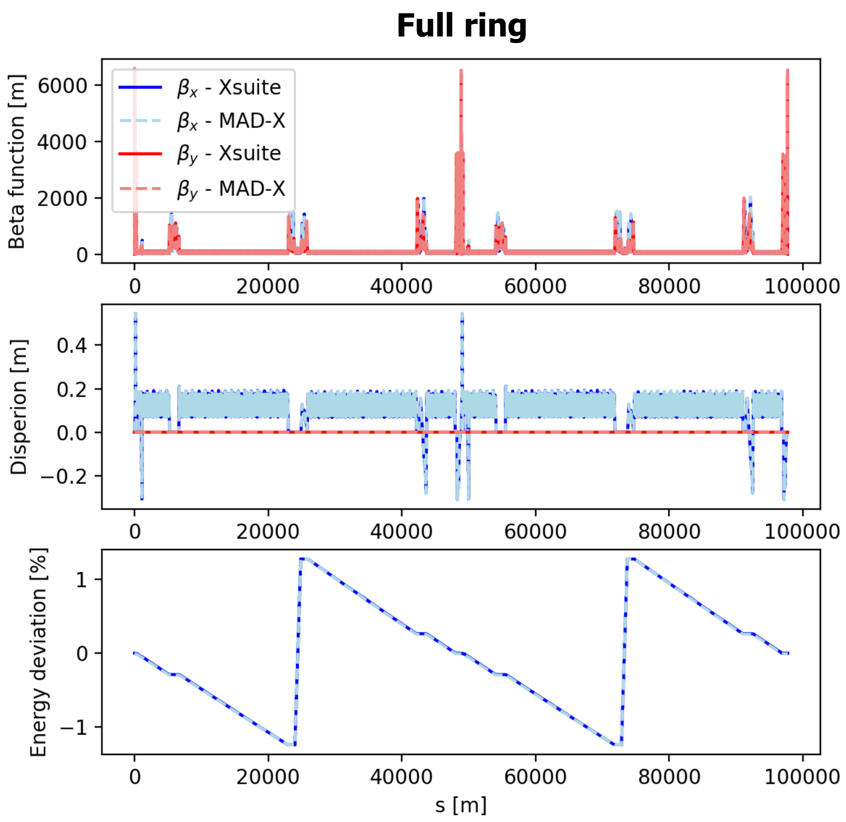
Synchrotron radiation

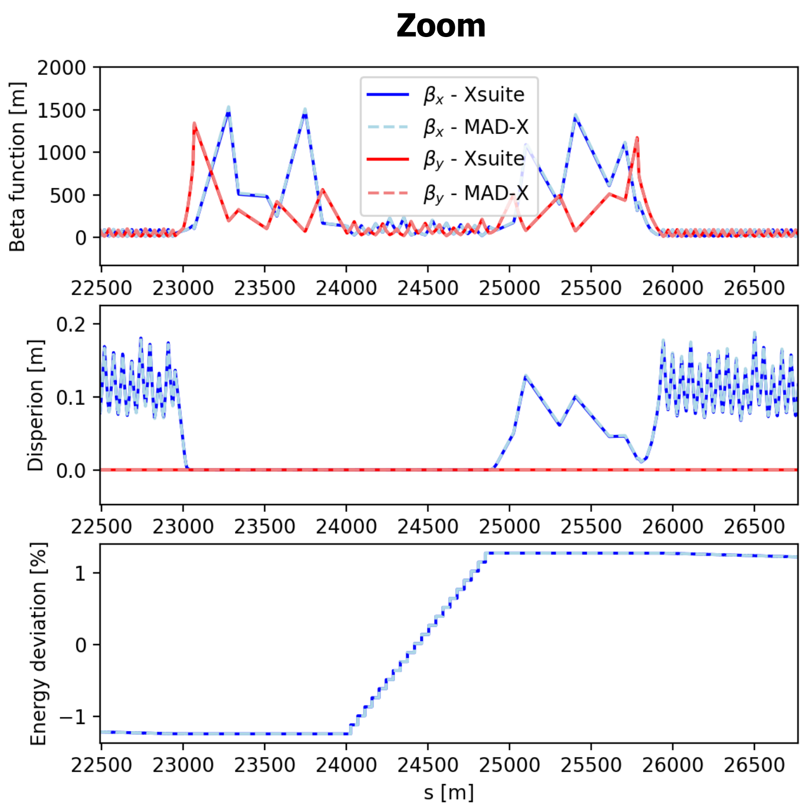
A. Abramov, A. Latina, A. Poyet, G. Simon, M. Zampetakis

Tested in **extreme configuration of FCC-ee** (tt mode, 175 GeV)

- ~2% energy loss, tapered lattice

Excellent agreement vs MAD-X on closed orbit and linear optics





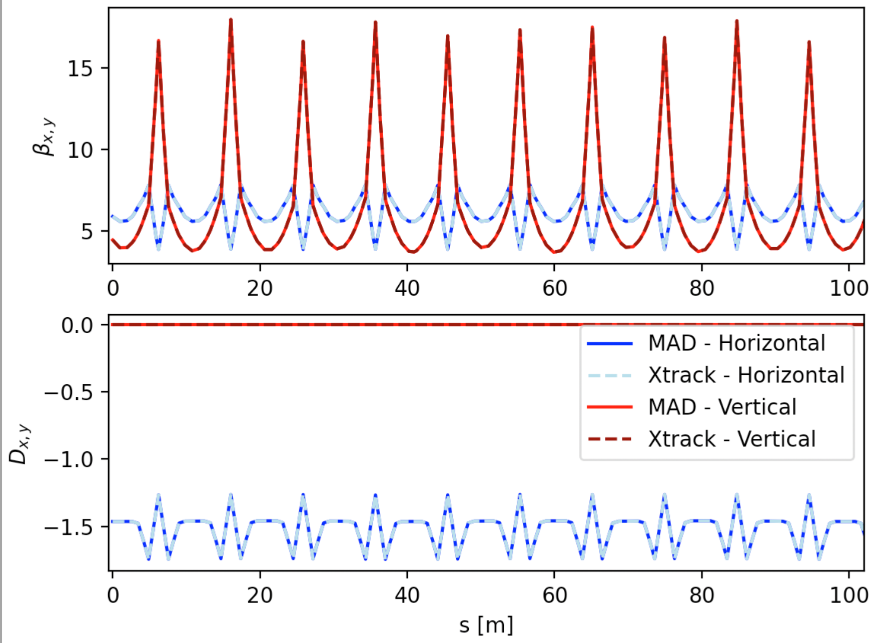
Low energy accelerator tests: PSB and ELENA

F. Asvesta, V. Rodin

Xsuite track and twiss tested also **for low energy machines** → check that effect or relativistic beta is correctly modelled

PSB

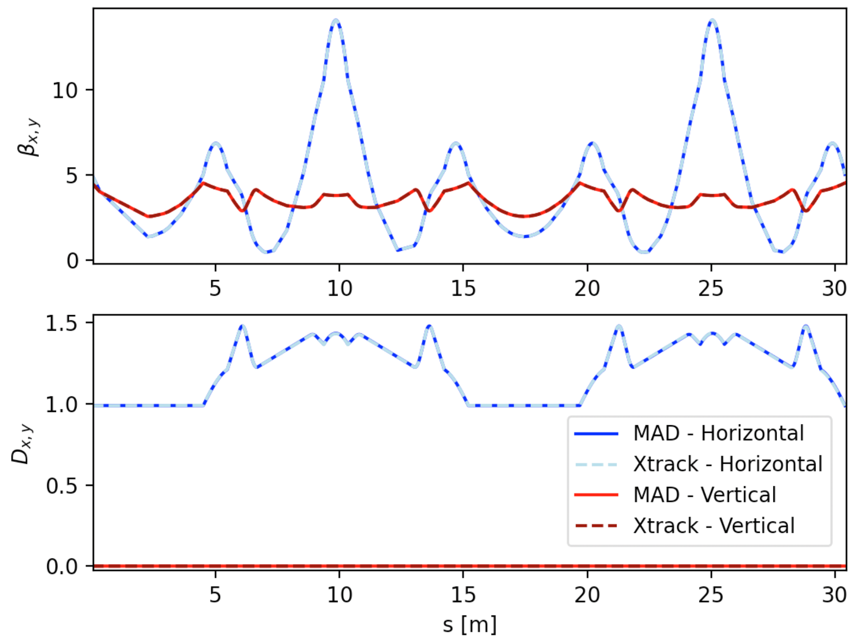
Protons, E_{kin} = 160 MeV, β = 0.520



```
MAD-X:  Qx  = 4.15000000   Qy  = 4.51000000
Xsuite: Qx  = 4.15000002   Qy  = 4.50999998

MAD-X:  Q'x = -3.5533572   Q'y = -7.1875114
Xsuite: Q'x = -3.5532929   Q'y = -7.1875384

MAD-X:  alpha_p = 0.0590347
Xsuite: alpha_p = 0.0590353
```

ELENA

Antiprotons, E_{kin} = 100 keV, β = 0.0146

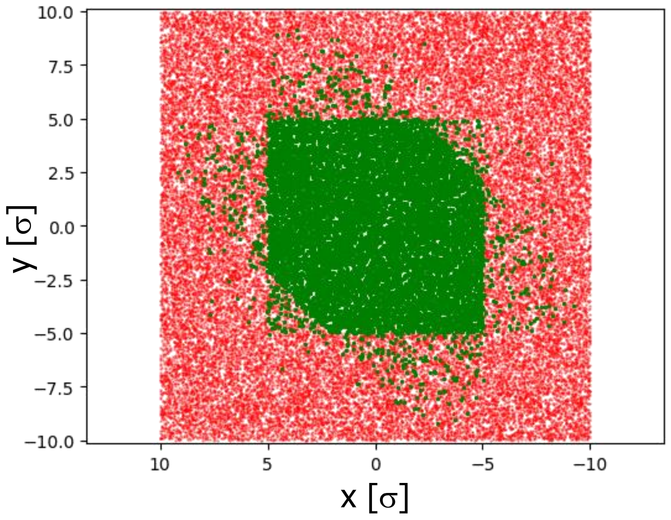


```
MAD-X:  Qx  = 2.45400000   Qy  = 1.41600000
Xsuite: Qx  = 2.45403134   Qy  = 1.41600000

MAD-X:  Q'x = -3.5579063   Q'y = -0.9370119
Xsuite: Q'x = -3.5563392   Q'y = -0.9366512

MAD-X:  alpha_p = 0.2512849
Xsuite: alpha_p = 0.2515006
```

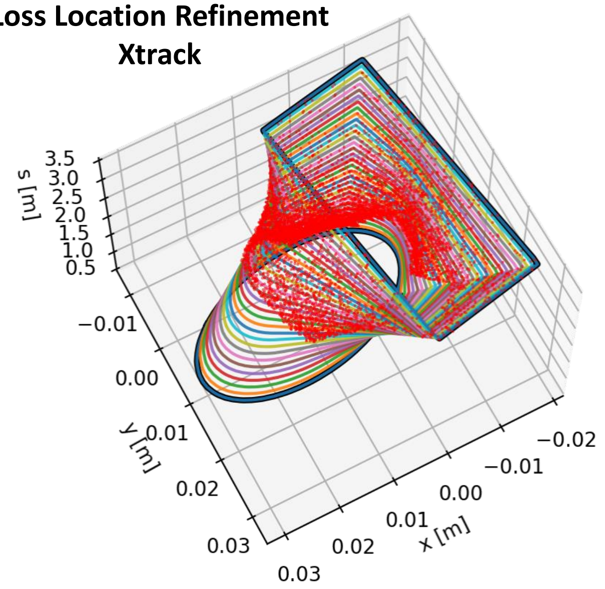
Collimation simulations

A. Abramov , D. Demetriadou, F. Van Der Veken

Xcoll - K2 simulation
(LHC primary collimators)



Loss Location Refinement Xtrack



The **Xcoll package** is being developed to manage and simulate collimators within an Xsuite beamline.

- It **installs collimators** in an Xsuite beamlines
- It **configures the collimator gaps** based **settings in sigmas** provided by the user (uses beam sizes from Xsuite twiss)
- It runs different **engines simulating particle-matter interaction** inside collimator jaw
    - **K2**: ported from Sixtrack, being translated in python (almost done)
    - **Geant4-BDSIM**: tested in full loss-map studies for HL-LHC and FCC-ee (developed in dedicated package, still to be be integrated in Xcoll)
    - **FLUKA**: coupling still to be ported from Sixtrack

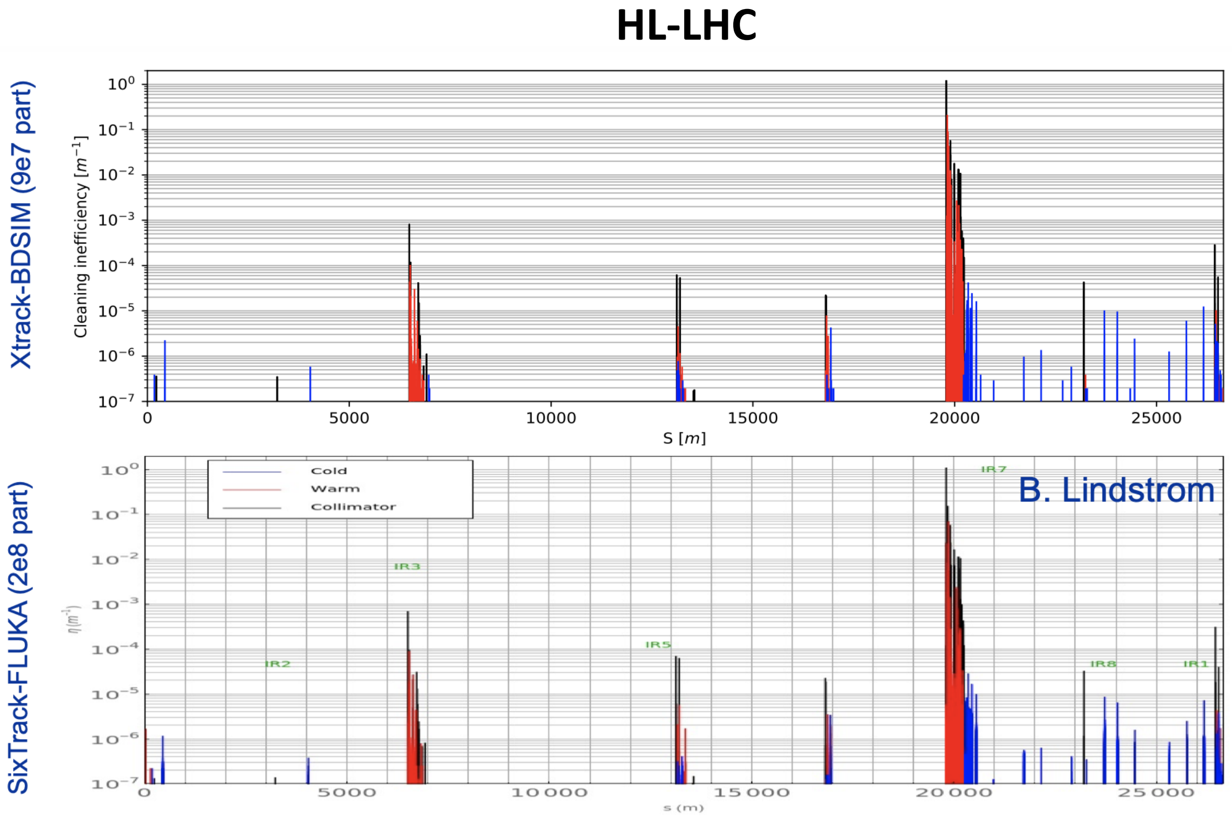
Collimation studies also require **precise localization of particles lost in the aperture**

- Dedicated module implemented in Xtrack for this purpose

Collimation simulations

A. Abramov , D. Demetriadou, F. Van Der Veken

First **loss-map simulations** conducted with **Xsuite + BDSIM/Geant4** for HL-LHC and FCC-ee



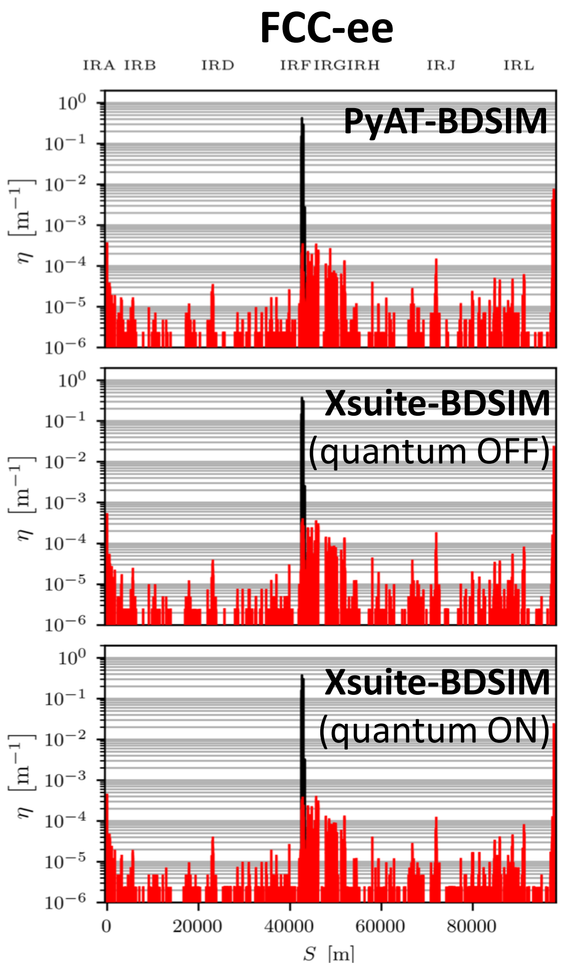


Figure 2: Loss map for collimation losses in the full FCC-ee ring, showing results from pyAT-BDSIM (top) and Xtrack-BDSIM (middle) with only radiation damping enabled, and Xtrack-BDSIM (bottom) with quantum fluctuations enabled.

For more details: https://indico.cern.ch/event/112293/

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**
- **A look under the hood** (optional)
  - Multiplatform programming with Xobjects
- **Summary and final remarks**

Summary

Xsuite development **experience so far**:

- Shows **feasibility of integrated modular code** covering a wide range of applications
- 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!

Thanks for your attention!

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**
- **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:

```python
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

```python
# 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:

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

Xobjects – data access from C

```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.]
```

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 (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
}
```