Read the Docs Template Documentation

Release 0.1

Read the Docs

Aug 31, 2021

CONTENTS

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 ensambles 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.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 User's guide

1.1.1 Installation

We assume that you have a recent python installation (python 3.7+). It this is not the case you can make one following the dedicated section on *how to get a miniconda installation*.

Table of Contents

Basic installation

The Xsuite packages can be cloned from GitHub and installed with pip:

```
$ git clone https://github.com/xsuite/xobjects
$ pip install -e xobjects

$ git clone https://github.com/xsuite/xline
$ pip install -e xline

$ git clone https://github.com/xsuite/xpart
$ pip install -e xpart

$ git clone https://github.com/xsuite/xtrack
```

(continues on next page)

(continued from previous page)

```
$ pip install -e xtrack

$ git clone https://github.com/xsuite/xfields
$ pip install -e xfields
```

(The installation without the `-e` option is still untested).

This installation allows using Xsuite on CPU. To use Xsuite on GPU, with the cupy and/or pyopencl you need to install the corresponding packages, as described in the *dedicated section*.

Optional dependencies

To import MAD-X lattices you will need the cpymad package, which can be installed as follow:

```
$ pip install cpymad
```

To import lattices from a set of sixtrack input files (fort.2, fort.3, etc.) you will need the sixtracktools package, which can be installed as follow:

```
$ git clone https://github.com/sixtrack/sixtracktools
$ pip install -e sixtracktools
```

Some of the tests rely on pyheadtail to test the corresponding interface:

```
$ git clone https://github.com/pycomplete/pyheadtail
$ pip install cython
$ pip install -e pyheadtail
```

GPU support

In the following section we describe the steps to install the two supported GPU platforms, i.e. cupy and pyopencl.

Installation of cupy

In order to use the *cupy context*, the cupy package needs to be installed. In Anacoda or Miniconda (if you don't have Anaconda or Miniconda, see dedicated section on *how to get a miniconda installation*) this can be done as follows for example for CUDA version 10.1.243:

```
$ conda install mamba -n base -c conda-forge
$ pip install cupy-cuda101
$ mamba install cudatoolkit=10.1.243
```

Remember to check your CUDA version e.g. via `$ nvcc --version` and use the appropriate tag.

Installation of PyOpenCL

In order to use the *pyopencl context*, the PyOpenCL package needs to be installed. In Anacoda or Miniconda this can be done as follows:

```
$ conda config --add channels conda-forge
$ conda install pyopencl
```

Check that there is an OpenCL installation in the system:

```
$ ls /etc/OpenCL/vendors
```

Make the OpenCL installation visible to pyopencl:

```
$ conda install ocl-icd-system
```

For the PyOpenCL context we will need the gpyfft and the clfft libraries. For this purpose we need to install cython.

```
$ pip install cython
```

Then we can install clfft.

```
$ conda install -c conda-forge clfft
```

We locate the library and headers here:

```
$ ls ~/miniconda3/pkgs/clfft-2.12.2-h83d4a3d_1/
# gives: include  info  lib
```

(Or locate the directory via `find $(dirname $(dirname $(type -P conda)))/pkgs -name "clfft*" -type d` .)

We obtain gpyfft from github:

```
$ git clone https://github.com/geggo/gpyfft
```

and we install gpyfft with pip providing extra flags as follows:

```
$ pip install --global-option=build_ext --global-option="-I/home/giadarol/miniconda3/
↪pkgs/clfft-2.12.2-h83d4a3d_1/include" --global-option="-L/home/giadarol/miniconda3/
↪pkgs/clfft-2.12.2-h83d4a3d_1/lib" gpyfft/
```

Alternatively (if the command above does not work) we can edit the `setup.py` of gpyfft to provide the right paths to your clfft installation (and potentially the OpenCL directory of your platform):

```
if 'Linux' in system:
    CLFFT_DIR = os.path.expanduser('~/miniconda3/pkgs/clfft-2.12.2-h83d4a3d_1/')
    CLFFT_LIB_DIRS = [r'/usr/local/lib64']
    CLFFT_INCL_DIRS = [os.path.join(CLFFT_DIR, 'include'), ] # remove the 'src' part
    CL_INCL_DIRS = ['/opt/rocm-4.0.0/opencl/include']
```

And install gpyfft locally.

```
$ pip install -e gpyfft/
```

Install Miniconda

If you don't have a miniconda installation, you can quickly get one ready for xsuite installation with the following steps.

On Linux

```
$ cd ~
$ wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
$ bash Miniconda3-latest-Linux-x86_64.sh
$ source miniconda3/bin/activate
$ pip install numpy scipy matplotlib pandas ipython pytest
```

On MacOS

```
$ cd ~
$ curl https://repo.anaconda.com/miniconda/Miniconda3-latest-MacOSX-x86_64.sh >␣
↪miniconda_inst.sh
$ bash miniconda_inst.sh
$ source miniconda3/bin/activate
$ conda install clang_osx-64
$ pip install numpy scipy matplotlib pandas ipython pytest
```

1.1.2 Single-particle tracking

This page describes the basic usage of Xsuite to perform tracking simulations. Instructions on how to install Xsuite are provided in the dedicated *installation page*.

Table of Contents

A simple example

A simple tracking simulation can be configured and executed with the following python code. More details on the different steps will be discussed in the following section.

```
import numpy as np

import xobjects as xo
import xline as xl
import xtrack as xt

## Generate a simple sequence
sequence = xl.Line(
    elements=[xl.Drift(length=2.),
              xl.Multipole(knl=[0, 1.], ksl=[0,0]),
              xl.Drift(length=1.),
              xl.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
# context = xo.ContextCupy()       # For CUDA GPUs
# context = xo.ContextPyopencl()   # For OpenCL GPUs

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

## Build particle object on context
n_part = 200
particles = xt.Particles(_context=context,
                         p0c=6500e9,
                         x=np.random.uniform(-1e-3, 1e-3, n_part),
                         px=np.random.uniform(-1e-5, 1e-5, n_part),
                         y=np.random.uniform(-2e-3, 2e-3, n_part),
                         py=np.random.uniform(-3e-5, 3e-5, 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)
n_turns = 100
tracker.track(particles, num_turns=n_turns,
              turn_by_turn_monitor=True)

## Turn-by-turn data is available at:
tracker.record_last_track.x
tracker.record_last_track.px
# etc...
```

Step-by-step description

In this sections we will discussed in some more detail the difference steps outlined in the example above.

Getting the Xline machine model

The first step to perform a tracking simulation consists in creating or importing the lattice description of a ring or a beam line.

This is done with the Xline package, which allows:

- creating a lattice directly in python script
- importing the lattice from a MAD-X model
- importing the lattice from a set of Sixtrack input files (fort.2, fort.3, etc.)

These three options will be briefly described in the following.

We can create a simple lattice in python as follows:

```
import xline as xl

sequence = xl.Line(
    elements=[xl.Drift(length=2.),
              xl.Multipole(knl=[0, 1.], ksl=[0,0]),
              xl.Drift(length=1.),
              xl.Multipole(knl=[0, -1.], ksl=[0,0])],
    element_names=['drift_0', 'quad_0', 'drift_1', 'quad_1'])
```

The lattice can be manipulated in python after its creation. For example we can change the strength of the first quadrupole as follows:

```
q1 = sequence.elements[1]
q1.knl = 2.
```

Importing a MAD-X lattice

Xline can import a MAD-X lattice using the cpymad interface of MAD-X.

Assuming that we have a MAD-X script called `myscript.madx` that creates and manipulates (e.g. matches) a thin sequence called "lhcb1", we can execute the script using cpymad and import transform the sequence into and Xline object using the following instructions:

```
import xline as xl
from cpymad.madx import Madx

mad = Madx()
mad.call("mad/lhcwbb.seq")

line = xl.Line.from_madx_sequence(mad.sequence['lhcb1'])
```

Importing lattice from sixtrack input

Xline can import a lattice from a set of sixtrack input files using the sixtracktools package.

Assuming that we have a sixtrack input files (fort.2, fort.3, etc.) in a folder called `sixtrackfiles` we can import the lattice using the following instructions:

```
import xline as xl
import sixtracktools as st

sequence = xl.Line.from_sixinput(st.sixinput('./sixtrackfiles'))
```

Once a Xline lattice is available, it can be used to track particles CPU or GPU.

Create a Context (CPU or GPU)

The first step consists in choosing the hardware on which the simulation will run as xsuite can run on different kinds of hardware (CPUs and GPUs). The user selects the hardware to be used by creating a *context object*, that is then passed to all other Xsuite components.

To run on conventional CPUs you need the context is created with the following instructions:

```
import xobjects as xo
context = xo.ContextCpu()
```

Similarly to run on GPUs using cupy or pyopenl you can use one of the following:

```
context = xo.ContextCupy()
```

```
context = xo.ContextPyopencl()
```

Create an Xtrack tracker object

An Xtrack tracker object needs to be created to track particles on the chosen computing platform (defined by the context) using the Xline sequence created or imported as described above:

```
import xtrack as xt
tracker = xt.Tracker(_context=context, sequence=sequence)
```

This step transfers the machine model to the required platform and compiles the required tracking code.

Generate particles to be tracked

The particles to be tracked can be allocated on the chosen platform using the following instruction (in this example particle coordinates are randomly generated):

```
import numpy as np
n_part = 100
particles = xt.Particles(_context=context,
                         p0c=6500e9,
                         x=np.random.uniform(-1e-3, 1e-3, n_part),
```

(continues on next page)

(continued from previous page)

```
                        px=np.random.uniform(-1e-5, 1e-5, n_part),
                        y=np.random.uniform(-2e-3, 2e-3, n_part),
                        py=np.random.uniform(-3e-5, 3e-5, n_part),
                        zeta=np.random.uniform(-1e-2, 1e-2, n_part),
                        delta=np.random.uniform(-1e-4, 1e-4, n_part),
                        )
```

The coordinates of the particle object are accessible with the conventional python syntax. For example to access the x coordinate of the particle 20, one can use the following instruction:

```
particles.x[20]
```

Track particles

The tracker object can now be used to track the generated particles over the specified lattice for an arbitrary number of turns:

```
num_turns = 100
tracker.track(particles, num_turns=num_turns)
```

This returns the particles state after 100 revolutions over the lattice.

Record turn-by-turn data

Optionally the particles coordinates can be saved at each turn. This feature can be activated when calling the tracking method:

```
n_turns = 100
tracker.track(particles, num_turns=n_turns,
              turn_by_turn_monitor=True)
```

The data can be retrieved as follows:

```
tracker.record_last_track.x # Shape is (n_part, n_turns)
tracker.record_last_track.px
# etc...
```

1.1.3 Tracking with collective elements

A collective beam element is an element that needs access to the entire particle set (in read and/or write mode). The following example shows how to handle such elements in Xsuite.

Example

A typical example of collective element is a space-charge interaction. We can create a space-charge beam element as follows:

```
import xobjects as xo
import xfields as xf

context = xo.ContextCpu()

spcharge = xf.SpaceChargeBiGaussian(_context=context,
    update_on_track = ['sigma_x', 'sigma_y'], length=2,
    longitudinal_profile=xf.LongitudinalProfileQGaussian(
        _context=context, number_of_particles=1e11, sigma_z=0.2))
```

This creates a space-charge element where the transverse beam sizes are updated based on the particle set at each interaction. Such an element can be included in an *xtrack tracker* similarly to single-particle elements.

```
import xline as xl
import xtrack as xt

## Generate a simple sequence including the spacecharge element
myqf = xl.Multipole(knl=[0, 1.])
myqd = xl.Multipole(knl=[0, -1.])
mydrift = xl.Drift(length=1.)
sequence = xl.Line(
    elements = [myqf, mydrift, myqd, mydrift,
                spcharge,
                myqf, mydrift, myqd, mydrift,],
    element_names = ['qf1', 'drift1', 'qd1', 'drift2',
                        'spcharge'
                        'qf2', 'drift3', 'qd2', 'drift4'])

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

## Build particle object on context
n_part = 200
particles = xt.Particles(_context=context,
                         p0c=6500e9,
                         x=np.random.uniform(-1e-3, 1e-3, n_part),
                         px=np.random.uniform(-1e-5, 1e-5, n_part),
                         y=np.random.uniform(-2e-3, 2e-3, n_part),
                         py=np.random.uniform(-3e-5, 3e-5, 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)
n_turns = 100
tracker.track(particles, num_turns=n_turns,
              turn_by_turn_monitor=True)
```

How does it work?

To decide whether or not an element needs to be treated as collective, the tracker inspects its `iscollective` attribute. In our example:

```
print(qf.iscollective)
# Gives "False"

print(spcharge.iscollective)
# Gives "True"
```

Based in this information the sequence is devided in parts that are either collective elements or xtrack trackers simulating groups of consecutive non-collective elements.

We can visualize this in our example:

```
print(tracker._parts)
# Gives:
# [<xtrack.tracker.Tracker object at 0x7f5ba8ce7760>,
#  <xfields.beam_elements.spacecharge.SpaceChargeBiGaussian object at 0x7f5ba8e1bd30>,
#  <xtrack.tracker.Tracker object at 0x7f5ba8ce7610>]
```

where the first part tracks the particles through to the first potion of the machine up to the space-charge element, the second part simulates the space-charge interaction, the third part trackes the particles from the space-charge element to the end of the sequence.

As all xsuite and xsuite-compatible beam elements need to expose a `.track` method the instruction:

```
tracker.track(particles)
```

is equivalent to the loop:

```
for pp in tracker._parts:
    pp.track(particles)
```

Any python object exposing a '.track' method can be used as beam_element. If the attribute `iscollective` is not present the element is handled as collective.

1.1.4 Quasi-frozen and PIC space-charge simulations

Xfields provides tools to configure qausi-frozen and Particle-In-Cell space-charge simulations by automatically replacing in an Xline sequence the frozen space-charge lenses with the corresponding collective beam elements. This is illustrated in the following example.

Example

Import modules

We import all the required modules

```
import json
import numpy as np
```

(continues on next page)

(continued from previous page)

```
import xobjects as xo
import xline as xl
import xpart as xp
import xtrack as xt
import xfields as xf
```

Machine model

For this example we load from the SPS Xtrack test_data folder a sequence with frozen space-charge lenses together with the corresponding particle on the closed orbit and linearized one-turn matrix. The same folder contains also example code to generate these files from the MAD-X model of the accelerator.

```
fname_sequence = ('xtrack/test_data/sps_w_spacecharge/'
                  'line_with_spacecharge_and_particle.json')

fname_optics = ('xtrack/test_data/sps_w_spacecharge/'
                'optics_and_co_at_start_ring.json')

with open(fname_sequence, 'r') as fid:
    seq_dict = json.load(fid)
with open(fname_optics, 'r') as fid:
    co_opt_dict = json.load(fid)

sequence = xl.Line.from_dict(seq_dict['line'])
part_on_co = xp.Particles.from_dict(co_opt_dict['particle_on_madx_co'])
RR = np.array(co_opt_dict['RR_madx']) # Linear one-turn matrix
```

Choice of the context

We choose the hardware on which we want to run by buildind an Xobjects context:

```
context = xo.ContextCupy()
#context = xo.ContextPyopencl('0.0')
#context = xo.ContextCpu()
```

Configuration quasi-frozen or PIC space-charge elements

We use the Xfields functions `replace_spaceharge_with_quasi_frozen` or `replace_spaceharge_with_PIC` to replace the frozen space-charge lenses with PIC or quasi-frozen collective elements:

```
mode = 'pic' # Can be 'pic', 'quasi-frozen' or 'forzen'

if mode == 'frozen':
    pass # Already configured in line
elif mode == 'quasi-frozen':
    xf.replace_spaceharge_with_quasi_frozen(
                sequence, _buffer=context.new_buffer(),
                update_mean_x_on_track=True,
```

(continues on next page)

(continued from previous page)

```
                update_mean_y_on_track=True)
elif mode == 'pic':
    pic_collection, all_pics = xf.replace_spaceharge_with_PIC(
                _context=context, sequence=sequence,
                n_sigmas_range_pic_x=8,
                n_sigmas_range_pic_y=8,
                nx_grid=256, ny_grid=256, nz_grid=100,
                n_lims_x=7, n_lims_y=3,
                z_range=(-0.7, 0.7))
else:
    raise ValueError(f'Invalid mode: {mode}')
```

Build Xtrack tracker

We build an Xtrack tracker:

```
tracker = xt.Tracker(_context=context,
                     sequence=sequence)
```

As discussed *here*, the tracker is built in such a way that particles are tracked asynchronously by separate threads in the non-collective sections of the sequence and are regrouped at each collescive element (in our case the PIC or quasi-forzen space-charge lenses).

Generation of matched particle set

We use Xpart to generate a matched particle distribution and we transfer it to the context:

```
part = xp.generate_matched_gaussian_bunch(
        num_particles=int(1e6), total_intensity_particles=1e11,
        nemitt_x=2.5e-6, nemitt_y=2.5e-6, sigma_z=22.5e-2,
        particle_on_co=part_on_co, R_matrix=RR,
        circumference=6911., alpha_momentum_compaction=0.0030777,
        rf_harmonic=4620, rf_voltage=3e6, rf_phase=0)

# Transfer particles to context
xtparticles = xt.Particles(_context=context, **part.to_dict())
```

Simulate

The simulation can be started by calling the `track` method of the tracker:

```
tracker.track(xtparticles, num_turns=3)
```

A *ParticlesMonitor* object can be passed to the track method to record all or a fraction of the particles coordinated.

1.1.5 Interface to PyHEADTAIL

PyHEADTAIL elements cannot be natively used by an Xsuite tracker due to different naming conventions for the particles coordinates. A specific interface has been introduced in Xsuite which indtroduces additional properties in the Particles objects in order to make them compatible with PyHEADTAIL beam elements. The interface can be enabled by calling the function `nable_pyheadtail_interface` right after importing xtrack, as illustrated in the following example.

```
import pathlib
import json
import numpy as np

import xobjects as xo
import xline as xl
import xtrack as xt
xt.enable_pyheadtail_interface()

fname_sequence = '../../test_data/lhc_no_bb/line_and_particle.json'
num_turns = int(100)
n_part = 200

######################
# Choose a context #
######################

context = xo.ContextCpu()

##################
# Get a sequence #
##################

with open(fname_sequence, 'r') as fid:
    input_data = json.load(fid)
sequence = xl.Line.from_dict(input_data['line'])

#########################
# Add PyHEADTAIL damper #
#########################

from PyHEADTAIL.feedback.transverse_damper import TransverseDamper
damper = TransverseDamper(dampingrate_x=10., dampingrate_y=15.)
sequence.append_element(damper, 'Damper')

##################
# Build TrackJob #
##################

tracker = xt.Tracker(_context=context, sequence=sequence)

######################
# Get some particles #
######################
```

(continues on next page)

(continued from previous page)

```
particles = xt.Particles(_context=context,
                         p0c=6500e9,
                         x=np.random.uniform(-1e-3, 1e-3, n_part)+1e-3,
                         px=np.random.uniform(-1e-7, 1e-7, n_part),
                         y=np.random.uniform(-0.5e-3, 0.5e-3, n_part)-1.2e-3,
                         py=np.random.uniform(-1e-7, 1e-7, n_part),
                         zeta=0*np.random.uniform(-1e-2, 1e-2, n_part),
                         delta=0.*np.random.uniform(-1e-5, 1e-5, n_part))

#########
# Track #
#########

tracker.track(particles, num_turns=num_turns, turn_by_turn_monitor=True)

########
# Plot #
########

res = tracker.record_last_track

import matplotlib.pyplot as plt
plt.close('all')
fig1 = plt.figure(1)
sp1 = fig1.add_subplot(2,1,1)
sp2 = fig1.add_subplot(2,1,2)
sp1.plot(np.mean(res.x, axis=0))
sp2.plot(np.mean(res.y, axis=0))
plt.show()
```

1.2 Developer's guide

1.2.1 Definition a new beam element for Xtrack

Table of Contents

In this page we illustrate how to introduce a new beam element in Xtrack. We will use for illustration the "SRotation" element which performs the following transformation of the particle coordinates:

- x = cos(theta) * x + sin(theta) * y
- y = -sin(theta) * x + cos(theta) * y
- px = cos(theta) * px + sin(theta) * py
- py = -sin(theta) * px + cos(theta) * py

The element is fully described by the rotation angle theta.

Definition of the data structure

New beam elements are defined as python classes inheriting from the class `BeamElement` of xtrack. In each element class we define a dictionary called `_xofields`, which specifies names and types of the data to be made accessible to the C tracking code.

Although our beam element is defined by the single parameter (theta), it is convenient to store the quantities sin(theta) and cos(theta) to avoid recalculating them multiple times:

```
import xobjects as xo
import xtrack as xt

class SRotation(BeamElement):

    _xofields={
        'cos_z': xo.Float64,
        'sin_z': xo.Float64,
        }
```

Allocation of beam elements on CPU or GPU

Objects of the defined class can be allocated as follows:

```
srot = SRotation(sin_z=1., cos_z=0)
```

By default the objects are allocated in the CPU memory. They can be allocated in the memory of a GPU by providing an xobject context or buffer. For example:

```
ctx = xo.ContextCupy()

# Object allocated on the GPU
srot = SRotation(sin_z=1., cos_z=0, _context=ctx)
```

Python access to beam-element data

The fields specified in `_xofields` are automatically exposed as attributes of the objects that can be read and set with the standard python syntax, also if the object is allocated on the GPU:

```
print(srot.sin_z)
# returns 1.0

srot.sin_z = 0.9

print(srot.sin_z)
# returns 0.9
```

Additional attributes and methods can be added to the class. If the `__init__` method is defined, the `__init__` of the parent class needs to be called to initialize the `xobject`, i.e. the data structure accessible from the C code.

Custom `__init__` method

In our example we want to initialize the object providing the rotation angle and not its sine and cosine and we introduce a property called `angle` that allows setting or getting the angle from the stored sine and cosine. This can be done as follows:

```
import numpy as np

import xobjects as xo
import xtrack as xt

class SRotation(BeamElement):

    def __init__(self, angle=0, **kwargs):
        anglerad = angle / 180 * np.pi
        kwargs['cos_z']=np.cos(anglerad)
        kwargs['sin_z']=np.sin(anglerad)
        super().__init__(**kwargs)

    @property
    def angle(self):
        return np.arctan2(self.sin_z, self.cos_z) * (180.0 / np.pi)

    @angle.setter
    def angle(self, value):
        anglerad = value / 180 * np.pi
        self.cos_z = np.cos(anglerad)
        self.sin_z = np.sin(anglerad)
```

Definition of the tracking function

Accessing beam-element data from C

The class definition from previous section automatically generates a set of functions (API) to access and manipulate in C the data spcified in `_xofields`. The C API for the defined class can be inspected as follows:

```
source, kernels, cdefs = SRotation.XoStruct._gen_c_api()
print(source)
```

By printing source we can see that C methods are available to set, get and get a pointer to the fields specified in `_xofields`:

```
/*gpufun*/ double SRotationData_get_cos_z(const SRotationData/*restrict*/ obj);
/*gpufun*/ void SRotationData_set_cos_z(SRotationData/*restrict*/ obj, double value);
/*gpufun*/ /*gpuglmem*/double* SRotationData_getp_cos_z(SRotationData/*restrict*/ obj);

/*gpufun*/ double SRotationData_get_sin_z(const SRotationData/*restrict*/ obj);
/*gpufun*/ void SRotationData_set_sin_z(SRotationData/*restrict*/ obj, double value);
/*gpufun*/ /*gpuglmem*/double* SRotationData_getp_sin_z(SRotationData/*restrict*/ obj);
```

Note the annotations `/*gpufun*/` that indicates that these are device functions on GPU and `/*gpuglmem*/` that indicates that the annotated pointer refers to the GPU global memory space.

These methods can be used to write a C header file containing the tracking code for the beam element. The method takes two arguments, the element data in a data type called `<ElementName>Data`, i.e. `SRotationData` in our example and a `LocalParticle` which is associated to methods to set and and get the particle coordinates. The `LocalParticle` represents one particle of the particle set provided to the simulation. On CPU it is possible to change the particle pointed by the local particle by changing the index `ipart`.

Writing the tracking code

For our example beam elements the tracking code can be writte as follows:

```
#ifndef XTRACK_SROTATION_H
#define XTRACK_SROTATION_H

/*gpufun*/
void SRotation_track_local_particle(SRotationData el, LocalParticle* part0){


    double const sin_z = SRotationData_get_sin_z(el);
    double const cos_z = SRotationData_get_cos_z(el);

    //start_per_particle_block (part0->part)

        double const x  = LocalParticle_get_x(part);
        double const y  = LocalParticle_get_y(part);
        double const px = LocalParticle_get_px(part);
        double const py = LocalParticle_get_py(part);

        double const x_hat  =  cos_z * x  + sin_z * y;
        double const y_hat  = -sin_z * x  + cos_z * y;
```

(continues on next page)

(continued from previous page)

```
        double const px_hat =  cos_z * px + sin_z * py;
        double const py_hat = -sin_z * px + cos_z * py;


        LocalParticle_set_x(part, x_hat);
        LocalParticle_set_y(part, y_hat);

        LocalParticle_set_px(part, px_hat);
        LocalParticle_set_py(part, py_hat);

    //end_per_particle_block

}

#endif
```

You can note in the code above the `/*gpufun*/` annotation specifying that the function is to be executed on the device for the GPU contexts.

The annotations `` `//start_per_particle_block` `` and `` `//end_per_particle_block` `` map part0 to part and introduce a loop over the particle when needed (i.e. for the CPU contexts). Parallelization over CPU cores is also applied if this is set in the context.

Once ready the code needs to be associated to the class. This is done with the following instruction:

```
from pathlib import Path

SRotation.XoStruct.extra_sources = [Path('./srotation.h')]
```

1.2.2 Multiplatform programming with xobjects

Table of Contents

* *Inspecting the source code*

Xobjects provides low-level functionalities to the other Xsuite packages, allowing to use the same code on different platforms (CPU and GPU).

Data management

The following example shows how to use Xobjects to allocate and manipulate date in the memory of the computing platform (CPU or GPU)

Definition of a simple data structure class

A Xobjects data structure can be defined as follows:

```
import xobjects as xo

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

The structure has four fields, three of them (`a`, `b` and `c`) are arrays and one of them (`s`) is a scalar.

Allocation of a data object on CPU or GPU

The memory on which data is stored is chosen by defining a context object, which is passed to the class constructor when the objects are instantiated. For example we can allocate an object of our data structure as follows:

```
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)
```

If `ctx = xo.ContextCpu()` the object is allocated on CPU, if `ctx = xo.ContextCupy()` the object is allocated in GPU.

Access to the data

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
```

Numpy-like access

Xobjects arrays can be viewed as numpy arrays (or numpy-like on GPUs). For example for our object:

```
arr = obj.a.to_nplike()
type(arr) # gives: numpy.ndarray
```

This object is a view and not a copy, which means that when we operate on the numpy arrays the underlying xobject is also modified. For example:

```
print(obj.a[0], obj.b[0]) # gives: 1.0 4.0
a_nplike = obj.a.to_nplike()
b_nplike = obj.b.to_nplike()

# We use np array algebra
a_nplike[:] = b_nplike - 1

print(obj.a[0], obj.b[0]) # gives: 3.0 4.0

# Usage of the .sum method of the numpy array
obj.s = a_nplike.sum()
```

Kernel functions in C

Autogenerated C API

The definition of a Xobject in Python, automatically triggers the generation of a set of functions (C-API) that can be used in C code to access the data allocated in Python. The available functions for a given Xobject can be inspected using the method `_gen_c_decl()`. For our example structure this can be done by:

```
print(DataStructure._gen_c_decl())
```

which provides the following set of C functions:

```
typedef /*gpuglmem*/ struct DataStructure_s * DataStructure;
/*gpufun*/ DataStructure DataStructure_getp(DataStructure/*restrict*/ obj);
/*gpufun*/ ArrNFloat64 DataStructure_getp_a(DataStructure/*restrict*/ obj);
/*gpufun*/ int64_t DataStructure_len_a(DataStructure/*restrict*/ obj);
/*gpufun*/ double DataStructure_get_a(const DataStructure/*restrict*/ obj, int64_t i0);
/*gpufun*/ void DataStructure_set_a(DataStructure/*restrict*/ obj, int64_t i0, double␣
↪value);
/*gpufun*/ /*gpuglmem*/double* DataStructure_getp1_a(DataStructure/*restrict*/ obj,␣
↪int64_t i0);
/*gpufun*/ ArrNFloat64 DataStructure_getp_b(DataStructure/*restrict*/ obj);
/*gpufun*/ int64_t DataStructure_len_b(DataStructure/*restrict*/ obj);
/*gpufun*/ double DataStructure_get_b(const DataStructure/*restrict*/ obj, int64_t i0);
/*gpufun*/ void DataStructure_set_b(DataStructure/*restrict*/ obj, int64_t i0, double␣
↪value);
/*gpufun*/ /*gpuglmem*/double* DataStructure_getp1_b(DataStructure/*restrict*/ obj,␣
↪int64_t i0);
/*gpufun*/ ArrNFloat64 DataStructure_getp_c(DataStructure/*restrict*/ obj);
/*gpufun*/ int64_t DataStructure_len_c(DataStructure/*restrict*/ obj);
```

(continues on next page)

(continued from previous page)

```
/*gpufun*/ double DataStructure_get_c(const DataStructure/*restrict*/ obj, int64_t i0);
/*gpufun*/ void DataStructure_set_c(DataStructure/*restrict*/ obj, int64_t i0, double␣
↪value);
/*gpufun*/ /*gpuglmem*/double* DataStructure_getp1_c(DataStructure/*restrict*/ obj,␣
↪int64_t i0);
/*gpufun*/ double DataStructure_get_s(const DataStructure/*restrict*/ obj);
/*gpufun*/ void DataStructure_set_s(DataStructure/*restrict*/ obj, double value);
/*gpufun*/ /*gpuglmem*/double* DataStructure_getp_s(DataStructure/*restrict*/ obj);
```

Writing a C kernel

A C function that can be parallelized when running on GPU is called Kernel. As an example, using our example data structure, we write a C kernel function (running on CPU and GPU) that performs the element-by-element product between the arrays `obj.a` and `obj.b` and writes it in `obj.c`. In the kernel code we use methods of the autogenerated C API to access the data in our example DataStructure.

```
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

}
'''
```

Note the xobject annotation `/*gpukern*/` that specifies that the function is a kernel, as well as annotations `//vectorize_over ii nelem` and `//end_vectorize` which identifies the variable on which the calculation can be performed in parallel and the corresponding range (i.e. 0 <= ii < nelem).

Compiling the kernel

The Xobject contex that we have alredy created to allocate the object in memory can also be used to compile the C code and access it from Python. This can be done with the method `add_kernels` by providing the source code and the description of the kernels from the source code that we would like to access 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 argument `n_threads` can be used to specify the name of an argument of the C function from which the number of threads to be used in the GPU can be inferred.

Calling the kernel

The kernel can be called from Python as follows

```
# obj.a contains [3., 4., 5.]
# obj.b contains [4., 5., 6.]
# obj.c contains [0., 0., 0.]

ctx.kernels.myprod(ob=obj, nelem=len(obj.a))

# obj.a contains [3., 4., 5.]
# obj.b contains [4., 5., 6.]
# obj.c contains [12., 20., 30.]
```

Inspecting the source code

Before compiling, the context specializes our source code for tgw chosen platform. Such autogenerated specialized code can be inspected from Python as follows:

```
print(ctx.kernels.myprod.specialized_source)
```

For our example kernel `mymul`, if the chosen context is a `ContextCpu` the generated specialized source is:

```
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

}
```

If the chosen context is a `ContextCupy` the generated specialized source is:

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

    int ii; //autovectorized
    ii=blockDim.x * blockIdx.x + threadIdx.x;//autovectorized
    if (ii<nelem){ //autovectorized
        double a_ii = DataStructure_get_a(ob, ii);
        double b_ii = DataStructure_get_b(ob, ii);

        double c_ii = a_ii * b_ii;
```

(continues on next page)

(continued from previous page)

```
        DataStructure_set_c(ob, ii, c_ii);
    }//end autovectorized
}
```

If the chosen context is a `ContextCupy` the generated specialized source is:

```
__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

}
```

1.2.3 Code autogeneration

The xsuite library uses code autogeneration to specialize kernel code for the different contexts. Three contexts are presently available: `cpu`, `cuda`, and `opencl`.

An example annotated source file can be found here.

The developer writes a single C source code, providing additional information through the comment strings (annotations) described in the following.

`vectorize_over` block

The syntax is the following:

```
for (int myvar=0; myvar<myvarlim; myvar++){ //vectorize_over myvar myvarlim

    [MY CODE]

}//end_vectorize
```

This is translated into a for loop in the CPU implementation and in a kernel function for the parallel implementations (cupy, pyopencl).

The generated cpu code will be:

```
for (int myvar=0; myvar<myvarlim; myvar++){ //autovectorized

    [MY CODE]

    }//end autovectorized
```

The generated CUDA code will be:

```
int myvar; //autovectorized
myvar = blockDim.x * blockIdx.x + threadIdx.x; //autovectorized
if (myvar<myvarlim) { //autovectorized

    [MY CODE]

}//end autovectorized
```

The corresponding generated OpenCL code will be:

```
int myvar; //autovectorized
myvar = get_global_id(0); //autovectorized

    [MY CODE]

//end autovectorized
```

only_for_context directive

The \\only_for_context directive can be used to include a givem line only for a certain context. For example with the following code the line marked line is included only in the GPU implementation.

```
#include <atomicadd.h> //only_for_context cpu
```

gpufun directive The *gpufun*\ directive is used to qualify device functions. The code generator replaces it with __device__ in the CUDA code.

gpukern directive

The *gpukern*\ directive is used to qualify kernel functions. The code generator replaces it with __global__ in the CUDA code and with __kernel in the OpenCL code.

gpuglmem directive

The *gpuglmem*\ directive is used to qualify pointers to locations in the device global memoru. The code generator replaces it with __global in the OpenCL code.

1.3 Physics guide

The physics models implemented in Xsuite are being documented in the guide available at the following link:

https://github.com/xsuite/xsuite/blob/master/docs/physics_manual/physics_man.pdf

1.4 API reference

The API of the Xsuite library is documented in the following sections:

1.4.1 Xobjects API

The API of the Xobjects package is documented in the following sections:

Contexts

Xsuite supports different plaforms allowing the exploitation of different kinds of hardware (CPUs and GPUs). A context is initialized by instanciating objects from one of the context classes available Xobjects, which is then passed to the other Xsuite components (see example in Getting Started Guide). Contexts are interchangeable as they expose the same API. Custom kernel functions can be added to the contexts. General source code with annotations can be provided to define the kernels, which is then automatically specialized for the chosen platform (see *dedicated section*).

Three contexts are presently available:

- The *Cupy context*, based on cupy-cuda to run on NVidia GPUs
- The *Pyopencl context*, bases on PyOpenCL, to run on CPUs or GPUs throught PyOPENCL library.
- The *CPU context*, to use conventional CPUs

The corresponfig API is described in the following subsections.

Cupy context

class xobjects.**ContextCupy**(*default_block_size=256*, *device=None*)

Creates a Cupy Context object, that allows performing the computations on nVidia GPUs.

To select device use cupy.Device(<n>).use()

Parameters **default_block_size** (*int*) – CUDA thread size that is used by default for kernel execution in case a block size is not specified directly in the kernel object. The default value is 256.

Returns context object.

Return type *ContextCupy*

add_kernels(*sources*, *kernels*, *specialize=True*, *save_source_as=None*, *extra_cdef=None*, *extra_classes=[]*, *extra_headers=[]*)

Adds user-defined kernels to to the context. The kernel source code is provided as a string and/or in source files and must contain the kernel names defined in the kernel descriptions. :param sources: List of source codes that are concatenated before

compilation. The list can contain strings (raw source code), File objects and Path objects.

Parameters

- **kernels** (*dict*) – Dictionary with the kernel descriptions in the form given by the following examples. The descriptions define the kernel names, the type and name of the arguments and identify one input argument that defines the number of threads to be launched (only on cuda/opencl).

- **specialize_code** (*bool*) – If True, the code is specialized using annotations in the source code. Default is `True`
- **save_source_as** (*str*) – Filename for saving the specialized source code. Default is `None`.

Example:

```
# A simple kernel
src_code = '''
/*gpukern*/
void my_mul(const int n,
    /*gpuglmem*/ const double* x1,
    /*gpuglmem*/ const double* x2,
    /*gpuglmem*/       double* y) {
    int tid = 0 //vectorize_over tid
    y[tid] = x1[tid] * x2[tid];
    //end_vectorize
    }
'''

# Prepare description
kernel_descriptions = {
    "my_mul": xo.Kernel(
        args=[
            xo.Arg(xo.Int32, name="n"),
            xo.Arg(xo.Float64, pointer=True, const=True, name="x1"),
            xo.Arg(xo.Float64, pointer=True, const=True, name="x2"),
            xo.Arg(xo.Float64, pointer=True, const=False, name="y"),
        ],
        n_threads="n",
        ),
}

# Import kernel in context
ctx.add_kernels(
    sources=[src_code],
    kernels=kernel_descriptions,
    save_source_as=None,
)

# With a1, a2, b being arrays on the context, the kernel
# can be called as follows:
ctx.kernels.my_mul(n=len(a1), x1=a1, x2=a2, y=b)
```

nparray_to_context_array(*arr*)
Copies a numpy array to the device memory.

Parameters **arr** (*numpy.ndarray*) – Array to be transferred

Returns The same array copied to the device.

Return type cupy.ndarray

nparray_from_context_array(*dev_arr*)
Copies an array to the device to a numpy array.

Parameters **dev_arr** (*cupy.ndarray*) – Array to be transferred.

Returns The same data copied to a numpy array.

Return type numpy.ndarray

property nplike_lib
Module containing all the numpy features supported by cupy.

synchronize()
Ensures that all computations submitted to the context are completed. Equivalent to `cupy.cuda.stream.get_current_stream().synchronize()`

zeros(**args*, ***kwargs*)
Allocates an array of zeros on the device. The function has the same interface of numpy.zeros

plan_FFT(*data*, *axes*)
Generates an FFT plan object to be executed on the context.

Parameters

- **data** (*cupy.ndarray*) – Array having type and shape for which the FFT needs to be planned.
- **axes** (*sequence of ints*) – Axes along which the FFT needs to be performed.

Returns FFT plan for the required array shape, type and axes.

Return type FFTCupy

Example:

```
plan = context.plan_FFT(data, axes=(0,1))

data2 = 2*data

# Forward tranform (in place)
plan.transform(data2)

# Inverse tranform (in place)
plan.itransform(data2)
```

property kernels
Dictionary containing all the kernels that have been imported to the context. The syntax `context.kernels.mykernel` can also be used.

property buffers

minimum_alignment = 1

new_buffer(*capacity=1048576*)

PyOpenCL context

class xobjects.**ContextPyopencl**(*device=None*, *patch_pyopencl_array=True*, *minimum_alignment=None*)

classmethod get_devices()

classmethod print_devices()

minimum_alignment = 1

find_minimum_alignment()

add_kernels(*sources=[]*, *kernels={}*, *specialize=True*, *save_source_as=None*, *extra_cdef=None*, *extra_classes={}*, *extra_headers=()*)

Adds user-defined kernels to to the context. The kernel source code is provided as a string and/or in source files and must contain the kernel names defined in the kernel descriptions. :param sources: List of source codes that are concatenated before

compilation. The list can contain strings (raw source code), File objects and Path objects.

Parameters

- **kernels** (*dict*) – Dictionary with the kernel descriptions in the form given by the following examples. The descriptions define the kernel names, the type and name of the arguments and identify one input argument that defines the number of threads to be launched (only on cuda/opencl).
- **specialize_code** (*bool*) – If True, the code is specialized using annotations in the source code. Default is `True`
- **save_source_as** (*str*) – Filename for saving the specialized source code. Default is `None`.

Example:

```
# A simple kernel
src_code = '''
/*gpukern*/
void my_mul(const int n,
    /*gpuglmem*/ const double* x1,
    /*gpuglmem*/ const double* x2,
    /*gpuglmem*/       double* y) {
    int tid = 0 //vectorize_over tid
    y[tid] = x1[tid] * x2[tid];
    //end_vectorize
    }
'''

# Prepare description
kernel_descriptions = {
    "my_mul": xo.Kernel(
        args=[
            xo.Arg(xo.Int32, name="n"),
            xo.Arg(xo.Float64, pointer=True, const=True, name="x1"),
            xo.Arg(xo.Float64, pointer=True, const=True, name="x2"),
            xo.Arg(xo.Float64, pointer=True, const=False, name="y"),
```

(continues on next page)

(continued from previous page)

```
        ],
        n_threads="n",
        ),
}

# Import kernel in context
ctx.add_kernels(
    sources=[src_code],
    kernels=kernel_descriptions,
    save_source_as=None,
)

# With a1, a2, b being arrays on the context, the kernel
# can be called as follows:
ctx.kernels.my_mul(n=len(a1), x1=a1, x2=a2, y=b)
```

nparray_to_context_array(*arr*)

Copies a numpy array to the device memory. :param arr: Array to be transferred :type arr: numpy.ndarray

Returns The same array copied to the device.

Return type pyopencl.array.Array

nparray_from_context_array(*dev_arr*)

Copies an array to the device to a numpy array.

Parameters **dev_arr** (*pyopencl.array.Array*) – Array to be transferred.

Returns The same data copied to a numpy array.

Return type numpy.ndarray

property nplike_lib

Module containing all the numpy features supported by PyOpenCL (optionally with patches to operate with non-contiguous arrays).

synchronize()

Ensures that all computations submitted to the context are completed. No action is performed by this function in the Pyopencl context. The method is provided so that the Pyopencl context has an identical API to the Cupy one.

zeros(**args*, ***kwargs*)

Allocates an array of zeros on the device. The function has the same interface of numpy.zeros

plan_FFT(*data*, *axes*, *wait_on_call=True*)

Generates an FFT plan object to be executed on the context.

Parameters

- **data** (*pyopencl.array.Array*) – Array having type and shape for which the FFT needs to be planned.
- **axes** (*sequence of ints*) – Axes along which the FFT needs to be performed.

Returns FFT plan for the required array shape, type and axes.

Return type FFTPyopencl

Example:

```
plan = context.plan_FFT(data, axes=(0,1))

data2 = 2*data

# Forward tranform (in place)
plan.transform(data2)

# Inverse tranform (in place)
plan.itransform(data2)
```

property kernels
Dictionary containing all the kernels that have been imported to the context. The syntax `context.kernels.mykernel` can also be used.

property buffers

new_buffer(*capacity=1048576*)

CPU context

class xobjects.**ContextCpu**(*omp_num_threads=0*)
Creates a CPU Platform object, that allows performing the computations on conventional CPUs.

Returns platform object.

Return type ContextCpu

add_kernels(*sources=[]*, *kernels=[]*, *specialize=True*, *save_source_as=None*, *extra_compile_args=['-O3', '-Wno-unused-function']*, *extra_link_args=['-O3']*, *extra_cdef=None*, *extra_classes=[]*, *extra_headers=[]*)
Adds user-defined kernels to to the context. The kernel source code is provided as a string and/or in source files and must contain the kernel names defined in the kernel descriptions. :param sources: List of source codes that are concatenated before

compilation. The list can contain strings (raw source code), File objects and Path objects.

Parameters

- **kernels** (*dict*) – Dictionary with the kernel descriptions in the form given by the following examples. The descriptions define the kernel names, the type and name of the arguments and identify one input argument that defines the number of threads to be launched (only on cuda/opencl).
- **specialize_code** (*bool*) – If True, the code is specialized using annotations in the source code. Default is `True`
- **save_source_as** (*str*) – Filename for saving the specialized source code. Default is `None`.

Example:

```
# A simple kernel
src_code = '''
/*gpukern*/
void my_mul(const int n,
    /*gpuglmem*/ const double* x1,
```

(continues on next page)

(continued from previous page)

```
    /*gpuglmem*/ const double* x2,
    /*gpuglmem*/       double* y) {
    int tid = 0 //vectorize_over tid
    y[tid] = x1[tid] * x2[tid];
    //end_vectorize
    }
'''

# Prepare description
kernel_descriptions = {
    "my_mul": xo.Kernel(
        args=[
            xo.Arg(xo.Int32, name="n"),
            xo.Arg(xo.Float64, pointer=True, const=True, name="x1"),
            xo.Arg(xo.Float64, pointer=True, const=True, name="x2"),
            xo.Arg(xo.Float64, pointer=True, const=False, name="y"),
        ],
        n_threads="n",
        ),
}

# Import kernel in context
ctx.add_kernels(
    sources=[src_code],
    kernels=kernel_descriptions,
    save_source_as=None,
)

# With a1, a2, b being arrays on the context, the kernel
# can be called as follows:
ctx.kernels.my_mul(n=len(a1), x1=a1, x2=a2, y=b)
```

nparray_to_context_array(*arr*)

Moves a numpy array to the device memory. No action is performed by this function in the CPU context. The method is provided so that the CPU context has an identical API to the GPU ones.

Parameters **arr** (*numpy.ndarray*) – Array to be transferred

Returns The same array (no copy!).

Return type numpy.ndarray

nparray_from_context_array(*dev_arr*)

Moves an array to the device to a numpy array. No action is performed by this function in the CPU context. The method is provided so that the CPU context has an identical API to the GPU ones.

Parameters **dev_arr** (*numpy.ndarray*) – Array to be transferred

Returns The same array (no copy!)

Return type numpy.ndarray

property nplike_lib

Module containing all the numpy features. Numpy members should be accessed through `nplike_lib` to keep compatibility with the other contexts.

synchronize()
Ensures that all computations submitted to the context are completed. No action is performed by this function in the CPU context. The method is provided so that the CPU context has an identical API to the GPU ones.

zeros(**args*, ***kwargs*)
Allocates an array of zeros on the device. The function has the same interface of numpy.zeros

plan_FFT(*data*, *axes*)
Generate an FFT plan object to be executed on the context.

Parameters

- **data** (*numpy.ndarray*) – Array having type and shape for which the FFT needs to be planned.
- **axes** (*sequence of ints*) – Axes along which the FFT needs to be performed.

Returns FFT plan for the required array shape, type and axes.

Return type FFTCpu

Example:

```
plan = context.plan_FFT(data, axes=(0,1))

data2 = 2*data

# Forward tranform (in place)
plan.transform(data2)

# Inverse tranform (in place)
plan.itransform(data2)
```

property kernels
Dictionary containing all the kernels that have been imported to the context. The syntax `context.kernels.mykernel` can also be used.

property buffers

minimum_alignment = 1

new_buffer(*capacity=1048576*)

1.4.2 Xline API

Table of Contents

* *RFMultipole*
* *DipoleEdge*
* *XYShift*
* *SRotation*
* *LimitEllipse*
* *LimitRect*
* *BeamBeam4D*
* *BeamBeam6D*
* *SCCoasting*
* *SCInterpolatedProfile*
* *SCQGaussProfile*
* *BeamMonitor*

Line

class xline.**Line**(*elements: tuple = ()*, *element_names: tuple = ()*)

Bases: xline.base_classes.Element

Fields:

- elements []: List of elements
- element_names []: List of element names

to_dict(*keepextra=True*)

classmethod from_dict(*dct*, *keepextra=True*)

to_json(*filename*, *keepextra=True*)

classmethod from_json(*filename*, *keepextra=True*)

append_line(*line*)

track(*p*)

track_elem_by_elem(*p*, *start=True*, *end=False*)

insert_element(*idx*, *element*, *name*)

append_element(*element*, *name*)

get_length()

get_s_elements(*mode='upstream'*)

remove_inactive_multipoles(*inplace=False*)

remove_zero_length_drifts(*inplace=False*)

merge_consecutive_drifts(*inplace=False*)

merge_consecutive_multipoles(*inplace=False*)

get_elements_of_type(*types*)

get_element_ids_of_type(*types*, *start_idx_offset=0*)

linear_normal_form(*M*)

find_closed_orbit_and_linear_OTM(*p0c*, *guess=None*, *d=1e-07*, *tol=1e-10*, *max_iterations=20*, *longitudinal_coordinate='zeta'*)

find_closed_orbit(*p0c*, *guess=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]*, *method='Nelder-Mead'*, ***kwargs*)

enable_beambeam()

disable_beambeam()

beambeam_store_closed_orbit_and_dipolar_kicks(*particle_on_CO*, *separation_given_wrt_closed_orbit_4D=True*, *separation_given_wrt_closed_orbit_6D=True*)

classmethod from_sixinput(*sixinput*, *classes=<module 'xline.elements' from '/home/docs/checkouts/readthedocs.org/user_builds/xsuite/checkouts/latest/docs/xline/xline/eler*

classmethod from_madx_sequence(*sequence*, *classes=<module 'xline.elements' from '/home/docs/checkouts/readthedocs.org/user_builds/xsuite/checkouts/latest/docs/xline/xli*, *ignored_madtypes=[]*, *exact_drift=False*, *drift_threshold=1e-06*, *install_apertures=False*, *apply_madx_errors=False*)

find_element_ids(*element_name*)

Find element_name in this Line instance's self.elements_name list. Assumes the names are unique.

Return index before and after the element, taking into account attached _aperture instances (LimitRect, LimitEllipse, ...) which would follow the element occurrence in the list.

Raises IndexError if element_name not found in this Line.

element_names: tuple = ()

elements: tuple = ()

Beam elements

Drift

class xline.**Drift**(*length: int = 0*)

Bases: xline.base_classes.Element

Drift in expanded form

Fields:

- length [m]: Length of the drift

track(*p*)

length: int = 0

Multipole

class xline.**Multipole**(*knl: List = <factory>*, *ksl: List = <factory>*, *hxl: int = 0*, *hyl: int = 0*, *length: int = 0*)

Bases: xline.base_classes.Element

Fields:

- knl [m^-n]: Normalized integrated strength of normal components
- ksl [m^-n]: Normalized integrated strength of skew components
- hxl [rad]: Rotation angle of the reference trajectoryin the horizzontal plane
- hyl [rad]: Rotation angle of the reference trajectory in the vertical plane
- length [m]: Length of the originating thick multipole

property order

track(*p*)

hxl: int = 0

hyl: int = 0

length: int = 0

knl: List

ksl: List

Cavity

class xline.**Cavity**(*voltage: int = 0*, *frequency: int = 0*, *lag: int = 0*)

Bases: xline.base_classes.Element

Radio-frequency cavity

Fields:

- voltage [V]: Integrated energy change
- frequency [Hz]: Frequency of the cavity
- lag [degree]: Delay in the cavity sin(lag - w tau)

track(*p*)

frequency: int = 0

lag: int = 0

voltage: int = 0

RFMultipole

class xline.**RFMultipole**(*voltage: int = 0*, *frequency: int = 0*, *lag: int = 0*, *knl: List = <factory>*, *ksl: List = <factory>*, *pn: List = <factory>*, *ps: List = <factory>*)

Bases: xline.base_classes.Element

H= -l sum Re[(kn[n](zeta) + i ks[n](zeta)) (x+iy)**(n+1)/ n]

kn[n](z) = k_n cos(2pi w tau + pn/180*pi) ks[n](z) = k_n cos(2pi w tau + pn/180*pi)

Fields:

- voltage [volt]: Voltage
- frequency [hertz]: Frequency
- lag [degree]: Delay in the cavity sin(lag - w tau)
- knl []: …
- ksl []: …
- pn []: …
- ps []: …

property order

track(*p*)

frequency: int = 0

lag: int = 0

voltage: int = 0

knl: List

ksl: List

pn: List

ps: List

DipoleEdge

class xline.**DipoleEdge**(*h: int = 0*, *e1: int = 0*, *hgap: int = 0*, *fint: int = 0*)

Bases: xline.base_classes.Element

Fields:

- h [1/m]: Curvature
- e1 [rad]: Face angle
- hgap [m]: Equivalent gap
- fint []: Fringe integral

track(*p*)

e1: int = 0

fint: int = 0

h: int = 0

hgap: int = 0

XYShift

class xline.**XYShift**(*dx: int = 0*, *dy: int = 0*)

Bases: xline.base_classes.Element

shift of the reference

Fields:

- dx [m]: Horizontal shift
- dy [m]: Vertical shift

track(*p*)

dx: int = 0

dy: int = 0

SRotation

class xline.**SRotation**(*angle: int = 0*)

Bases: xline.base_classes.Element

anti-clockwise rotation of the reference frame

Fields:

- angle []: Rotation angle

track(*p*)

angle: int = 0

LimitEllipse

class xline.**LimitEllipse**(*a: float = 1.0*, *b: float = 1.0*)

Bases: xline.base_classes.Element

Fields:

- a [m]: Horizontal semiaxis
- b [m]: Vertical semiaxis

track(*particle*)

a: float = 1.0

b: float = 1.0

LimitRect

class xline.**LimitRect**(*min_x: float = - 1.0*, *max_x: float = 1.0*, *min_y: float = - 1.0*, *max_y: float = 1.0*)

Bases: `xline.base_classes.Element`

Fields:

- min_x [m]: Minimum horizontal aperture
- max_x [m]: Maximum horizontal aperture
- min_y [m]: Minimum vertical aperture
- max_y [m]: Minimum vertical aperture

track(*particle*)

max_x: float = 1.0

max_y: float = 1.0

min_x: float = -1.0

min_y: float = -1.0

BeamBeam4D

class xline.**BeamBeam4D**(*charge: int = 0*, *sigma_x: float = 1.0*, *sigma_y: float = 1.0*, *beta_r: float = 1.0*, *x_bb: int = 0*, *y_bb: int = 0*, *d_px: int = 0*, *d_py: int = 0*, *min_sigma_diff: float = 1e-28*, *enabled: bool = True*)

Bases: `xline.base_classes.Element`

Interaction with a transverse-Gaussian strong beam (4D modelling).

Fields:

- charge [e]: Charge of the interacting distribution (strong beam)
- sigma_x [m]: Horizontal size of the strong beam (r.m.s.)
- sigma_y [m]: Vertical size of the strong beam (r.m.s.)
- beta_r []: Relativistic beta of the stron beam
- x_bb [m]: H. position of the strong beam w.r.t. the reference trajectory
- y_bb [m]: V. position of the strong beam w.r.t. the reference trajectory
- d_px []: H. kick subtracted after the interaction.
- d_py []: V. kick subtracted after the interaction.
- min_sigma_diff [m]: Threshold to detect round beam
- enabled []: Switch for closed orbit search

track(*p*)

beta_r: float = 1.0

charge: int = 0

d_px: int = 0

d_py: int = 0

enabled: bool = True

min_sigma_diff: *float = 1e-28*

sigma_x: *float = 1.0*

sigma_y: *float = 1.0*

x_bb: *int = 0*

y_bb: *int = 0*

BeamBeam6D

class xline.**BeamBeam6D**(*phi: int = 0, alpha: int = 0, x_bb_co: int = 0, y_bb_co: int = 0, charge_slices: float = 0.0, zeta_slices: float = 0.0, sigma_11: float = 1.0, sigma_12: int = 0, sigma_13: int = 0, sigma_14: int = 0, sigma_22: int = 0, sigma_23: int = 0, sigma_24: int = 0, sigma_33: float = 1.0, sigma_34: int = 0, sigma_44: int = 0, x_co: int = 0, px_co: int = 0, y_co: int = 0, py_co: int = 0, zeta_co: int = 0, delta_co: int = 0, d_x: int = 0, d_px: int = 0, d_y: int = 0, d_py: int = 0, d_zeta: int = 0, d_delta: int = 0, min_sigma_diff: float = 1e-28, threshold_singular: float = 1e-28, enabled: bool = True*)

Bases: `xline.base_classes.Element`

Interaction with a transverse-Gaussian strong beam (6D modelling).

http://cds.cern.ch/record/2306400

Fields:

- phi [rad]: Crossing angle (>0 weak beam increasesx in the direction motion)
- alpha [rad]: Crossing plane tilt angle (>0 x tends to y)
- x_bb_co [m]: H. position of the strong beam w.r.t. the closed orbit
- y_bb_co [m]: V. position of the strong beam w.r.t. the closed orbit
- charge_slices [qe]: Charge of the interacting slices (strong beam)
- zeta_slices [m]: Longitudinal position of the interactingslices (>0 head of the strong).
- sigma_11 [m^2]: Sigma_11 element of the sigma matrix of the strong beam
- sigma_12 [m]: Sigma_12 element of the sigma matrix of the strong beam
- sigma_13 [m^2]: Sigma_13 element of the sigma matrix of the strong beam
- sigma_14 [m]: Sigma_14 element of the sigma matrix of the strong beam
- sigma_22 []: Sigma_22 element of the sigma matrix of the strong beam
- sigma_23 [m]: Sigma_23 element of the sigma matrix of the strong beam
- sigma_24 []: Sigma_24 element of the sigma matrix of the strong beam
- sigma_33 [m^2]: Sigma_33 element of the sigma matrix of the strong beam
- sigma_34 [m]: Sigma_34 element of the sigma matrix of the strong beam
- sigma_44 []: Sigma_44 element of the sigma matrix of the strong beam
- x_co [m]: x coordinate the closed orbit (weak beam).
- px_co []: px coordinate the closed orbit (weak beam).
- y_co [m]: y coordinate the closed orbit (weaek beam).

- py_co []: py coordinate the closed orbit (weaek beam).
- zeta_co [m]: zeta coordinate the closed orbit (weaek beam).
- delta_co []: delta coordinate the closed orbit (weaek beam).
- d_x [m]: Quantity subtracted from x after the interaction.
- d_px []: Quantity subtracted from px after the interaction.
- d_y [m]: Quantity subtracted from y after the interaction.
- d_py []: Quantity subtracted from py after the interaction.
- d_zeta [m]: Quantity subtracted from sigma after the interaction.
- d_delta []: Quantity subtracted from delta after the interaction.
- min_sigma_diff [m]: Threshold to detect round beam
- threshold_singular []: Threshold to detect small denominator
- enabled []: Switch for closed orbit search

track(*p*)

alpha: int = 0

charge_slices: float = 0.0

d_delta: int = 0

d_px: int = 0

d_py: int = 0

d_x: int = 0

d_y: int = 0

d_zeta: int = 0

delta_co: int = 0

enabled: bool = True

min_sigma_diff: float = 1e-28

phi: int = 0

px_co: int = 0

py_co: int = 0

sigma_11: float = 1.0

sigma_12: int = 0

sigma_13: int = 0

sigma_14: int = 0

sigma_22: int = 0

sigma_23: int = 0

sigma_24: int = 0

sigma_33: float = 1.0

sigma_34: int = 0

sigma_44: int = 0

threshold_singular: float = 1e-28

x_bb_co: int = 0

x_co: int = 0

y_bb_co: int = 0

y_co: int = 0

zeta_co: int = 0

zeta_slices: float = 0.0

SCCoasting

class xline.**SCCoasting**(*number_of_particles: float = 0.0*, *circumference: float = 1.0*, *sigma_x: float = 1.0*, *sigma_y: float = 1.0*, *length: float = 0.0*, *x_co: float = 0.0*, *y_co: float = 0.0*, *min_sigma_diff: float = 1e-08*, *enabled: bool = True*)

Bases: `xline.base_classes.Element`

Space charge for a coasting beam.

Fields:

- number_of_particles []: Number of particles in the beam
- circumference [m]: Machine circumference
- sigma_x [m]: Horizontal size of the beam (r.m.s.)
- sigma_y [m]: Vertical size of the beam (r.m.s.)
- length [m]: Integration length of space charge kick
- x_co [m]: Horizontal closed orbit offset
- y_co [m]: Vertical closed orbit offset
- min_sigma_diff [m]: Threshold to detect round beam
- enabled []: Switch to disable space charge effect

track(*p*)

circumference: float = 1.0

enabled: bool = True

length: float = 0.0

min_sigma_diff: float = 1e-08

number_of_particles: float = 0.0

sigma_x: float = 1.0

sigma_y: float = 1.0

x_co: float = 0.0

y_co: float = 0.0

SCInterpolatedProfile

class xline.**SCInterpolatedProfile**(*number_of_particles: float = 0.0*, *line_density_profile: List = <factory>*, *dz: float = 1.0*, *z0: float = -0.5*, *sigma_x: float = 1.0*, *sigma_y: float = 1.0*, *length: float = 0.0*, *x_co: float = 0.0*, *y_co: float = 0.0*, *method: int = 0*, *min_sigma_diff: float = 1e-08*, *enabled: bool = True*)

Bases: `xline.base_classes.Element`

Space charge for a bunched beam with discretised profile.

Fields:

- number_of_particles []: Number of particles in the bunch
- line_density_profile [1/m]: Discretised list of density values with integral normalised to 1
- dz [m]: Unit distance in zeta between profile points
- z0 [m]: Start zeta position of line density profile
- sigma_x [m]: Horizontal size of the beam (r.m.s.)
- sigma_y [m]: Vertical size of the beam (r.m.s.)
- length [m]: Integration length of space charge kick
- x_co [m]: Horizontal closed orbit offset
- y_co [m]: Vertical closed orbit offset
- method []: Interpolation method; 0 == linear (default), 1 == cubic spline
- min_sigma_diff [m]: Threshold to detect round beam
- enabled []: Switch to disable space charge effect

track(*p*)

dz: float = 1.0

enabled: bool = True

length: float = 0.0

method: int = 0

min_sigma_diff: float = 1e-08

number_of_particles: float = 0.0

sigma_x: float = 1.0

sigma_y: float = 1.0

x_co: float = 0.0

y_co: float = 0.0

z0: float = -0.5

line_density_profile: List

SCQGaussProfile

class xline.SCQGaussProfile(*number_of_particles: float = 0.0*, *bunchlength_rms: float = 1.0*, *sigma_x: float = 1.0*, *sigma_y: float = 1.0*, *length: float = 0.0*, *x_co: float = 0.0*, *y_co: float = 0.0*, *min_sigma_diff: float = 1e-08*, *enabled: bool = True*, *q_parameter: float = 1.0*)

Bases: xline.base_classes.Element

Space charge for a bunched beam with generalised Gaussian profile.

Fields:

- number_of_particles []: Number of particles in the bunch
- bunchlength_rms [m]: Length of the bunch (r.m.s.)
- sigma_x [m]: Horizontal size of the beam (r.m.s.)
- sigma_y [m]: Vertical size of the beam (r.m.s.)
- length [m]: Integration length of space charge kick
- x_co [m]: Horizontal closed orbit offset
- y_co [m]: Vertical closed orbit offset
- min_sigma_diff [m]: Threshold to detect round beam
- enabled []: Switch to disable space charge effect
- q_parameter []: q parameter of generalised Gaussian distribution (q=1 for standard Gaussian)

track(*p*)

bunchlength_rms: float = 1.0

enabled: bool = True

length: float = 0.0

min_sigma_diff: float = 1e-08

number_of_particles: float = 0.0

q_parameter: float = 1.0

sigma_x: float = 1.0

sigma_y: float = 1.0

x_co: float = 0.0

y_co: float = 0.0

BeamMonitor

class xline.BeamMonitor(*num_stores: int = 0*, *start: int = 0*, *skip: int = 1*, *max_particle_id: int = 0*, *min_particle_id: int = 0*, *is_rolling: bool = False*, *is_turn_ordered: bool = True*, *data: List = <factory>*)

Bases: xline.base_classes.Element

Fields:

- num_stores []: ...
- start []: ...

- skip []: …
- max_particle_id []:
- min_particle_id []:
- is_rolling []:
- is_turn_ordered []:
- data []: …

offset(*particle*)

track(*p*)

is_rolling: bool = False

is_turn_ordered: bool = True

max_particle_id: int = 0

min_particle_id: int = 0

num_stores: int = 0

skip: int = 1

start: int = 0

data: List

1.4.3 Xpart API

The particle class

class xpart.**Particles**(*s=0.0*, *x=0.0*, *px=0.0*, *y=0.0*, *py=0.0*, *delta=None*, *ptau=None*, *psigma=None*, *rvv=None*, *zeta=None*, *tau=None*, *sigma=None*, *mass0=938272081.0*, *q0=1.0*, *p0c=None*, *energy0=None*, *gamma0=None*, *beta0=None*, *chi=None*, *mass_ratio=None*, *charge_ratio=None*, *particle_id=None*, *parent_particle_id=None*, *at_turn=None*, *state=None*, *weight=None*, *at_element=None*, *mathlib=<class 'xline.mathlibs.MathlibDefault'>*, ***args*)

Particle objects have the following fields:

- s [m]: Reference accumulated pathlength
- x [m]: Horizontal position
- px[1]: Px / (m/m0 * p0c) = beta_x gamma /(beta0 gamma0)
- y [m]: Vertical position
- py [1]: Py / (m/m0 * p0c)
- delta[1]: Pc / (m/m0 * p0c) - 1
- ptau [1]: Energy / (m/m0 * p0c) - 1
- psigma [1]: ptau/beta0
- rvv [1]: beta/beta0
- rpp [1]: 1/(1+delta) = (m/m0 * p0c) / Pc
- zeta [m]: beta (s/beta0 - ct)

- tau [m]:
- sigma [m]: s - beta0 ct = rvv * zeta
- mass0 [eV]:
- q0 [e]: Reference charge
- p0c [eV]: Reference momentum
- energy0 [eV]: Reference energy
- gamma0 [1]: Reference relativistic gamma
- beta0 [1]: Reference relativistix beta
- chi [1]: q/ q0 * m0/m = qratio / mratio
- mass_ratio [1]: mass/mass0
- charge_ratio [1]: q / q0
- particle_id [int]: Identifier of the particle
- at_turn [int]: Number of tracked turns
- state [int]: It is `0` if the particle is lost, 1 otherwise
- weight [int]: particle weight in number of particles (for collective sims.)
- at_element [int]: Identifier of the last element through which the particle has been
- parent_particle_id [int]: Identifier of the parent particle (secondary production processes)

clight = 299792458

pi = 3.141592653589793

echarge = 1.602176565e-19

emass = 510998.928

pmass = 938272081.0

epsilon0 = 8.854187817e-12

mu0 = 1.2566370614359173e-06

eradius = 5.0234346981155707e-51

pradius = 2.7358479460235164e-54

anumber = 6.02214129e+23

kboltz = 1.3806488e-23

copy(*index=None*)

property Px

property Py

property energy

property pc

property mass

property beta

property rvv

property rpp

add_to_energy(*energy*)

property delta

property psigma

property tau

property sigma

property ptau

property mass0

property beta0

property gamma0

property p0c

property energy0

property mass_ratio

property charge_ratio

property chi

remove_lost_particles(*keep_memory=True*)

to_dict()

classmethod from_dict(*dct*)

to_json(*filename*)

classmethod from_json(*filename*)

compare(*particle*, *rel_tol=1e-06*, *abs_tol=1e-15*)

classmethod from_madx_twiss(*twiss*)

classmethod from_madx_track(*mad*)

classmethod from_list(*lst*)

1.4.4 Xtrack API

Table of Contents

Tracker

class xtrack.**Tracker**(*_context=None*, *_buffer=None*, *_offset=None*, *sequence=None*, *track_kernel=None*, *element_classes=None*, *particles_class=None*, *skip_end_turn_actions=False*, *particles_monitor_class=None*, *global_xy_limit=1.0*, *local_particle_src=None*, *save_source_as=None*)

Beam elements

Drift

class xtrack.**Drift**(*_xobject=None*, ***kwargs*)

Bases: xtrack.dress.DressedDriftData, *xtrack.base_element.BeamElement*

Beam element modeling a drift section. Parameters:

- length [m]: Length of the drift section. Default is 0.

Multipole

class xtrack.**Multipole**(*order=None*, *knl=None*, *ksl=None*, *bal=None*, ***kwargs*)

Bases: xtrack.dress.DressedMultipoleData, *xtrack.base_element.BeamElement*

Beam element modeling a thin magnetic multipole. Parameters:

- order [int]: Horizontal shift. Default is 0.
- knl [m^-n, array]: Normalized integrated strength of the normal components.
- ksl [m^-n, array]: Normalized integrated strength of the skew components.
- hxl [rad]: Rotation angle of the reference trajectoryin the horizzontal plane.
- hyl [rad]: Rotation angle of the reference trajectory in the vertical plane.
- length [m]: Length of the originating thick multipole.

property knl

property ksl

Cavity

class xtrack.**Cavity**(*_xobject=None*, ***kwargs*)

Bases: xtrack.dress.DressedCavityData, *xtrack.base_element.BeamElement*

Beam element modeling an RF cavity. Parameters:

- voltage [V]: Voltage of the RF cavity. Default is 0.
- frequency [Hz]: Frequency of the RF cavity. Default is 0.
- lag [deg]: Phase seen by the reference particle. Default is 0.

RFMultipole

class xtrack.**RFMultipole**(*order=None*, *knl=None*, *ksl=None*, *pn=None*, *ps=None*, *bal=None*, *p=None*, ***kwargs*)

Bases: xtrack.dress.DressedRFMultipoleData, *xtrack.base_element.BeamElement*

Beam element modeling a thin modulated multipole, with strengths dependent on the z coordinate:

kn(z) = k_n cos(2pi w tau + pn/180*pi)

ks[n](z) = k_n cos(2pi w tau + pn/180*pi)

Its parameters are:

- order [int]: Horizontal shift. Default is 0.
- frequency [Hz]: Frequency of the RF cavity. Default is 0.
- knl [m^-n, array]: Normalized integrated strength of the normal components.
- ksl [m^-n, array]: Normalized integrated strength of the skew components.
- pn [deg, array]: Phase of the normal components.
- ps [deg, array]: Phase of the skew components.
- voltage [V]: Longitudinal voltage. Default is 0.
- lag [deg]: Longitudinal phase seen by the reference particle. Default is 0.

property knl

property ksl

set_knl(*value*, *order*)

set_ksl(*value*, *order*)

property pn

property ps

set_pn(*value*, *order*)

set_ps(*value*, *order*)

DipoleEdge

class xtrack.**DipoleEdge**(*r21=None*, *r43=None*, *h=None*, *e1=None*, *hgap=None*, *fint=None*, ***kwargs*)
Bases: xtrack.dress.DressedDipoleEdgeData, *xtrack.base_element.BeamElement*

Beam element modeling a dipole edge. Parameters:

- h [1/m]: Curvature.
- e1 [rad]: Face angle.
- hgap [m]: Equivalent gap.
- fint []: Fringe integral.

XYShift

class xtrack.**XYShift**(*_xobject=None*, ***kwargs*)
Bases: xtrack.dress.DressedXYShiftData, *xtrack.base_element.BeamElement*

Beam element modeling an transverse shift of the reference system. Parameters:

- dx [m]: Horizontal shift. Default is 0.
- dy [m]: Vertical shift. Default is 0.

SRotation

class xtrack.**SRotation**(*angle=0*, ***nargs*)
Bases: xtrack.dress.DressedSRotationData, *xtrack.base_element.BeamElement*

Beam element modeling an rotation of the reference system around the s axis. Parameters:

- angle [deg]: Rotation angle. Default is 0.

property angle

LimitEllipse

class xtrack.**LimitEllipse**(*a_squ=None*, *b_squ=None*, ***kwargs*)
Bases: xtrack.dress.DressedLimitEllipseData, *xtrack.base_element.BeamElement*

set_half_axes(*a*, *b*)

set_half_axes_squ(*a_squ*, *b_squ*)

LimitRect

class xtrack.**LimitRect**(*_xobject=None*, ***kwargs*)
Bases: xtrack.dress.DressedLimitRectData, *xtrack.base_element.BeamElement*

Monitors

class xtrack.**ParticlesMonitor**(*_context=None*, *_buffer=None*, *_offset=None*, *start_at_turn=None*, *stop_at_turn=None*, *num_particles=None*, *particle_id_range=None*, *auto_to_numpy=True*)

Bases: xtrack.dress.DressedParticlesMonitorData

at_element

at_turn

beta0

charge_ratio

chi

delta

gamma0

p0c

parent_particle_id

particle_id

psigma

px

py

rpp

rvv

s

state

weight

x

y

zeta

BeamElement base class

class xtrack.base_element.**BeamElement**(*_xobject=None*, ***kwargs*)

compile_track_kernel(*save_source_as=None*)

iscollective = False

to_dict()

track(*particles*)

xoinitialize(*_xobject=None*, ***kwargs*)

1.4.5 Xfields API

Table of Contents

Beam elements

Space Charge 3D

class xfields.**SpaceCharge3D**(*_context=None*, *_buffer=None*, *_offset=None*, *update_on_track=True*, *length=None*, *apply_z_kick=True*, *x_range=None*, *y_range=None*, *z_range=None*, *nx=None*, *ny=None*, *nz=None*, *dx=None*, *dy=None*, *dz=None*, *x_grid=None*, *y_grid=None*, *z_grid=None*, *rho=None*, *phi=None*, *solver=None*, *gamma0=None*, *fftplan=None*)

Simulates the effect of space charge on a bunch.

Parameters

- **context** (*XfContext*) – identifies the *context* on which the computation is executed.
- **update_on_track** (*bool*) – If `True` the beam field map is update at each interaction. If `False` the initial field map is used at each interaction (frozen model). The default is `True`.
- **length** (*float*) – the length of the space-charge interaction in meters.
- **apply_z_kick** (*bool*) – If `True`, the longitudinal kick on the particles is applied.
- **x_range** (*tuple*) – Horizontal extent (in meters) of the computing grid.
- **y_range** (*tuple*) – Vertical extent (in meters) of the computing grid.
- **z_range** (*tuple*) – Longitudina extent (in meters) of the computing grid.
- **nx** (*int*) – Number of cells in the horizontal direction.
- **ny** (*int*) – Number of cells in the vertical direction.
- **nz** (*int*) – Number of cells in the vertical direction.
- **dx** (*float*) – Horizontal cell size in meters. It can be provided alternatively to `nx`.

- **dy** (*float*) – Vertical cell size in meters. It can be provided alternatively to ny.
- **dz** (*float*) – Longitudinal cell size in meters.It can be provided alternatively to nz.
- **x_grid** (*np.ndarray*) – Equispaced array with the horizontal grid points (cell centers). It can be provided alternatively to x_range, dx/nx.
- **y_grid** (*np.ndarray*) – Equispaced array with the horizontal grid points (cell centers). It can be provided alternatively to y_range, dy/ny.
- **z_grid** (*np.ndarray*) – Equispaced array with the horizontal grid points (cell centers). It can be provided alternatively to z_range, dz/nz.
- **rho** (*np.ndarray*) – initial charge density at the grid points in Coulomb/m^3.
- **phi** (*np.ndarray*) – initial electric potential at the grid points in Volts. If not provided the phi is calculated from rho using the Poisson solver (if available).
- **solver** (*str or solver object*) – Defines the Poisson solver to be used to compute phi from rho. Accepted values are FFTSolver3D and FFTSolver2p5D. A Xfields solver object can also be provided. In case update_on_track``is ``False and phi is provided by the user, this argument can be omitted.
- **gamma0** (*float*) – Relativistic gamma factor of the beam. This is required only if the solver is FFTSolver3D.

Returns A space-charge 3D beam element.

Return type (*SpaceCharge3D*)

property iscollective
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

track(*particles*)
Computes and applies the space-charge forces for the provided set of particles.

Parameters **particles** (*Particles Object*) – Particles to be tracked.

Space Charge Bi-Gaussian

class xfields.**SpaceChargeBiGaussian**(*_context=None*, *_buffer=None*, *_offset=None*, *update_on_track=False*, *length=None*, *apply_z_kick=False*, *longitudinal_profile=None*, *mean_x=0.0*, *mean_y=0.0*, *sigma_x=None*, *sigma_y=None*, *min_sigma_diff=1e-10*)

track(*particles*)

property iscollective
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

property mean_x

property mean_y

property sigma_x

property sigma_y

classmethod from_xline(*xline_spacecharge=None*, *_context=None*, *_buffer=None*, *_offset=None*)

Beam-beam Bi-Gaussian 2D

class xfields.**BeamBeamBiGaussian2D**(*_context=None*, *_buffer=None*, *_offset=None*, *n_particles=None*, *q0=None*, *beta0=None*, *mean_x=0.0*, *mean_y=0.0*, *sigma_x=None*, *sigma_y=None*, *d_px=0.0*, *d_py=0.0*, *min_sigma_diff=1e-10*)

Simulates the effect of beam-beam on a bunch.

Parameters

- **context** (*xobjects context*) – identifies the *context* on which the computation is executed.
- **n_particles** (*float64*) – Number of particles in the colliding bunch.
- **q0** (*float64*) – Number of particles in the colliding bunch.
- **beta0** (*float64*) – Relativistic beta of the colliding bunch.
- **mean_x** (*float64*) – Horizontal position (in meters) of the colliding bunch. It can be updated after the object creation. Default is `0.`.
- **mean_y** (*float64*) – Vertical position (in meters) of the Gaussian distribution. It can be updated after the object creation. Default is `0.`.
- **sigma_x** (*float64*) – Horizontal r.m.s. size (in meters) of the colliding bunch. It can be updated after the object creation. Default is `None`.
- **sigma_y** (*float64*) – Vertical r.m.s. size (in meters) of the colliding bunch. It can be updated after the object creation. Default is `None`.

Returns A beam-beam element.

Return type (*BeamBeamBiGaussian2D*)

update(***kwargs*)

property mean_x

property mean_y

property sigma_x

property sigma_y

classmethod from_xline(*xline_beambeam=None*, *_context=None*, *_buffer=None*, *_offset=None*)

Beam-beam Bi-Gaussian 3D

class xfields.**BeamBeamBiGaussian3D**(*_xobject=None*, ***kwargs*)

classmethod from_xline(*xline_beambeam=None*, *_context=None*, *_buffer=None*, *_offset=None*)

Field Maps

Beam-beam Tri-linear Iterpolated Field Map

class xfields.fieldmaps.**TriLinearInterpolatedFieldMap**(*_context=None*, *_buffer=None*, *_offset=None*, *x_range=None*, *y_range=None*, *z_range=None*, *nx=None*, *ny=None*, *nz=None*, *dx=None*, *dy=None*, *dz=None*, *x_grid=None*, *y_grid=None*, *z_grid=None*, *rho=None*, *phi=None*, *solver=None*, *scale_coordinates_in_solver=(1.0, 1.0, 1.0)*, *updatable=True*, *fftplan=None*)

Builds a linear interpolator for a 3D field map. The map can be updated using the Parcle In Cell method.

Parameters

- **context** (*xobjects context*) – identifies the *context* on which the computation is executed.
- **x_range** (*tuple*) – Horizontal extent (in meters) of the computing grid.
- **y_range** (*tuple*) – Vertical extent (in meters) of the computing grid.
- **z_range** (*tuple*) – Longitudina extent (in meters) of the computing grid.
- **nx** (*int*) – Number of cells in the horizontal direction.
- **ny** (*int*) – Number of cells in the vertical direction.
- **nz** (*int*) – Number of cells in the vertical direction.
- **dx** (*float*) – Horizontal cell size in meters. It can be provided alternatively to nx.
- **dy** (*float*) – Vertical cell size in meters. It can be provided alternatively to ny.
- **dz** (*float*) – Longitudinal cell size in meters.It can be provided alternatively to nz.
- **x_grid** (*np.ndarray*) – Equispaced array with the horizontal grid points (cell centers). It can be provided alternatively to x_range, dx/nx.
- **y_grid** (*np.ndarray*) – Equispaced array with the horizontal grid points (cell centers). It can be provided alternatively to y_range, dy/ny.
- **z_grid** (*np.ndarray*) – Equispaced array with the horizontal grid points (cell centers). It can be provided alternatively to z_range, dz/nz.
- **rho** (*np.ndarray*) – initial charge density at the grid points in Coulomb/m^3.
- **phi** (*np.ndarray*) – initial electric potential at the grid points in Volts. If not provided the phi is calculated from rho using the Poisson solver (if available).
- **solver** (*str or solver object*) – Defines the Poisson solver to be used to compute phi from rho. Accepted values are FFTSolver3D and FFTSolver2p5D. A Xfields solver object can also be provided. In case update_on_track``is ``False and phi is provided by the user, this argument can be omitted.
- **scale_coordinates_in_solver** (*tuple*) – Three coefficients used to rescale the grid coordinates in the definition of the solver. The default is (1.,1.,1.).
- **updatable** (*bool*) – If True the field map can be updated after creation. Default is True.

Returns Interpolator object.

Return type (*TriLinearInterpolatedFieldMap*)

get_values_at_points(*x*, *y*, *z*, *return_rho=True*, *return_phi=True*, *return_dphi_dx=True*, *return_dphi_dy=True*, *return_dphi_dz=True*)

Returns the charge density, the field potential and its derivatives at the points specified by x, y, z. The output can be customized (see below). Zeros are returned for points outside the grid.

Parameters

- **x** (*float64 array*) – Horizontal coordinates at which the field is evaluated.
- **y** (*float64 array*) – Vertical coordinates at which the field is evaluated.
- **z** (*float64 array*) – Longitudinal coordinates at which the field is evaluated.
- **return_rho** (*bool*) – If `True`, the charge density at the given points is returned.
- **return_phi** (*bool*) – If `True`, the potential at the given points is returned.
- **return_dphi_dx** (*bool*) – If `True`, the horizontal derivative of the potential at the given points is returned.
- **return_dphi_dy** – If `True`, the vertical derivative of the potential at the given points is returned.
- **return_dphi_dz** – If `True`, the longitudinal derivative of the potential at the given points is returned.

Returns The required quantities at the provided points.

Return type (tuple of float64 array)

update_from_particles(*particles=None*, *x_p=None*, *y_p=None*, *z_p=None*, *ncharges_p=None*, *state_p=None*, *q0_coulomb=None*, *reset=True*, *update_phi=True*, *solver=None*, *force=False*)

Updates the charge density at the grid using a given set of particles, which can be provided by a particles object or by individual arrays. The potential can be optionally updated accordingly.

Parameters

- **particles** (*xtrack.Particles*) – xtrack particle object.
- **x_p** (*float64 array*) – Horizontal coordinates of the macroparticles.
- **y_p** (*float64 array*) – Vertical coordinates of the macroparticles.
- **z_p** (*float64 array*) – Longitudinal coordinates of the macroparticles.
- **ncharges_p** (*float64 array*) – Number of reference charges in the macroparticles.
- **state_p** (*int64, array*) – particle state (>0 active, lost otherwise)
- **q0_coulomb** (*float64*) – Reference charge in Coulomb.
- **reset** (*bool*) – If `True` the stored charge density is overwritten with the provided one. If `False` the provided charge density is added to the stored one. The default is `True`.
- **update_phi** (*bool*) – If `True` the stored potential is recalculated from the stored charge density.
- **solver** (*Solver object*) – solver object to be used to solve Poisson's equation (compute phi from rho). If `None` is provided the solver attached to the fieldmap is used (if any). The default is `None`.
- **force** (*bool*) – If `True` the potential is updated even if the map is declared as not updateable. The default is `False`.

update_rho(*rho*, *reset=True*, *force=False*)
Updates the charge density on the grid.

Parameters

- **rho** (*float64 array*) – Charge density at the grid points in C/m^3.
- **reset** (*bool*) – If True the stored charge density is overwritten with the provided one. If False the provided charge density is added to the stored one. The default is True.
- **force** (*bool*) – If True the charge density is updated even if the map is declared as not updateable. The default is False.

update_phi(*phi*, *reset=True*, *force=False*)
Updates the potential on the grid. The stored derivatives are also updated.

Parameters

- **rho** (*float64 array*) – Potential at the grid points.
- **reset** (*bool*) – If True the stored potential is overwritten with the provided one. If False the provided potential is added to the stored one. The default is True.
- **force** (*bool*) – If True the potential is updated even if the map is declared as not updateable. The default is False.

update_phi_from_rho(*solver=None*)
Updates the potential on the grid (phi) from the charge density on the grid (phi). It requires a Poisson solver object. If none is provided the one attached to the fieldmap is used (if any).

Parameters **solver** (*Solver object*) – solver object to be used to solve Poisson's equation. If None is provided the solver attached to the fieldmap is used (if any). The default is None.

generate_solver(*solver*, *fftplan*)
Generates a Poisson solver associated to the defined grid.

Parameters

- **solver** (*str*) – Defines the Poisson solver to be used
- **and** (*to compute phi from rho. Accepted values are FFTSolver3D*) –
- **FFTSolver2p5D.** –

Returns Solver object associated to the defined grid.

Return type (Solver)

property x_grid
Array with the horizontal grid points (cell centers).

property y_grid
Array with the vertical grid points (cell centers).

property z_grid
Array with the longitudinal grid points (cell centers).

property nx
Number of cells in the horizontal direction.

property ny
Number of cells in the vertical direction.

property nz
Number of cells in the longitudinal direction.

property dx
Horizontal cell size in meters.

property dy
Vertical cell size in meters.

property dz
Longitudinal cell size in meters.

property rho

property phi
Electric potential at the grid points in Volts.

property dphi_dx

property dphi_dy

property dphi_dz

Beam-beam Bi-Gaussian Field Map

class xfields.fieldmaps.**BiGaussianFieldMap**(*_context=None*, *_buffer=None*, *_offset=None*, *mean_x=0.0*, *mean_y=0.0*, *sigma_x=None*, *sigma_y=None*, *min_sigma_diff=1e-10*, *updatable=True*)

Builds a field-map object that provides the fields generated by a normalized 2D Gaussian distribution (Bassetti-Erskine formula).

Parameters

- **context** (*xobjects context*) – identifies the *context* on which the computation is executed.
- **mean_x** (*float64*) – Horizontal position (in meters) of the Gaussian distribution. It can be updated after the object creation. Default is `0.`.
- **mean_y** (*float64*) – Vertical position (in meters) of the Gaussian distribution. It can be updated after the object creation. Default is `0.`.
- **sigma_x** (*float64*) – Horizontal r.m.s. size (in meters) of the Gaussian distribution. It can be updated after the object creation. Default is `None`.
- **sigma_y** (*float64*) – Vertical r.m.s. size (in meters) of tthe Gaussian distribution. It can be updated after the object creation. Default is `None`.
- **min_sigma_diff** (*float64*) – Difference between sigma_x and sigma_y (in meters) below which round distribution is assumed.
- **updatable** (*bool*) – If `True` the field map can be updated after creation. Default is `True`.

Returns Field map object.

Return type (*BiGaussianFieldMap*)

update_from_particles(*x_p*, *y_p*, *z_p*, *ncharges_p*, *q0_coulomb*, *reset=True*, *update_phi=True*, *solver=None*, *force=False*)

update_rho(*rho*, *reset*)

update_phi(*phi*, *reset=True*, *force=False*)

update_phi_from_rho(*solver=None*)

generate_solver(*solver*)

Solvers

FFT Solver 3D

class xfields.solvers.**FFTSolver3D**(*dx*, *dy*, *dz*, *nx*, *ny*, *nz*, *context=None*, *fftplan=None*)

Creates a Poisson solver object that solves the full 3D Poisson equation using the FFT method (free space).

Parameters

- **nx** (*int*) – Number of cells in the horizontal direction.
- **ny** (*int*) – Number of cells in the vertical direction.
- **nz** (*int*) – Number of cells in the vertical direction.
- **dx** (*float*) – Horizontal cell size in meters.
- **dy** (*float*) – Vertical cell size in meters.
- **dz** (*float*) – Longitudinal cell size in meters.
- **context** (*XfContext*) – identifies the *context* on which the computation is executed.

Returns Poisson solver object.

Return type (*FFTSolver3D*)

solve(*rho*)

Solves Poisson's equation in free space for a given charge density.

Parameters **rho** (*float64 array*) – charge density at the grid points in Coulomb/m^3.

Returns electric potential at the grid points in Volts.

Return type phi (float64 array)

FFT Solver 2.5D

class xfields.solvers.**FFTSolver2p5D**(*dx*, *dy*, *dz*, *nx*, *ny*, *nz*, *context=None*, *fftplan=None*)

Creates a Poisson solver object that solve's Poisson equation in the 2.5D aaoroximation equation the FFT method (free space).

Parameters

- **nx** (*int*) – Number of cells in the horizontal direction.
- **ny** (*int*) – Number of cells in the vertical direction.
- **nz** (*int*) – Number of cells in the vertical direction.
- **dx** (*float*) – Horizontal cell size in meters.
- **dy** (*float*) – Vertical cell size in meters.
- **dz** (*float*) – Longitudinal cell size in meters.
- **context** (*XfContext*) – identifies the *context* on which the computation is executed.

Returns Poisson solver object.

Return type (*FFTSolver3D*)

solve(*rho*)

Solves Poisson's equation in free space for a given charge density.

Parameters **rho** (*float64 array*) – charge density at the grid points in Coulomb/m^3.

Returns electric potential at the grid points in Volts.

Return type phi (float64 array)

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z