Cheetah

Release 0.6.3

Jan Kaiser, Chenran Xu

Mar 28, 2024

EXAMPLES

Cheetah is a particle tracking accelerator we built specifically to speed up the training of reinforcement learning models.

GitHub repository: https://github.com/desy-ml/cheetah

Paper: https://arxiv.org/abs/2401.05815

CHAPTER

ONE

INSTALLATION

Simply install *Cheetah* from PyPI by running the following command.

```
pip install cheetah-accelerator
```

CHAPTER

TWO

EXAMPLES

We provide some examples to demonstrate some features of *Cheetah* and show how to use them. They provide a good entry point to using *Cheetah*, but they do not represent its full functionality. To move beyond the examples, please refer to the in-depth documentation. If you feel like other examples should be added, feel free to open an issue on GitHub.

2.1 Tracking through a simple lattice

In this example, we create a custom lattice and track a beam through it. We start with some imports.

[1]:
```
import torch

from cheetah import (
    BPM,
    Drift,
    HorizontalCorrector,
    ParticleBeam,
    Segment,
    VerticalCorrector,
)
```

Lattices in *Cheetah* are represented by `Segments`. A `Segment` is created as follows.

[2]:
```
segment = Segment(
    elements=[
        BPM(name="BPM1SMATCH"),
        Drift(length=torch.tensor(1.0)),
        BPM(name="BPM6SMATCH"),
        Drift(length=torch.tensor(1.0)),
        VerticalCorrector(length=torch.tensor(0.3), name="V7SMATCH"),
        Drift(length=torch.tensor(0.2)),
        HorizontalCorrector(length=torch.tensor(0.3), name="H10SMATCH"),
        Drift(length=torch.tensor(7.0)),
        HorizontalCorrector(length=torch.tensor(0.3), name="H12SMATCH"),
        Drift(length=torch.tensor(0.05)),
        BPM(name="BPM13SMATCH"),
    ]
)
```

Note that many values must be passed to lattice elements as `torch.Tensors`. This is because *Cheetah* uses automatic differentiation to compute the gradient of the beam position at the end of the lattice with respect to the element strengths. This is necessary for gradient-based magnet setting optimisation.

Named lattice elements (i.e. elements that were given a `name` keyword argument) can be accessed by name and their parameters changed like so.

[3]:
```
segment.V7SMATCH.angle = torch.tensor(3.142e-3)
```

Next, we create a beam to track through the lattice. In this particular example, we import a beam from an Astra particle distribution file. Note that we are using a `ParticleBeam` here, which is a beam defined by individual particles. This is the most precise way to track a beam through a lattice, but also slower than the alternative `ParameterBeam` which is defined by the beam's parameters. Instead of importing beams from other simulation codes, you can also create beams from scratch, either using their parameters or their Twiss parameters.

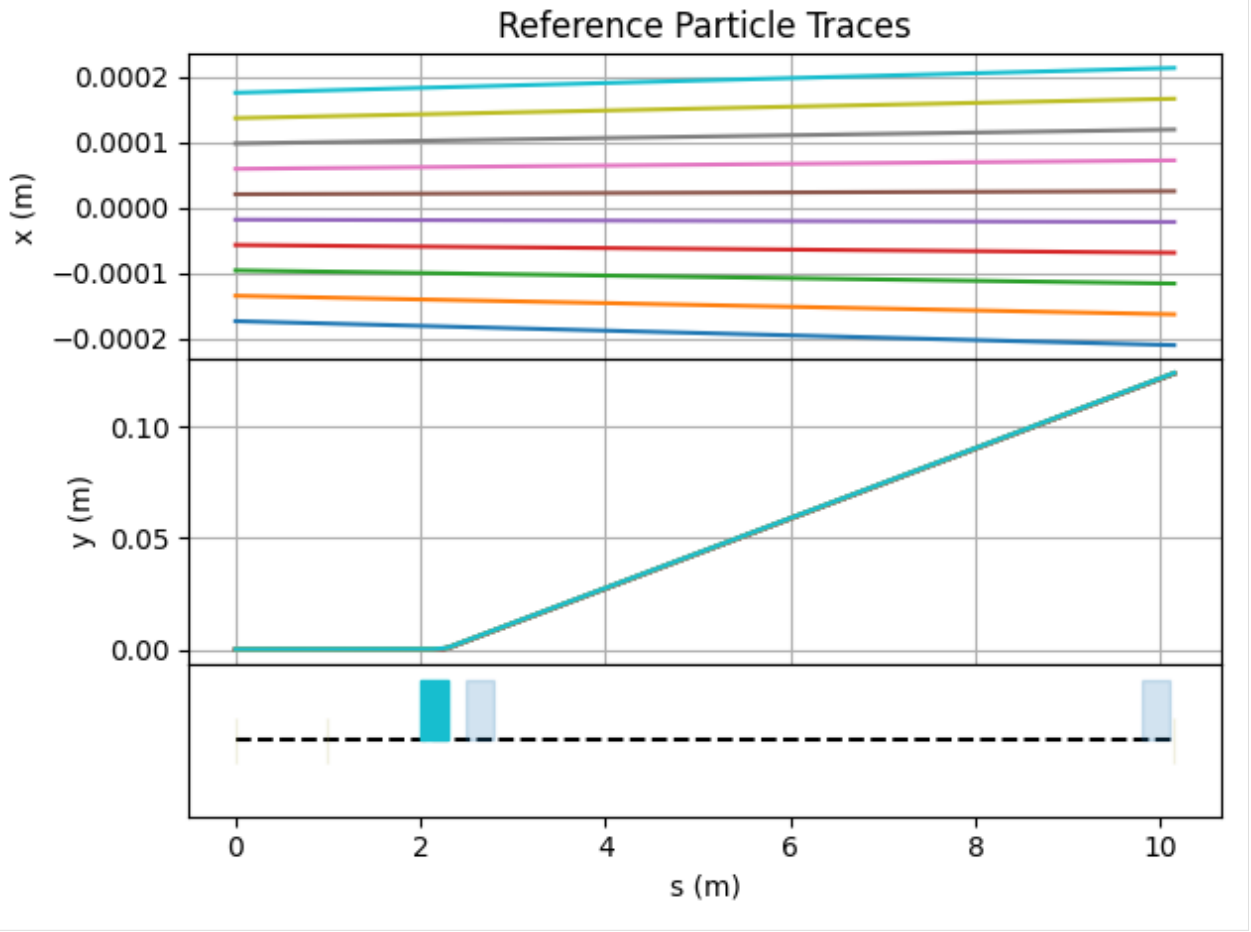
[4]:
```
incoming_beam = ParticleBeam.from_astra("../../tests/resources/ACHIP_EA1_2021.1351.001")
```

In order to track a beam through the segment, simply call the segment's `track` method.

[5]:
```
outgoing_beam = segment.track(incoming_beam)
```

You may plot a segment with reference particle traces bay calling

[6]:
```
segment.plot_overview(beam=incoming_beam)
```



where the optional keyword argument `beam` is the incoming beam represented by the reference particles. Cheetah will use a default incoming beam, if no beam is passed.

2.2 Converting lattices from other simulation codes

In this example, we demonstrate how to convert lattices from other simulation codes. At the moment, *Cheetah* supports the conversion of lattices *Ocelot* and *Bmad*.

[1]:
```
import ocelot
import torch

from cheetah import Segment
```

```
math_op.py: module Numba is not installed. Install it if you want speed up correlation␣
↪calculations
```

```
[INFO    ] : : beam.py: module NUMBA is not installed. Install it to speed up calculation
[INFO    ] : : : : : : : : high_order.py: module NUMBA is not installed. Install it to␣
↪speed up calculation
[INFO    ] radiation_py.py: module NUMBA is not installed. Install it to speed up␣
↪calculation
[INFO    ] radiation_py.py: module NUMBA is not installed. Install it to speed up␣
↪calculation
[INFO    ] csr.py: module NUMBA is not installed. Install it to speed up calculation
[INFO    ] csr.py: module PYFFTW is not installed. Install it to speed up calculation.
[INFO    ] csr.py: module NUMEXPR is not installed. Install it to speed up calculation
[INFO    ] wake3D.py: module NUMBA is not installed. Install it to speed up calculation
```

```
initializing ocelot...
import: module NUMBA is not installed. Install it to speed up calculation
import: module PYFFTW is not installed. Install it to speed up calculation
import: module NUMEXPR is not installed. Install it to speed up calculation
```

Lattice conversions can conveniently be done using class methods defined by the `Segment` class.

To convert an *Ocelot* cell that is stored as a Python variable, simply pass it to `Segment.from_ocelot()`:

[2]:
```
ocelot_cell = [
    ocelot.Drift(l=1.0),
    ocelot.Quadrupole(l=0.2, k1=4.2),
    ocelot.Drift(l=1.0),
]

ocelot_converted = Segment.from_ocelot(ocelot_cell)
ocelot_converted
```

[2]:
```
Segment(elements=ModuleList(
  (0): Drift(length=tensor(1.), name='ID_27628570_', device='cpu')
  (1): Quadrupole(length=tensor(0.2000), k1=tensor(4.2000), misalignment=tensor([0., 0.
↪]), tilt=tensor(0.), name='ID_99960050_', device="'cpu'")
  (2): Drift(length=tensor(1.), name='ID_94109468_', device='cpu')
), name='unnamed_element_0', device='cpu')
```

Bmad on the other are read from `.bmad` files. To convert the following *Bmad* lattice

[3]:
```
!cat ../../tests/resources/bmad_tutorial_lattice.bmad
```

```
! Lattice file: simple.bmad
beginning[beta_a] = 10. ! m a-mode beta function
```

(continues on next page)

(continued from previous page)

```
beginning[beta_b] = 10. ! m b-mode beta function
beginning[e_tot] = 10e6 ! eV    Or can set beginning[p0c]

parameter[geometry] = open  ! Or closed
parameter[particle] = electron  ! Reference particle.

d: drift, L = 0.5
b: sbend, L = 0.5, g = 1, e1 = 0.1, dg = 0.001 ! g = 1/design_radius
q: quadrupole, L = 0.6, k1 = 0.23

lat: line = (d, b, q) ! List of lattice elements
use, lat ! Line used to construct the lattice
```

, pass the file path to `Segment.from_bmad()`.

[4]:
```
bmad_converted = Segment.from_bmad("../../tests/resources/bmad_tutorial_lattice.bmad")
bmad_converted
```

[4]:
```
Segment(elements=ModuleList(
  (0): Drift(length=tensor(0.5000), name='d', device='cpu')
  (1): Dipole(length=tensor(0.5000), angle=tensor(0.), e1=tensor(0.1000),e2=tensor(0.),
↪tilt=tensor(0.),fringe_integral=tensor(0.),fringe_integral_exit=tensor(0.),
↪gap=tensor(0.),name='b', device='cpu')
  (2): Quadrupole(length=tensor(0.6000), k1=tensor(0.2300), misalignment=tensor([0., 0.
↪]), tilt=tensor(0.), name='q', device="'cpu'")
), name='lat', device='cpu')
```

2.3 Optimising *Cheetah* for speed

One of Cheetah's standout features is its computational speed. This is achieved through some optimisations under the hood, which the user never needs to worry about. Often, however, there further optimisations that can be made when knowledge on how the model will be used is available. For example, in many cases, one might load a large lattice of an entire facility that has thousands of elements, but then only ever changes a handful of these elements for the experiments at hand. For this case, Cheetah offers some opt-in optimisation features that can help speed up simulations significantly by an order of magnitude or more in some cases.

[1]:
```
import cheetah
import torch
```

[2]:
```
incoming_beam = cheetah.ParameterBeam.from_astra(
    "../../tests/resources/ACHIP_EA1_2021.1351.001"
)
```

Let's define a large lattice. With many quadrupole magnets and drift sections in the center and a pair of steerers at each end. We assume that the quadrupole magnets are at their design settings and will never be touched. Only the two steerers at each end are of interest to us, for example because we would like to train a neural network policy to steer the beam using these steerers. Furthermore, as many lattices do, there are a bunch of markers in this lattice. These markers may be helpful to mark certain positions along the beamline, but they don't actually add anything to the physics of the simulation.

[3]:
```
original_segment = cheetah.Segment(
    elements=[
        cheetah.HorizontalCorrector(
            length=torch.tensor(0.1), angle=torch.tensor(0.0), name="HCOR_1"
        ),
        cheetah.Drift(length=torch.tensor(0.3)),
        cheetah.VerticalCorrector(
            length=torch.tensor(0.1), angle=torch.tensor(0.0), name="VCOR_1"
        ),
        cheetah.Drift(length=torch.tensor(0.3)),
    ]
    + [
        cheetah.Quadrupole(length=torch.tensor(0.1), k1=torch.tensor(4.2)),
        cheetah.Drift(length=torch.tensor(0.2)),
        cheetah.Quadrupole(length=torch.tensor(0.1), k1=torch.tensor(-4.2)),
        cheetah.Drift(length=torch.tensor(0.2)),
        cheetah.Marker(),
        cheetah.Quadrupole(length=torch.tensor(0.1), k1=torch.tensor(0.0)),
        cheetah.Drift(length=torch.tensor(0.2)),
    ]
    * 150
    + [
        cheetah.HorizontalCorrector(
            length=torch.tensor(0.1), angle=torch.tensor(0.0), name="HCOR_2"
        ),
        cheetah.Drift(length=torch.tensor(0.3)),
        cheetah.VerticalCorrector(
            length=torch.tensor(0.1), angle=torch.tensor(0.0), name="VCOR_2"
        ),
        cheetah.Drift(length=torch.tensor(0.3)),
    ]
)
```

[4]:
```
len(original_segment.elements)
```

[4]:
```
1058
```

First, we test how long it takes to track a beam through this segment without any optimisations beyond the ones automatically done under the hood.

[5]:
```
%%timeit
original_segment.track(incoming_beam)
```

```
66.1 ms ± 178 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Just by removing unused markers, we already see a small performance improvement.

[6]:
```
markers_removed_segment = original_segment.without_inactive_markers()
```

[7]:
```
%%timeit
markers_removed_segment.track(incoming_beam)
```

```
65.3 ms ± 203 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Drift sections tend to be the cheapest elements to compute. At the same time, many elements in a lattice may be

switched off at any given time. When they are switched off, they behave almost exactly like drift sections, but they still require additional computations to arrive at this result. We can however safely replace them by actual `Drift` elements, which clearly speeds up computations.

[8]:
```
inactive_to_drifts_segment = original_segment.inactive_elements_as_drifts(
    except_for=["HCOR_1", "VCOR_1", "HCOR_2", "VCOR_2"]
)
len(inactive_to_drifts_segment.elements)
```

[8]:
```
1058
```

[9]:
```
%%timeit
inactive_to_drifts_segment.track(incoming_beam)
```

```
50 ms ± 198 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

The most significant improvement can be made by merging elements that are not expected to be changed in the future. For this, Cheetah offers the `transfer_maps_merged` method. This will by default merge the transfer maps of all elements in the segment. In almost all realistic applications, however, there are some elements the settings of which we wish to change in the future. By passing a list of their names to `except_for`, we can instruct Cheetah to only merge elements in between the passed elements.

NOTE: Transfer map merging can only be done for a constant incoming beam energy, because the transfer maps need to be computed before they can be merged, and computing them might require the beam energy at the entrance of the element that the transfer map belongs to. If you want to try a different beam energy, you will need to reapply the optimisations to the original lattice while passing a beam with the desired energy.

[10]:
```
transfer_maps_merged_segment = original_segment.transfer_maps_merged(
    incoming_beam=incoming_beam, except_for=["HCOR_1", "VCOR_1", "HCOR_2", "VCOR_2"]
)
len(transfer_maps_merged_segment.elements)
```

[10]:
```
8
```

[11]:
```
%%timeit
transfer_maps_merged_segment.track(incoming_beam)
```

```
96.2 µs ± 121 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

[12]:
```
transfer_maps_merged_segment
```

[12]:
```
Segment(elements=ModuleList(
  (0): HorizontalCorrector(length=tensor(0.1000), angle=tensor(0.), name='HCOR_1',␣
↪device='cpu')
  (1): Drift(length=tensor(0.3000), name='unnamed_element_0', device='cpu')
  (2): VerticalCorrector(length=tensor(0.1000), angle=tensor(0.), name='VCOR_1', device=
↪'cpu')
  (3): CustomTransferMap(name='unnamed_element_615', device='cpu')
  (4): HorizontalCorrector(length=tensor(0.1000), angle=tensor(0.), name='HCOR_2',␣
↪device='cpu')
  (5): Drift(length=tensor(0.3000), name='unnamed_element_9', device='cpu')
  (6): VerticalCorrector(length=tensor(0.1000), angle=tensor(0.), name='VCOR_2', device=
↪'cpu')
  (7): CustomTransferMap(name='unnamed_element_616', device='cpu')
), name='unnamed', device='cpu')
```

It is also possible and often advisable to combine optimisations. However, note that this might not always yield as much of an improvement as one may have hoped looking at the improvements delivered by each optimisation on its own. This is usually because these optimisations share some of their effects, i.e. if the first optimisation has already performed a change on the lattice that the second optimisation would have done as well, the second optimisation will not lead to a further speed improvement.

[13]:
```
fully_optimized_segment = (
    original_segment.without_inactive_markers()
    .inactive_elements_as_drifts(except_for=["HCOR_1", "VCOR_1", "HCOR_2", "VCOR_2"])
    .transfer_maps_merged(
        incoming_beam=incoming_beam, except_for=["HCOR_1", "VCOR_1", "HCOR_2", "VCOR_2"]
    )
)
len(fully_optimized_segment.elements)
```

[13]:
```
8
```

[14]:
```
fully_optimized_segment
```

[14]:
```
Segment(elements=ModuleList(
  (0): HorizontalCorrector(length=tensor(0.1000), angle=tensor(0.), name='HCOR_1',␣
↪device='cpu')
  (1): Drift(length=tensor(0.3000), name='unnamed_element_617', device='cpu')
  (2): VerticalCorrector(length=tensor(0.1000), angle=tensor(0.), name='VCOR_1', device=
↪'cpu')
  (3): CustomTransferMap(name='unnamed_element_1221', device='cpu')
  (4): HorizontalCorrector(length=tensor(0.1000), angle=tensor(0.), name='HCOR_2',␣
↪device='cpu')
  (5): Drift(length=tensor(0.3000), name='unnamed_element_1219', device='cpu')
  (6): VerticalCorrector(length=tensor(0.1000), angle=tensor(0.), name='VCOR_2', device=
↪'cpu')
  (7): CustomTransferMap(name='unnamed_element_1222', device='cpu')
), name='unnamed', device='cpu')
```

[15]:
```
%%timeit
fully_optimized_segment.track(incoming_beam)
```

```
96.9 µs ± 780 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

[ ]:

2.4 Gradient-based optimisation using *Cheetah* and *PyTorch*

Cheetah is a differentiable beam dynamics simulation engine, making it ideally suited to gradient-based optimisation, for example for optimisation magnet settings, lattice geometries or even for system identification. *Cheetah*'s tight integration with *PyTorch* makes this particularly easy as it opens up the use of *PyTorch*'s automatic differentiation capabilities and toolchain.

In this example, we demonstrate how to use *Cheetah* for **magnet setting optimisation** and how to **add custom normalisation** to that same task.

[1]:
```
import matplotlib.pyplot as plt
import torch
```

(continues on next page)

(continued from previous page)

```
import torch.nn as nn
import torch.nn.functional as F

import cheetah
```

2.4.1 Simple example (without normalisation)

We start by creating the lattice section and incoming beam.

[2]:
```
ares_ea = cheetah.Segment.from_lattice_json("ARESlatticeStage3v1_9.json").subcell(
    "AREASOLA1", "AREABSCR1"
)
```

[3]:
```
incoming_beam = cheetah.ParticleBeam.from_astra(
    "../../tests/resources/ACHIP_EA1_2021.1351.001"
)
incoming_beam
```

[3]:
```
ParticleBeam(n=100000, mu_x=tensor(8.2413e-07), mu_xp=tensor(5.9885e-08), mu_y=tensor(-1.
↪7276e-06), mu_yp=tensor(-1.1746e-07), sigma_x=tensor(0.0002), sigma_xp=tensor(3.6794e-
↪06), sigma_y=tensor(0.0002), sigma_yp=tensor(3.6941e-06), sigma_s=tensor(8.0116e-06),␣
↪sigma_p=tensor(0.0023), energy=tensor(1.0732e+08, dtype=torch.float64))
```

By default, *Cheetah* assumes that no part of its simulation requires differentiation and therefore does not track gradients, all parameters are of type `torch.Tensor`. To enable gradient tracking for parameters you would like to optimise over, you need to wrap them in `torch.nn.Parameter`, either when creating your elements and beams, or later on.

In this example, we would like to optimise over the settings of three quadrupoles and two steerers in the experimental area at the *ARES* accelerator facility at DESY. In this case, we will need to redefine the `k1` and `angle` parameters of the magnets as `torch.nn.Parameter`.

Note: You could simply wrap the value of the parameters as the value it already has, e.g.

```
ares_ea.AREAMQZM1.k1 = nn.Parameter(ares_ea.AREAMQZM1.k1)
```

[4]:
```
ares_ea.AREAMQZM1.k1 = nn.Parameter(ares_ea.AREAMQZM1.k1)
ares_ea.AREAMQZM2.k1 = nn.Parameter(ares_ea.AREAMQZM2.k1)
ares_ea.AREAMCVM1.angle = nn.Parameter(ares_ea.AREAMCVM1.angle)
ares_ea.AREAMQZM3.k1 = nn.Parameter(ares_ea.AREAMQZM3.k1)
ares_ea.AREAMCHM1.angle = nn.Parameter(ares_ea.AREAMCHM1.angle)
```

Next, we define the function that will do the actual optimisation. The goal of our optimisation is to tune the transverse beam parameters `[mu_x, sigma_x, mu_y, sigma_y]` towards some target beam parameters on a diagnostic screen at the end of the considered lattice segment. Hence, we pass the target beam parameters to the `train` function and make use of *PyTorch*'s `torch.nn.function.mse_loss` function. Note that we can easily make use of *PyTorch*'s `Adam` optimiser implementation. As a result the following code looks very similar to a standard *PyTorch* optimisation loop for the training of neural networks.

[5]:
```
def train(num_steps: int, target_beam_parameters: torch.Tensor, lr=0.1) -> dict:
    beam_parameter_history = []
    magnet_setting_history = []
    loss_history = []
```

(continues on next page)

(continued from previous page)

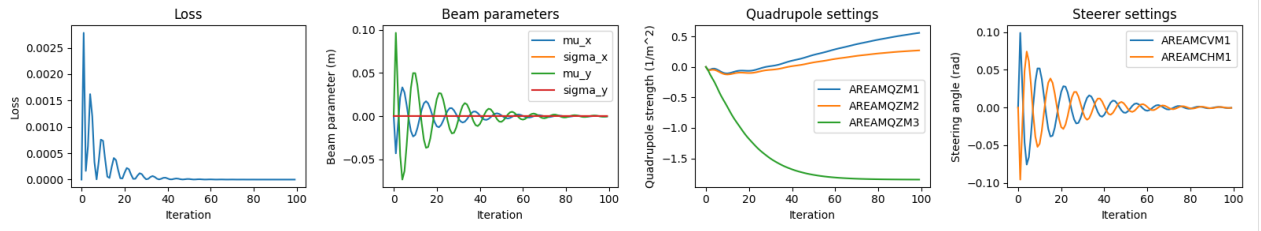
```
    optimizer = torch.optim.Adam(ares_ea.parameters(), lr=lr)

    for _ in range(num_steps):
        optimizer.zero_grad()

        outgoing_beam = ares_ea.track(incoming_beam)

        observed_beam_parameters = torch.stack(
            [
                outgoing_beam.mu_x,
                outgoing_beam.sigma_x,
                outgoing_beam.mu_y,
                outgoing_beam.sigma_y,
            ]
        )
        loss = F.mse_loss(observed_beam_parameters, target_beam_parameters)

        loss.backward()

        # Log magnet settings and beam parameters
        loss_history.append(loss.item())
        beam_parameter_history.append(observed_beam_parameters.detach().numpy())
        magnet_setting_history.append(
            torch.stack(
                [
                    ares_ea.AREAMQZM1.k1,
                    ares_ea.AREAMQZM2.k1,
                    ares_ea.AREAMCVM1.angle,
                    ares_ea.AREAMQZM3.k1,
                    ares_ea.AREAMCHM1.angle,
                ]
            )
            .detach()
            .numpy()
        )
        optimizer.step()

    history = {
        "loss": loss_history,
        "beam_parameters": beam_parameter_history,
        "magnet_settings": magnet_setting_history,
    }
    return history
```

We now simply run the optimisation function with a target beam that is centred on the origin and focused to be as small as possible.

[6]:
```
history = train(num_steps=100, target_beam_parameters=torch.zeros(4))
```

The returned `history` dictionary allows us to plot the evolution of the optimisation process.

[7]:
```
plt.figure(figsize=(16, 3))

plt.subplot(1, 4, 1)
plt.plot(history["loss"])
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.title("Loss")

plt.subplot(1, 4, 2)
plt.plot([record[0] for record in history["beam_parameters"]], label="mu_x")
plt.plot([record[1] for record in history["beam_parameters"]], label="sigma_x")
plt.plot([record[2] for record in history["beam_parameters"]], label="mu_y")
plt.plot([record[3] for record in history["beam_parameters"]], label="sigma_y")
plt.xlabel("Iteration")
plt.ylabel("Beam parameter (m)")
plt.title("Beam parameters")
plt.legend()

plt.subplot(1, 4, 3)
plt.plot([record[0] for record in history["magnet_settings"]], label="AREAMQZM1")
plt.plot([record[1] for record in history["magnet_settings"]], label="AREAMQZM2")
plt.plot([record[3] for record in history["magnet_settings"]], label="AREAMQZM3")
plt.xlabel("Iteration")
plt.ylabel("Quadrupole strength (1/m^2)")
plt.title("Quadrupole settings")
plt.legend()

plt.subplot(1, 4, 4)
plt.plot([record[2] for record in history["magnet_settings"]], label="AREAMCVM1")
plt.plot([record[4] for record in history["magnet_settings"]], label="AREAMCHM1")
plt.xlabel("Iteration")
plt.ylabel("Steering angle (rad)")
plt.title("Steerer settings")
plt.legend()

plt.tight_layout()
plt.show()
```



Success! We can observe that the optimisation converges to a solution that is close to the target beam parameters.

However, we can also observe that the quadrupole converges very slowly, indicating that the learning rate is too small, while the steerers keep overshooting the target, indicating that the learning rate is too large. This is a common problem in gradient-based optimisation caused by the very different scales of `k1` and `angle`, and can be solved by **normalising** the parameters under optimisation.

2.4.2 Normalising parameters in gradient-based optimisation

In the following example we demonstrate how to **normalise** the parameters under optimisation with *Cheetah*. The same principle can also be applied to other custom mechanisms one might like to build around the lattice optimisation process, e.g. to add custom constraints, coupled parameters, etc.

To achieve this, we wrap the lattice section in a `torch.nn.Module` and define a `forward` function that applies the normalisation to the parameters before passing them to the lattice section.

Note that this time, simply for the fun of it, we also start with randomly initialised magnet settings.

[8]:
```
class NormalizedARESExperimentalArea(nn.Module):
    """
    Wrapper around the AREA experimental area that holds normalised versions of the
    magnet settings as its trainable parameters.
    """

    QUADRUPOLE_LIMIT = 5.0
    STEERER_LIMIT = 6.1782e-3

    def __init__(self) -> None:
        super().__init__()
        self.ares_ea = cheetah.Segment.from_lattice_json(
            "ARESlatticeStage3v1_9.json"
        ).subcell("AREASOLA1", "AREABSCR1")

        # self.normalized_quadrupole_strengths = nn.Parameter(
        #     torch.tensor([10.0, -10.0, 10.0]) / self.QUADRUPOLE_LIMIT
        # )
        # self.normalized_steering_angles = nn.Parameter(
        #     torch.tensor([1e-3, -1e-3]) / self.STEERER_LIMIT
        # )

        self.normalized_quadrupole_strengths = nn.Parameter(torch.randn(3) * 2 - 1)
        self.normalized_steering_angles = nn.Parameter(torch.randn(2) * 2 - 1)

    def forward(self, incoming_beam: cheetah.Beam):
        self.ares_ea.AREAMQZM1.k1 = (
            self.normalized_quadrupole_strengths[0] * self.QUADRUPOLE_LIMIT
        )
        self.ares_ea.AREAMQZM2.k1 = (
            self.normalized_quadrupole_strengths[1] * self.QUADRUPOLE_LIMIT
        )
        self.ares_ea.AREAMCVM1.angle = (
            self.normalized_steering_angles[0] * self.STEERER_LIMIT
        )
        self.ares_ea.AREAMQZM3.k1 = (
            self.normalized_quadrupole_strengths[2] * self.QUADRUPOLE_LIMIT
        )
        self.ares_ea.AREAMCHM1.angle = (
            self.normalized_steering_angles[1] * self.STEERER_LIMIT
        )

        return self.ares_ea.track(incoming_beam)
```

[9]:
```
normalized_ares_ea = NormalizedARESExperimentalArea()
```

We then redefine the `train` function to use the `torch.nn.Module` instead of the lattice section directly.

Note that we also chose to apply normalisation to the beam parameters. This is not strictly necessary, but can help to improve the stability of the optimisation process.

[10]:
```
def train_normalized(num_steps: int, target_beam_parameters: torch.Tensor):
    beam_parameter_history = []
    magnet_setting_history = []
    loss_history = []

    optimizer = torch.optim.Adam(normalized_ares_ea.parameters(), lr=1e-1)

    for _ in range(num_steps):
        optimizer.zero_grad()

        outgoing_beam = normalized_ares_ea(incoming_beam)
        observed_beam_parameters = torch.stack(
            [
                outgoing_beam.mu_x,
                outgoing_beam.sigma_x,
                outgoing_beam.mu_y,
                outgoing_beam.sigma_y,
            ]
        )
        loss = F.mse_loss(
            observed_beam_parameters / 2e-3, target_beam_parameters / 2e-3
        )

        loss.backward()

        # Log magnet settings and beam parameters
        loss_history.append(loss.item())
        beam_parameter_history.append(observed_beam_parameters.detach().numpy())
        magnet_setting_history.append(
            torch.stack(
                [
                    normalized_ares_ea.ares_ea.AREAMQZM1.k1,
                    normalized_ares_ea.ares_ea.AREAMQZM2.k1,
                    normalized_ares_ea.ares_ea.AREAMCVM1.angle,
                    normalized_ares_ea.ares_ea.AREAMQZM3.k1,
                    normalized_ares_ea.ares_ea.AREAMCHM1.angle,
                ]
            )
            .detach()
            .numpy()
        )

        optimizer.step()

    history = {
        "loss": loss_history,
```

(continues on next page)

(continued from previous page)

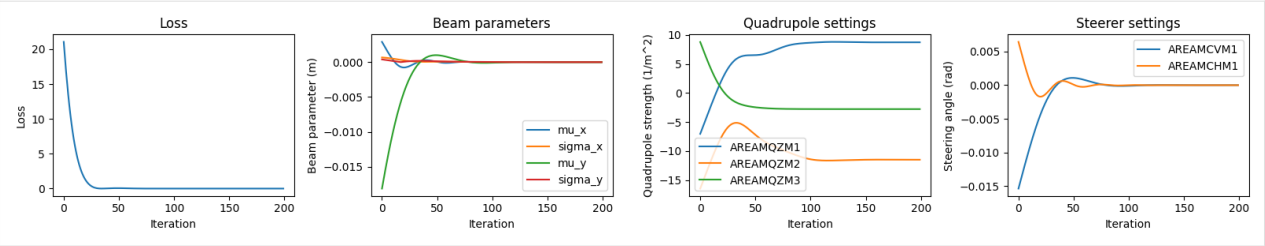
```
        "beam_parameters": beam_parameter_history,
        "magnet_settings": magnet_setting_history,
    }
    return history
```

Now we run or new `train_normalized` function with the same target beam as before.

[11]:
```
history = train_normalized(num_steps=200, target_beam_parameters=torch.zeros(4))
```

Then we plot the evolution of the optimisation process again.

[12]:
```
plt.figure(figsize=(16, 3))

plt.subplot(1, 4, 1)
plt.plot(history["loss"])
# plt.yscale("log")
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.title("Loss")

plt.subplot(1, 4, 2)
plt.plot([record[0] for record in history["beam_parameters"]], label="mu_x")
plt.plot([record[1] for record in history["beam_parameters"]], label="sigma_x")
plt.plot([record[2] for record in history["beam_parameters"]], label="mu_y")
plt.plot([record[3] for record in history["beam_parameters"]], label="sigma_y")
plt.xlabel("Iteration")
plt.ylabel("Beam parameter (m)")
plt.title("Beam parameters")
plt.legend()

plt.subplot(1, 4, 3)
plt.plot([record[0] for record in history["magnet_settings"]], label="AREAMQZM1")
plt.plot([record[1] for record in history["magnet_settings"]], label="AREAMQZM2")
plt.plot([record[3] for record in history["magnet_settings"]], label="AREAMQZM3")
plt.xlabel("Iteration")
plt.ylabel("Quadrupole strength (1/m^2)")
plt.title("Quadrupole settings")
plt.legend()

plt.subplot(1, 4, 4)
plt.plot([record[2] for record in history["magnet_settings"]], label="AREAMCVM1")
plt.plot([record[4] for record in history["magnet_settings"]], label="AREAMCHM1")
plt.xlabel("Iteration")
plt.ylabel("Steering angle (rad)")
plt.title("Steerer settings")
plt.legend()

plt.tight_layout()
plt.show()
```



As you can see this already looks much better than it did without normalisation.

CHAPTER

THREE

DOCUMENTATION

For more advanced usage, please refer to the in-depth documentation.

3.1 Accelerator

class accelerator.**Aperture**(*x_max: Tensor | Parameter | None = None*, *y_max: Tensor | Parameter | None = None*, *shape: Literal['rectangular', 'elliptical'] = 'rectangular'*, *is_active: bool = True*, *name: str | None = None*, *device=None*, *dtype=torch.float32*)

Physical aperture.

Parameters

- **x_max** – half size horizontal offset in [m]
- **y_max** – half size vertical offset in [m]
- **shape** – Shape of the aperture. Can be "rectangular" or "elliptical".
- **is_active** – If the aperture actually blocks particles.
- **name** – Unique identifier of the element.

property defining_features: list[str]

List of features that define the element. Used to compare elements for equality and to save them.

NOTE: When overriding this property, make sure to call the super method and extend the list it returns.

property is_skippable: bool

Whether the element can be skipped during tracking. If *True*, the element's transfer map is combined with the transfer maps of surrounding skipable elements.

plot(*ax: Axes*, *s: float*) → None

Plot a representation of this element into a *matplotlib* Axes at position *s*.

Parameters

- **ax** – Axes to plot the representation into.
- **s** – Position of the object along s in meters.

split(*resolution: Tensor*) → list[*Element*]

Split the element into slices no longer than *resolution*. Some elements may not be splittable, in which case a list containing only the element itself is returned.

Parameters

resolution – Length of the longest allowed split in meters.

Returns
Ordered sequence of sliced elements.

`track`(*incoming: Beam*) → Beam

Track particles through the element. The input can be a *ParameterBeam* or a *ParticleBeam*.

Parameters
`incoming` – Beam of particles entering the element.

Returns
Beam of particles exiting the element.

`training: bool`

`transfer_map`(*energy: Tensor*) → Tensor

Generates the element's transfer map that describes how the beam and its particles are transformed when traveling through the element. The state vector consists of 6 values with a physical meaning: (in the trace space notation)

- x: Position in x direction
- xp: Angle in x direction
- y: Position in y direction
- yp: Angle in y direction
- s: Position in longitudinal direction, the zero value is set to the

reference position (usually the center of the pulse) - p: Relative energy deviation from the reference particle

$p = \frac{\Delta E}{p_0 C}$

As well as a seventh value used to add constants to some of the prior values if necessary. Through this seventh state, the addition of constants can be represented using a matrix multiplication.

Parameters
`energy` – Reference energy of the Beam. Read from the fed-in Cheetah Beam.

Returns
A 7x7 Matrix for further calculations.

`class accelerator.BPM`(*is_active: bool = False*, *name: str | None = None*)

Beam Position Monitor (BPM) in a particle accelerator.

Parameters

- `is_active` – If *True* the BPM is active and will record the beam's position. If *False* the BPM is inactive and will not record the beam's position.
- `name` – Unique identifier of the element.

`property defining_features: list[str]`

List of features that define the element. Used to compare elements for equality and to save them.

NOTE: When overriding this property, make sure to call the super method and extend the list it returns.

`property is_skippable: bool`

Whether the element can be skipped during tracking. If *True*, the element's transfer map is combined with the transfer maps of surrounding skipable elements.

plot(*ax: Axes*, *s: float*) → None

Plot a representation of this element into a *matplotlib* Axes at position *s*.

Parameters

- **ax** – Axes to plot the representation into.
- **s** – Position of the object along s in meters.

split(*resolution: Tensor*) → list[*Element*]

Split the element into slices no longer than *resolution*. Some elements may not be splittable, in which case a list containing only the element itself is returned.

Parameters
resolution – Length of the longest allowed split in meters.

Returns
Ordered sequence of sliced elements.

track(*incoming: Beam*) → Beam

Track particles through the element. The input can be a *ParameterBeam* or a *ParticleBeam*.

Parameters
incoming – Beam of particles entering the element.

Returns
Beam of particles exiting the element.

training: bool

transfer_map(*energy: Tensor*) → Tensor

Generates the element's transfer map that describes how the beam and its particles are transformed when traveling through the element. The state vector consists of 6 values with a physical meaning: (in the trace space notation)

- x: Position in x direction
- xp: Angle in x direction
- y: Position in y direction
- yp: Angle in y direction
- s: Position in longitudinal direction, the zero value is set to the

reference position (usually the center of the pulse) - p: Relative energy deviation from the reference particle

$p = \frac{\Delta E}{p_0 C}$

As well as a seventh value used to add constants to some of the prior values if necessary. Through this seventh state, the addition of constants can be represented using a matrix multiplication.

Parameters
energy – Reference energy of the Beam. Read from the fed-in Cheetah Beam.

Returns
A 7x7 Matrix for further calculations.

class accelerator.**Cavity**(*length: Tensor | Parameter*, *voltage: Tensor | Parameter | None = None*, *phase: Tensor | Parameter | None = None*, *frequency: Tensor | Parameter | None = None*, *name: str | None = None*, *device=None*, *dtype=torch.float32*)

Accelerating cavity in a particle accelerator.

Parameters

- **length** – Length in meters.
- **voltage** – Voltage of the cavity in volts.
- **phase** – Phase of the cavity in degrees.
- **frequency** – Frequency of the cavity in Hz.
- **name** – Unique identifier of the element.

property defining_features: list[str]

List of features that define the element. Used to compare elements for equality and to save them.

NOTE: When overriding this property, make sure to call the super method and extend the list it returns.

property is_active: bool

property is_skippable: bool

Whether the element can be skipped during tracking. If *True*, the element's transfer map is combined with the transfer maps of surrounding skipable elements.

plot(*ax: Axes*, *s: float*) → None

Plot a representation of this element into a *matplotlib* Axes at position *s*.

Parameters

- **ax** – Axes to plot the representation into.
- **s** – Position of the object along s in meters.

split(*resolution: Tensor*) → list[*Element*]

Split the element into slices no longer than *resolution*. Some elements may not be splittable, in which case a list containing only the element itself is returned.

Parameters
resolution – Length of the longest allowed split in meters.

Returns
Ordered sequence of sliced elements.

track(*incoming: Beam*) → Beam

Track particles through the cavity. The input can be a *ParameterBeam* or a *ParticleBeam*. For a cavity, this does a little more than just the transfer map multiplication done by most elements.

Parameters
incoming – Beam of particles entering the element.

Returns
Beam of particles exiting the element.

training: bool

transfer_map(*energy: Tensor*) → Tensor

Generates the element's transfer map that describes how the beam and its particles are transformed when traveling through the element. The state vector consists of 6 values with a physical meaning: (in the trace space notation)

- x: Position in x direction
- xp: Angle in x direction
- y: Position in y direction
- yp: Angle in y direction

- s: Position in longitudinal direction, the zero value is set to the

reference position (usually the center of the pulse) - p: Relative energy deviation from the reference particle

$p = \frac{\Delta E}{p_0 C}$

As well as a seventh value used to add constants to some of the prior values if necessary. Through this seventh state, the addition of constants can be represented using a matrix multiplication.

Parameters
energy – Reference energy of the Beam. Read from the fed-in Cheetah Beam.

Returns
A 7x7 Matrix for further calculations.

class accelerator.**CustomTransferMap**(*transfer_map: Tensor | Parameter*, *length: Tensor | None = None*, *name: str | None = None*, *device=None*, *dtype=torch.float32*)

This element can represent any custom transfer map.

defining_features() → list[str]

List of features that define the element. Used to compare elements for equality and to save them.

NOTE: When overriding this property, make sure to call the super method and extend the list it returns.

classmethod from_merging_elements(*elements: list[*Element*]*, *incoming_beam: Beam*) → *CustomTransferMap*

Combine the transfer maps of multiple successive elements into a single transfer map. This can be used to speed up tracking through a segment, if no changes are made to the elements in the segment or the energy of the beam being tracked through them.

Parameters

- **elements** – List of consecutive elements to combine.
- **incoming_beam** – Beam entering the first element in the segment. NOTE: That this is required because the separate original transfer maps have to be computed before being combined and some of them may depend on the energy of the beam.

property is_skippable: bool

Whether the element can be skipped during tracking. If *True*, the element's transfer map is combined with the transfer maps of surrounding skipable elements.

plot(*ax: Axes*, *s: float*) → None

Plot a representation of this element into a *matplotlib* Axes at position *s*.

Parameters

- **ax** – Axes to plot the representation into.
- **s** – Position of the object along s in meters.

split(*resolution: Tensor*) → list[*Element*]

Split the element into slices no longer than *resolution*. Some elements may not be splittable, in which case a list containing only the element itself is returned.

Parameters
resolution – Length of the longest allowed split in meters.

Returns
Ordered sequence of sliced elements.

training: bool

transfer_map(*energy: Tensor*) → Tensor

Generates the element's transfer map that describes how the beam and its particles are transformed when traveling through the element. The state vector consists of 6 values with a physical meaning: (in the trace space notation)

- x: Position in x direction
- xp: Angle in x direction
- y: Position in y direction
- yp: Angle in y direction
- s: Position in longitudinal direction, the zero value is set to the

reference position (usually the center of the pulse) - p: Relative energy deviation from the reference particle

$p = \frac{\triangle E}{p_0 C}$

As well as a seventh value used to add constants to some of the prior values if necessary. Through this seventh state, the addition of constants can be represented using a matrix multiplication.

Parameters
energy – Reference energy of the Beam. Read from the fed-in Cheetah Beam.

Returns
A 7x7 Matrix for further calculations.

class accelerator.**Dipole**(*length: Tensor | Parameter*, *angle: Tensor | Parameter | None = None*, *e1: Tensor | Parameter | None = None*, *e2: Tensor | Parameter | None = None*, *tilt: Tensor | Parameter | None = None*, *fringe_integral: Tensor | Parameter | None = None*, *fringe_integral_exit: Tensor | Parameter | None = None*, *gap: Tensor | Parameter | None = None*, *name: str | None = None*, *device=None*, *dtype=torch.float32*)

Dipole magnet (by default a sector bending magnet).

Parameters

- **length** – Length in meters.
- **angle** – Deflection angle in rad.
- **e1** – The angle of inclination of the entrance face [rad].
- **e2** – The angle of inclination of the exit face [rad].
- **tilt** – Tilt of the magnet in x-y plane [rad].
- **fringe_integral** – Fringe field integral (of the enterance face).
- **fringe_integral_exit** – (only set if different from *fint*) Fringe field integral of the exit face.
- **gap** – The magnet gap [m], NOTE in MAD and ELEGANT: HGAP = gap/2
- **name** – Unique identifier of the element.

property defining_features: list[str]

List of features that define the element. Used to compare elements for equality and to save them.

NOTE: When overriding this property, make sure to call the super method and extend the list it returns.

property hx: Tensor

property is_active

property is_skippable: bool

Whether the element can be skipped during tracking. If *True*, the element's transfer map is combined with the transfer maps of surrounding skipable elements.

plot(*ax: Axes*, *s: float*) → None

Plot a representation of this element into a *matplotlib* Axes at position *s*.

Parameters

- **ax** – Axes to plot the representation into.
- **s** – Position of the object along s in meters.

split(*resolution: Tensor*) → list[*Element*]

Split the element into slices no longer than *resolution*. Some elements may not be splittable, in which case a list containing only the element itself is returned.

Parameters
resolution – Length of the longest allowed split in meters.

Returns
Ordered sequence of sliced elements.

training: bool

transfer_map(*energy: Tensor*) → Tensor

Generates the element's transfer map that describes how the beam and its particles are transformed when traveling through the element. The state vector consists of 6 values with a physical meaning: (in the trace space notation)

- x: Position in x direction
- xp: Angle in x direction
- y: Position in y direction
- yp: Angle in y direction
- s: Position in longitudinal direction, the zero value is set to the

reference position (usually the center of the pulse) - p: Relative energy deviation from the reference particle

$p = \frac{\Delta E}{p_0 C}$

As well as a seventh value used to add constants to some of the prior values if necessary. Through this seventh state, the addition of constants can be represented using a matrix multiplication.

Parameters
energy – Reference energy of the Beam. Read from the fed-in Cheetah Beam.

Returns
A 7x7 Matrix for further calculations.

class accelerator.**Drift**(*length: Tensor | Parameter*, *name: str | None = None*, *device=None*, *dtype=torch.float32*)

Drift section in a particle accelerator.

Note: the transfer map now uses the linear approximation. Including the R_56 = L / (beta**2 * gamma **2)

Parameters

- **length** – Length in meters.
- **name** – Unique identifier of the element.

property defining_features: list[str]

List of features that define the element. Used to compare elements for equality and to save them.

NOTE: When overriding this property, make sure to call the super method and extend the list it returns.

property is_skippable: bool

Whether the element can be skipped during tracking. If *True*, the element's transfer map is combined with the transfer maps of surrounding skipable elements.

plot(*ax: Axes*, *s: float*) → None

Plot a representation of this element into a *matplotlib* Axes at position *s*.

Parameters

- **ax** – Axes to plot the representation into.
- **s** – Position of the object along s in meters.

split(*resolution: Tensor*) → list[*Element*]

Split the element into slices no longer than *resolution*. Some elements may not be splittable, in which case a list containing only the element itself is returned.

Parameters
resolution – Length of the longest allowed split in meters.

Returns
Ordered sequence of sliced elements.

training: bool

transfer_map(*energy: Tensor*) → Tensor

Generates the element's transfer map that describes how the beam and its particles are transformed when traveling through the element. The state vector consists of 6 values with a physical meaning: (in the trace space notation)

- x: Position in x direction
- xp: Angle in x direction
- y: Position in y direction
- yp: Angle in y direction
- s: Position in longitudinal direction, the zero value is set to the

reference position (usually the center of the pulse) - p: Relative energy deviation from the reference particle

$p = \frac{\Delta E}{p_0 C}$

As well as a seventh value used to add constants to some of the prior values if necessary. Through this seventh state, the addition of constants can be represented using a matrix multiplication.

Parameters
energy – Reference energy of the Beam. Read from the fed-in Cheetah Beam.

Returns
A 7x7 Matrix for further calculations.

class accelerator.**Element**(*name: str | None = None*)

Base class for elements of particle accelerators.

Parameters
name – Unique identifier of the element.

abstract property defining_features: list[str]

List of features that define the element. Used to compare elements for equality and to save them.

NOTE: When overriding this property, make sure to call the super method and extend the list it returns.

forward(*incoming: Beam*) → Beam

Forward function required by *torch.nn.Module*. Simply calls *track*.

abstract property is_skippable: bool

Whether the element can be skipped during tracking. If *True*, the element's transfer map is combined with the transfer maps of surrounding skipable elements.

abstract plot(*ax: Axes*, *s: float*) → None

Plot a representation of this element into a *matplotlib* Axes at position *s*.

Parameters

- **ax** – Axes to plot the representation into.
- **s** – Position of the object along s in meters.

abstract split(*resolution: Tensor*) → list[*Element*]

Split the element into slices no longer than *resolution*. Some elements may not be splittable, in which case a list containing only the element itself is returned.

Parameters
resolution – Length of the longest allowed split in meters.

Returns
Ordered sequence of sliced elements.

track(*incoming: Beam*) → Beam

Track particles through the element. The input can be a *ParameterBeam* or a *ParticleBeam*.

Parameters
incoming – Beam of particles entering the element.

Returns
Beam of particles exiting the element.

training: bool

transfer_map(*energy: Tensor*) → Tensor

Generates the element's transfer map that describes how the beam and its particles are transformed when traveling through the element. The state vector consists of 6 values with a physical meaning: (in the trace space notation)

- x: Position in x direction
- xp: Angle in x direction
- y: Position in y direction
- yp: Angle in y direction
- s: Position in longitudinal direction, the zero value is set to the

reference position (usually the center of the pulse) - p: Relative energy deviation from the reference particle

$p = \frac{\Delta E}{p_0 C}$

As well as a seventh value used to add constants to some of the prior values if necessary. Through this seventh state, the addition of constants can be represented using a matrix multiplication.

Parameters
energy – Reference energy of the Beam. Read from the fed-in Cheetah Beam.

Returns
A 7x7 Matrix for further calculations.

class accelerator.**HorizontalCorrector**(*length: Tensor | Parameter*, *angle: Tensor | Parameter | None = None*, *name: str | None = None*, *device=None*, *dtype=torch.float32*)

Horizontal corrector magnet in a particle accelerator. Note: This is modeled as a drift section with

a thin-kick in the horizontal plane.

Parameters

- **length** – Length in meters.
- **angle** – Particle deflection angle in the horizontal plane in rad.
- **name** – Unique identifier of the element.

property defining_features: list[str]

List of features that define the element. Used to compare elements for equality and to save them.

NOTE: When overriding this property, make sure to call the super method and extend the list it returns.

property is_active: bool

property is_skippable: bool

Whether the element can be skipped during tracking. If *True*, the element's transfer map is combined with the transfer maps of surrounding skipable elements.

plot(*ax: Axes*, *s: float*) → None

Plot a representation of this element into a *matplotlib* Axes at position *s*.

Parameters

- **ax** – Axes to plot the representation into.
- **s** – Position of the object along s in meters.

split(*resolution: Tensor*) → list[*Element*]

Split the element into slices no longer than *resolution*. Some elements may not be splittable, in which case a list containing only the element itself is returned.

Parameters
resolution – Length of the longest allowed split in meters.

Returns
Ordered sequence of sliced elements.

training: bool

transfer_map(*energy: Tensor*) → Tensor

Generates the element's transfer map that describes how the beam and its particles are transformed when traveling through the element. The state vector consists of 6 values with a physical meaning: (in the trace space notation)

- x: Position in x direction
- xp: Angle in x direction

- y: Position in y direction
- yp: Angle in y direction
- s: Position in longitudinal direction, the zero value is set to the

reference position (usually the center of the pulse) - p: Relative energy deviation from the reference particle

$p = \frac{\Delta E}{p_0 C}$

As well as a seventh value used to add constants to some of the prior values if necessary. Through this seventh state, the addition of constants can be represented using a matrix multiplication.

Parameters
energy – Reference energy of the Beam. Read from the fed-in Cheetah Beam.

Returns
A 7x7 Matrix for further calculations.

class accelerator.**Marker**(*name: str | None = None*)

General Marker / Monitor element

Parameters
name – Unique identifier of the element.

property **defining_features: list[str]**

List of features that define the element. Used to compare elements for equality and to save them.

NOTE: When overriding this property, make sure to call the super method and extend the list it returns.

property **is_skippable: bool**

Whether the element can be skipped during tracking. If *True*, the element's transfer map is combined with the transfer maps of surrounding skipable elements.

plot(*ax: Axes*, *s: float*) → None

Plot a representation of this element into a *matplotlib* Axes at position *s*.

Parameters

- **ax** – Axes to plot the representation into.
- **s** – Position of the object along s in meters.

split(*resolution: Tensor*) → list[*Element*]

Split the element into slices no longer than *resolution*. Some elements may not be splittable, in which case a list containing only the element itself is returned.

Parameters
resolution – Length of the longest allowed split in meters.

Returns
Ordered sequence of sliced elements.

track(*incoming*)

Track particles through the element. The input can be a *ParameterBeam* or a *ParticleBeam*.

Parameters
incoming – Beam of particles entering the element.

Returns
Beam of particles exiting the element.

training: bool

transfer_map(*energy*)

Generates the element's transfer map that describes how the beam and its particles are transformed when traveling through the element. The state vector consists of 6 values with a physical meaning: (in the trace space notation)

- x: Position in x direction
- xp: Angle in x direction
- y: Position in y direction
- yp: Angle in y direction
- s: Position in longitudinal direction, the zero value is set to the

reference position (usually the center of the pulse) - p: Relative energy deviation from the reference particle

$p = \frac{\triangle E}{p_0 C}$

As well as a seventh value used to add constants to some of the prior values if necessary. Through this seventh state, the addition of constants can be represented using a matrix multiplication.

Parameters
energy – Reference energy of the Beam. Read from the fed-in Cheetah Beam.

Returns
A 7x7 Matrix for further calculations.

class accelerator.**Quadrupole**(*length: Tensor | Parameter*, *k1: Tensor | Parameter | None = None*, *misalignment: Tensor | Parameter | None = None*, *tilt: Tensor | Parameter | None = None*, *name: str | None = None*, *device=None*, *dtype=torch.float32*)

Quadrupole magnet in a particle accelerator.

Parameters

- **length** – Length in meters.
- **k1** – Strength of the quadrupole in rad/m.
- **misalignment** – Misalignment vector of the quadrupole in x- and y-directions.
- **tilt** – Tilt angle of the quadrupole in x-y plane [rad]. pi/4 for skew-quadrupole.
- **name** – Unique identifier of the element.

property defining_features: list[str]

List of features that define the element. Used to compare elements for equality and to save them.

NOTE: When overriding this property, make sure to call the super method and extend the list it returns.

property is_active: bool

property is_skippable: bool

Whether the element can be skipped during tracking. If *True*, the element's transfer map is combined with the transfer maps of surrounding skipable elements.

plot(*ax: Axes*, *s: float*) → None

Plot a representation of this element into a *matplotlib* Axes at position *s*.

Parameters

- **ax** – Axes to plot the representation into.
- **s** – Position of the object along s in meters.

split(*resolution: Tensor*) → list[*Element*]

Split the element into slices no longer than *resolution*. Some elements may not be splittable, in which case a list containing only the element itself is returned.

Parameters
resolution – Length of the longest allowed split in meters.

Returns
Ordered sequence of sliced elements.

training: bool

transfer_map(*energy: Tensor*) → Tensor

Generates the element's transfer map that describes how the beam and its particles are transformed when traveling through the element. The state vector consists of 6 values with a physical meaning: (in the trace space notation)

- x: Position in x direction
- xp: Angle in x direction
- y: Position in y direction
- yp: Angle in y direction
- s: Position in longitudinal direction, the zero value is set to the

reference position (usually the center of the pulse) - p: Relative energy deviation from the reference particle

$p = \frac{\Delta E}{p_0 C}$

As well as a seventh value used to add constants to some of the prior values if necessary. Through this seventh state, the addition of constants can be represented using a matrix multiplication.

Parameters
energy – Reference energy of the Beam. Read from the fed-in Cheetah Beam.

Returns
A 7x7 Matrix for further calculations.

class accelerator.**RBend**(*length: Tensor | Parameter | None*, *angle: Tensor | Parameter | None = None*, *e1: Tensor | Parameter | None = None*, *e2: Tensor | Parameter | None = None*, *tilt: Tensor | Parameter | None = None*, *fringe_integral: Tensor | Parameter | None = None*, *fringe_integral_exit: Tensor | Parameter | None = None*, *gap: Tensor | Parameter | None = None*, *name: str | None = None*, *device=None*, *dtype=torch.float32*)

Rectangular bending magnet.

Parameters

- **length** – Length in meters.
- **angle** – Deflection angle in rad.
- **e1** – The angle of inclination of the entrance face [rad].
- **e2** – The angle of inclination of the exit face [rad].
- **tilt** – Tilt of the magnet in x-y plane [rad].
- **fringe_integral** – Fringe field integral (of the enterance face).
- **fringe_integral_exit** – (only set if different from *fint*) Fringe field integral of the exit face.
- **gap** – The magnet gap [m], NOTE in MAD and ELEGANT: HGAP = gap/2

- **name** – Unique identifier of the element.

training: bool

class accelerator.**Screen**(*resolution: Tensor | Parameter | None = None*, *pixel_size: Tensor | Parameter | None = None*, *binning: Tensor | Parameter | None = None*, *misalignment: Tensor | Parameter | None = None*, *is_active: bool = False*, *name: str | None = None*, *device=None*, *dtype=torch.float32*)

Diagnostic screen in a particle accelerator.

Parameters

- **resolution** – Resolution of the camera sensor looking at the screen given as Tensor *(width, height)*.
- **pixel_size** – Size of a pixel on the screen in meters given as a Tensor *(width, height)*.
- **binning** – Binning used by the camera.
- **misalignment** – Misalignment of the screen in meters given as a Tensor *(x, y)*.
- **is_active** – If *True* the screen is active and will record the beam's distribution. If *False* the screen is inactive and will not record the beam's distribution.
- **name** – Unique identifier of the element.

property defining_features: list[str]

List of features that define the element. Used to compare elements for equality and to save them.

NOTE: When overriding this property, make sure to call the super method and extend the list it returns.

property effective_pixel_size: Tensor

property effective_resolution: Tensor

property extent: Tensor

get_read_beam() → Beam

property is_skippable: bool

Whether the element can be skipped during tracking. If *True*, the element's transfer map is combined with the transfer maps of surrounding skipable elements.

property pixel_bin_edges: tuple[Tensor, Tensor]

plot(*ax: Axes*, *s: float*) → None

Plot a representation of this element into a *matplotlib* Axes at position *s*.

Parameters

- **ax** – Axes to plot the representation into.
- **s** – Position of the object along s in meters.

property reading: Tensor

set_read_beam(*value: Beam*) → None

split(*resolution: Tensor*) → list[*Element*]

Split the element into slices no longer than *resolution*. Some elements may not be splittable, in which case a list containing only the element itself is returned.

Parameters
resolution – Length of the longest allowed split in meters.

Returns
Ordered sequence of sliced elements.

track(*incoming: Beam*) → Beam

Track particles through the element. The input can be a *ParameterBeam* or a *ParticleBeam*.

Parameters
incoming – Beam of particles entering the element.

Returns
Beam of particles exiting the element.

training: bool

transfer_map(*energy: Tensor*) → Tensor

Generates the element's transfer map that describes how the beam and its particles are transformed when traveling through the element. The state vector consists of 6 values with a physical meaning: (in the trace space notation)

- x: Position in x direction
- xp: Angle in x direction
- y: Position in y direction
- yp: Angle in y direction
- s: Position in longitudinal direction, the zero value is set to the

reference position (usually the center of the pulse) - p: Relative energy deviation from the reference particle

$p = \frac{\Delta E}{p_0 C}$

As well as a seventh value used to add constants to some of the prior values if necessary. Through this seventh state, the addition of constants can be represented using a matrix multiplication.

Parameters
energy – Reference energy of the Beam. Read from the fed-in Cheetah Beam.

Returns
A 7x7 Matrix for further calculations.

class accelerator.**Segment**(*elements: list[*Element*]*, *name: str = 'unnamed'*)

Segment of a particle accelerator consisting of several elements.

Parameters

- **cell** – List of Cheetah elements that describe an accelerator (section).
- **name** – Unique identifier of the element.

property defining_features: list[str]

List of features that define the element. Used to compare elements for equality and to save them.

NOTE: When overriding this property, make sure to call the super method and extend the list it returns.

flattened() → *Segment*

Return a flattened version of the segment, i.e. one where all subsegments are resolved and their elements entered into a top-level segment.

classmethod from_bmad(*bmad_lattice_file_path: str*, *environment_variables: dict | None = None*) → *Segment*

Read a Cheetah segment from a Bmad lattice file.

NOTE: This function was designed at the example of the LCLS lattice. While this lattice is extensive, this function might not properly convert all features of a Bmad lattice. If you find that this function does not work for your lattice, please open an issue on GitHub.

Parameters

- **bmad_lattice_file_path** – Path to the Bmad lattice file.
- **environment_variables** – Dictionary of environment variables to use when parsing the lattice file.

Returns

Cheetah *Segment* representing the Bmad lattice.

classmethod from_lattice_json(*filepath: str*) → *Segment*

Load a Cheetah model from a JSON file.

Parameters

filename – Name/path of the file to load the lattice from.

Returns

Loaded Cheetah *Segment*.

classmethod from_nx_tables(*filepath: Path | str*) → *Element*

Read an NX Tables CSV-like file generated for the ARES lattice into a Cheetah *Segment*.

NOTE: This format is specific to the ARES accelerator at DESY.

Parameters

filepath – Path to the NX Tables file.

Returns

Converted Cheetah *Segment*.

classmethod from_ocelot(*cell*, *name: str | None = None*, *warnings: bool = True*, *device=None*, *dtype=torch.float32*, ***kwargs*) → *Segment*

Translate an Ocelot cell to a Cheetah *Segment*.

NOTE Objects not supported by Cheetah are translated to drift sections. Screen objects are created only from *ocelot.Monitor* objects when the string "BSC" is contained in their *id* attribute. Their screen properties are always set to default values and most likely need adjusting afterwards. BPM objects are only created from *ocelot.Monitor* objects when their id has a substring "BPM".

Parameters

- **cell** – Ocelot cell, i.e. a list of Ocelot elements to be converted.
- **name** – Unique identifier for the entire segment.
- **warnings** – Whether to print warnings when objects are not supported by Cheetah or converted with potentially unexpected behavior.

Returns

Cheetah segment closely resembling the Ocelot cell.

inactive_elements_as_drifts(*except_for: list[str] | None = None*) → *Segment*

Return a segment where all inactive elements (that have a length) are replaced by drifts. This can be used to speed up tracking through the segment and is a valid thing to as inactive elements should basically be no different from drift sections.

Parameters
except_for – List of names of elements that should not be replaced by drifts despite being inactive. Usually these are the elements that are currently inactive but will be activated later.

Returns
Segment with inactive elements replaced by drifts.

property is_skippable: bool

Whether the element can be skipped during tracking. If *True*, the element's transfer map is combined with the transfer maps of surrounding skipable elements.

property length: Tensor

plot(*ax: Axes*, *s: float*) → None

Plot a representation of this element into a *matplotlib* Axes at position *s*.

Parameters

- **ax** – Axes to plot the representation into.
- **s** – Position of the object along s in meters.

plot_overview(*fig: Figure | None = None*, *beam: Beam | None = None*, *n: int = 10*, *resolution: float = 0.01*) → None

Plot an overview of the segment with the lattice and traced reference particles.

Parameters

- **fig** – Figure to plot the overview into.
- **beam** – Entering beam from which the reference particles are sampled.
- **n** – Number of reference particles to plot. Must not be larger than number of particles passed in *beam*.
- **resolution** – Minimum resolution of the tracking of the reference particles in the plot.

plot_reference_particle_traces(*axx: Axes*, *axy: Axes*, *beam: Beam | None = None*, *num_particles: int = 10*, *resolution: float = 0.01*) → None

Plot *n* reference particles along the segment view in x- and y-direction.

Parameters

- **axx** – Axes to plot the particle traces into viewed in x-direction.
- **axy** – Axes to plot the particle traces into viewed in y-direction.
- **beam** – Entering beam from which the reference particles are sampled.
- **num_particles** – Number of reference particles to plot. Must not be larger than number of particles passed in *beam*.
- **resolution** – Minimum resolution of the tracking of the reference particles in the plot.

plot_twiss(*beam: Beam*, *ax: Any | None = None*) → None

Plot twiss parameters along the segment.

plot_twiss_over_lattice(*beam: Beam*, *figsize=(8, 4)*) → None

Plot twiss parameters in a plot over a plot of the lattice.

split(*resolution: Tensor*) → list[*Element*]

Split the element into slices no longer than *resolution*. Some elements may not be splittable, in which case a list containing only the element itself is returned.

Parameters
resolution – Length of the longest allowed split in meters.

Returns
Ordered sequence of sliced elements.

subcell(*start: str*, *end: str*) → *Segment*

Extract a subcell *[start, end]* from an this segment.

to_lattice_json(*filepath: str*, *title: str | None = None*, *info: str = 'This is a placeholder lattice description'*) → None

Save a Cheetah model to a JSON file.

Parameters

- **filename** – Name/path of the file to save the lattice to.
- **title** – Title of the lattice. If not provided, defaults to the name of the *Segment* object. If that also does not have a name, defaults to "Unnamed Lattice".
- **info** – Information about the lattice. Defaults to "This is a placeholder lattice description".

track(*incoming: Beam*) → Beam

Track particles through the element. The input can be a *ParameterBeam* or a *ParticleBeam*.

Parameters
incoming – Beam of particles entering the element.

Returns
Beam of particles exiting the element.

training: bool

transfer_map(*energy: Tensor*) → Tensor

Generates the element's transfer map that describes how the beam and its particles are transformed when traveling through the element. The state vector consists of 6 values with a physical meaning: (in the trace space notation)

- x: Position in x direction
- xp: Angle in x direction
- y: Position in y direction
- yp: Angle in y direction
- s: Position in longitudinal direction, the zero value is set to the

reference position (usually the center of the pulse) - p: Relative energy deviation from the reference particle

$p = \frac{\Delta E}{p_0 C}$

As well as a seventh value used to add constants to some of the prior values if necessary. Through this seventh state, the addition of constants can be represented using a matrix multiplication.

Parameters
energy – Reference energy of the Beam. Read from the fed-in Cheetah Beam.

Returns
A 7x7 Matrix for further calculations.

transfer_maps_merged(*incoming_beam: Beam*, *except_for: list[str] | None = None*) → *Segment*

Return a segment where the transfer maps of skipable elements are merged into elements of type *CustomTransferMap*. This can be used to speed up tracking through the segment.

Parameters

- **incoming_beam** – Beam that is incoming to the segment. NOTE: This beam is needed to determine the energy of the beam when entering each element, as the transfer maps of merged elements might depend on the beam energy.
- **except_for** – List of names of elements that should not be merged despite being skippable. Usually these are the elements that are changed from one tracking to another.

Returns
Segment with merged transfer maps.

without_inactive_markers(*except_for: list[str] | None = None*) → *Segment*

Return a segment where all inactive markers are removed. This can be used to speed up tracking through the segment.

NOTE: *is_active* has not yet been implemented for Markers. Therefore, this function currently removes all markers.

Parameters
except_for – List of names of elements that should not be removed despite being inactive.

Returns
Segment without inactive markers.

without_inactive_zero_length_elements(*except_for: list[str] | None = None*) → *Segment*

Return a segment where all inactive zero length elements are removed. This can be used to speed up tracking through the segment.

NOTE: If *is_active* is not implemented for an element, it is assumed to be inactive and will be removed.

Parameters
except_for – List of names of elements that should not be removed despite being inactive and having a zero length.

Returns
Segment without inactive zero length elements.

class accelerator.**Solenoid**(*length: Tensor | Parameter | None = None*, *k: Tensor | Parameter | None = None*, *misalignment: Tensor | Parameter | None = None*, *name: str | None = None*, *device=None*, *dtype=torch.float32*)

Solenoid magnet.

Implemented according to A.W.Chao P74

Parameters

- **length** – Length in meters.
- **k** – Normalised strength of the solenoid magnet B0/(2*Brho). B0 is the field inside the solenoid, Brho is the momentum of central trajectory.
- **misalignment** – Misalignment vector of the solenoid magnet in x- and y-directions.
- **name** – Unique identifier of the element.

property defining_features: list[str]

List of features that define the element. Used to compare elements for equality and to save them.

NOTE: When overriding this property, make sure to call the super method and extend the list it returns.

property is_active: bool

is_skippable() → bool

Whether the element can be skipped during tracking. If *True*, the element's transfer map is combined with the transfer maps of surrounding skipable elements.

plot(*ax: Axes*, *s: float*) → None

Plot a representation of this element into a *matplotlib* Axes at position *s*.

Parameters

- **ax** – Axes to plot the representation into.
- **s** – Position of the object along s in meters.

split(*resolution: Tensor*) → list[*Element*]

Split the element into slices no longer than *resolution*. Some elements may not be splittable, in which case a list containing only the element itself is returned.

Parameters
resolution – Length of the longest allowed split in meters.

Returns
Ordered sequence of sliced elements.

training: bool

transfer_map(*energy: Tensor*) → Tensor

Generates the element's transfer map that describes how the beam and its particles are transformed when traveling through the element. The state vector consists of 6 values with a physical meaning: (in the trace space notation)

- x: Position in x direction
- xp: Angle in x direction
- y: Position in y direction
- yp: Angle in y direction
- s: Position in longitudinal direction, the zero value is set to the

reference position (usually the center of the pulse) - p: Relative energy deviation from the reference particle

$p = \frac{\Delta E}{p_0 C}$

As well as a seventh value used to add constants to some of the prior values if necessary. Through this seventh state, the addition of constants can be represented using a matrix multiplication.

Parameters
energy – Reference energy of the Beam. Read from the fed-in Cheetah Beam.

Returns
A 7x7 Matrix for further calculations.

class accelerator.**Undulator**(*length: Tensor | Parameter*, *is_active: bool = False*, *name: str | None = None*, *device=None*, *dtype=torch.float32*)

Element representing an undulator in a particle accelerator.

NOTE Currently behaves like a drift section but is plotted distinctively.

Parameters

- **length** – Length in meters.
- **is_active** – Indicates if the undulator is active or not. Currently has no effect.

- **name** – Unique identifier of the element.

property defining_features: list[str]

List of features that define the element. Used to compare elements for equality and to save them.

NOTE: When overriding this property, make sure to call the super method and extend the list it returns.

property is_skippable: bool

Whether the element can be skipped during tracking. If *True*, the element's transfer map is combined with the transfer maps of surrounding skipable elements.

plot(*ax: Axes*, *s: float*) → None

Plot a representation of this element into a *matplotlib* Axes at position *s*.

Parameters

- **ax** – Axes to plot the representation into.
- **s** – Position of the object along s in meters.

split(*resolution: Tensor*) → list[*Element*]

Split the element into slices no longer than *resolution*. Some elements may not be splittable, in which case a list containing only the element itself is returned.

Parameters
resolution – Length of the longest allowed split in meters.

Returns
Ordered sequence of sliced elements.

training: bool

transfer_map(*energy: Tensor*) → Tensor

Generates the element's transfer map that describes how the beam and its particles are transformed when traveling through the element. The state vector consists of 6 values with a physical meaning: (in the trace space notation)

- x: Position in x direction
- xp: Angle in x direction
- y: Position in y direction
- yp: Angle in y direction
- s: Position in longitudinal direction, the zero value is set to the

reference position (usually the center of the pulse) - p: Relative energy deviation from the reference particle

$p = \frac{\Delta E}{p_0 C}$

As well as a seventh value used to add constants to some of the prior values if necessary. Through this seventh state, the addition of constants can be represented using a matrix multiplication.

Parameters
energy – Reference energy of the Beam. Read from the fed-in Cheetah Beam.

Returns
A 7x7 Matrix for further calculations.

class accelerator.**VerticalCorrector**(*length: Tensor | Parameter*, *angle: Tensor | Parameter | None = None*, *name: str | None = None*, *device=None*, *dtype=torch.float32*)

Verticle corrector magnet in a particle accelerator. Note: This is modeled as a drift section with

a thin-kick in the vertical plane.

Parameters

- **length** – Length in meters.
- **angle** – Particle deflection angle in the vertical plane in rad.
- **name** – Unique identifier of the element.

property defining_features: list[str]

List of features that define the element. Used to compare elements for equality and to save them.

NOTE: When overriding this property, make sure to call the super method and extend the list it returns.

property is_active: bool

property is_skippable: bool

Whether the element can be skipped during tracking. If *True*, the element's transfer map is combined with the transfer maps of surrounding skipable elements.

plot(*ax: Axes*, *s: float*) → None

Plot a representation of this element into a *matplotlib* Axes at position *s*.

Parameters

- **ax** – Axes to plot the representation into.
- **s** – Position of the object along s in meters.

split(*resolution: Tensor*) → list[*Element*]

Split the element into slices no longer than *resolution*. Some elements may not be splittable, in which case a list containing only the element itself is returned.

Parameters
resolution – Length of the longest allowed split in meters.

Returns
Ordered sequence of sliced elements.

training: bool

transfer_map(*energy: Tensor*) → Tensor

Generates the element's transfer map that describes how the beam and its particles are transformed when traveling through the element. The state vector consists of 6 values with a physical meaning: (in the trace space notation)

- x: Position in x direction
- xp: Angle in x direction
- y: Position in y direction
- yp: Angle in y direction
- s: Position in longitudinal direction, the zero value is set to the

reference position (usually the center of the pulse) - p: Relative energy deviation from the reference particle

$p = \frac{\triangle E}{p_0 C}$

As well as a seventh value used to add constants to some of the prior values if necessary. Through this seventh state, the addition of constants can be represented using a matrix multiplication.

Parameters
energy – Reference energy of the Beam. Read from the fed-in Cheetah Beam.

Returns
A 7x7 Matrix for further calculations.

3.2 Astralavista

converters.astralavista.**from_astrabeam**(*path: str*) → tuple[ndarray, float, ndarray]

Read from a ASTRA beam distribution, and prepare for conversion to a Cheetah ParticleBeam or ParameterBeam.

Adapted from the implementation in Ocelot: https://github.com/ocelot-collab/ocelot/blob/master/ocelot/adaptors/astra2ocelot.py

Parameters
path – Path to the ASTRA beam distribution file.

Returns
(particles, energy, q_array) Particle 6D phase space information, mean energy, and the charge array of the particle beam.

3.3 DontBmad

converters.dontbmad.**assign_property**(*line: str*, *context: dict*) → dict

Assign a property of an element to the context.

Parameters
- **line** – Line of a property assignment to be parsed.
- **context** – Dictionary of variables to assign the property to and from which to read variables.

Returns
Updated context.

converters.dontbmad.**assign_variable**(*line: str*, *context: dict*) → dict

Assign a variable to the context.

Parameters
- **line** – Line of a variable assignment to be parsed.
- **context** – Dictionary of variables to assign the variable to and from which to read variables.

Returns
Updated context.

converters.dontbmad.**convert_bmad_lattice**(*bmad_lattice_file_path: Path*, *environment_variables: dict | None = None*) → Element

Convert a Bmad lattice file to a Cheetah *Segment*.

NOTE: This function was designed at the example of the LCLS lattice. While this
lattice is extensive, this function might not properly convert all features of a Bmad lattice. If you find that this function does not work for your lattice, please open an issue on GitHub.

Parameters

- **bmad_lattice_file_path** – Path to the Bmad lattice file.
- **environment_variables** – Dictionary of environment variables to use when parsing the lattice file.

Returns
Cheetah *Segment* representing the Bmad lattice.

converters.dontbmad.**convert_element**(*name: str*, *context: dict*) → Element

Convert a parsed Bmad element dict to a cheetah Element.

Parameters

- **name** – Name of the (top-level) element to convert.
- **context** – Context dictionary parsed from Bmad lattice file(s).

Returns
Converted cheetah Element. If you are calling this function yourself as a user of Cheetah, this is most likely a *Segment*.

converters.dontbmad.**define_element**(*line: str*, *context: dict*) → dict

Define an element in the context.

Parameters

- **line** – Line of an element definition to be parsed.
- **context** – Dictionary of variables to define the element in and from which to read variables.

Returns
Updated context.

converters.dontbmad.**define_line**(*line: str*, *context: dict*) → dict

Define a beam line in the context.

Parameters

- **line** – Line of a beam line definition to be parsed.
- **context** – Dictionary of variables to define the beam line in and from which to read variables.

Returns
Updated context.

converters.dontbmad.**define_overlay**(*line: str*, *context: dict*) → dict

Define an overlay in the context.

Parameters

- **line** – Line of an overlay definition to be parsed.
- **context** – Dictionary of variables to define the overlay in and from which to read variables.

Returns
Updated context.

converters.dontbmad.**evaluate_expression**(*expression: str*, *context: dict*) → Any

Evaluate an expression in the context of a dictionary of variables.

Parameters

- **expression** – Expression to evaluate.

- **context** – Dictionary of variables to evaluate the expression in the context of.

Returns
Result of evaluating the expression.

converters.dontbmad.**merge_delimiter_continued_lines**(*lines: list[str]*, *delimiter: str*, *remove_delimiter: bool = False*) → list[str]

Merge lines ending with some character as a delimitter with the following line.

Parameters

- **lines** – List of lines to merge.
- **delimitter** – Character to use as a delimitter.
- **remove_delimitter** – Whether to remove the delimitter from the merged line.

Returns
List of lines with ampersand-continued lines merged.

converters.dontbmad.**parse_lines**(*lines: str*) → dict

Parse a list of lines from a Bmad lattice file. They should be cleaned and merged before being passed to this function.

Parameters
lines – List of lines to parse.

Returns
Dictionary of variables defined in the lattice file.

converters.dontbmad.**parse_use_line**(*line: str*, *context: dict*) → dict

Parse a use line.

Parameters

- **line** – Line of a use statement to be parsed.
- **context** – Dictionary of variables to define the overlay in and from which to read variables.

Returns
Updated context.

converters.dontbmad.**read_clean_lines**(*lattice_file_path: Path*) → list[str]

Recursevely read lines from Bmad lattice files, removing comments and empty lines, and replacing lines calling external files with the lines of the external file.

Parameters
lattice_file_path – Path to the root Bmad lattice file.

Returns
List of lines from the root Bmad lattice file and all external files.

converters.dontbmad.**resolve_object_name_wildcard**(*wildcard_pattern: str*, *context: dict*) → list

Return a list of object names that match the given wildcard pattern.

Parameters

- **wildcard_pattern** – Wildcard pattern to match.
- **context** – Dictionary of variables among which to search for matching object.

Returns
List of object names that match the given wildcard pattern, both in terms of name and element type.

converters.dontbmad.**validate_understood_properties**(*understood: list[str]*, *properties: dict*) → None

Validate that all properties are understood. This function primarily ensures that properties not understood by Cheetah are not ignored silently.

Raises an *AssertionError* if a property is found that is not understood.

Parameters

- **understood** – List of properties understood (or purpusefully ignored) by Cheetah.
- **properties** – Dictionary of properties to validate.

Returns

None

3.4 Error

3.5 LatticeJSON

class latticejson.**CompactJSONEncoder**(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

A JSON Encoder which only indents the first two levels.

Taken from https://github.com/nobeam/latticejson/blob/main/latticejson/format.py

encode(*obj*, *level=0*)

Return a JSON string representation of a Python data structure.

```
>>> from json.encoder import JSONEncoder
>>> JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

latticejson.**convert_element**(*element: Element*)

Deconstruct an element into its name, class and parameters for saving to JSON.

Parameters

element – Cheetah element

Returns

Tuple of element name, element class, and element parameters

latticejson.**convert_segment**(*segment: Segment*) → Tuple[dict, dict]

Deconstruct a segment into its name, a list of its elements and a dictionary of its element parameters for saving to JSON.

Parameters

segment – Cheetah segment.

Returns

Tuple of elments and lattices dictionaries found in segment, including the segment itself.

latticejson.**feature2nontorch**(*value: Any*) → Any

if necesary, convert an the value of a feature of a *cheetah.Element* to a non-torch type that can be saved to LatticeJSON.

Parameters
value – Value of the feature that might be in some kind of PyTorch format, such as *torch.Tensor* or *torch.nn.Parameter*.

Returns
Value of the feature if it is not in a PyTorch format, otherwise the value converted to a non-PyTorch format.

latticejson.**load_cheetah_model**(*filename: str*) → Segment

Load a Cheetah model from a JSON file.

Parameters
filename – Name/path of the file to load the lattice from.

Returns
Loaded Cheetah *Segment*.

latticejson.**nontorch2feature**(*value: Any*) → Any

Convert a value like a *float*, *int*, etc. to a *torch.Tensor* if necessary. Values of type *str* and *bool* are not converted, because all currently existing *cheetah.Element* subclasses expect these values to not be of type *torch.Tensor*.

Parameters
value – Value to convert to a *torch.Tensor* if necessary.

Returns
Value converted to a *torch.Tensor* if necessary.

latticejson.**parse_element**(*name: str*, *lattice_dict: dict*) → Element

Parse an *Element* named *name* from a *lattice_dict*.

Parameters

- **name** – Name of the *Element* to parse.
- **lattice_dict** – Dictionary containing the lattice information.

latticejson.**parse_segment**(*name: str*, *lattice_dict: dict*) → Segment

Parse a *Segment* named *name* from a *lattice_dict*.

Parameters

- **name** – Name of the *Segment* to parse.
- **lattice_dict** – Dictionary containing the lattice information.

latticejson.**save_cheetah_model**(*segment: Segment*, *filename: str*, *title: str | None = None*, *info: str = 'This is a placeholder lattice description'*) → None

Save a cheetah model to json file accoding to the lattice-json convention c.f. https://github.com/nobeam/latticejson

Parameters

- **segment** – Cheetah *Segment* to save.
- **filename** – Name/path of the file to save the lattice to.
- **title** – Title of the lattice. If not provided, defaults to the name of the *Segment* object. If that also does not have a name, defaults to "Unnamed Lattice".
- **info** – Information about the lattice. Defaults to "This is a placeholder lattice description".

3.6 NOcelot

converters.nocelot.**ocelot2cheetah**(*element*, *warnings: bool = True*, *device=None*, *dtype=torch.float32*) → Element

Translate an Ocelot element to a Cheetah element.

NOTE Object not supported by Cheetah are translated to drift sections. Screen objects are created only from *ocelot.Monitor* objects when the string "BSC" is contained in their *id* attribute. Their screen properties are always set to default values and most likely need adjusting afterwards. BPM objects are only created from *ocelot.Monitor* objects when their id has a substring "BPM".

Parameters

- **element** – Ocelot element object representing an element of particle accelerator.
- **warnings** – Whether to print warnings when elements might not be converted as expected.

Returns

Cheetah element object representing an element of particle accelerator.

converters.nocelot.**subcell_of_ocelot**(*cell: list*, *start: str*, *end: str*) → list

Extract a subcell *[start, end]* from an Ocelot cell.

3.7 Particles

class particles.**Beam**(**args*, ***kwargs*)

property alpha_x: Tensor

Alpha function in x direction in rad.

property alpha_y: Tensor

Alpha function in y direction in rad.

property beta_x: Tensor

Beta function in x direction in meters.

property beta_y: Tensor

Beta function in y direction in meters.

property emittance_x: Tensor

Emittance of the beam in x direction in m*rad.

property emittance_y: Tensor

Emittance of the beam in y direction in m*rad.

empty = "I'm an empty beam!"

classmethod from_astra(*path: str*, ***kwargs*) → *Beam*

Load an Astra particle distribution as a Cheetah Beam.

classmethod from_ocelot(*parray*) → *Beam*

Convert an Ocelot ParticleArray *parray* to a Cheetah Beam.

classmethod from_parameters(*mu_x: Tensor | None = None*, *mu_xp: Tensor | None = None*, *mu_y: Tensor | None = None*, *mu_yp: Tensor | None = None*, *sigma_x: Tensor | None = None*, *sigma_xp: Tensor | None = None*, *sigma_y: Tensor | None = None*, *sigma_yp: Tensor | None = None*, *sigma_s: Tensor | None = None*, *sigma_p: Tensor | None = None*, *cor_x: Tensor | None = None*, *cor_y: Tensor | None = None*, *cor_s: Tensor | None = None*, *energy: Tensor | None = None*, *total_charge: Tensor | None = None*) → *Beam*

Create beam that with given beam parameters.

Parameters

- **n** – Number of particles to generate.
- **mu_x** – Center of the particle distribution on x in meters.
- **mu_xp** – Center of the particle distribution on x'=px/px' (trace space) in rad.
- **mu_y** – Center of the particle distribution on y in meters.
- **mu_yp** – Center of the particle distribution on y' in rad.
- **sigma_x** – Sigma of the particle distribution in x direction in meters.
- **sigma_xp** – Sigma of the particle distribution in x' direction in rad.
- **sigma_y** – Sigma of the particle distribution in y direction in meters.
- **sigma_yp** – Sigma of the particle distribution in y' direction in rad.
- **sigma_s** – Sigma of the particle distribution in s direction in meters.
- **sigma_p** – Sigma of the particle distribution in p direction in meters.
- **energy** – Energy of the beam in eV.
- **total_charge** – Total charge of the beam in C.

classmethod from_twiss(*beta_x: Tensor | None = None*, *alpha_x: Tensor | None = None*, *emittance_x: Tensor | None = None*, *beta_y: Tensor | None = None*, *alpha_y: Tensor | None = None*, *emittance_y: Tensor | None = None*, *energy: Tensor | None = None*, *total_charge: Tensor | None = None*) → *Beam*

Create a beam from twiss parameters.

Parameters

- **beta_x** – Beta function in x direction in meters.
- **alpha_x** – Alpha function in x direction in rad.
- **emittance_x** – Emittance in x direction in m*rad.
- **beta_y** – Beta function in y direction in meters.
- **alpha_y** – Alpha function in y direction in rad.
- **emittance_y** – Emittance in y direction in m*rad.
- **energy** – Energy of the beam in eV.
- **total_charge** – Total charge of the beam in C.

property mu_p: Tensor

property mu_s: Tensor

property mu_x: Tensor

property mu_xp: Tensor

property mu_y: Tensor

property mu_yp: Tensor

property normalized_emittance_x: Tensor

Normalized emittance of the beam in x direction in m*rad.

property normalized_emittance_y: Tensor

Normalized emittance of the beam in y direction in m*rad.

property parameters: dict

Return an iterator over module parameters.

This is typically passed to an optimizer.

Args:

recurse (bool): if True, then yields parameters of this module
and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields:
Parameter: module parameter

Example:

```
>>> # xdoctest: +SKIP("undefined vars")
>>> for param in model.parameters():
>>>     print(type(param), param.size())
<class 'torch.Tensor'> (20L,)
<class 'torch.Tensor'> (20L, 1L, 5L, 5L)
```

property relativistic_beta: Tensor

property relativistic_gamma: Tensor

property sigma_p: Tensor

property sigma_s: Tensor

property sigma_x: Tensor

property sigma_xp: Tensor

property sigma_xxp: Tensor

property sigma_y: Tensor

property sigma_yp: Tensor

property sigma_yyp: Tensor

training: bool

transformed_to(*mu_x: Tensor | None = None*, *mu_xp: Tensor | None = None*, *mu_y: Tensor | None = None*, *mu_yp: Tensor | None = None*, *sigma_x: Tensor | None = None*, *sigma_xp: Tensor | None = None*, *sigma_y: Tensor | None = None*, *sigma_yp: Tensor | None = None*, *sigma_s: Tensor | None = None*, *sigma_p: Tensor | None = None*, *energy: Tensor | None = None*, *total_charge: Tensor | None = None*) → *Beam*

Create version of this beam that is transformed to new beam parameters.

Parameters

- **mu_x** – Center of the particle distribution on x in meters.
- **mu_xp** – Center of the particle distribution on x' in rad.
- **mu_y** – Center of the particle distribution on y in meters.
- **mu_yp** – Center of the particle distribution on y' in rad.
- **sigma_x** – Sigma of the particle distribution in x direction in meters.
- **sigma_xp** – Sigma of the particle distribution in x' direction in rad.
- **sigma_y** – Sigma of the particle distribution in y direction in meters.
- **sigma_yp** – Sigma of the particle distribution in y' direction in rad.
- **sigma_s** – Sigma of the particle distribution in s direction in meters.
- **sigma_p** – Sigma of the particle distribution in p direction,

dimensionless. :param energy: Energy of the beam in eV. :param total_charge: Total charge of the beam in C.

class particles.**ParameterBeam**(*mu: Tensor*, *cov: Tensor*, *energy: Tensor*, *total_charge: Tensor | None = None*, *device=None*, *dtype=torch.float32*)

Beam of charged particles, where each particle is simulated.

Parameters

- **mu** – Mu vector of the beam.
- **cov** – Covariance matrix of the beam.
- **energy** – Energy of the beam in eV.
- **total_charge** – Total charge of the beam in C.
- **device** – Device to use for the beam. If "auto", use CUDA if available. Note: Compuationally it would be faster to use CPU for ParameterBeam.

classmethod from_astra(*path: str*, *device=None*, *dtype=torch.float32*) → *ParameterBeam*

Load an Astra particle distribution as a Cheetah Beam.

classmethod from_ocelot(*parray*, *device=None*, *dtype=torch.float32*) → *ParameterBeam*

Load an Ocelot ParticleArray *parray* as a Cheetah Beam.

classmethod from_parameters(*mu_x: Tensor | None = None*, *mu_xp: Tensor | None = None*, *mu_y: Tensor | None = None*, *mu_yp: Tensor | None = None*, *sigma_x: Tensor | None = None*, *sigma_xp: Tensor | None = None*, *sigma_y: Tensor | None = None*, *sigma_yp: Tensor | None = None*, *sigma_s: Tensor | None = None*, *sigma_p: Tensor | None = None*, *cor_x: Tensor | None = None*, *cor_y: Tensor | None = None*, *cor_s: Tensor | None = None*, *energy: Tensor | None = None*, *total_charge: Tensor | None = None*, *device=None*, *dtype=torch.float32*) → *ParameterBeam*

Create beam that with given beam parameters.

Parameters

- **n** – Number of particles to generate.
- **mu_x** – Center of the particle distribution on x in meters.
- **mu_xp** – Center of the particle distribution on x'=px/px' (trace space) in rad.
- **mu_y** – Center of the particle distribution on y in meters.
- **mu_yp** – Center of the particle distribution on y' in rad.
- **sigma_x** – Sigma of the particle distribution in x direction in meters.
- **sigma_xp** – Sigma of the particle distribution in x' direction in rad.
- **sigma_y** – Sigma of the particle distribution in y direction in meters.
- **sigma_yp** – Sigma of the particle distribution in y' direction in rad.
- **sigma_s** – Sigma of the particle distribution in s direction in meters.
- **sigma_p** – Sigma of the particle distribution in p direction in meters.
- **energy** – Energy of the beam in eV.
- **total_charge** – Total charge of the beam in C.

classmethod from_twiss(*beta_x: Tensor | None = None*, *alpha_x: Tensor | None = None*, *emittance_x: Tensor | None = None*, *beta_y: Tensor | None = None*, *alpha_y: Tensor | None = None*, *emittance_y: Tensor | None = None*, *sigma_s: Tensor | None = None*, *sigma_p: Tensor | None = None*, *cor_s: Tensor | None = None*, *energy: Tensor | None = None*, *total_charge: Tensor | None = None*, *device=None*, *dtype=torch.float32*) → *ParameterBeam*

Create a beam from twiss parameters.

Parameters

- **beta_x** – Beta function in x direction in meters.
- **alpha_x** – Alpha function in x direction in rad.
- **emittance_x** – Emittance in x direction in m*rad.
- **beta_y** – Beta function in y direction in meters.
- **alpha_y** – Alpha function in y direction in rad.
- **emittance_y** – Emittance in y direction in m*rad.
- **energy** – Energy of the beam in eV.
- **total_charge** – Total charge of the beam in C.

property mu_p: Tensor

property mu_s: Tensor

property mu_x: Tensor

property mu_xp: Tensor

property mu_y: Tensor

property mu_yp: Tensor

property sigma_p: Tensor

property sigma_s: Tensor

property sigma_x: Tensor

property sigma_xp: Tensor

property sigma_xxp: Tensor

property sigma_y: Tensor

property sigma_yp: Tensor

property sigma_yyp: Tensor

training: bool

transformed_to(*mu_x: Tensor | None = None*, *mu_xp: Tensor | None = None*, *mu_y: Tensor | None = None*, *mu_yp: Tensor | None = None*, *sigma_x: Tensor | None = None*, *sigma_xp: Tensor | None = None*, *sigma_y: Tensor | None = None*, *sigma_yp: Tensor | None = None*, *sigma_s: Tensor | None = None*, *sigma_p: Tensor | None = None*, *energy: Tensor | None = None*, *total_charge: Tensor | None = None*, *device=None*, *dtype=torch.float32*) → *ParameterBeam*

Create version of this beam that is transformed to new beam parameters.

Parameters

- **n** – Number of particles to generate.
- **mu_x** – Center of the particle distribution on x in meters.
- **mu_xp** – Center of the particle distribution on x' in rad.
- **mu_y** – Center of the particle distribution on y in meters.
- **mu_yp** – Center of the particle distribution on y' in rad.
- **sigma_x** – Sigma of the particle distribution in x direction in meters.
- **sigma_xp** – Sigma of the particle distribution in x' direction in rad.
- **sigma_y** – Sigma of the particle distribution in y direction in meters.
- **sigma_yp** – Sigma of the particle distribution in y' direction in rad.
- **sigma_s** – Sigma of the particle distribution in s direction in meters.
- **sigma_p** – Sigma of the particle distribution in p, dimensionless.
- **energy** – Energy of the beam in eV.
- **total_charge** – Total charge of the beam in C.

class particles.**ParticleBeam**(*particles: Tensor*, *energy: Tensor*, *particle_charges: Tensor | None = None*, *device=None*, *dtype=torch.float32*)

Beam of charged particles, where each particle is simulated.

Parameters

- **particles** – List of 7-dimensional particle vectors.
- **energy** – Energy of the beam in eV.
- **total_charge** – Total charge of the beam in C.

- **device** – Device to move the beam's particle array to. If set to *"auto"* a CUDA GPU is selected if available. The CPU is used otherwise.

classmethod from_astra(*path: str*, *device=None*, *dtype=torch.float32*) → *ParticleBeam*

Load an Astra particle distribution as a Cheetah Beam.

classmethod from_ocelot(*parray*, *device=None*, *dtype=torch.float32*) → *ParticleBeam*

Convert an Ocelot ParticleArray *parray* to a Cheetah Beam.

classmethod from_parameters(*num_particles: Tensor | None = None*, *mu_x: Tensor | None = None*, *mu_y: Tensor | None = None*, *mu_xp: Tensor | None = None*, *mu_yp: Tensor | None = None*, *sigma_x: Tensor | None = None*, *sigma_y: Tensor | None = None*, *sigma_xp: Tensor | None = None*, *sigma_yp: Tensor | None = None*, *sigma_s: Tensor | None = None*, *sigma_p: Tensor | None = None*, *cor_x: Tensor | None = None*, *cor_y: Tensor | None = None*, *cor_s: Tensor | None = None*, *energy: Tensor | None = None*, *total_charge: Tensor | None = None*, *device=None*, *dtype=torch.float32*) → *ParticleBeam*

Generate Cheetah Beam of random particles.

Parameters

- **num_particles** – Number of particles to generate.
- **mu_x** – Center of the particle distribution on x in meters.
- **mu_y** – Center of the particle distribution on y in meters.
- **mu_xp** – Center of the particle distribution on x' in rad.
- **mu_yp** – Center of the particle distribution on y' in metraders.
- **sigma_x** – Sigma of the particle distribution in x direction in meters.
- **sigma_y** – Sigma of the particle distribution in y direction in meters.
- **sigma_xp** – Sigma of the particle distribution in x' direction in rad.
- **sigma_yp** – Sigma of the particle distribution in y' direction in rad.
- **sigma_s** – Sigma of the particle distribution in s direction in meters.
- **sigma_p** – Sigma of the particle distribution in p, dimensionless.
- **cor_x** – Correlation between x and x'.
- **cor_y** – Correlation between y and y'.
- **cor_s** – Correlation between s and p.
- **energy** – Energy of the beam in eV.
- **device** – Device to move the beam's particle array to. If set to *"auto"* a CUDA GPU is selected if available. The CPU is used otherwise.

Total_charge
Total charge of the beam in C.

classmethod from_twiss(*num_particles: Tensor | None = None*, *beta_x: Tensor | None = None*, *alpha_x: Tensor | None = None*, *emittance_x: Tensor | None = None*, *beta_y: Tensor | None = None*, *alpha_y: Tensor | None = None*, *emittance_y: Tensor | None = None*, *energy: Tensor | None = None*, *sigma_s: Tensor | None = None*, *sigma_p: Tensor | None = None*, *cor_s: Tensor | None = None*, *total_charge: Tensor | None = None*, *device=None*, *dtype=torch.float32*) → *ParticleBeam*

Create a beam from twiss parameters.

Parameters

- **beta_x** – Beta function in x direction in meters.
- **alpha_x** – Alpha function in x direction in rad.
- **emittance_x** – Emittance in x direction in m*rad.
- **beta_y** – Beta function in y direction in meters.
- **alpha_y** – Alpha function in y direction in rad.
- **emittance_y** – Emittance in y direction in m*rad.
- **energy** – Energy of the beam in eV.
- **total_charge** – Total charge of the beam in C.

classmethod make_linspaced(*num_particles: Tensor | None = None*, *mu_x: Tensor | None = None*, *mu_y: Tensor | None = None*, *mu_xp: Tensor | None = None*, *mu_yp: Tensor | None = None*, *sigma_x: Tensor | None = None*, *sigma_y: Tensor | None = None*, *sigma_xp: Tensor | None = None*, *sigma_yp: Tensor | None = None*, *sigma_s: Tensor | None = None*, *sigma_p: Tensor | None = None*, *energy: Tensor | None = None*, *total_charge: Tensor | None = None*, *device=None*, *dtype=torch.float32*) → *ParticleBeam*

Generate Cheetah Beam of *n* linspaced particles.

Parameters

- **n** – Number of particles to generate.
- **mu_x** – Center of the particle distribution on x in meters.
- **mu_y** – Center of the particle distribution on y in meters.
- **mu_xp** – Center of the particle distribution on x' in rad.
- **mu_yp** – Center of the particle distribution on y' in rad.
- **sigma_x** – Sigma of the particle distribution in x direction in meters.
- **sigma_y** – Sigma of the particle distribution in y direction in meters.
- **sigma_xp** – Sigma of the particle distribution in x' direction in rad.
- **sigma_yp** – Sigma of the particle distribution in y' direction in rad.
- **sigma_s** – Sigma of the particle distribution in s direction in meters.
- **sigma_p** – Sigma of the particle distribution in p, dimensionless.
- **energy** – Energy of the beam in eV.
- **device** – Device to move the beam's particle array to. If set to *"auto"* a CUDA GPU is selected if available. The CPU is used otherwise.

property mu_p: Tensor | None

property mu_s: Tensor | None

property mu_x: Tensor | None

property mu_xp: Tensor | None

property mu_y: float | None

property mu_yp: Tensor | None

property num_particles: Tensor

property ps: Tensor | None

property sigma_p: Tensor | None

property sigma_s: Tensor | None

property sigma_x: Tensor | None

property sigma_xp: Tensor | None

property sigma_xxp: Tensor

property sigma_y: Tensor | None

property sigma_yp: Tensor | None

property sigma_yyp: Tensor

property ss: Tensor | None

property total_charge: Tensor

training: bool

transformed_to(*mu_x: Tensor | None = None*, *mu_y: Tensor | None = None*, *mu_xp: Tensor | None = None*, *mu_yp: Tensor | None = None*, *sigma_x: Tensor | None = None*, *sigma_y: Tensor | None = None*, *sigma_xp: Tensor | None = None*, *sigma_yp: Tensor | None = None*, *sigma_s: Tensor | None = None*, *sigma_p: Tensor | None = None*, *energy: Tensor | None = None*, *total_charge: Tensor | None = None*, *device=None*, *dtype=torch.float32*) → *ParticleBeam*

Create version of this beam that is transformed to new beam parameters.

Parameters

- **n** – Number of particles to generate.
- **mu_x** – Center of the particle distribution on x in meters.
- **mu_y** – Center of the particle distribution on y in meters.
- **mu_xp** – Center of the particle distribution on x' in rad.
- **mu_yp** – Center of the particle distribution on y' in rad.
- **sigma_x** – Sigma of the particle distribution in x direction in meters.
- **sigma_y** – Sigma of the particle distribution in y direction in meters.
- **sigma_xp** – Sigma of the particle distribution in x' direction in rad.
- **sigma_yp** – Sigma of the particle distribution in y' direction in rad.
- **sigma_s** – Sigma of the particle distribution in s direction in meters.
- **sigma_p** – Sigma of the particle distribution in p, dimensionless.
- **energy** – Energy of the beam in eV.
- **total_charge** – Total charge of the beam in C.
- **device** – Device to move the beam's particle array to. If set to *"auto"* a CUDA GPU is selected if available. The CPU is used otherwise.

property xps: Tensor | None

property xs: Tensor | None

property yps: Tensor | None

property ys: Tensor | None

3.8 Track Methods

Utility functions for creating transfer maps for the elements.

track_methods.**base_rmatrix**(*length: Tensor*, *k1: Tensor*, *hx: Tensor*, *tilt: Tensor | None = None*, *energy: Tensor | None = None*) → Tensor

Create a universal transfer matrix for a beamline element.

Parameters

- **length** – Length of the element in m.
- **k1** – Quadrupole strength in 1/m**2.
- **hx** – Curvature (1/radius) of the element in 1/m**2.
- **tilt** – Roation of the element relative to the longitudinal axis in rad.
- **energy** – Beam energy in eV.

Returns

Transfer matrix for the element.

track_methods.**misalignment_matrix**(*misalignment: Tensor*) → tuple[Tensor, Tensor]

Shift the beam for tracking beam through misaligned elements

track_methods.**rotation_matrix**(*angle: Tensor*) → Tensor

Rotate the transfer map in x-y plane

Parameters

angle – Rotation angle in rad, for example *angle = np.pi/2* for vertical = dipole.

Returns

Rotation matrix to be multiplied to the element's transfer matrix.

3.9 Utils

class utils.**UniqueNameGenerator**(*prefix: str*)

Generates a unique name given a prefix.

CHAPTER

FOUR

CITE CHEETAH

If you use Cheetah, please cite the following two papers:

```
@misc{kaiser2024cheetah,
  title         = {Cheetah: Bridging the Gap Between Machine Learning and Particle␣
↪Accelerator Physics with High-Speed, Differentiable Simulations},
  author        = {Kaiser, Jan and Xu, Chenran and Eichler, Annika and {Santamaria␣
↪Garcia}, Andrea},
  year          = {2024},
  eprint        = {2401.05815},
  archiveprefix = {arXiv},
  primaryclass  = {physics.acc-ph}
}
@inproceedings{stein2022accelerating,
  title     = {Accelerating Linear Beam Dynamics Simulations for Machine Learning␣
↪Applications},
  author    = {Stein, Oliver and Kaiser, Jan and Eichler, Annika},
  year      = {2022},
  booktitle = {Proceedings of the 13th International Particle Accelerator Conference}
}
```

CHAPTER

FIVE

FOR DEVELOPERS

Activate your virtual environment. (Optional)

Install the cheetah package as editable

```
pip install -e .
```

We suggest installing pre-commit hooks to automatically conform with the code formatting in commits:

```
pip install pre-commit
pre-commit install
```

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

G

H

I

L

M

T

U

V

W

X

Y