DiPAS

Release 0.1.dev50+g3cd93de

Dominik Vilsmeier

Apr 16, 2020

CONTENTS:

CHAPTER

ONE

INSTALLATION

DiPAS requires `Python >= 3.7` and can be installed from PyPI: `pip install dipas`.

On Windows it is recommended to use Anaconda or Miniconda for the Python setup in order to make sure the project's dependencies are handled correctly.

CHAPTER

TWO

CONVENTIONS

Regarding the various conventions we mostly follow MADX in order to provide a smooth transition from one program to the other and hence the user may refer to[1] (Chapter 1) for details.

- Units for the various physical quantities follow the MADX definitions (see[1], Chapter 1.8).
- The phase-space coordinates used for particle tracking are `(x, px, y, py, t, pt)` (see[2] for details).
- A right-handed curvilinear coordinate system is assumed (see[1], Chapter 1.1).

Similar analogies with MADX hold for the various commands and their parameters.

[1] Hans Grote, Frank Schmidt, Laurent Deniau and Ghislain Roy, "The MAD-X Program (Methodical Accelerator Design) Version 5.02.08 - User's Reference Manual, 2016

[2] F. Christoph Iselin, "The MAD program (Methodical Accelerator Design) Version 8.13 - Physical Methods Manual", 1994

CHAPTER

THREE

COMPATIBILITY WITH MADX

The DiPAS package ships with a MADX parser which can parse most of the MADX syntax. Hence parsing lattices from MADX files should work without problems in most cases. Due to the (dynamically typed) nature of the parser a few noteworthy differences to the MADX syntax exist however and are explained in the following sub-sections. An essential add-on to the syntax is described as well. If desired, the parsing process can be further customized via the `madx.parser` module attributes. Please refer to this module's documentation for more information. The MADX parser is fully contained in the `madx` subpackage. The main entry functions for parsing MADX scripts are `madx.parse_file` and `madx.parse_script`.

3.1 Declaring variables used for the optimization process (add-on)

This is an addition to the existing MADX syntax and in order not to interfere with it, this is realized via placement of special comments. Since the purpose of differentiable simulations is to optimize some set of parameters, a seamless syntax for indicating the relevant parameters is desirable (we call these to-be-optimized-for parameters "*flow variables*"). This can be done directly in the MADX scripts, by placing special comments of the form `// <optional text goes here> [flow] variable`, i.e. a comment that is concluded with the string `[flow] variable`. These can be placed in three different ways to mark a variable (or attribute) as an optimization parameter.

On the same line as the variable definition:

```
q1_k1 = 0;  // [flow] variable
q1: quadrupole, l=1, k1=q1_k1;
```

On the line preceding the variable definition:

```
// [flow] variable
q1_k1 = 0;
q1: quadrupole, l=1, k1=q1_k1;
```

On the same line that sets an attribute value:

```
q1: quadrupole, l=1, k1=0;
q1->k1 = 0;  // [flow] variable
```

All of the above three cases will create a Quadrupole element with variable (to be optimized) `k1` attribute with initial value set to `0`.

The same syntax also works with error definitions, for example:

```
SELECT, flag = error, class = quadrupole;
dx = 0.001;  // [flow] variable
EALIGN, dx = dx;
```

This will cause all Quadrupole elements to have an initial alignment error of `dx = 0.001` which are however variable during the optimization process.

Flow variables also work with deferred expressions:

```
SELECT, flag = "error", class = "quadrupole";
dx := ranf() - 0.5;  // [flow] variable
EALIGN, dx = dx;
```

Here again each Quadrupole's `dx` alignment error will be optimized for and has a random initial value in `[-0.5, 0.5]`.

3.2 Variations to the MADX syntax

- **Beam command** - For `BEAM` commands the particle type as well as the beam energy must be unambiguously specified (via `particle` or `{mass, charge}` and one of `energy, pc, beta, gamma, brho`).
- **String literals** - String literals without spaces may be unquoted only for the following set of command attributes: `{'particle', 'range', 'class', 'pattern', 'flag', 'file', 'period', 'sequence', 'refer', 'apertype', 'name'}`. Which attributes are considered to be string attributes is regulated by the `madx.command_str_attributes` variable and users may add their own names if appropriate. All other string attributes must use quotation marks for correct parsing.
- **Variable names** - All variable names containing a dot `.` will be renamed by replacing the dot with an underscore. In case a similar name (with an underscore) is already used somewhere else in the script a warning will be issued. The string which will be used to replace dots in variable names can be configure via `madx.replacement_string_for_dots_in_variable_names`. It needs to be part of the set of valid characters for Python names (see the docs, basically this is `[A-Za-z0-9_]`).
- **Aperture checks** - Aperture checks on Drift spaces will be performed at the entrance of the drift space (as opposed to MADX). In case intermediary aperture checks for long drift spaces are desired, appropriate markers can be placed in-between. The `maxaper` attribute of the `RUN` command is ignored. Only explicitly defined apertures of the elements are considered.
- **Random number generation** - All the random functions from MADX are supported however the underlying random number generator (RNG) is (potentially) different. For that reason, even if the same seed for the RNG is used, the values generated by MADX and by DiPAS will likely differ. For that reason is it important, when comparing results obtained with DiPAS and MADX, to always generate a new MADX script from the particular DiPAS lattice to ensure the same numerical values from random functions. If the original MADX script (from which the DiPAS lattice was parsed) is used, then these values might differ and hence the results are not comparable. For error definitions the user can load and assign the specific errors which were generated by MADX (see *build.assign_errors*).
- **Single-argument commands** - Commands that have a single argument without specifying an argument name, such as `SHOW` or `EXEC`, are interpreted to indicate a (single) flag, analogue to `OPTION`. For example using `OPTION` MADX considers the following usages equivalent: `OPTION, warn;` and `OPTION, warn = true;` (i.e. `warn` being a positive flag). The DiPAS parser treats other commands in a similar manner, for example `SHOW, q1;` will be converted to `SHOW, q1 = true;`. The same holds also for `VALUE` but expressions here need to be unquoted, otherwise this will result in a parsing error. That means when inspecting the resulting command list these are still useful with the subtlety that the single-arguments are stored as argument names together with the argument value `"true"`.

3.3 Unsupported statement types

- Program flow constructs such as `if / else` or `while`.
- Macro definitions.
- Commands that take a single quoted string as argument without specifying an argument name such as `TITLE` or `SYSTEM`.
- Template beamlines defined via `label(arg): LINE = (arg);` ("normal" beamline definitions (without `arg`) can be used though).

CHAPTER

FOUR

USAGE

The DiPAS package can be used for various tasks, among which are

- parsing MADX scripts,
- building lattices, either from scripts or programmatically,
- (differentiable) particle tracking,
- (differentiable) optics calculations, such as closed orbit search and Twiss computation,
- serializing lattices to MADX scripts,
- run MADX scripts and related tasks,
- visualizing lattices.

4.1 Building lattices

Lattices can be built by parsing MADX scripts or programmatically using the API of the package.

4.1.1 Parsing MADX scripts

The main functions for parsing MADX scripts to lattices are `build.from_file` and `build.from_script`. The only difference is that the former expects the file name to the script and the latter the raw script as a string:

```
from dipas.build import from_file, from_script

lattice = from_file('example.madx')

with open('example.madx') as fh:  # alternatively build from script string
    lattice = from_script(fh.read())
```

In case the MADX script contains an unknown element, a warning will be issued and the element is skipped. The supported elements can be found by inspecting the `elements.elements` dict; keys are MADX command names and values are the corresponding PyTorch backend modules.

[1]:
```
from pprint import pprint
from dipas.elements import elements

pprint(elements)
```

```
{'dipedge': <class 'dipas.elements.Dipedge'>,
 'drift': <class 'dipas.elements.Drift'>,
 'hkicker': <class 'dipas.elements.HKicker'>,
 'hmonitor': <class 'dipas.elements.HMonitor'>,
 'instrument': <class 'dipas.elements.Instrument'>,
 'kicker': <class 'dipas.elements.Kicker'>,
 'marker': <class 'dipas.elements.Marker'>,
 'monitor': <class 'dipas.elements.Monitor'>,
 'placeholder': <class 'dipas.elements.Placeholder'>,
 'quadrupole': <class 'dipas.elements.Quadrupole'>,
 'rbend': <class 'dipas.elements.RBend'>,
 'sbend': <class 'dipas.elements.SBend'>,
 'sbendbody': <class 'dipas.elements.SBendBody'>,
 'sextupole': <class 'dipas.elements.Sextupole'>,
 'tkicker': <class 'dipas.elements.TKicker'>,
 'vkicker': <class 'dipas.elements.VKicker'>,
 'vmonitor': <class 'dipas.elements.VMonitor'>}
```

Similarly we can check the supported alignment errors and aperture types:

[2]:
```
from dipas.elements import alignment_errors, aperture_types

pprint(alignment_errors)
pprint(aperture_types)
```

```
{'dpsi': <class 'dipas.elements.LongitudinalRoll'>,
 'dx': <class 'dipas.elements.Offset'>,
 'dy': <class 'dipas.elements.Offset'>,
 'mrex': <class 'dipas.elements.BPMError'>,
 'mrey': <class 'dipas.elements.BPMError'>,
 'mscalx': <class 'dipas.elements.BPMError'>,
 'mscaly': <class 'dipas.elements.BPMError'>,
 'tilt': <class 'dipas.elements.Tilt'>}
{'circle': <class 'dipas.elements.ApertureCircle'>,
 'ellipse': <class 'dipas.elements.ApertureEllipse'>,
 'rectangle': <class 'dipas.elements.ApertureRectangle'>,
 'rectellipse': <class 'dipas.elements.ApertureRectEllipse'>}
```

As can be seen seen from the above element dictionary, a general `MULTIPOLE` is not yet supported and so attempting to load a script with such a definition will raise a warning:

[3]:
```
from importlib import resources
from dipas.build import from_file
import dipas.test.sequences

with resources.path(dipas.test.sequences, 'hades.seq') as path:
    lattice = from_file(path)
```

```
/home/dominik/Projects/DiPAS/dipas/build.py:560: UserWarning: Skipping element (no␣
↪equivalent implementation found): Command(keyword='multipole',␣
↪local_attributes={'knl': array([0.]), 'at': 8.6437999}, label='gts1mu1', base=None)
  warnings.warn(f'Skipping element (no equivalent implementation found): {command}')
/home/dominik/Projects/DiPAS/dipas/build.py:560: UserWarning: Skipping element (no␣
↪equivalent implementation found): Command(keyword='multipole',␣
↪local_attributes={'knl': array([0.]), 'at': 28.6437973}, label='gte3mu1', base=None)
  warnings.warn(f'Skipping element (no equivalent implementation found): {command}')
/home/dominik/Projects/DiPAS/dipas/build.py:560: UserWarning: Skipping element (no␣
↪equivalent implementation found): Command(keyword='multipole',␣
↪local_attributes={'knl': array([0.]), 'at': 52.4014301}, label='ghhtmu1', base=None)
```

(continues on next page)

(continued from previous page)

```
  warnings.warn(f'Skipping element (no equivalent implementation found): {command}')
/home/dominik/Projects/DiPAS/dipas/build.py:560: UserWarning: Skipping element (no␣
↪equivalent implementation found): Command(keyword='multipole',␣
↪local_attributes={'knl': array([0.]), 'at': 100.2795473}, label='gth3mu1',␣
↪base=None)
  warnings.warn(f'Skipping element (no equivalent implementation found): {command}')
/home/dominik/Projects/DiPAS/dipas/build.py:560: UserWarning: Skipping element (no␣
↪equivalent implementation found): Command(keyword='multipole',␣
↪local_attributes={'knl': array([0.]), 'at': 125.7672306}, label='gtp1mu1',␣
↪base=None)
  warnings.warn(f'Skipping element (no equivalent implementation found): {command}')
```

This issues a few warnings of the following form:

```
.../dipas/build.py:174: UserWarning: Skipping element (no equivalent implementation␣
↪found): Command(keyword='multipole', local_attributes={'knl': array([0.]), 'at': 8.
↪6437999}, label='gts1mu1', base=None)
```

In order to not accidentally miss any such non-supported elements one can configure Python to raise an error whenever a warning is encountered (see the docs for more details):

```
import warnings

warnings.simplefilter('error')

with resources.path(dipas.test.sequences, 'hades.seq') as path:
    lattice = from_file(path)
```

This will convert the previous warning into an error.

4.1.2 Using the build API

We can also build a lattice using the `build.Lattice` class:

[4]:
```
from dipas.build import Lattice

with Lattice(beam=dict(particle='proton', beta=0.6)) as lattice:
    lattice.Drift(l=2)
    lattice.Quadrupole(k1=0.25, l=1, label='q1')
    lattice.Drift(l=3)
    lattice.HKicker(kick=0.1, label='hk1')
```

When used as a context manager (i.e. inside `with`) we just need to invoke the various element functions in order to append them to the lattice.

We can get an overview of the lattice by printing it:

[5]:
```
print(lattice)
```

```
[    0.000000]  Drift(l=tensor(2.), label=None)
[    2.000000]  Quadrupole(l=tensor(1.), k1=tensor(0.2500), dk1=tensor(0.),␣
↪label='q1')
[    3.000000]  Drift(l=tensor(3.), label=None)
[    6.000000]  HKicker(l=tensor(0.), hkick=tensor(0.1000), vkick=tensor(0.),␣
↪kick=tensor(0.1000), label='hk1')
```

The number in brackets `[...]` indicates the position along the lattice in meters, followed by a description of the element

Besides usage as a context manager other ways of adding elements exist:

[6]:
```
lattice = Lattice({'particle': 'proton', 'beta': 0.6})
lattice += lattice.Drift(l=2)
lattice.append(lattice.Quadrupole(k1=0.25, l=1, label='q1'))
lattice += [lattice.Drift(l=3), lattice.HKicker(kick=0.1, label='hk1')]
```

This creates the same lattice as before. Note that because `lattice` is not used as a context manager, invoking the element functions, such as `lattice.Quadrupole`, will not automatically add the element to the lattice; we can do so via `lattice += ...`, `lattice.append` or `lattice.extend`.

We can also specify positions along the lattice directly, which will also take care of inserting implicit drift spaces:

[7]:
```
lattice = Lattice({'particle': 'proton', 'beta': 0.6})
lattice[2.0] = lattice.Quadrupole(k1=0.25, l=1, label='q1')
lattice['q1', 3.0] = lattice.HKicker(kick=0.1, label='hk1')
```

This again creates the same lattice as before. We can specify an absolute position along the lattice by just using a float or we can specify a position relative to another element by using a tuple and referring to the other element via its label.

> **Note:** When using a relative position via tuple, the position is taken relative to the *exit* of the referred element.

After building the lattice in such a way there's one step left to obtain the same result as via `build.from_file` or `build.from_script`. These methods return a `elements.Segment` instance which provides further functionality for tracking and conversion to thin elements for example. We can simply convert our lattice to a Segment as follows:

[8]:
```
from dipas.elements import Segment

lattice = Segment(lattice)  # 'lattice' from before
```

4.1.3 Using the element types directly

Another option for building a lattice is to access the element classes directly. This can be done via `elements.<cls_name>` or by using the `elements.elements` dict which maps MADX command names to corresponding backend classes:

[9]:
```
from dipas.build import Beam
import dipas.elements as elements

beam = Beam(particle='proton', beta=0.6).to_dict()
sequence = [
    elements.Drift(l=2, beam=beam),
    elements.Quadrupole(k1=0.25, l=1, beam=beam, label='q1'),
    elements.elements['drift'](l=3, beam=beam),
    elements.elements['hkicker'](kick=0.1, label='hk1')
]
lattice = elements.Segment(sequence)
```

This creates the same lattice as in the previous section. Note that we had to use `Beam(...).to_dict()` and pass the result to the element classes. This is because the elements expect both `beta` and `gamma` in the `beam` dict and won't compute it themselves. `build.Beam` however does the job for us:

[10]:
```
from pprint import pprint

pprint(beam)
```

```
{'beta': 0.6,
 'brho': 2.34730408386391,
 'charge': 1,
 'energy': 1.1728401016249999,
 'gamma': 1.25,
 'mass': 0.9382720813,
 'particle': 'proton',
 'pc': 0.7037040609749998}
```

Taking care of the beam definition was done automatically by using the `build.Lattice` class as in the previous section.

4.2 Inspecting lattices

Once we have a lattice in form of a `elements.Segment` instance, such as returned from `build.from_file`, we can inspect its elements in various ways:

[1]:
```
from importlib import resources
import warnings
from dipas.build import from_file
from dipas.elements import Quadrupole, HKicker, SBend
import dipas.test.sequences

warnings.simplefilter('ignore')

with resources.path(dipas.test.sequences, 'hades.seq') as path:
    lattice = from_file(path)

print(len(lattice[Quadrupole]))
```

```
21
```

Here `lattice[Quadrupole]` returns a list containing all quadrupoles in the lattice. This can be done with any lattice element class:

[2]:
```
print('HKicker', end='\n\n')
for kicker in lattice[HKicker]:
    print(kicker)

print('\nSBend', end='\n\n')
for sbend in lattice[SBend]:
    print(sbend)
```

```
HKicker

HKicker(l=tensor(0.), hkick=tensor(0.), vkick=tensor(0.), kick=tensor(0.),␣
↪label='gte2kx1')
HKicker(l=tensor(0.), hkick=tensor(0.), vkick=tensor(0.), kick=tensor(0.),␣
↪label='gth1kx1')
HKicker(l=tensor(0.), hkick=tensor(0.), vkick=tensor(0.), kick=tensor(0.),␣
↪label='gth2kx1')
HKicker(l=tensor(0.), hkick=tensor(0.), vkick=tensor(0.), kick=tensor(0.),␣
↪label='ghadkx1')
```

(continues on next page)

(continued from previous page)

```
SBend

Tilt(psi=tensor(0.3795),
     target=SBend(l=tensor(1.4726), angle=tensor(-0.1308), e1=tensor(0.), e2=tensor(0.
↪), fint=tensor(0.), fintx=tensor(0.), hgap=tensor(0.), h1=tensor(0.), h2=tensor(0.),
↪ dk0=tensor(0.), label='ghadmu1')
    > Dipedge(l=tensor(0.), h=tensor(-0.0888), e1=tensor(0.), fint=tensor(0.),␣
↪hgap=tensor(0.), label=None)
    > SBendBody(l=tensor(1.4726), angle=tensor(-0.1308), dk0=tensor(0.), label=None)
    > Dipedge(l=tensor(0.), h=tensor(-0.0888), e1=tensor(0.), fint=tensor(0.),␣
↪hgap=tensor(0.), label=None))
Tilt(psi=tensor(-0.3795),
     target=SBend(l=tensor(1.4726), angle=tensor(-0.1311), e1=tensor(0.), e2=tensor(0.
↪), fint=tensor(0.), fintx=tensor(0.), hgap=tensor(0.), h1=tensor(0.), h2=tensor(0.),
↪ dk0=tensor(0.), label='ghadmu2')
    > Dipedge(l=tensor(0.), h=tensor(-0.0891), e1=tensor(0.), fint=tensor(0.),␣
↪hgap=tensor(0.), label=None)
    > SBendBody(l=tensor(1.4726), angle=tensor(-0.1311), dk0=tensor(0.), label=None)
    > Dipedge(l=tensor(0.), h=tensor(-0.0891), e1=tensor(0.), fint=tensor(0.),␣
↪hgap=tensor(0.), label=None))
```

Note that the `SBend`s are tilted which is indicated by the wrapping `Tilt` object. Also the two dipole edge elements are reported in terms of `Dipedge` elements.

We can select a specific element from a multi-element selection directly by providing a tuple:

[3]:
```
print(lattice[Quadrupole, 5])
print(lattice[Quadrupole, 5] is lattice[Quadrupole][5])
```

```
Quadrupole(l=tensor(1.), k1=Parameter containing: tensor(0.0298, requires_grad=True),␣
↪dk1=tensor(0.), label='gth1qd11')
True
```

As shown, the same result can of course be obtained by indexing the resulting list of multiple elements. One case where tuples are the only way however is if we want to set a specific element in the sequence. For example if we want to tilt the second `HKicker` then we can do:

[4]:
```
from dipas.elements import Tilt

lattice[HKicker, 1] = Tilt(lattice[HKicker][1], psi=0.5)
for kicker in lattice[HKicker]:
    print(kicker)
```

```
HKicker(l=tensor(0.), hkick=tensor(0.), vkick=tensor(0.), kick=tensor(0.),␣
↪label='gte2kx1')
Tilt(psi=tensor(0.5000),
     target=HKicker(l=tensor(0.), hkick=tensor(0.), vkick=tensor(0.), kick=tensor(0.),
↪ label='gth1kx1'))
HKicker(l=tensor(0.), hkick=tensor(0.), vkick=tensor(0.), kick=tensor(0.),␣
↪label='gth2kx1')
HKicker(l=tensor(0.), hkick=tensor(0.), vkick=tensor(0.), kick=tensor(0.),␣
↪label='ghadkx1')
```

Note that we specified `(HKicker, 1)` because indices start at zero. Selections also work with modifiers such as `Tilt` or alignment errors such as `Offset`:

[5]:
```
from dipas.elements import Offset

print('Elements with offset: ', lattice[Offset])

for element in lattice[Tilt]:
    print(element)
```

```
Elements with offset:  []
Tilt(psi=tensor(0.5000),
     target=HKicker(l=tensor(0.), hkick=tensor(0.), vkick=tensor(0.), kick=tensor(0.),
↪ label='gth1kx1'))
Tilt(psi=tensor(0.3795),
     target=SBend(l=tensor(1.4726), angle=tensor(-0.1308), e1=tensor(0.), e2=tensor(0.
↪), fint=tensor(0.), fintx=tensor(0.), hgap=tensor(0.), h1=tensor(0.), h2=tensor(0.),
↪ dk0=tensor(0.), label='ghadmu1')
    > Dipedge(l=tensor(0.), h=tensor(-0.0888), e1=tensor(0.), fint=tensor(0.),␣
↪hgap=tensor(0.), label=None)
    > SBendBody(l=tensor(1.4726), angle=tensor(-0.1308), dk0=tensor(0.), label=None)
    > Dipedge(l=tensor(0.), h=tensor(-0.0888), e1=tensor(0.), fint=tensor(0.),␣
↪hgap=tensor(0.), label=None))
Tilt(psi=tensor(-0.3795),
     target=SBend(l=tensor(1.4726), angle=tensor(-0.1311), e1=tensor(0.), e2=tensor(0.
↪), fint=tensor(0.), fintx=tensor(0.), hgap=tensor(0.), h1=tensor(0.), h2=tensor(0.),
↪ dk0=tensor(0.), label='ghadmu2')
    > Dipedge(l=tensor(0.), h=tensor(-0.0891), e1=tensor(0.), fint=tensor(0.),␣
↪hgap=tensor(0.), label=None)
    > SBendBody(l=tensor(1.4726), angle=tensor(-0.1311), dk0=tensor(0.), label=None)
    > Dipedge(l=tensor(0.), h=tensor(-0.0891), e1=tensor(0.), fint=tensor(0.),␣
↪hgap=tensor(0.), label=None))
```

There are no offset elements in the lattice but as we see there are three tilted elements: the two `SBend`s from before and the `HKicker` that we tilted manually.

We can also select elements by their label:

[6]:
```
print(lattice['gth1kx1'])
print(lattice['gth1kx1'] is lattice[HKicker, 1])
```

```
Tilt(psi=tensor(0.5000),
     target=HKicker(l=tensor(0.), hkick=tensor(0.), vkick=tensor(0.), kick=tensor(0.),
↪ label='gth1kx1'))
True
```

If there are multiple elements that share a label, a list will be returned instead. Again we can use a tuple index to select a specific element:

[7]:
```
from dipas.elements import Drift

print(lattice[Drift, 0].label)
lattice[Drift, 1].label = 'pad_drift_0'
print(lattice['pad_drift_0'])
print(lattice['pad_drift_0', 1])
```

```
pad_drift_0
[Drift(l=tensor(3.9057), label='pad_drift_0'), Drift(l=tensor(0.8420),␣
↪label='pad_drift_0')]
Drift(l=tensor(0.8420), label='pad_drift_0')
```

By using regular expression patterns we can select all elements whose labels match the specified pattern:

[8]:
```
import re

pattern = re.compile(r'[a-z0-9]+kx1')
for element in lattice[pattern]:
    print(element)
```

```
HKicker(l=tensor(0.), hkick=tensor(0.), vkick=tensor(0.), kick=tensor(0.),␣
↪label='gte2kx1')
Tilt(psi=tensor(0.5000),
     target=HKicker(l=tensor(0.), hkick=tensor(0.), vkick=tensor(0.), kick=tensor(0.),
↪ label='gth1kx1'))
HKicker(l=tensor(0.), hkick=tensor(0.), vkick=tensor(0.), kick=tensor(0.),␣
↪label='gth2kx1')
HKicker(l=tensor(0.), hkick=tensor(0.), vkick=tensor(0.), kick=tensor(0.),␣
↪label='ghadkx1')
```

Here we need to use a compiled `re` object because strings will be interpreted as element labels, not patterns. An exception is if the string contains an asterisk `*` which will be interpreted as a shell-style wildcard pattern (internally it is converted to a regex while replacing `*` with `.*?`). Thus using `lattice['*kx1']` selects all `HKicker` elements as before.

Last but not least we can select elements by their index position along the lattice:

[9]:
```
print(lattice[4])  # Selecting the 5-th element.
print(lattice[19])  # Selecting the 20-th element.
```

```
Drift(l=tensor(0.3370), label='pad_drift_2')
Monitor(l=tensor(0.), label='gte2dg4')
```

Sub-segments can be selected by using slice syntax. Here the start and stop parameters must be unambiguous element identifiers, as described above (e.g. unique labels, or a multi-selector such as `Quadrupole` with an occurrence count, i.e. `(Quadrupole, 5)`).

[10]:
```
print(lattice[:6], end='\n\n')
print(lattice[(Drift, 1):'gte1dg1'])
```

```
Segment(elements=[Drift(l=tensor(3.9057), label='pad_drift_0'),
 VKicker(l=tensor(0.), hkick=tensor(0.), vkick=tensor(0.), kick=tensor(0.),␣
↪label='gte1ky1'),
 Drift(l=tensor(0.8420), label='pad_drift_0'),
 Quadrupole(l=tensor(0.6660), k1=Parameter containing: tensor(0.5668,␣
↪requires_grad=True), dk1=tensor(0.), label='gte1qd11'),
 Drift(l=tensor(0.3370), label='pad_drift_2'),
 Monitor(l=tensor(0.), label='gte1dg1')])

Segment(elements=[Drift(l=tensor(0.8420), label='pad_drift_0'),
 Quadrupole(l=tensor(0.6660), k1=Parameter containing: tensor(0.5668,␣
↪requires_grad=True), dk1=tensor(0.), label='gte1qd11'),
 Drift(l=tensor(0.3370), label='pad_drift_2'),
 Monitor(l=tensor(0.), label='gte1dg1')])
```

4.3 Modifying lattices

We can modify single lattice elements or the lattice itself. In the previous section there was already a hint about how to replace specific lattice elements. This can be done via `lattice[identifier] = ...` where `identifier` must unambiguously identify a lattice element. That is `lattice[identifier]` (not setting, but getting the element) should return a single element, not a list of elements. Note that `identifer` can be a tuple as well, in order to

narrow down the selection. For example, let's offset the `Quadrupole` with label `"gte1qd11"` and tilt the second `Kicker`:

[1]:
```
from importlib import resources
import warnings
from dipas.build import from_file
from dipas.elements import HKicker, Offset, Tilt
import dipas.test.sequences

warnings.simplefilter('ignore')

with resources.path(dipas.test.sequences, 'hades.seq') as path:
    lattice = from_file(path)  # Load a fresh lattice.

print(lattice['gte1qd11'])  # Returns a single element, good.
lattice['gte1qd11'] = Offset(lattice['gte1qd11'], dx=0.25, dy=0.50)
print(lattice['gte1qd11'], end='\n\n')

print(lattice[HKicker, 1])  # Returns a single element, good.
lattice[HKicker, 1] = Tilt(lattice[HKicker, 1], psi=1.0)
print(lattice[HKicker, 1])
```

```
Quadrupole(l=tensor(0.6660), k1=Parameter containing: tensor(0.5668,␣
↪requires_grad=True), dk1=tensor(0.), label='gte1qd11')
Offset(dx=tensor(0.2500), dy=tensor(0.5000),
       target=Quadrupole(l=tensor(0.6660), k1=Parameter containing: tensor(0.5668,␣
↪requires_grad=True), dk1=tensor(0.), label='gte1qd11'))

HKicker(l=tensor(0.), hkick=tensor(0.), vkick=tensor(0.), kick=tensor(0.),␣
↪label='gth1kx1')
Tilt(psi=tensor(1.),
     target=HKicker(l=tensor(0.), hkick=tensor(0.), vkick=tensor(0.), kick=tensor(0.),
↪ label='gth1kx1'))
```

We can of course also modify attributes of single elements. For example let's introduce some random errors to the quadrupole gradient strengths:

[2]:
```
import numpy as np
from dipas.elements import Quadrupole

for quad in lattice[Quadrupole]:
    quad.element.k1.data += np.random.normal(scale=0.1)
```

Two things are worth noting here:

1. We used `quad.element.k1` instead of just `quad.k1`. This is because `lattice[Quadrupole]` returns a list of all `Quadrupole` elements, potentially wrapped by alignment error classes. Because we applied an offset to the first quadrupole beforehand, the first `quad` is actually an `Offset` object. By using `quad.element` we ensure that we always get the underlying `Quadrupole` object. Using `element` on a `Quadrupole` itself will just return the same object.
2. We used `k1.data` instead of just `k1`. This is because the MADX sequence file that we used to parse the lattice from actually contained optimization parameter definition (see the next section for more details) and so we need to use `.data` to modify the actual number of the tensor.

4.4 Converting thick to thin elements

Not all lattice elements support thick tracking and so converting these elements to thin slices is necessary before doing particle tracking or optics calculations. Elements can be converted to their thin representation using the `makethin` method:

[1]:
```
from dipas.build import Lattice

with Lattice({'particle': 'proton', 'beta': 0.6}) as lattice:
    lattice.HKicker(kick=0.5, l=1.0, label='hk1')
    lattice.Quadrupole(k1=0.625, l=5.0, label='q1')

kicker, quad = lattice
print(kicker)
print(kicker.makethin(5))
```

```
HKicker(l=tensor(1.), hkick=tensor(0.5000), vkick=tensor(0.), kick=tensor(0.5000),␣
↪label='hk1')
HKicker(l=tensor(1.), hkick=tensor(0.5000), vkick=tensor(0.), kick=tensor(0.5000),␣
↪label='hk1')
    > Drift(l=tensor(0.0833), label='hk1__d0')
    > HKicker(l=tensor(0.), hkick=tensor(0.1000), vkick=tensor(0.), kick=tensor(0.
↪1000), label='hk1__0')
    > Drift(l=tensor(0.2083), label='hk1__d1')
    > HKicker(l=tensor(0.), hkick=tensor(0.1000), vkick=tensor(0.), kick=tensor(0.
↪1000), label='hk1__1')
    > Drift(l=tensor(0.2083), label='hk1__d2')
    > HKicker(l=tensor(0.), hkick=tensor(0.1000), vkick=tensor(0.), kick=tensor(0.
↪1000), label='hk1__2')
    > Drift(l=tensor(0.2083), label='hk1__d3')
    > HKicker(l=tensor(0.), hkick=tensor(0.1000), vkick=tensor(0.), kick=tensor(0.
↪1000), label='hk1__3')
    > Drift(l=tensor(0.2083), label='hk1__d4')
    > HKicker(l=tensor(0.), hkick=tensor(0.1000), vkick=tensor(0.), kick=tensor(0.
↪1000), label='hk1__4')
    > Drift(l=tensor(0.0833), label='hk1__d5')
```

The `makethin` method returns a `elements.ThinElement` object, a special version of a more general `Segment`. This `ThinElement` contains the thin kicker slices as well as the drift space before, between and after the slices. The distribution of drift space depends on the selected slicing style. By default the `TEAPOT` style is used. Other available slicing styles include `SIMPLE` and `EDGE`. For more details consider the documentation of the `elements.ThinElement.create_thin_sequence` method.

Let's compare the `SIMPLE` and `EDGE` style for the quadrupole element:

[2]:
```
print('EDGE', end='\n\n')
print(quad.makethin(5, style='edge'), end='\n\n')
print('SIMPLE', end='\n\n')
print(quad.makethin(5, style='simple'), end='\n\n')
```

```
EDGE

Quadrupole(l=tensor(5.), k1=tensor(0.6250), dk1=tensor(0.), label='q1')
    > Drift(l=tensor(0.), label='q1__d0')
    > ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), dk1l=tensor(0.), label='q1__0')
    > Drift(l=tensor(1.2500), label='q1__d1')
    > ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), dk1l=tensor(0.), label='q1__1')
    > Drift(l=tensor(1.2500), label='q1__d2')
```

(continues on next page)

(continued from previous page)

```
    > ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), dk1l=tensor(0.), label='q1__2')
    > Drift(l=tensor(1.2500), label='q1__d3')
    > ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), dk1l=tensor(0.), label='q1__3')
    > Drift(l=tensor(1.2500), label='q1__d4')
    > ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), dk1l=tensor(0.), label='q1__4')
    > Drift(l=tensor(0.), label='q1__d5')

SIMPLE

Quadrupole(l=tensor(5.), k1=tensor(0.6250), dk1=tensor(0.), label='q1')
    > Drift(l=tensor(0.5000), label='q1__d0')
    > ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), dk1l=tensor(0.), label='q1__0')
    > Drift(l=tensor(1.), label='q1__d1')
    > ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), dk1l=tensor(0.), label='q1__1')
    > Drift(l=tensor(1.), label='q1__d2')
    > ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), dk1l=tensor(0.), label='q1__2')
    > Drift(l=tensor(1.), label='q1__d3')
    > ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), dk1l=tensor(0.), label='q1__3')
    > Drift(l=tensor(1.), label='q1__d4')
    > ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), dk1l=tensor(0.), label='q1__4')
    > Drift(l=tensor(0.5000), label='q1__d5')
```

`EDGE` places the outermost slices directly at the edges of the thick element, while `SIMPLE` adds a margin that is half the in-between distance of slices.

We can also convert whole lattices represented by `Segment` objects to thin elements. Here we can choose the number of slices as well as the style via a dict which maps element identifiers to the particular values. The identifiers can be strings for comparing element labels, regex patterns for matching element labels or lattice element types, similar to element selection via `lattice[identifier]` (see inspecting lattices).

[3]:
```
from dipas.elements import HKicker, Quadrupole, Segment

lattice = Segment(lattice)
thin = lattice.makethin({HKicker: 2, 'q1': 5}, style={'hk1': 'edge', Quadrupole:
↪'simple'})
print(thin)
```

```
Segment(elements=[HKicker(l=tensor(1.), hkick=tensor(0.5000), vkick=tensor(0.),␣
↪kick=tensor(0.5000), label='hk1')
    > Drift(l=tensor(0.), label='hk1__d0')
    > HKicker(l=tensor(0.), hkick=tensor(0.2500), vkick=tensor(0.), kick=tensor(0.
↪2500), label='hk1__0')
    > Drift(l=tensor(1.), label='hk1__d1')
    > HKicker(l=tensor(0.), hkick=tensor(0.2500), vkick=tensor(0.), kick=tensor(0.
↪2500), label='hk1__1')
    > Drift(l=tensor(0.), label='hk1__d2'),
 Quadrupole(l=tensor(5.), k1=tensor(0.6250), dk1=tensor(0.), label='q1')
    > Drift(l=tensor(0.5000), label='q1__d0')
    > ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), dk1l=tensor(0.), label='q1__0')
    > Drift(l=tensor(1.), label='q1__d1')
    > ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), dk1l=tensor(0.), label='q1__1')
    > Drift(l=tensor(1.), label='q1__d2')
    > ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), dk1l=tensor(0.), label='q1__2')
    > Drift(l=tensor(1.), label='q1__d3')
    > ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), dk1l=tensor(0.), label='q1__3')
    > Drift(l=tensor(1.), label='q1__d4')
```

(continues on next page)

(continued from previous page)

```
    > ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), dk1l=tensor(0.), label='q1__4')
    > Drift(l=tensor(0.5000), label='q1__d5')])
```

The `ThinElements` represent their thick counterparts which are still accessible via the `.base` attribute. Also the base label is inherited (element access works as explained in inspecting lattices):

[4]:
```
print('q1.base: ', thin['q1'].base)
print('q1.label: ', thin[1].label)
for drift in thin['q1']['q1__d*']:
    print(drift)
```

```
q1.base:  Quadrupole(l=tensor(5.), k1=tensor(0.6250), dk1=tensor(0.), label='q1')
q1.label:  q1
Drift(l=tensor(0.5000), label='q1__d0')
Drift(l=tensor(1.), label='q1__d1')
Drift(l=tensor(1.), label='q1__d2')
Drift(l=tensor(1.), label='q1__d3')
Drift(l=tensor(1.), label='q1__d4')
Drift(l=tensor(0.5000), label='q1__d5')
```

We can also flatten such a nested `Segment`, containing `ThinElements`, using the `flat` (or `flatten`) method:

[5]:
```
print(thin.flat())
```

```
Segment(elements=[Drift(l=tensor(0.), label='hk1__d0'),
 HKicker(l=tensor(0.), hkick=tensor(0.2500), vkick=tensor(0.), kick=tensor(0.2500),␣
↪label='hk1__0'),
 Drift(l=tensor(1.), label='hk1__d1'),
 HKicker(l=tensor(0.), hkick=tensor(0.2500), vkick=tensor(0.), kick=tensor(0.2500),␣
↪label='hk1__1'),
 Drift(l=tensor(0.), label='hk1__d2'),
 Drift(l=tensor(0.5000), label='q1__d0'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), dk1l=tensor(0.), label='q1__0'),
 Drift(l=tensor(1.), label='q1__d1'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), dk1l=tensor(0.), label='q1__1'),
 Drift(l=tensor(1.), label='q1__d2'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), dk1l=tensor(0.), label='q1__2'),
 Drift(l=tensor(1.), label='q1__d3'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), dk1l=tensor(0.), label='q1__3'),
 Drift(l=tensor(1.), label='q1__d4'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), dk1l=tensor(0.), label='q1__4'),
 Drift(l=tensor(0.5000), label='q1__d5')])
```

`flatten()` returns a generator over all the nested elements.

4.5 Specifying optimization parameters

Optimization parameters can be specified either directly in the MADX files that are parsed or they can set on the lattice elements after parsing (or building in general), similar to modifying lattice elements as seen in the previous section.

4.5.1 Inside MADX scripts

For details please consider the documentation part about "Compatibility with MADX".

Optimization parameters ("flow variables") can be indicates by placing dedicated comments in MADX scripts. There comments should be of the form `// <some optional text> [flow] variable` and should either precede or conclude the line of a variable definition or attribute assignment. Let's peek into one of the example scripts:

[1]:
```
from importlib import resources
from pprint import pprint
import dipas.test.sequences

script = resources.read_text(dipas.test.sequences, 'hades.seq')
script = script.splitlines()
pprint(script[:4])
```

```
['beam, particle=ion, charge=6, energy=28.5779291448, mass=11.1779291448;',
 '',
 'k1l_GTE1QD11 := 0.3774561583995819;       // [flow] variable',
 'k1l_GTE1QD12 := -0.35923901200294495;     // [flow] variable']
```

Here we can see that the two variables `k1l_GTE1QD11` and `k1l_GTE1QD12` have been declared as optimization parameters. Now let's check the lattice element after parsing the script:

[2]:
```
import warnings
from dipas.build import from_script

warnings.simplefilter('ignore')

lattice = from_script('\n'.join(script))
print(lattice['gte1qd11'])
print(lattice['gte1qd12'])
print(type(lattice['gte1qd11'].k1))
```

```
Quadrupole(l=tensor(0.6660), k1=Parameter containing: tensor(0.5668,␣
↪requires_grad=True), dk1=tensor(0.), label='gte1qd11')
Quadrupole(l=tensor(0.6660), k1=Parameter containing: tensor(-0.5394,␣
↪requires_grad=True), dk1=tensor(0.), label='gte1qd12')
<class 'torch.nn.parameter.Parameter'>
```

Here we can see that the `k1` parameters are indicates as `Parameter` which can be optimized for using PyTorch's optimization machinery.

4.5.2 Using the API

Now let's modify the script so that the first quadrupole's `k1` attribute won't be parsed to a parameter:

[3]:
```
script[2] = script[2].split(';')[0] + ';'
pprint(script[:4])
lattice = from_script('\n'.join(script))
print(lattice['gte1qd11'])
print(lattice['gte1qd12'])
```

```
['beam, particle=ion, charge=6, energy=28.5779291448, mass=11.1779291448;',
 '',
 'k1l_GTE1QD11 := 0.3774561583995819;',
 'k1l_GTE1QD12 := -0.35923901200294495;     // [flow] variable']
Quadrupole(l=tensor(0.6660), k1=tensor(0.5668), dk1=tensor(0.), label='gte1qd11')
Quadrupole(l=tensor(0.6660), k1=Parameter containing: tensor(-0.5394,␣
↪requires_grad=True), dk1=tensor(0.), label='gte1qd12')
```

Now the `lattice['gte1qd11'].k1` attribute is set as a tensor. If we want to optimize for that value nevertheless we can simply convert it to a parameter manually:

[4]:
```
import torch

lattice['gte1qd11'].k1 = torch.nn.Parameter(lattice['gte1qd11'].k1)
print(lattice['gte1qd11'])
```

```
Quadrupole(l=tensor(0.6660), k1=Parameter containing: tensor(0.5668,␣
↪requires_grad=True), dk1=tensor(0.), label='gte1qd11')
```

Similarly we could convert the k1-values of all quadrupoles to parameters:

[5]:
```
from dipas.elements import Quadrupole

for q in lattice[Quadrupole]:
    q.k1 = torch.nn.Parameter(q.k1)
```

For the example lattice however the k1-values are already parameters.

4.6 Particle tracking

Particle tracking can be performed by referring to the various methods of the lattice elements or similarly the lattice itself. For example linear optics tracking can be done via the `linear` instance method:

[1]:
```
from importlib import resources
from dipas.build import from_file
from dipas.elements import Kicker
import dipas.test.sequences
import torch
torch.manual_seed(1)

with resources.path(dipas.test.sequences, 'cryring.seq') as path:
    lattice = from_file(path)

lattice = lattice.makethin({Kicker: 3})  # Need to make thin for tracking.

particles = 0.001 * torch.randn(6, 1000)  # 1000 particles
print(particles[[0, 2], :].std(dim=1))

tracked = lattice.linear(particles)
print(tracked[[0, 2], :].std(dim=1))
```

```
tensor([0.0010, 0.0010])
tensor([0.0083, 0.0014])
```

This tracks one turn through the lattice. By default no aperture checks are performed. We can enable aperture checks by setting the parameter `aperture=True`:

[2]:
```
particles = 0.01 * torch.randn(6, 1000)
tracked = lattice.linear(particles, aperture=True)
print(tracked.shape)
```

```
torch.Size([6, 43])
```

So we lost most of the particles in this case. To get an idea of where they were lost, we can instruct the tracking method to record the loss:

[3]:
```
tracked, loss = lattice.linear(particles, aperture=True, recloss=True)
print(tracked.shape)
```

```
torch.Size([6, 43])
```

Setting `recloss=True` records the loss values at each element and adds them as a separate return value in form of a dict, mapping element labels to loss values. The loss values themselves are determined by the particular aperture type (see `elements.aperture_types`). The loss value is computed for each particle arriving at the entrance of an element. If the loss value is greater than zero the particle lost, otherwise it is tracked further. Let's see the loss values for the first ten elements:

[4]:
```
for label, loss_val in list(loss.items())[:10]:
    print(f'{label}: {len(loss_val)}')
```

```
p_0: 1000
drift_0: 1000
p_lp2end: 998
drift_1: 998
yr01lb3: 998
drift_2: 998
yr01lb4: 997
drift_3: 994
yr01df3: 989
drift_4: 989
```

That means all 1,000 particles arrived at the entrance of element `p_0` (which is a marker) and thus also arrives at element `drift_0`. Note that even though `drift_0` is a `k1 = 0` quadrupole, serving as an aperture-checked drift in MADX, the tracking here performs aperture checks also for `Drift` spaces. Since at the next marker only 998 particles arrive, this means we lost two particles at the previous element. We can confirm that by checking the loss values greater than zero:

[5]:
```
l_drift_0 = loss['drift_0']
print(l_drift_0[l_drift_0 > 0])
```

```
tensor([0.0012, 0.0029])
```

Instead of returning a loss history we can also ask for an accumulated version of the loss value. This will sum the loss values which are greater than zero at every element:

[6]:
```
tracked, loss = lattice.linear(particles, aperture=True, recloss='sum')
print(tracked.shape)
print(loss)
```

```
torch.Size([6, 43])
tensor(151.6977)
```

This is helpful for particle loss optimization because if our lattice contained optimization parameters, we could inject the corresponding gradients via `loss.backward()`.

We can also use more fine-grained control over the loss history by specifying one or more multi-element selectors that will be matched against elements (these multi-element selectors are `str`, `re.Pattern` or lattice element types).

[7]:
```
from dipas.elements import SBend

tracked, loss = lattice.linear(particles, aperture=True, recloss=SBend)
for k, v in loss.items():
    print(k, len(v))
```

```
yr01mh 961
yr02mh 351
yr03mh 107
```

(continues on next page)

(continued from previous page)

```
yr04mh 91
yr05mh 89
yr06mh 86
yr07mh 69
yr08mh 66
yr09mh 63
yr10mh 60
yr11mh 60
yr12mh 60
```

Again the lengths of the loss values indicate how many particles arrived at a particular element. Using a wildcard expression we can record the loss at all the quadrupoles for example:

[8]:
```
tracked, loss = lattice.linear(particles, aperture=True, recloss='yr*qs*')
print(len(loss))
print(set(type(lattice[label]) for label in loss))
```

```
18
{<class 'dipas.elements.Quadrupole'>}
```

The same options are available for observing particle coordinates at specific elements. For that purpose we can use the `observe` parameter. We can provide similar values as for `recloss` (except for `"sum"` which doesn't make sense here):

[9]:
```
tracked, locations = lattice.linear(particles, aperture=True, observe='yr*qs*')
print(len(locations))
print(set(type(lattice[label]) for label in locations))
```

```
18
{<class 'dipas.elements.Quadrupole'>}
```

By inspecting the shape of the corresponding position we can see how many particles were successfully tracked through an element, i.e. made it to the element's exit. This number is the number of particles that arrived at an element (the `len(loss_value)`) minus the number of particles that were lost at the element (`len(loss_value[loss_value > 0])`). The loss is computed at the entrance of an element and the coordinates are recorded at the exit of elements:

[10]:
```
tracked, locations, loss = lattice.linear(particles, aperture=True, observe='yr*qs*',␣
↪recloss='yr*qs*')
print(loss['yr02qs1'].shape[-1])
print(len(loss['yr02qs1'][loss['yr02qs1'] > 0]))
print(locations['yr02qs1'].shape[-1])
print(loss['yr02qs1'].shape[-1] - len(loss['yr02qs1'][loss['yr02qs1'] > 0]) ==␣
↪locations['yr02qs1'].shape[-1])
```

```
724
81
643
True
```

Irrespective of the tracking method used (e.g. `linear` in the above examples), drift spaces will always be tracked through by using the exact solutions to the equations of motion (referred to by the `exact` tracking method). If this behavior is undesired and drift spaces should use the specified tracking method instead of `exact` this can be done by specifying the parameter `exact_drift=False`.

[11]:
```
print(lattice.linear(particles, exact_drift=False).std(dim=1))
print(lattice.linear(particles, exact_drift=True).std(dim=1))
```

```
tensor([0.0863, 0.0151, 0.0141, 0.0089, 3.2844, 0.0103])
tensor([0.0861, 0.0151, 0.0141, 0.0089, 3.2826, 0.0103])
```

4.7 Optics calculations

Optics calculations can be performed via the `compute` module.

4.7.1 Closed orbit search

Using `compute.closed_orbit` we can perform closed orbit search for a given lattice:

[1]:
```
from importlib import resources
from dipas.build import from_file
from dipas.compute import closed_orbit, linear_closed_orbit
from dipas.elements import Kicker, HKicker, VKicker
import dipas.test.sequences

with resources.path(dipas.test.sequences, 'cryring.seq') as path:
    lattice = from_file(path)

thin = lattice.makethin({Kicker: 1})
print(closed_orbit(thin))
```

```
tensor([[0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.]])
```

As can be seen from the above example we first need to convert all elements that don't support thick tracking (the kicker magnets) to thin elements because the closed orbit search is performed by tracking the closed orbit through the lattice. Since all kickers are turned off (`kick == 0`) the closed orbit is just zero. Let's add some kicks:

[2]:
```
import random
random.seed(1)

for kicker in lattice[HKicker] + lattice[VKicker]:
    kicker.kick = random.uniform(-0.001, 0.001)

thin_with_kicks = lattice.makethin({Kicker: 1})  # Use 'makethin' again to make the␣
↪added kicks effective.
print(closed_orbit(thin_with_kicks, order=1).flatten())
print(linear_closed_orbit(thin_with_kicks).flatten())
```

```
tensor([0.0012, 0.0005, 0.0055, 0.0002, 0.0000, 0.0000])
tensor([0.0012, 0.0005, 0.0055, 0.0002, 0.0000, 0.0000])
```

One important thing to note is that we need to assign the kicks to the original lattice, since the `thin` version doesn't contain the original kickers anymore. Then for the new kicker values we need to `makethin` the `lattice` again before performing the closed orbit search. This seems somewhat repetitive but it is important in order to maintain the relation between thick and thin elements. Especially if optimization parameters are involved, it is important to always `makethin` the original lattice (which stores the optimization parameters) in order to always get the up-to-date values of the optimization parameters. `linear_closed_orbit` computes the closed orbit directly from first order transfer maps (using a slightly different algorithm).

4.7.2 Twiss computation

We can use `compute.twiss` in order to compute lattice functions as well as phase advance and tunes:

[3]:
```
import dipas.compute as compute

twiss = compute.twiss(thin)  # Returns a dict.
print(list(twiss), end='\n\n')
print('Q1:', twiss['Q1'], ', Q2:', twiss['Q2'], end='\n\n')
print('Coupling Matrix:', twiss['coupling_matrix'], sep='\n', end='\n\n')
print('One-Turn Matrix:', twiss['one_turn_matrix'], sep='\n', end='\n\n')
print('Lattice:', twiss['lattice'].columns, sep='\n')
```

```
['lattice', 'coupling_matrix', 'Q1', 'Q2', 'one_turn_matrix']

Q1: tensor(2.4200) , Q2: tensor(2.4200)

Coupling Matrix:
tensor([[0., 0.],
        [0., 0.]])

One-Turn Matrix:
tensor([[-8.7631e-01,  9.2467e-01,  0.0000e+00,  0.0000e+00,  1.6421e-17,
         -8.3255e+00],
        [-2.5099e-01, -8.7631e-01,  0.0000e+00,  0.0000e+00, -2.9032e-17,
         -1.1137e+00],
        [ 0.0000e+00,  0.0000e+00, -8.7631e-01,  1.0994e+00,  0.0000e+00,
          0.0000e+00],
        [ 0.0000e+00,  0.0000e+00, -2.1111e-01, -8.7631e-01,  0.0000e+00,
          0.0000e+00],
        [ 1.1137e+00,  8.3255e+00,  0.0000e+00,  0.0000e+00,  1.0000e+00,
          3.1825e+02],
        [ 5.8220e-18, -1.3981e-17,  0.0000e+00,  0.0000e+00,  7.7534e-35,
          1.0000e+00]])

Lattice:
Index(['x', 'px', 'y', 'py', 'bx', 'ax', 'mx', 'by', 'ay', 'my', 'dx', 'dpx',
       'dy', 'dpy'],
      dtype='object')
```

Here `twiss['lattice']` is a `pandas.DataFrame` with element labels as index and lattice functions as columns.

4.7.3 Transfer maps

By using `compute.transfer_maps` we can compute the transfer maps along the lattice. The parameter `method` let's us specify how the transfer maps are computed. The following options are available:

- `method='local'`: Compute the local maps of elements, including closed orbit contribution.
- `method='accumulate'`: Compute the cumulative transfer maps w.r.t. to the start of the lattice.
- `method='reduce'`: Compute the combined transfer map for the whole segment.

[4]:
```
from dipas.compute import transfer_maps

maps = dict(transfer_maps(thin, method='accumulate', labels=True))
print(*maps[lattice[HKicker, 0].label], sep='\n')
```

```
tensor([[0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.]])
tensor([[1.0000, 0.8328, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 1.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 1.0000, 0.8328, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 1.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 1.0000, 6.1278],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000]])
tensor([[[  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,  -1.2038],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,  -1.2038,   0.0000,   0.0000,   0.0000,   0.0000]],

        [[  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000]],

        [[  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,  -1.2038],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,  -1.2038,   0.0000,   0.0000]],

        [[  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000]],

        [[  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,  -1.2038,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,  -1.2038,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000, -26.5733]],

        [[  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000]]])
```

Since the kicker represents the full element as a sequence of drift spaces and thin slices, its first order map (i.e. the transfer matrix) is not the identity matrix.

If we want the element local maps instead we can use `method='local'` (here `0.3755` is the length of the kicker):

[5]:
```
maps = dict(transfer_maps(thin, method='local', labels=True))
print('Length of Kicker:', lattice[HKicker, 0].l)
print('Transfer Map:', *maps[lattice[HKicker, 0].label], sep='\n')
```

```
Length of Kicker: tensor(0.3755)
Transfer Map:
tensor([[0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.]])
tensor([[1.0000, 0.3755, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 1.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 1.0000, 0.3755, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 1.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.0000, 1.0000, 2.7629],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000]])
tensor([[[  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,  -0.5428],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,  -0.5428,   0.0000,   0.0000,   0.0000,   0.0000]],

        [[  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000]],

        [[  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,  -0.5428],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,  -0.5428,   0.0000,   0.0000]],

        [[  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000]],

        [[  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,  -0.5428,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,  -0.5428,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000, -11.9816]],

        [[  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000],
```

(continues on next page)

(continued from previous page)

```
         [  0.0000,   0.0000,   0.0000,   0.0000,   0.0000,   0.0000]]])
```

4.7.4 Orbit Response Matrix

Using `compute.orm` we can compute the orbit response matrix for a given lattice. We need to specify the kickers and monitors to be used, which can be done in a similar way as for selecting lattice elements in general: either we can specify an identifier that selects multiple elements directly, such as a lattice element type or a regex, or we can specify a list of single element identifiers, such as unambiguous labels for example. Let's compute the horizontal ORM for one of the example lattices:

[6]:
```
from importlib import resources
from dipas.build import from_file
from dipas.compute import orm
from dipas.elements import HKicker, HMonitor
import dipas.test.sequences

with resources.path(dipas.test.sequences, 'cryring.seq') as path:
    lattice = from_file(path)

orm_x, orm_y = orm(lattice, kickers=HKicker, monitors=HMonitor)
```

Here we don't need to call `makethin` beforehand because this will be done inside the `orm` function. This is necessary because the `orm` function will temporarily vary the kicker strengths and, as explained above, for each change to the original lattice we need to create a new thin version (i.e. changes to the original lattice are *not* automatically mapped to any thin versions that have been created before).

[7]:
```
print('ORM.shape: ', orm_x.shape)
print('Number of HKickers and HMonitors: ', (len(lattice[HKicker]),␣
↪len(lattice[HMonitor])), end='\n\n')
print('ORM-X\n\n', orm_x, end='\n\n')
print('ORM-Y\n\n', orm_y)
```

```
ORM.shape:  torch.Size([12, 9])
Number of HKickers and HMonitors:  (12, 9)

ORM-X

 tensor([[ 1.6402,  0.8140,  0.4583, -0.3921,  1.1925,  1.4039, -1.9135, -0.4376,
           1.7387],
         [ 1.5428,  1.0098,  0.6885, -0.8058,  1.3288,  1.7722, -2.1046, -0.6838,
           1.6842],
         [ 0.8887,  1.7728,  1.6933, -2.7059,  1.7242,  3.2348, -2.6066, -1.7705,
           1.0460],
         [ 2.0276,  0.9366,  1.2629, -2.4109,  0.4483,  1.8029, -0.5499, -1.3686,
          -0.8998],
         [ 1.6111,  1.2233,  0.8969, -0.0308, -1.3899, -1.3615,  2.2668,  0.2506,
          -2.3613],
         [ 1.0921,  1.8082,  1.6400,  1.2010, -2.0740, -2.7341,  3.2892,  1.0372,
          -2.6678],
         [-2.9513, -1.3954, -0.7518,  1.1115,  1.5688,  0.9301, -2.6391,  0.2900,
           3.4040],
         [ 3.2348,  0.5891, -0.1673,  1.0460,  1.4899,  0.8887,  1.2091, -1.1510,
          -2.7059],
         [-2.7059,  0.4745,  1.1597, -2.6066, -0.4258,  1.0460,  0.8887,  1.8197,
           1.2091],
```

(continues on next page)

(continued from previous page)

```
        [ 1.2091, -1.3684, -1.7372,  3.2348, -0.7906, -2.6066,  1.0460,  1.8726,
          0.8887],
        [ 2.1282, -0.1676, -0.6953,  1.6820,  0.5093, -0.4424, -0.9556,  0.7970,
          2.0117],
        [ 2.0422,  0.0053, -0.4921,  1.3166,  0.6296, -0.1171, -1.1243,  0.5795,
          1.9636]])

ORM-Y

 tensor([[0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

4.8 Serializing lattices

We can serialize lattices into MADX scripts using the following functions from the `build` module:

- `create_script` - Creates a full MADX script including beam command and optionally error definitions as well as particle tracking.
- `sequence_script` - Serializes a lattice into a corresponding `SEQUENCE; ENDSEQUENCE;` block.
- `track_script` - Serializes particle coordinates, plus some additional configuration, into a corrsponding `TRACK; ENDTRACK;` block.
- `error_script` - Parses error definitions from a given lattice and serializes them into a list of `SELECT` and `EALIGN` statements.

For example:

[1]:
```
from dipas.build import Lattice, create_script, sequence_script, track_script, error_
↪script
from dipas.elements import Segment

with Lattice(dict(particle='proton', gamma=1.25)) as lattice:
    lattice.Quadrupole(k1=0.125, l=1, label='qf')
    lattice.SBend(angle=0.05, l=6, label='s1')
    lattice.Quadrupole(k1=-0.125, l=1, label='qd')
    lattice.SBend(angle=0.05, l=6, label='s2')

lattice = Segment(lattice)
print(sequence_script(lattice))
```

```
seq: sequence, l = 14.0, refer = entry;
    qf: quadrupole, k1 = 0.125, l = 1.0, at = 0.0;
    s1: sbend, angle = 0.05, e1 = 0.0, e2 = 0.0, fint = 0.0, fintx = 0.0, h1 = 0.0,␣
↪h2 = 0.0, hgap = 0.0, l = 6.0, at = 1.0;
```

(continues on next page)

(continued from previous page)

```
    qd: quadrupole, k1 = -0.125, l = 1.0, at = 7.0;
    s2: sbend, angle = 0.05, e1 = 0.0, e2 = 0.0, fint = 0.0, fintx = 0.0, h1 = 0.0,␣
↪h2 = 0.0, hgap = 0.0, l = 6.0, at = 8.0;
endsequence;
```

Now let's create the `TRACK` block:

[2]:
```
import torch

particles = torch.rand(6, 10)
print(track_script(particles, observe=['qf', 'qd'], aperture=True, recloss=True,␣
↪turns=1, maxaper=[1]*6))
```

```
track, aperture = true, recloss = true, onepass = true, dump = true, onetable = true;
    start, x = 0.2014635703501506, px = 0.2662095882206157, y = 0.8166112346953612,␣
↪py = 0.23180824594831628, t = 0.39202893025542407, pt = 0.5399201490740829;
    start, x = 0.14669278350197978, px = 0.24835279589455972, y = 0.27761962358694936,
↪ py = 0.8467406897590475, t = 0.9640333584721231, pt = 0.2787731815670358;
    start, x = 0.40343846523065574, px = 0.8326952974366683, y = 0.30562559574588954,␣
↪py = 0.6813336597554878, t = 0.03341727991931742, pt = 0.830850622929301;
    start, x = 0.5470678008425934, px = 0.7779247137419353, y = 0.9958333764630148,␣
↪py = 0.29537177766723244, t = 0.11764775607056943, pt = 0.3208520680352198;
    start, x = 0.1049864797528981, px = 0.3752285141312387, y = 0.7203908988777403,␣
↪py = 0.63130467146256, t = 0.9473415670404751, pt = 0.10299125216411009;
    start, x = 0.39960444505659465, px = 0.3735462666225813, y = 0.7160529579941585,␣
↪py = 0.1100296996131459, t = 0.6683040540297257, pt = 0.5498750540523619;
    start, x = 0.0228851551084277, px = 0.876894016227671, y = 0.9587570774187312, py␣
↪= 0.6097960547963239, t = 0.833647004978768, pt = 0.6247855455798537;
    start, x = 0.9209046996988439, px = 0.11623483047061944, y = 0.8216139551556022,␣
↪py = 0.6395245166612197, t = 0.15343851365337768, pt = 0.9395649827597847;
    start, x = 0.20836412057646447, px = 0.32546809192785775, y = 0.7780847178261877,␣
↪py = 0.9400753865683356, t = 0.35463141128309483, pt = 0.10358923206153958;
    start, x = 0.4458752860950078, px = 0.19209207672465944, y = 0.43257782895074914,␣
↪py = 0.027717813984675876, t = 0.5167697678230143, pt = 0.4922091373466405;

    observe, place = qf;
    observe, place = qd;

    run, turns = 1, maxaper = {1, 1, 1, 1, 1, 1};

    write, table = trackloss, file;
endtrack;
```

Let's introduce some alignment errors to the defocusing quadrupole:

[3]:
```
from dipas.elements import LongitudinalRoll, Offset, Tilt

lattice['qd'] = Tilt(lattice['qd'], psi=0.78)  # Technically this is not an alignment␣
↪error, but it does modify the element.
lattice['qd'] = LongitudinalRoll(lattice['qd'], psi=0.35)
lattice['qd'] = Offset(lattice['qd'], dx=0.01, dy=0.02)

print(sequence_script(lattice))
```

```
seq: sequence, l = 14.0, refer = entry;
    qf: quadrupole, k1 = 0.125, l = 1.0, at = 0.0;
    s1: sbend, angle = 0.05, e1 = 0.0, e2 = 0.0, fint = 0.0, fintx = 0.0, h1 = 0.0,␣
↪h2 = 0.0, hgap = 0.0, l = 6.0, at = 1.0;
```

(continues on next page)

(continued from previous page)

```
    qd: quadrupole, k1 = -0.125, l = 1.0, tilt = 0.78, at = 7.0;
    s2: sbend, angle = 0.05, e1 = 0.0, e2 = 0.0, fint = 0.0, fintx = 0.0, h1 = 0.0,␣
↪h2 = 0.0, hgap = 0.0, l = 6.0, at = 8.0;
endsequence;
```

Here we can see that the output from `sequence_script` now contains the `tilt` for the `"qd"` quadrupole and the alignment errors are summarized and assigned in the part coming from `error_script`.

Now let's build the complete MADX script:

[4]:
```
print(create_script(
    dict(particle='proton', gamma=1.25),
    sequence=lattice,
    errors=True,  # Extracts the errors from the provided `sequence`.
    track=track_script(particles, ['qf', 'qd'])
))
```

```
beam, particle = proton, gamma = 1.25;

seq: sequence, l = 14.0, refer = entry;
    qf: quadrupole, k1 = 0.125, l = 1.0, at = 0.0;
    s1: sbend, angle = 0.05, e1 = 0.0, e2 = 0.0, fint = 0.0, fintx = 0.0, h1 = 0.0,␣
↪h2 = 0.0, hgap = 0.0, l = 6.0, at = 1.0;
    qd: quadrupole, k1 = -0.125, l = 1.0, tilt = 0.78, at = 7.0;
    s2: sbend, angle = 0.05, e1 = 0.0, e2 = 0.0, fint = 0.0, fintx = 0.0, h1 = 0.0,␣
↪h2 = 0.0, hgap = 0.0, l = 6.0, at = 8.0;
endsequence;

use, sequence = seq;

eoption, add = true;
select, flag = error, clear = true;
select, flag = error, range = "qd";
ealign, dx = 0.01, dy = 0.02;
ealign, dpsi = 0.35;

track, aperture = true, recloss = true, onepass = true, dump = true, onetable = true;
    start, x = 0.2014635703501506, px = 0.2662095882206157, y = 0.8166112346953612,␣
↪py = 0.23180824594831628, t = 0.39202893025542407, pt = 0.5399201490740829;
    start, x = 0.14669278350197978, px = 0.24835279589455972, y = 0.27761962358694936,
↪ py = 0.8467406897590475, t = 0.9640333584721231, pt = 0.2787731815670358;
    start, x = 0.40343846523065574, px = 0.8326952974366683, y = 0.30562559574588954,␣
↪py = 0.6813336597554878, t = 0.03341727991931742, pt = 0.830850622929301;
    start, x = 0.5470678008425934, px = 0.7779247137419353, y = 0.9958333764630148,␣
↪py = 0.29537177766723244, t = 0.11764775607056943, pt = 0.3208520680352198;
    start, x = 0.1049864797528981, px = 0.3752285141312387, y = 0.7203908988777403,␣
↪py = 0.63130467146256, t = 0.9473415670404751, pt = 0.10299125216411009;
    start, x = 0.39960444505659465, px = 0.3735462666225813, y = 0.7160529579941585,␣
↪py = 0.1100296996131459, t = 0.6683040540297257, pt = 0.5498750540523619;
    start, x = 0.0228851551084277, px = 0.876894016227671, y = 0.9587570774187312, py␣
↪= 0.6097960547963239, t = 0.833647004978768, pt = 0.6247855455798537;
    start, x = 0.9209046996988439, px = 0.11623483047061944, y = 0.8216139551556022,␣
↪py = 0.6395245166612197, t = 0.15343851365337768, pt = 0.9395649827597847;
    start, x = 0.20836412057646447, px = 0.32546809192785775, y = 0.7780847178261877,␣
↪py = 0.9400753865683356, t = 0.35463141128309483, pt = 0.10358923206153958;
    start, x = 0.4458752860950078, px = 0.19209207672465944, y = 0.43257782895074914,␣
↪py = 0.027717813984675876, t = 0.5167697678230143, pt = 0.4922091373466405;
```

(continues on next page)

(continued from previous page)

```
    observe, place = qf;
    observe, place = qd;

    run, turns = 1, maxaper = {0.1, 0.01, 0.1, 0.01, 1.0, 0.1};

    write, table = trackloss, file;
endtrack;
```

In case we wanted to add optics calculations via `TWISS` we can just append the relevant command manually:

[5]:
```
script = create_script(dict(particle='proton', gamma=1.25), sequence=sequence_
↪script(lattice))
script += '\nselect, flag = twiss, full;\ntwiss, save, file = "twiss";'
print(script)
```

```
beam, particle = proton, gamma = 1.25;

seq: sequence, l = 14.0, refer = entry;
    qf: quadrupole, k1 = 0.125, l = 1.0, at = 0.0;
    s1: sbend, angle = 0.05, e1 = 0.0, e2 = 0.0, fint = 0.0, fintx = 0.0, h1 = 0.0,␣
↪h2 = 0.0, hgap = 0.0, l = 6.0, at = 1.0;
    qd: quadrupole, k1 = -0.125, l = 1.0, tilt = 0.78, at = 7.0;
    s2: sbend, angle = 0.05, e1 = 0.0, e2 = 0.0, fint = 0.0, fintx = 0.0, h1 = 0.0,␣
↪h2 = 0.0, hgap = 0.0, l = 6.0, at = 8.0;
endsequence;

use, sequence = seq;
select, flag = twiss, full;
twiss, save, file = "twiss";
```

4.9 MADX utilities

Several utilities for interfacing with the MADX program exist in the module `madx.utils`. The following functions can be used to run MADX scripts:

- `run_file` - Runs a bunch of dependent MADX script files together with possibilities for further configuration (see the API docs for more details).
- `run_script` - Runs a single script with the possibility for further configuration (see the API docs for more details).
- `run_orm` - Compute the orbit response matrix for a given MADX sequence definition.

The resulting files are returned as pandas data frames by default but the functions can be configured to return the raw file contents instead. The stdout and stderr is returned as well.

These functions can also be imported from the `dipas.madx` subpackage directly.

4.9.1 Running MADX scripts

For example we can run `TWISS` computations on one of the example scripts:

[1]:
```
from importlib import resources
import os.path
from dipas.madx import run_file, run_script, run_orm
import dipas.test.sequences

with resources.path(dipas.test.sequences, 'cryring.seq') as path:
    result = run_file(path, madx=os.path.expanduser('~/bin/madx'))

print(list(result))
print(result['stderr'])
```

```
['stdout', 'stderr']
```

`result['stdout']` contains the whole echo from the script, which is pretty long, so we won't print it here. The empty list which was passed to `run_file` is a list of resulting files that should be retrieved, but since that example script just contained the sequence definition, no files were created anyway. Let's change that and also remove the echo:

[2]:
```
script = resources.read_text(dipas.test.sequences, 'cryring.seq')
script = 'option, -echo;\n' + script
script += 'twiss, save, file = "twiss";'
result = run_script(script, ['twiss'], madx=os.path.expanduser('~/bin/madx'))
```

Here we added a `TWISS` command to the script, which generates the file "twiss" which we then specified as a result in the call to `run_script`. Let's check the `result` now:

[3]:
```
print(list(result))
print(type(result['twiss']))
print(result['twiss'].columns)
```

```
['stdout', 'stderr', 'twiss']
<class 'pandas.core.frame.DataFrame'>
Index(['NAME', 'KEYWORD', 'S', 'BETX', 'ALFX', 'MUX', 'BETY', 'ALFY', 'MUY',
       'X',
       ...
       'SIG54', 'SIG55', 'SIG56', 'SIG61', 'SIG62', 'SIG63', 'SIG64', 'SIG65',
       'SIG66', 'N1'],
      dtype='object', length=256)
```

The `result['twiss']` is a pandas data frame, containing exactly the information from the generates "twiss" file.

Instead of adding the `twiss` command manually we can also specify the keyword argument `twiss=True` for `run_script` which will take care of the necessary actions (+ add `'twiss'` as a requested result).

[4]:
```
with resources.path(dipas.test.sequences, 'cryring.seq') as path:
    result = run_file(path, twiss=True, madx=os.path.expanduser('~/bin/madx'))

print(list(result))
print(type(result['twiss']))
```

```
['stdout', 'stderr', 'twiss']
<class 'tuple'>
```

Alternatively we could also provide a `dict` for the `twiss` keyword argument in order to specify the various arguments for the `twiss` command.

4.9.2 Retrieving meta data from output files

If we wanted the meta information, at the beginning of the "twiss" file and prefixed by "@", as well we can specify the resulting files that we want to retrieve as a dict instead: keys are file names and values are bools, indicating whether we want the meta information for that particular file or not:

[5]:
```
result = run_script(script, {'twiss': True}, madx=os.path.expanduser('~/bin/madx'))
print(type(result['twiss']))
print(type(result['twiss'][0]))
print(type(result['twiss'][1]))
```

```
<class 'tuple'>
<class 'pandas.core.frame.DataFrame'>
<class 'dict'>
```

There also exists a shorthand syntax for requesting meta data, namely specifying `'<filename>+meta'` in the results list:

[6]:
```
result = run_script(script, ['twiss+meta'], madx=os.path.expanduser('~/bin/madx'))
print(type(result['twiss']))
```

```
<class 'tuple'>
```

[7]:
```
from pprint import pprint

pprint(result['twiss'][1])
```

```
{'ALFA': 0.1884508489,
 'BCURRENT': 0.0,
 'BETXMAX': 7.14827132,
 'BETYMAX': 7.958073519,
 'BV_FLAG': 1.0,
 'CHARGE': 1.0,
 'DATE': '15/04/20',
 'DELTAP': 0.0,
 'DQ1': -4.394427712,
 'DQ2': -10.92147539,
 'DXMAX': 5.852905623,
 'DXRMS': 4.993097628,
 'DYMAX': 0.0,
 'DYRMS': 0.0,
 'ENERGY': 1.0,
 'ET': 0.001,
 'EX': 1.0,
 'EY': 1.0,
 'GAMMA': 1.065788933,
 'GAMMATR': 2.303567544,
 'KBUNCH': 1.0,
 'LENGTH': 54.17782237,
 'MASS': 0.9382720813,
 'NAME': 'TWISS',
 'NPART': 0.0,
 'ORBIT5': -0.0,
 'ORIGIN': '5.05.02 Linux 64',
 'PARTICLE': 'PROTON',
 'PC': 0.3458981085,
 'Q1': 2.42,
 'Q2': 2.419999999,
```

(continues on next page)

(continued from previous page)

```
 'SEQUENCE': 'CRYRING',
 'SIGE': 0.001,
 'SIGT': 1.0,
 'SYNCH_1': 0.0,
 'SYNCH_2': 0.0,
 'SYNCH_3': 0.0,
 'SYNCH_4': 0.0,
 'SYNCH_5': 0.0,
 'TIME': '23.15.29',
 'TITLE': 'no-title',
 'TYPE': 'TWISS',
 'XCOMAX': 0.0,
 'XCORMS': 0.0,
 'YCOMAX': 0.0,
 'YCORMS': 0.0}
```

If meta information is requested, the result is a tuple containing the actual file content as a data frame and the meta data as a dict.

Requested output files will be converted according to their suffix (if present) or if they have a special file name. File names ending in `"one"` are assumed to have emerged from `TRACKONE` and are parsed accordingly. File names ending in `.madx` or `.seq` are assumed to be MADX files and parsed versions are returned (see `madx.parser.parse_file`). Otherwise it is attempted to convert the file from TFS format to a `pandas.DataFrame`. If this fails, then the raw file content is returned as a string. We can also request explicitly that a file should be treated according to a given format via `<filename>;tfs`. Here the file format is specified after a semicolon. The following formats are available:

- `madx`: Will be parsed according to `madx.parser.parse_file`.
- `raw`: Returns the raw file content as a string.
- `tfs`: Converts from TFS format to `pandas.DataFrame`.
- `trackone`: Converts from special TRACKONE-TFS format to `pandas.DataFrame`.

For more details see `madx.utils.convert`.

[8]:
```
result = run_script(script, ['twiss;raw'], madx=os.path.expanduser('~/bin/madx'))
print(type(result['twiss']), end='\n\n')
pprint(result['twiss'].splitlines()[:5])
```

```
<class 'str'>

['@ NAME             %05s "TWISS"',
 '@ TYPE             %05s "TWISS"',
 '@ SEQUENCE         %07s "CRYRING"',
 '@ PARTICLE         %06s "PROTON"',
 '@ MASS             %le        0.9382720813']
```

4.9.3 Configure scripts before running

Now let's inspect the resulting data frame, for example the orbit:

[9]:
```
result = run_script(script, ['twiss'], madx=os.path.expanduser('~/bin/madx'))
twiss = result['twiss']
print(twiss[['X', 'Y']].describe())
```

```
           X      Y
count  184.0  184.0
mean     0.0    0.0
std      0.0    0.0
min      0.0    0.0
25%      0.0    0.0
50%      0.0    0.0
75%      0.0    0.0
max      0.0    0.0
```

Since the lattice does not contain any zeroth order kicks the orbit is just zero. We can change that by modifying (configuring) the script while we run it. For that purpose we can use the `variables` parameter. This parameter allows for replacing in variable definition of the form `name :?= value;` the `value` with a `new_value`.

[10]:
```
result = run_script(script, ['twiss'], variables={'k02kh': 0.005}, madx=os.path.
↪expanduser('~/bin/madx'))
print(result['twiss'][['X', 'Y']].describe())
```

```
                X      Y
count  184.000000  184.0
mean     0.002166    0.0
std      0.009515    0.0
min     -0.016486    0.0
25%     -0.008191    0.0
50%      0.005374    0.0
75%      0.008827    0.0
max      0.016840    0.0
```

Since we configured one of the horizontal kickers to have a non-zero kick strength the horizontal orbit changed but the vertical orbit remained zero (no coupling in the lattice).

4.9.4 Compute the Orbit Response Matrix

Using the `madx.utils.run_orm` function we can compute the orbit response matrix for the given lattice, by specifying a list of kicker and monitor labels. The ORM will be computed using these kickers and measured at these monitors:

[11]:
```
kickers = ['yr04kh', 'yr06kh', 'yr08kh', 'yr10kh']
monitors = ['yr02dx1', 'yr03dx1', 'yr03dx4']
orm = run_orm(script, kickers=kickers, monitors=monitors, madx=os.path.expanduser('~/
↪bin/madx'))
print(orm, end='\n\n')
print(orm['X'], end='\n\n')
```

```
               X                           Y
         yr02dx1   yr03dx1   yr03dx4 yr02dx1 yr03dx1 yr03dx4
yr04kh  1.092090  1.808234  1.640022     0.0     0.0     0.0
yr06kh -2.951312 -1.395442 -0.751784     0.0     0.0     0.0
yr08kh  3.234832  0.589074 -0.167347     0.0     0.0     0.0
yr10kh -2.705921  0.474549  1.159677     0.0     0.0     0.0

         yr02dx1   yr03dx1   yr03dx4
yr04kh  1.092090  1.808234  1.640022
yr06kh -2.951312 -1.395442 -0.751784
yr08kh  3.234832  0.589074 -0.167347
yr10kh -2.705921  0.474549  1.159677
```

Since we chose only horizontal kickers, the vertical ORM is zero (no coupling).

4.10 Visualizing lattices

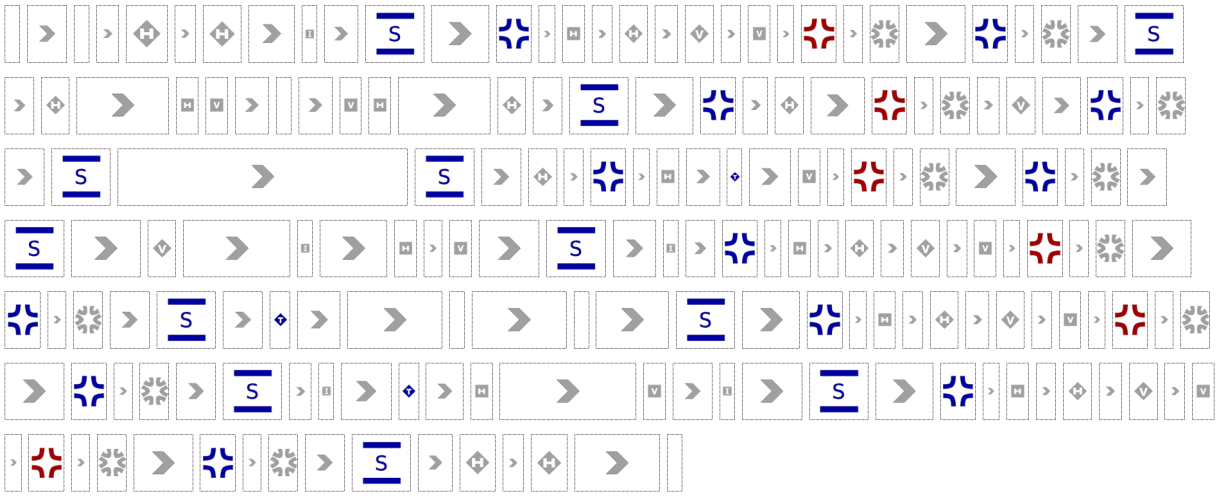
Lattices can be visualized by serializing them into an HTML file. This can be done also via *build.sequence_script* by supplying the argument `markup='html'`. The resulting HTML sequence file can be viewed in any modern browser and elements can be inspected by using the browser's inspector tool (e.g. `<ctrl> + <shift> + C` for Firefox).

Let's visualize one of the example sequences:

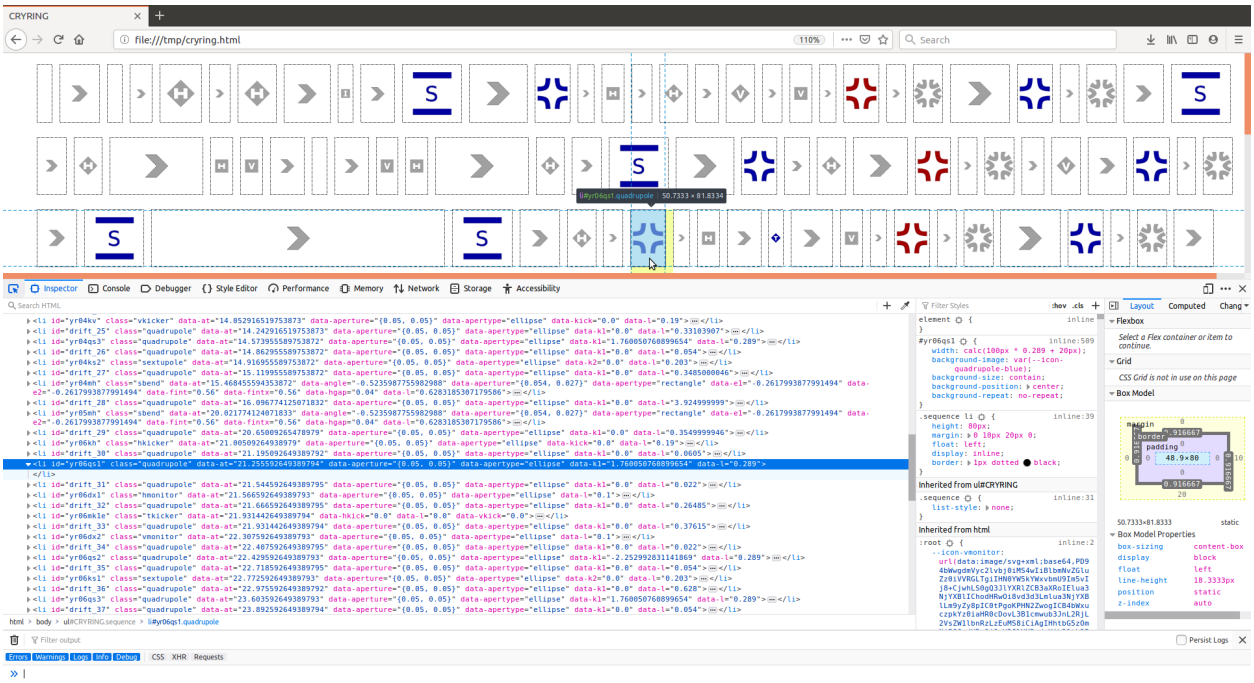
```
>>> from importlib import resources
>>> from dipas.build import from_file, sequence_script
>>> import dipas.test.sequences
>>>
>>> with resources.path(dipas.test.sequences, 'cryring.seq') as path:
...     lattice = from_file(path)
...
>>> with open('cryring.html', 'w') as fh:
...     fh.write(sequence_script(lattice, markup='html'))
...
```

Note: The same result can also be obtained by using the `madx-to-html` command line utility that ships with the *dipas* package. Just run `madx-to-html cryring.seq` to generate a corresponding `cryring.html` file. Running `madx-to-html --help` shows all options for the utility.

This produces the following HTML page, as viewed from the browser:



Using the browser's inspector tool we can inspect the elements and view their attributes:



Legend: The following information is encoded in the visualization:

- Drift spaces are displayed as >,
- Kickers are displayed as diamonds; the letters “H”, “V”, “T” indicate the type of kicker (no letter indicates a bare *KICKER*).
- *RBend*, *SBend*, *Quadrupole* and *Sextupole* are displayed by their number of coils; for *RBend* and *SBend* those are displayed as two horizontal bars, the letters “R”, “S” indicate the type of magnet.
- Monitors and Instruments are displayed as rectangles,
- *HKicker*, *VKicker*, *Quadrupole*, *Sextupole*, *RBend* and *SBend* elements are displayed in blue if their particular main multipole component (*hkick*, *vkick*, *k1*, *k2*, *angle* and *angle*) has a positive sign (except for *RBend* and *SBend*, where the sign is inverted, because a positive *angle* bends towards negative x-direction), are displayed in red if that component is negative (with the exception of *RBend* and *SBend* again) and are displayed in grey if that component is zero. A further exception are quadrupoles with `k1 == 0` which are displayed as drift spaces (>).
- *Kicker* and *TKicker* elements are always displayed in blue,
- Elements that do not actively influence the trajectory of the beam are displayed in grey (such as monitors, instruments),
- Placeholders are displayed as drift spaces,
- Markers are displayed as blank elements.

CHAPTER

FIVE

EXAMPLES

The following examples show the setup and implementation of various use cases with the DiPAS package.

The relevant example and code files as well as the complete walkthrough can be found at the repository.

5.1 Inverse Modeling of Quadrupole Gradient Errors by Matching the Orbit Response Matrix

This example introduces errors to quadrupole gradient strengths and the goal of the differentiable simulation is to infer these errors by matching the Orbit Response Matrix (ORM) was well as the tunes of the resulting lattice. MADX simulations are used to provide the reference data corresponding to the lattice with errors.

Running the example script will perform the following steps:

1. Define the lattice
2. Assign a random field error to the third quadrupole of each triplet: `file = "errors"`
3. Compute Twiss of the sequence with errors: `file = "twiss"`

Here we only assign one error per triplet since the magnets that form a triplet are located very close together. For that reason the compensation of neighboring field errors is quite effective which considerably slows down the convergence of the optimization process. Assigning one error per triplet is equivalent to having only one free variable per triplet (e.g. if all magnets shared the same power supply).

[1]:
```
import os.path
from dipas.madx import run_file

result = run_file('example.madx', results=['twiss+meta', 'errors'],
                  madx=os.path.expanduser('~/bin/madx'))

twiss_ref = result['twiss']
errors = result['errors']

twiss_ref[0].set_index('NAME', inplace=True)  # [0] is the twiss data, [1] is the␣
↪meta data ("@"-prefixed in the TFS file)
errors.set_index('NAME', inplace=True)
```

Let's check the K1L values and associated errors for all magnets. As mentioned above, only the third magnet in each triplet (`*QS3`) has been assigned an error:

[2]:
```
import pandas as pd

k1_values = pd.DataFrame({
```

(continues on next page)

(continued from previous page)

```
    'K1L':  twiss_ref[0]['K1L'].loc[errors.index],
    'Errors': errors['K1L'],
})
print(k1_values)
```

```
              K1L    Errors
NAME
YR02QS1  0.508655  0.000000
YR02QS2 -0.651115  0.000000
YR02QS3  0.508655  0.011836
YR04QS1  0.508655  0.000000
YR04QS2 -0.651115  0.000000
YR04QS3  0.508655  0.011628
YR06QS1  0.508655  0.000000
YR06QS2 -0.651115  0.000000
YR06QS3  0.508655 -0.008112
YR08QS1  0.508655  0.000000
YR08QS2 -0.651115  0.000000
YR08QS3  0.508655 -0.003652
YR10QS1  0.508655  0.000000
YR10QS2 -0.651115  0.000000
YR10QS3  0.508655  0.011761
YR12QS1  0.508655  0.000000
YR12QS2 -0.651115  0.000000
YR12QS3  0.508655  0.006400
```

Now we load the lattice from the MADX file and declare the relevant quadrupole's k1-errors as optimization parameters in order to infer the actual values:

[3]:
```
from dipas.build import from_file
from dipas.elements import Quadrupole
import torch

lattice = from_file('example.madx', errors=False)  # use `errors=False` to load the␣
↪nominal optics
for quad in lattice['yr*qs3']:
    quad.dk1 = torch.nn.Parameter(quad.dk1)
    quad.update_transfer_map()  # make changes to `dk1` effective
print('# parameters: ', len(list(lattice.parameters())))
```

```
# parameters:  6
```

With the utility function `dipas.madx.run_orm` we can have MADX compute the Orbit Response Matrix for the given script file. Here we only consider the vertical component of the ORM. This will serve as the reference data against which the model will be matched.

[4]:
```
from dipas.elements import VKicker, VMonitor
from dipas.madx import run_orm

kicker_labels = [x.label for x in lattice[VKicker]]
monitor_labels = [x.label for x in lattice[VMonitor]]

orm_ref = run_orm('example.madx',
                  kickers=kicker_labels,
                  monitors=monitor_labels,
                  madx=os.path.expanduser('~/bin/madx'))
```

(continues on next page)

(continued from previous page)

```
orm_ref = orm_ref.loc[:, 'Y']  # only consider the vertical component
print(orm_ref)  # rows are kickers, columns are monitors
```

```
          yr02dx2   yr03dx2   yr03dx3   yr06dx2   yr07dx2   yr08dx2   yr10dx2  \
yr02kv  1.115240  1.983728  1.972641 -2.891744  1.823880  3.705735 -3.214685
yr04kv  0.809330  1.921067  1.954103  1.436141 -2.135845 -2.999211  3.510664
yr07kv -0.233959 -1.277466 -1.348631  2.203000  1.410669  2.204617 -2.361074
yr08kv  3.705474  0.733856  0.197256  1.247780  1.900128  1.201006  1.044550
yr10kv -3.056570  0.464584  0.998505 -3.012275 -0.807507  1.309568  1.132972
yr12kv  1.180532 -1.503768 -1.822981  3.617948 -0.628670 -3.078852  1.449289

          yr11dx2   yr12dx2
yr02kv -0.971380  1.443970
yr04kv -0.022372 -2.643008
yr07kv -0.178819  1.586732
yr08kv -2.170915 -2.916487
yr10kv  2.065269  1.186817
yr12kv  1.893791  1.050390
```

Using `dipas.compute.orm` we can compute the ORM for the given lattice, in dependency on the quadrupole gradient errors which we have previously declared as parameters:

[5]:
```
import dipas.compute as compute

orm_x, orm_y = compute.orm(lattice, kickers=VKicker, monitors=VMonitor)
orm_y = pd.DataFrame(data=orm_y.detach().numpy(), index=orm_ref.index, columns=orm_
↪ref.columns)
print(orm_y)
```

```
          yr02dx2   yr03dx2   yr03dx3   yr06dx2   yr07dx2   yr08dx2   yr10dx2  \
yr02kv  1.051808  1.992815  1.989703 -2.950579  1.775919  3.689564 -3.108786
yr04kv  0.859713  1.864128  1.881479  1.546831 -2.106485 -3.043805  3.452004
yr07kv -0.295694 -1.264483 -1.323158  2.149441  1.414799  2.254200 -2.355583
yr08kv  3.689564  0.712763  0.171017  1.415990  1.853362  1.051808  1.156167
yr10kv -2.950579  0.484488  1.006893 -3.108786 -0.747051  1.415990  1.051808
yr12kv  1.156167 -1.508437 -1.824636  3.689564 -0.626480 -3.108786  1.415990

          yr11dx2   yr12dx2
yr02kv -0.946485  1.415990
yr04kv -0.022300 -2.625416
yr07kv -0.165343  1.614370
yr08kv -2.118423 -2.950579
yr10kv  1.986812  1.156167
yr12kv  1.886741  1.051808
```

Since the above `lattice` has no gradient errors so far, the result is quite different. The goal is to align the two ORMs so that their values match.

Similarly we can compute the tunes via `dipas.compute.twiss`:

[6]:
```
from dipas.elements import Kicker

twiss = compute.twiss(lattice.makethin({Kicker: 2}, style={Kicker: 'edge'}))  # MADX␣
↪uses 'edge' style

print(f'Tunes:     Q1 = {twiss["Q1"]:.3f}, Q2 = {twiss["Q2"]:.3f}')
print(f'Reference: Q1 = {twiss_ref[1]["Q1"]:.3f}, Q2 = {twiss_ref[1]["Q2"]:.3f}')
```

```
Tunes:     Q1 = 2.420, Q2 = 2.420
Reference: Q1 = 2.439, Q2 = 2.411
```

In the following we setup and run the optimization process. For that purpose we need to define an `optimizer` as well as compute the necessary quantities during each step of the optimization.

[ ]:
```
import itertools as it
from dipas.elements import tensor

optimizer = torch.optim.Adam(lattice.parameters(), lr=1.8e-3, betas=(0.51, 0.96))

quadrupoles = lattice['yr*qs3']

Q1 = twiss_ref[1]["Q1"]
Q2 = twiss_ref[1]["Q2"]
orm_ref_y = torch.from_numpy(orm_ref.to_numpy())

cost_history = []
dk1_history = []

for step in it.count(1):
    optimizer.zero_grad()

    orm_y = compute.orm(lattice, kickers=VKicker, monitors=VMonitor)[1]
    cost1 = torch.nn.functional.mse_loss(orm_y, orm_ref_y)

    try:
        twiss = compute.twiss(lattice.makethin({Kicker: 2}, style={Kicker: 'edge'}))
    except compute.UnstableLatticeError:
        cost2 = tensor(0.)
    else:
        cost2 = (twiss['Q1'] - Q1)**2 + (twiss['Q2'] - Q2)**2

    cost = cost1 + cost2
    cost.backward(retain_graph=True)

    cost_history.append(cost.item())
    dk1_history.append([quad.dk1.item() for quad in quadrupoles])
    print(f'[Step {step:03d}] cost = {cost_history[-1]:.3e}')

    optimizer.step()

    if cost_history[-1] < 1e-12:  # if converged
        break

    for quad in quadrupoles:
        quad.update_transfer_map()  # make changes from `optimizer.step()` effective
```

We can check the k1-error values during the optimization in order to assess the convergence:

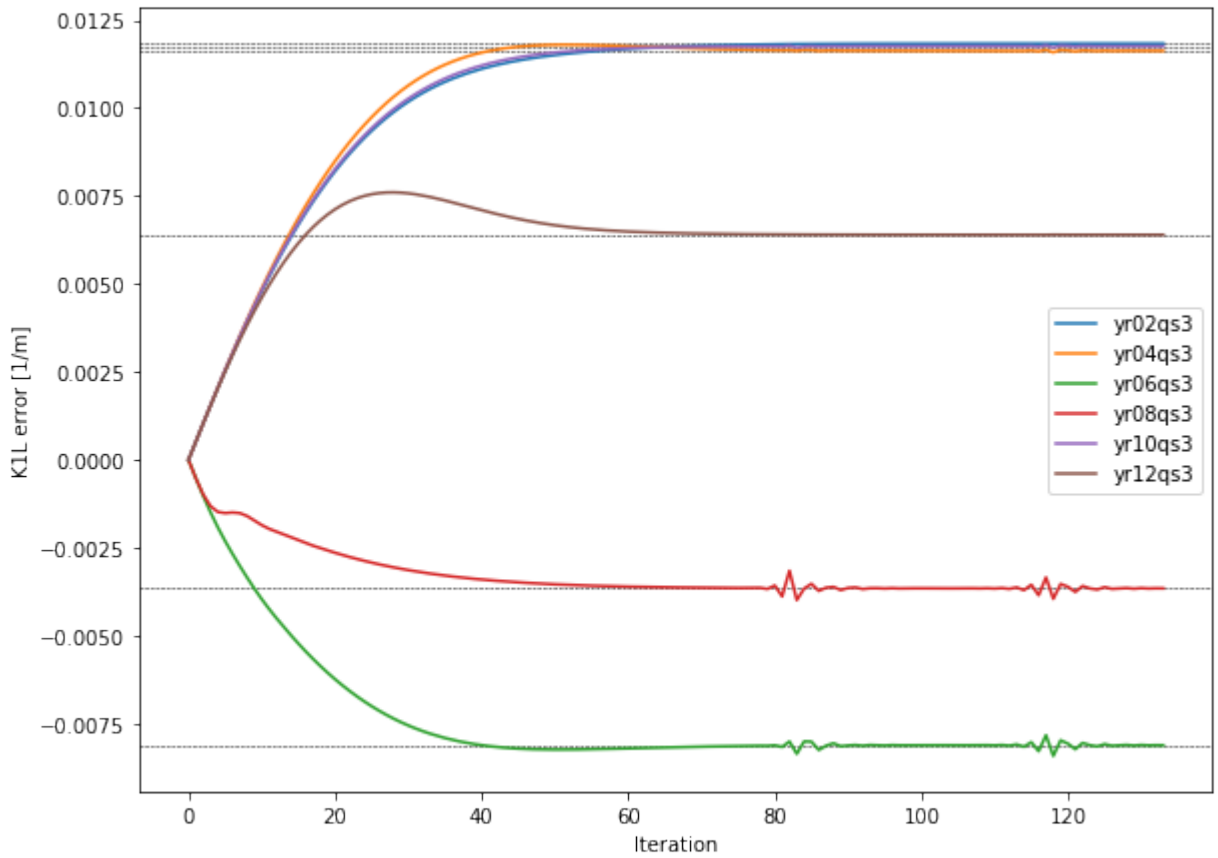
[8]:
```
%matplotlib inline

import matplotlib.pyplot as plt
import numpy as np

dk1_history = np.array(dk1_history)
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots(figsize=(9.6, 7.2))
ax.set(xlabel='Iteration', ylabel='K1L error [1/m]')
for i, quad in enumerate(quadrupoles):
    ax.plot(dk1_history[:, i]*quad.l.item(), label=quad.label)
    ax.axhline(errors.loc[quad.label.upper(), 'K1L'], lw=0.5, ls='--', color='black',␣
↪zorder=-100)
ax.legend()
```

[8]:
```
<matplotlib.legend.Legend at 0x7f84cd51f110>
```



The small wiggles towards the end of the `yr06qs3` and `yr08qs3` lines come from the particular structure of the parameter space close to the target values. The considered quadrupoles in the lattice have a certain capability to compensate each other's over- or underestimation of the true parameter values. This creates a region of strong compensation were the considered cost function (ORM + tunes) barely changes, resulting in a very slow, asymptotic convergence, as can be seen for iteration 40 or later. Perpendicular to that region however the cost increases very rapidly, so small misalignments of the optimizer momentum with respect to that region can lead to a digression from the "optimal" route, causing transverse oscillations in parameter space which are eventually damped away. Being mostly perpendicular to the direction towards the target this ususally doesn't hinder convergence. Nevertheless the convergence properties largely depend on the used optimizer and its settings, so a systematic screening of the available options is recommended.

We can also check the cost function during the optimization which reflects the above observed wiggles as well. Nevertheless, imagining a continuation of the cost trend line beyond iteration 75 arrives at approximately the same number of iterations needed to reach 10^{-12} MSE level (i.e. the wiggles don't hinder the convergence process).

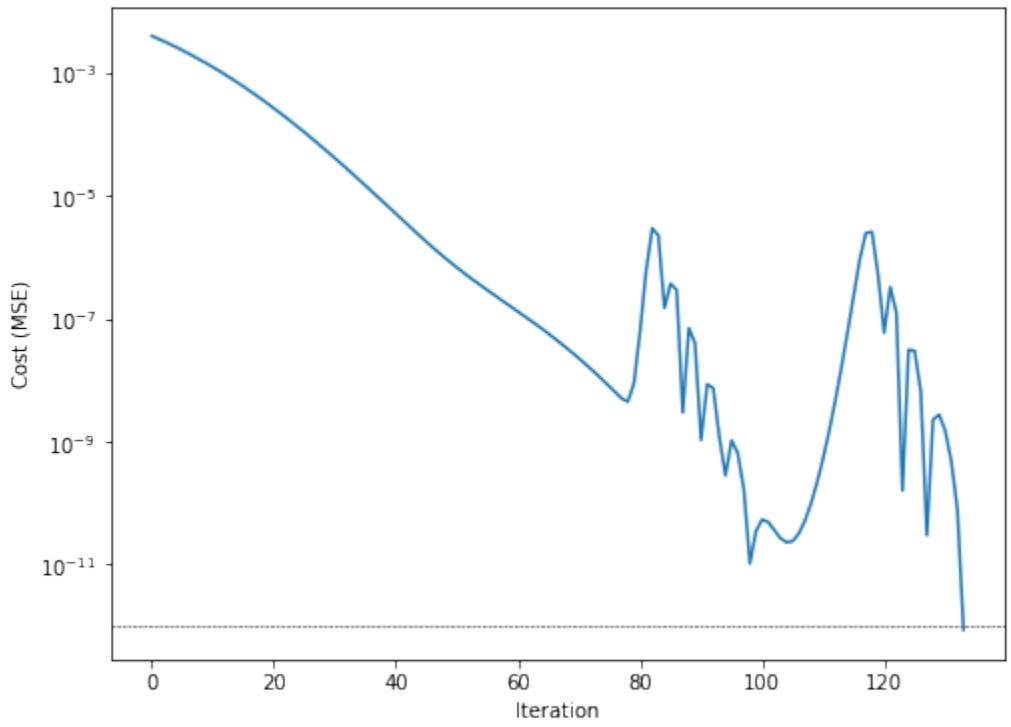
[9]:
```
fig, ax = plt.subplots(figsize=(8, 6))
```

(continues on next page)

(continued from previous page)

```
ax.set(xlabel='Iteration', ylabel='Cost (MSE)')
ax.set_yscale('log')
ax.plot(cost_history)
ax.axhline(1e-12, lw=0.5, ls='--', color='black', zorder=-100)
```

[9]:
```
<matplotlib.lines.Line2D at 0x7f84c63a3c10>
```



Finally we run a crosscheck with MADX, using the derived k1-error values, in order to confirm that the computed ORM does indeed match our computation:

[10]:
```
from dipas.build import create_script

script = create_script(
    beam=dict(particle='proton', energy=1),
    sequence=lattice,
    errors=True)

with open('crosscheck_orm.madx', 'w') as fh:
    fh.write(script)

orm_cc = run_orm('crosscheck_orm.madx',
                 kickers=kicker_labels,
                 monitors=monitor_labels,
                 madx=os.path.expanduser('~/bin/madx'))
orm_cc = orm_cc.loc[:, 'Y']

print('Deviation between computed and crosscheck ORM:', end='\n\n')
print(orm_cc - orm_ref)
```

```
Deviation between computed and crosscheck ORM:
```

(continues on next page)

(continued from previous page)

```
              yr02dx2       yr03dx2       yr03dx3       yr06dx2       yr07dx2  \
yr02kv -6.660000e-06 -2.938000e-06 -2.112000e-06  6.800000e-07 -1.357000e-06
yr04kv  4.068900e-06  1.354000e-06  8.110000e-07 -4.800000e-08  1.835000e-06
yr07kv -1.316000e-06  3.540000e-07  5.970000e-07 -1.382000e-06 -2.987000e-06
yr08kv -9.660000e-07  7.280000e-07  9.459000e-07 -1.028000e-06 -3.304000e-06
yr10kv -4.969000e-06 -3.296800e-06 -2.774300e-06  1.558000e-06  1.244100e-06
yr12kv  7.332000e-06  3.797000e-06  2.936000e-06 -1.186000e-06  1.344000e-07

              yr08dx2       yr10dx2       yr11dx2       yr12dx2
yr02kv -9.540000e-07 -4.630000e-06  3.543900e-06  7.109000e-06
yr04kv  1.531000e-06  1.902000e-06 -2.322250e-06 -3.865000e-06
yr07kv -2.521000e-06  9.580000e-07  9.788000e-07  3.590000e-07
yr08kv -3.103000e-06  1.396000e-06  7.440000e-07 -3.190000e-07
yr10kv  1.284000e-06 -5.831000e-06  2.325000e-06  6.553000e-06
yr12kv -2.680000e-07  6.340000e-06 -3.647000e-06 -8.422000e-06
```

5.2 Tune matching

In the following example some gradient errors will be introduced to quadrupoles and then the tunes will be rematched to their original values by varying the strength of horizontally focusing quadrupoles. To begin with we load the example MADX script:

[ ]:
```
with open('example.madx') as fh:
    script = fh.read()
```

As mentioned already, the script assigns gradient errors to all 18 quadrupoles and then varies the strength of the 12 horizontal quadrupoles in order to rematch the tune. Let's inspect the results by running the script via `dipas.madx.run_script`:

[2]:
```
import os
from dipas.madx import run_script

result = run_script(
    script,
    {'twiss': True, 'twiss_error': True, 'twiss_matched': True, 'errors': False},
    madx=os.path.expanduser('~/bin/madx')
)
twiss = result['twiss']
twiss_error = result['twiss_error']
twiss_matched = result['twiss_matched']
errors = result['errors']
```

```
/home/dominik/Projects/DiPAS/dipas/madx/utils.py:247: UserWarning: MADX issued the␣
↪following warnings: ['MTSIMP More variables than constraints seen. SIMPLEX may not␣
↪converge to optimal solution.']
  warnings.warn(f'MADX issued the following warnings: {warnings_list}')
```

Here we specified `True` for the twiss files since we want to retrieve the "@"-prefixed meta data besides the actual data frame (since the meta data contains the tune values). For the errors file we're not interested in meta data and so `False` will result in just the corresponding data frame. Let's inspect the tune values:

[3]:
```
print('Tune values:')
print(f' - original: {        twiss[1]["Q1"]:.3f}, {        twiss[1]["Q2"]:.3f}')
```

(continues on next page)

(continued from previous page)

```
print(f' - shifted : {  twiss_error[1]["Q1"]:.3f}, {  twiss_error[1]["Q2"]:.3f}')
print(f' - matched : {twiss_matched[1]["Q1"]:.3f}, {twiss_matched[1]["Q2"]:.3f}')
```

```
Tune values:
 - original: 2.420, 2.420
 - shifted : 2.362, 2.459
 - matched : 2.420, 2.420
```

Now let's do the same thing with gradient-based optimization via DiPAS. First we load the script via `dipas.build.from_script` and assign the errors via the `errors` data frame:

[4]:
```
from dipas.build import from_script

lattice = from_script(script, errors=errors)
```

We select the quadrupoles by checking for `k1 != 0` since the script defines drift spaces as `k1 == 0` quadrupoles:

[5]:
```
from dipas.elements import Quadrupole, Parameter

quadrupoles = [q for q in lattice[Quadrupole] if q.k1 != 0]
for q in quadrupoles:
    print(f'{q.label}, k1 = {q.k1: .6f}, dk1 = {q.dk1: .6f}')
```

```
yr02qs1, k1 =  1.760051, dk1 = -0.013478
yr02qs2, k1 = -2.252993, dk1 =  0.012333
yr02qs3, k1 =  1.760051, dk1 = -0.077924
yr04qs1, k1 =  1.760051, dk1 = -0.003707
yr04qs2, k1 = -2.252993, dk1 =  0.047466
yr04qs3, k1 =  1.760051, dk1 =  0.035394
yr06qs1, k1 =  1.760051, dk1 = -0.144814
yr06qs2, k1 = -2.252993, dk1 = -0.069407
yr06qs3, k1 =  1.760051, dk1 = -0.069233
yr08qs1, k1 =  1.760051, dk1 =  0.055320
yr08qs2, k1 = -2.252993, dk1 = -0.008849
yr08qs3, k1 =  1.760051, dk1 =  0.001484
yr10qs1, k1 =  1.760051, dk1 = -0.086816
yr10qs2, k1 = -2.252993, dk1 = -0.001718
yr10qs3, k1 =  1.760051, dk1 = -0.015357
yr12qs1, k1 =  1.760051, dk1 = -0.096350
yr12qs2, k1 = -2.252993, dk1 = -0.068815
yr12qs3, k1 =  1.760051, dk1 =  0.135457
```

For the matching we will use the horizontally focusing quadrupoles and so we'll select these and turn their `k1` attributes into parameters (being varied during the optimization). We have to call `update_transfer_map` as well in order for the change to `k1` to become effective. In general, after altering any (to-be-)parametrized attribute of a lattice element (also its value), we need to call the `update_transfer_map` method for bringing the change into effect.

[6]:
```
h_quadrupoles = [q for q in quadrupoles if q.k1 > 0]
for q in h_quadrupoles:
    q.k1 = Parameter(q.k1)
    q.update_transfer_map()
print(f'# Parameters: {len(list(lattice.parameters()))}')
```

```
# Parameters: 12
```

Next we'll prepare the optimization by creating an optimizer and defining a cost (loss) function. The cost function indicates the distance to the optimization target(s). Before we can use the lattice's transfer maps we need to convert

`Kicker` elements to thin counterparts since the transfer map for a thick kicker doesn't exist. We specify two slices placed at the edges of the original elements (this is the configuration used by MADX during TWISS computation).

We use the `dipas.compute.twiss` function for computing the tune values (alongside other lattice functions). When computing the gradients via `cost.backward` we specify `retain_graph=True` since at every iteration we're optimizing against the same data and so retaining the graph is required (e.g. the transfer map tensors of lattice elements will be reused at every iteration so their memory buffers need to be retained). At the end of each iteration, after the optimizer has updated the `k1` values, we need to call `update_transfer_map` again in order to activate the updates.

[7]:
```
import itertools as it
import dipas.compute as compute
from dipas.elements import Kicker
import torch

lattice = lattice.makethin({Kicker: 2}, style={Kicker: 'edge'})
print(f'# Parameters: {len(list(lattice.parameters()))}')

targets = {'Q1': torch.tensor(twiss[1]['Q1']), 'Q2': torch.tensor(twiss[1]['Q2'])}
optimizer = torch.optim.LBFGS(lattice.parameters())
cost_fn = torch.nn.MSELoss()

for step in it.count():
    def closure():
        optimizer.zero_grad()
        data = compute.twiss(lattice)
        Q1, Q2 = data['Q1'], data['Q2']
        cost = cost_fn(Q1, targets['Q1']) + cost_fn(Q2, targets['Q2'])
        print(f'Step {step:03d}: Q1 = {Q1:.3f}, Q2 = {Q2:.3f}, cost = {cost:.2e}')
        if cost < 1e-6:
            raise RuntimeError
        cost.backward(retain_graph=True)
        return cost

    try:
        optimizer.step(closure)
    except RuntimeError:
        break

    for q in h_quadrupoles:
        q.update_transfer_map()
```

```
# Parameters: 12
Step 000: Q1 = 2.362, Q2 = 2.459, cost = 4.88e-03
Step 000: Q1 = 2.362, Q2 = 2.459, cost = 4.88e-03
Step 001: Q1 = 2.386, Q2 = 2.446, cost = 1.87e-03
Step 001: Q1 = 2.386, Q2 = 2.446, cost = 1.87e-03
Step 002: Q1 = 2.408, Q2 = 2.435, cost = 3.69e-04
Step 002: Q1 = 2.408, Q2 = 2.435, cost = 3.69e-04
Step 003: Q1 = 2.418, Q2 = 2.430, cost = 1.10e-04
Step 003: Q1 = 2.418, Q2 = 2.430, cost = 1.10e-04
Step 004: Q1 = 2.422, Q2 = 2.428, cost = 7.26e-05
Step 004: Q1 = 2.422, Q2 = 2.428, cost = 7.26e-05
Step 005: Q1 = 2.423, Q2 = 2.428, cost = 6.80e-05
Step 005: Q1 = 2.423, Q2 = 2.428, cost = 6.80e-05
Step 006: Q1 = 2.424, Q2 = 2.427, cost = 6.25e-05
Step 006: Q1 = 2.424, Q2 = 2.427, cost = 6.25e-05
Step 007: Q1 = 2.425, Q2 = 2.425, cost = 5.13e-05
```

(continues on next page)

(continued from previous page)

```
Step 007: Q1 = 2.425, Q2 = 2.425, cost = 5.13e-05
Step 008: Q1 = 2.425, Q2 = 2.423, cost = 3.71e-05
Step 008: Q1 = 2.425, Q2 = 2.423, cost = 3.71e-05
Step 009: Q1 = 2.424, Q2 = 2.422, cost = 1.66e-05
Step 009: Q1 = 2.424, Q2 = 2.422, cost = 1.66e-05
Step 010: Q1 = 2.420, Q2 = 2.420, cost = 3.00e-07
```

Now let's crosscheck the solution by running it through MADX. We can use `dipas.build.create_script` in order to convert the `lattice` object to a corresponding MADX script.

[8]:
```
from dipas.build import create_script

twiss_check = run_script(
    create_script(sequence=lattice, errors=True, beam={'particle': 'proton', 'energy
↪': 1}),
    twiss=True,
    madx=os.path.expanduser('~/bin/madx')
)['twiss']
print(f'Tune values: {twiss_check[1]["Q1"]:.3f}, {twiss_check[1]["Q2"]:.3f}')
```

```
Tune values: 2.420, 2.420
```

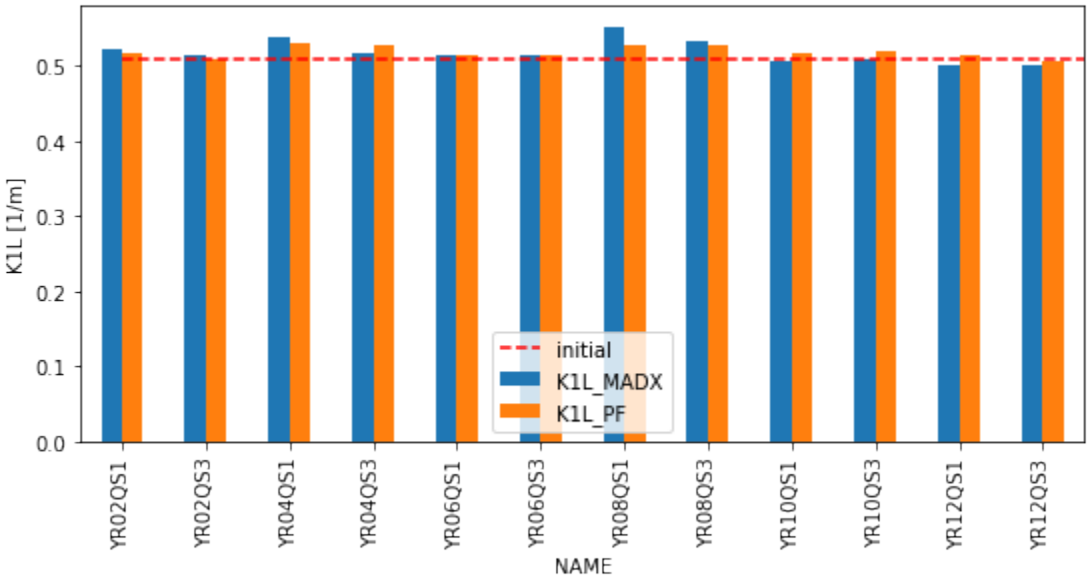
Finally let's compare the results to the solution which MADX originally computed:

[9]:
```
import pandas as pd

q_names = [q.label.upper() for q in h_quadrupoles]
results = twiss_matched[0].set_index('NAME').loc[q_names, ['K1L']]
results = results.assign(DP=pd.Series([q.k1.item()*q.l.item() for q in h_quadrupoles],
↪ index=q_names))
results.columns = ['K1L_MADX', 'K1L_PF']
print(results)
```

```
         K1L_MADX    K1L_PF
NAME
YR02QS1  0.521336  0.517564
YR02QS3  0.513679  0.509166
YR04QS1  0.538908  0.530332
YR04QS3  0.517392  0.527857
YR06QS1  0.514691  0.513474
YR06QS3  0.513379  0.513102
YR08QS1  0.552487  0.526501
YR08QS3  0.533606  0.526681
YR10QS1  0.506585  0.515837
YR10QS3  0.507948  0.518586
YR12QS1  0.500134  0.513845
YR12QS3  0.500342  0.506482
```

[10]:
```
%matplotlib inline

ax = results.plot(kind='bar', figsize=(9, 4))
ax.set_ylabel('K1L [1/m]')
ax.plot([0, len(results)], [0.5086546699]*2, '--', color='red', label='initial')
ax.legend()
```

[10]:
```
<matplotlib.legend.Legend at 0x7f6063591910>
```



[ ]:

5.3 Transfer line quadrupole tuning

The beamline consists of 21 quadrupoles whose strength will be varied in order to fulfill the following optimization targets:

- Beam spot size @ target position: $\sigma_x \leq 500\,\mu m, \sigma_y \leq 500\,\mu m$
- Beam spot size @ beam dump position: $\sigma_x \leq 12\,mm, \sigma_y \leq 12\,mm$
- Fractional beam loss along beamline less than 1%

Let's start with importing everything that we are going to need throughout the optimization process:

[1]:
```
from collections import deque
import itertools as it
import logging
import math
import os
from pprint import pprint
import statistics

import numpy as np
import pandas as pd
import torch

from dipas.build import from_file, create_script, track_script
from dipas.elements import configure, Quadrupole
from dipas.madx import run_script
```

In the following we define the optimization targets as above:

[2]:
```
optimization_targets = dict(
    target_rms_x = 500e-6,  # Beam spot size at target.
    target_rms_y = 500e-6,
    dump_rms_x = 12e-3,     # Beam spot size at beam dump.
    dump_rms_y = 12e-3,
    loss = 0.01             # Fractional loss along beamline.
)
```

Using the `build.from_file` function we can load the example lattice from a MADX script file:

[ ]:
```
configure(transfer_map_order=1)  # Using linear optics to save memory.
lattice = from_file('example.seq')
```

We are only interested in the part of the beamline up to the beam dump position, so we select the corresponding segment:

[4]:
```
lattice = lattice[:'dump']
print('Last lattice element:', lattice.elements[-1])
```

```
Last lattice element: Monitor(l=tensor(0.), label='dump')
```

In a next step we would declare the parameters of the optimization process, i.e. the quadrupoles' gradient strengths. However this step has been done already in the corresponding MADX script. The MADX parser supports special comments of the form `// [flow] variable` to indicate optimization parameters. These comments work for variable definitions as well as attribute updates (e.g. `some_element->k1 = 0.0;`). Let's view the corresponding section of the MADX script file:

[5]:
```
with open('example.seq') as fh:
    print(''.join(fh.readlines()[:30]))
```

```
beam, particle=ion, charge=6, energy=28.5779291448, mass=11.1779291448;

k1l_GTE1QD11 =  1e-6;  // [flow] variable
k1l_GTE1QD12 = -1e-6;  // [flow] variable

k1l_GTE2QT11 =  1e-6;  // [flow] variable
k1l_GTE2QT12 = -1e-6;  // [flow] variable
k1l_GTE2QT13 =  1e-6;  // [flow] variable

k1l_GTH1QD11 =  1e-6;  // [flow] variable
k1l_GTH1QD12 = -1e-6;  // [flow] variable

k1l_GTH2QD11 =  1e-6;  // [flow] variable
k1l_GTH2QD12 = -1e-6;  // [flow] variable
k1l_GTH2QD21 = -1e-6;  // [flow] variable
k1l_GTH2QD22 =  1e-6;  // [flow] variable

k1l_GHADQD11 = -1e-6;  // [flow] variable
k1l_GHADQD12 =  1e-6;  // [flow] variable
k1l_GHADQD21 = -1e-6;  // [flow] variable
k1l_GHADQD22 =  1e-6;  // [flow] variable

k1l_GHADQD31 = -1e-6;  // [flow] variable
k1l_GHADQD32 =  1e-6;  // [flow] variable
k1l_GHADQD41 =  1e-6;  // [flow] variable
k1l_GHADQD42 = -1e-6;  // [flow] variable

k1l_GHADQT51 =  1e-6;  // [flow] variable
```

(continues on next page)

(continued from previous page)

```
k11_GHADQT52 = -1e-6;  // [flow] variable
```

We can confirm that the parsed lattice contains the corresponding parameters already:

[6]:
```
for quad in lattice[Quadrupole]:
    print(f'{quad.label}: {quad.k1!r}')
print('Number of parameters:', len(list(lattice.parameters())))
```

```
gte1qd11: Parameter containing:
tensor(1.5015e-06, requires_grad=True)
gte1qd12: Parameter containing:
tensor(-1.5015e-06, requires_grad=True)
gte2qt11: Parameter containing:
tensor(1.0000e-06, requires_grad=True)
gte2qt12: Parameter containing:
tensor(-1.0000e-06, requires_grad=True)
gte2qt13: Parameter containing:
tensor(1.0000e-06, requires_grad=True)
gth1qd11: Parameter containing:
tensor(1.0000e-06, requires_grad=True)
gth1qd12: Parameter containing:
tensor(-1.0000e-06, requires_grad=True)
gth2qd11: Parameter containing:
tensor(1.0000e-06, requires_grad=True)
gth2qd12: Parameter containing:
tensor(-1.0000e-06, requires_grad=True)
gth2qd21: Parameter containing:
tensor(-1.0000e-06, requires_grad=True)
gth2qd22: Parameter containing:
tensor(1.0000e-06, requires_grad=True)
ghadqd11: Parameter containing:
tensor(-1.0000e-06, requires_grad=True)
ghadqd12: Parameter containing:
tensor(1.0000e-06, requires_grad=True)
ghadqd21: Parameter containing:
tensor(-1.0000e-06, requires_grad=True)
ghadqd22: Parameter containing:
tensor(1.0000e-06, requires_grad=True)
ghadqd31: Parameter containing:
tensor(-1.6667e-06, requires_grad=True)
ghadqd32: Parameter containing:
tensor(1.6667e-06, requires_grad=True)
ghadqd41: Parameter containing:
tensor(1.6667e-06, requires_grad=True)
ghadqd42: Parameter containing:
tensor(-1.6667e-06, requires_grad=True)
ghadqt51: Parameter containing:
tensor(1.0000e-06, requires_grad=True)
ghadqt52: Parameter containing:
tensor(-1.0000e-06, requires_grad=True)
Number of parameters: 21
```

If the optimization parameters were not already declared we could also do it manually using the following `for` loop:

[7]:
```
# for quad in lattice[Quadrupole]:
```

(continues on next page)

(continued from previous page)

```
#     quad.k1 = torch.nn.Parameter(quad.k1)
#     quad.update_transfer_map()  # Need to call this method in order for the change␣
↪to become effective.
```

In a next step we select the 21 quadrupoles and define some additional properties such as valid boundaries for their k1-values. An important aspect to note here is that k1-values which are marked as optimization parameters must never be zero. This is because internally the polarity of the magnet is derived from the sign of the k1-value (positive sign means horizontally focusing). For that reason we define a small epsilon-boundary instead (any value other than zero would do, no matter how small). Also note that this restriction only applies to k1-values that are `Parameters`. For non-parameters, if the k1-value is zero, the Quadrupole acts as a Drift space.

[8]:
```
quadrupoles = lattice[Quadrupole]
pprint(quadrupoles)
print()

QPL_limit = 11.1 / 14.62
QPK_limit = 6.88 / 14.62
QPK_magnets = {'gte1qd11', 'gte1qd12', 'ghadqd31', 'ghadqd32', 'ghadqd41', 'ghadqd42'}
polarity = {  # +1.0 means horizontally focusing.
    'gte1qd11':  1.0,
    'gte1qd12': -1.0,

    'gte2qt11':  1.0,
    'gte2qt12': -1.0,
    'gte2qt13':  1.0,

    'gth1qd11':  1.0,
    'gth1qd12': -1.0,

    'gth2qd11':  1.0,
    'gth2qd12': -1.0,
    'gth2qd21': -1.0,
    'gth2qd22':  1.0,

    'ghadqd11': -1.0,
    'ghadqd12':  1.0,
    'ghadqd21': -1.0,
    'ghadqd22':  1.0,

    'ghadqd31': -1.0,
    'ghadqd32':  1.0,
    'ghadqd41':  1.0,
    'ghadqd42': -1.0,

    'ghadqt51':  1.0,
    'ghadqt52': -1.0,
}
k1_bounds = {
    q.label: sorted([  # Lower bound must come first.
        polarity[q.label] * 1e-6,  # Variable strength quadrupoles must not be zero␣
↪to retain their polarity.
        polarity[q.label] * (QPK_limit if q.label in QPK_magnets else QPL_limit)
    ]) for q in quadrupoles
}
pprint(k1_bounds)
```

```
[Quadrupole(l=tensor(0.6660), k1=Parameter containing: tensor(1.5015e-06,␣
↪requires_grad=True), aperture=ApertureEllipse(aperture=tensor([0.0600, 0.0600]),␣
↪offset=tensor([0., 0.]), padding=0.0), label='gte1qd11'),
 Quadrupole(l=tensor(0.6660), k1=Parameter containing: tensor(-1.5015e-06,␣
↪requires_grad=True), aperture=ApertureCircle(aperture=0.06, offset=tensor([0., 0.]),
↪ padding=0.0), label='gte1qd12'),
 Quadrupole(l=tensor(1.), k1=Parameter containing: tensor(1.0000e-06,␣
↪requires_grad=True), aperture=ApertureEllipse(aperture=tensor([0.0600, 0.0600]),␣
↪offset=tensor([0., 0.]), padding=0.0), label='gte2qt11'),
 Quadrupole(l=tensor(1.), k1=Parameter containing: tensor(-1.0000e-06,␣
↪requires_grad=True), aperture=ApertureEllipse(aperture=tensor([0.0600, 0.0600]),␣
↪offset=tensor([0., 0.]), padding=0.0), label='gte2qt12'),
 Quadrupole(l=tensor(1.), k1=Parameter containing: tensor(1.0000e-06,␣
↪requires_grad=True), aperture=ApertureEllipse(aperture=tensor([0.0600, 0.0600]),␣
↪offset=tensor([0., 0.]), padding=0.0), label='gte2qt13'),
 Quadrupole(l=tensor(1.), k1=Parameter containing: tensor(1.0000e-06,␣
↪requires_grad=True), aperture=ApertureEllipse(aperture=tensor([0.0540, 0.0540]),␣
↪offset=tensor([0., 0.]), padding=0.0), label='gth1qd11'),
 Quadrupole(l=tensor(1.), k1=Parameter containing: tensor(-1.0000e-06,␣
↪requires_grad=True), aperture=ApertureEllipse(aperture=tensor([0.0540, 0.0540]),␣
↪offset=tensor([0., 0.]), padding=0.0), label='gth1qd12'),
 Quadrupole(l=tensor(1.), k1=Parameter containing: tensor(1.0000e-06,␣
↪requires_grad=True), aperture=ApertureEllipse(aperture=tensor([0.0540, 0.0540]),␣
↪offset=tensor([0., 0.]), padding=0.0), label='gth2qd11'),
 Quadrupole(l=tensor(1.), k1=Parameter containing: tensor(-1.0000e-06,␣
↪requires_grad=True), aperture=ApertureEllipse(aperture=tensor([0.0540, 0.0540]),␣
↪offset=tensor([0., 0.]), padding=0.0), label='gth2qd12'),
 Quadrupole(l=tensor(1.), k1=Parameter containing: tensor(-1.0000e-06,␣
↪requires_grad=True), aperture=ApertureEllipse(aperture=tensor([0.0540, 0.0540]),␣
↪offset=tensor([0., 0.]), padding=0.0), label='gth2qd21'),
 Quadrupole(l=tensor(1.), k1=Parameter containing: tensor(1.0000e-06,␣
↪requires_grad=True), aperture=ApertureEllipse(aperture=tensor([0.0540, 0.0540]),␣
↪offset=tensor([0., 0.]), padding=0.0), label='gth2qd22'),
 Quadrupole(l=tensor(1.), k1=Parameter containing: tensor(-1.0000e-06,␣
↪requires_grad=True), aperture=ApertureEllipse(aperture=tensor([0.0900, 0.0900]),␣
↪offset=tensor([0., 0.]), padding=0.0), label='ghadqd11'),
 Quadrupole(l=tensor(1.), k1=Parameter containing: tensor(1.0000e-06,␣
↪requires_grad=True), aperture=ApertureEllipse(aperture=tensor([0.0900, 0.0900]),␣
↪offset=tensor([0., 0.]), padding=0.0), label='ghadqd12'),
 Quadrupole(l=tensor(1.), k1=Parameter containing: tensor(-1.0000e-06,␣
↪requires_grad=True), aperture=ApertureEllipse(aperture=tensor([0.0600, 0.0600]),␣
↪offset=tensor([0., 0.]), padding=0.0), label='ghadqd21'),
 Quadrupole(l=tensor(1.), k1=Parameter containing: tensor(1.0000e-06,␣
↪requires_grad=True), aperture=ApertureEllipse(aperture=tensor([0.0600, 0.0600]),␣
↪offset=tensor([0., 0.]), padding=0.0), label='ghadqd22'),
 Quadrupole(l=tensor(0.6000), k1=Parameter containing: tensor(-1.6667e-06,␣
↪requires_grad=True), aperture=ApertureEllipse(aperture=tensor([0.0600, 0.0600]),␣
↪offset=tensor([0., 0.]), padding=0.0), label='ghadqd31'),
 Quadrupole(l=tensor(0.6000), k1=Parameter containing: tensor(1.6667e-06,␣
↪requires_grad=True), aperture=ApertureEllipse(aperture=tensor([0.0600, 0.0600]),␣
↪offset=tensor([0., 0.]), padding=0.0), label='ghadqd32'),
 Quadrupole(l=tensor(0.6000), k1=Parameter containing: tensor(1.6667e-06,␣
↪requires_grad=True), aperture=ApertureEllipse(aperture=tensor([0.0600, 0.0600]),␣
↪offset=tensor([0., 0.]), padding=0.0), label='ghadqd41'),
 Quadrupole(l=tensor(0.6000), k1=Parameter containing: tensor(-1.6667e-06,␣
↪requires_grad=True), aperture=ApertureEllipse(aperture=tensor([0.0600, 0.0600]),␣
↪offset=tensor([0., 0.]), padding=0.0), label='ghadqd42'),
```

(continues on next page)

(continued from previous page)

```
 Quadrupole(l=tensor(1.), k1=Parameter containing: tensor(1.0000e-06,␣
↪requires_grad=True), aperture=ApertureEllipse(aperture=tensor([0.0600, 0.0600]),␣
↪offset=tensor([0., 0.]), padding=0.0), label='ghadqt51'),
 Quadrupole(l=tensor(1.), k1=Parameter containing: tensor(-1.0000e-06,␣
↪requires_grad=True), aperture=ApertureEllipse(aperture=tensor([0.0600, 0.0600]),␣
↪offset=tensor([0., 0.]), padding=0.0), label='ghadqt52')]

{'ghadqd11': [-0.759233926128591, -1e-06],
 'ghadqd12': [1e-06, 0.759233926128591],
 'ghadqd21': [-0.759233926128591, -1e-06],
 'ghadqd22': [1e-06, 0.759233926128591],
 'ghadqd31': [-0.47058823529411764, -1e-06],
 'ghadqd32': [1e-06, 0.47058823529411764],
 'ghadqd41': [1e-06, 0.47058823529411764],
 'ghadqd42': [-0.47058823529411764, -1e-06],
 'ghadqt51': [1e-06, 0.759233926128591],
 'ghadqt52': [-0.759233926128591, -1e-06],
 'gte1qd11': [1e-06, 0.47058823529411764],
 'gte1qd12': [-0.47058823529411764, -1e-06],
 'gte2qt11': [1e-06, 0.759233926128591],
 'gte2qt12': [-0.759233926128591, -1e-06],
 'gte2qt13': [1e-06, 0.759233926128591],
 'gth1qd11': [1e-06, 0.759233926128591],
 'gth1qd12': [-0.759233926128591, -1e-06],
 'gth2qd11': [1e-06, 0.759233926128591],
 'gth2qd12': [-0.759233926128591, -1e-06],
 'gth2qd21': [-0.759233926128591, -1e-06],
 'gth2qd22': [1e-06, 0.759233926128591]}
```

The initial particle distribution at the entrance of the beamline is stored in a CSV file (5,000 particles):

[9]:
```
particles = pd.read_csv('particles.csv', index_col=0)
particles = torch.from_numpy(particles.values.T)
print('Particles:', particles.shape)
```

```
Particles: torch.Size([6, 5000])
```

Now let's run a tracking forward pass through the lattice in order to verify everything's set up correctly:

[10]:
```
x, history, loss = lattice.linear(particles, observe=['target', 'dump'], recloss='sum
↪')
print(x.shape)
print({k: v.shape for k, v in history.items()})
print(loss)
```

```
torch.Size([6, 1878])
{'target': torch.Size([6, 1878]), 'dump': torch.Size([6, 1878])}
tensor(1073.9377, grad_fn=<AddBackward0>)
```

Here we can see that out of the initial 5,000 particles only 1,878 make it to the end of the beamline. The remaining 3,122 are lost at the various elements in between and this is reflected in the loss value `loss`. This value is the sum over all elements and for each particle and element it indicates by how much the particle's spatial coordinates exceeded the element's aperture (so it is not directly related to the fraction of particles lost; this value can be computed from the shape of the tensors). If we wanted to know where exactly the particles are lost, we would need to specify `recloss=True` (or more generally `recloss=identifier`, see the documentation of `elements.Segment` for more details). We can also observe that the `loss` value is differentiable, as indicated by the `grad_fn` attribute.

Finally we setup the optimizer that computes the updates for the k1-values during the optimization process. For this

example we use the `Adam` optimizer:

[11]:
```
optimizer = torch.optim.Adam(lattice.parameters(), lr=0.001)
```

Now we're ready to start the optimization:

[ ]:
```
cost_history = []

for epoch in it.count(1):
    def closure():
        optimizer.zero_grad()

        __, history, loss = lattice.linear(particles, observe=['target', 'dump'],
                                           recloss='sum',      # Sum the loss per␣
↪element and per particle.
                                           exact_drift=False)  # Linear drifts speed␣
↪up the computation.
        particles_lost = 1.0 - history['dump'].shape[1] / particles.shape[1]
        if particles_lost > optimization_targets['loss']:
            cost = (loss / particles.shape[1]) / optimization_targets['loss']  #␣
↪Average loss per particle / target loss.
        else:
            cost = 0.  # Target fractional loss was reached, no need to optimize for␣
↪that (at the current iteration).

        log_dict = dict(epoch=epoch, particles_lost=f'{particles_lost:.2f}')

        for place in ('target', 'dump'):
            # Only compare spot sizes to targeted ones if no more than 50% of the␣
↪particles were lost.
            if history[place].shape[1] > particles.shape[1] // 2:
                x, y = history[place][[0, 2]]
                rms_x = x.std()
                rms_y = y.std()
                cost = (cost + torch.nn.functional.relu(rms_x / optimization_targets[f
↪'{place}_rms_x'] - 1.0)
                        + torch.nn.functional.relu(rms_y / optimization_targets[f'
↪{place}_rms_y'] - 1.0))
                log_dict.update({f'{place}_rms_x': f'{rms_x.data:.6f}', f'{place}_rms_
↪y': f'{rms_y:.6f}'})
            else:
                log_dict.update({f'{place}_rms_x': 'n.a.', f'{place}_rms_y': 'n.a.'})

        cost_history.append(cost.data.clone())
        log_dict['cost_to_optimize'] = cost.data.clone()
        print(log_dict)
        cost.backward(retain_graph=True)  # Transfer maps are reused at every␣
↪iteration so we need to retain the memory buffers.
        return cost

    optimizer.step(closure)

    if cost_history[-1] == 0:
        break

    with torch.no_grad():
        for q in quadrupoles:
            q.k1.data.clamp_(*k1_bounds[q.label])  # Squeeze k1-values back into␣
↪bounds if necessary.
```

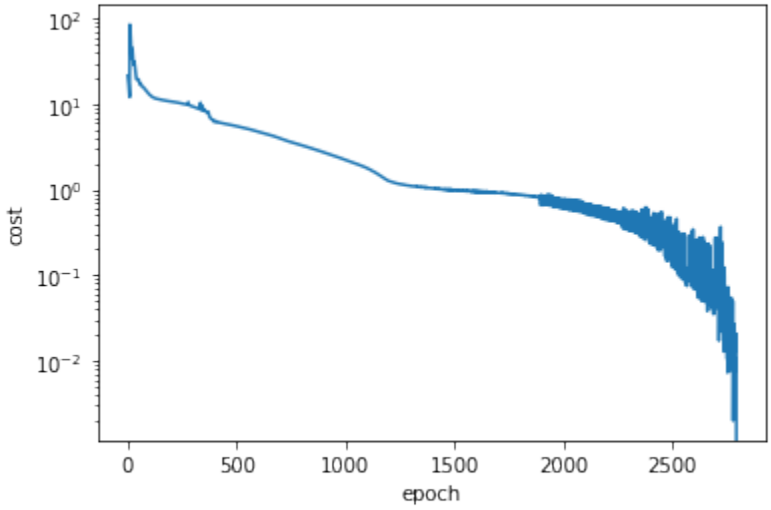
(continues on next page)

(continued from previous page)

```
    for q in quadrupoles:
        q.update_transfer_map()
```

Eventually the optimization process converged and we plot the cost value over all epochs, to review the progress of the optimization process:

[13]:
```
%matplotlib inline

import matplotlib.pyplot as plt

plt.plot(cost_history)
plt.yscale('log')
plt.xlabel('epoch')
plt.ylabel('cost')
plt.show()
```



As we can observe from the above plot, the optimization was a bit of a bumpy ride towards the end. Situations like this can often be improved by decreasing the learning rate when approaching the minimum. For that we could have used one of the `torch.optim.lr_scheduler` classes or manually stop the optimization at some point, decrease the learning rate and resume from where we stopped.

To conclude the example we will run a crosscheck with the MADX simulation tool, in order to verify the results. We can serialize the current version of the lattice using the functions `create_script, sequence_script, track_script` from the `build` module. With `madx.utils.run_script` we can run the thus generated script and get back the tracking results in form of `pd.DataFrame` objects.

[14]:
```
madx_script = create_script(
    sequence=lattice,
    track=track_script(particles, observe=['target', 'dump'], maxaper=[100]*6),  #␣
↪Aperture is already on the elements.
    beam=dict(charge=6, mass=11.1779291448, energy=28.5779291448)
)
with open('result.madx', 'w') as fh:
    fh.write(madx_script)

results = run_script(madx_script, ['trackone', 'trackloss'], twiss=True, madx=os.path.
↪expanduser('~/bin/madx'))
```

(continues on next page)

(continued from previous page)

```
print('\nCrosscheck with MADX:')
print('\tFraction of particles lost: ', len(results['trackloss'])/particles.shape[1])
print('\tBeam spot size at target:   ', results['trackone'].loc['target', ['X', 'Y']].
↪values.std(axis=0))
print('\tBeam spot size at beam dump:', results['trackone'].loc['dump', ['X', 'Y']].
↪values.std(axis=0))
```

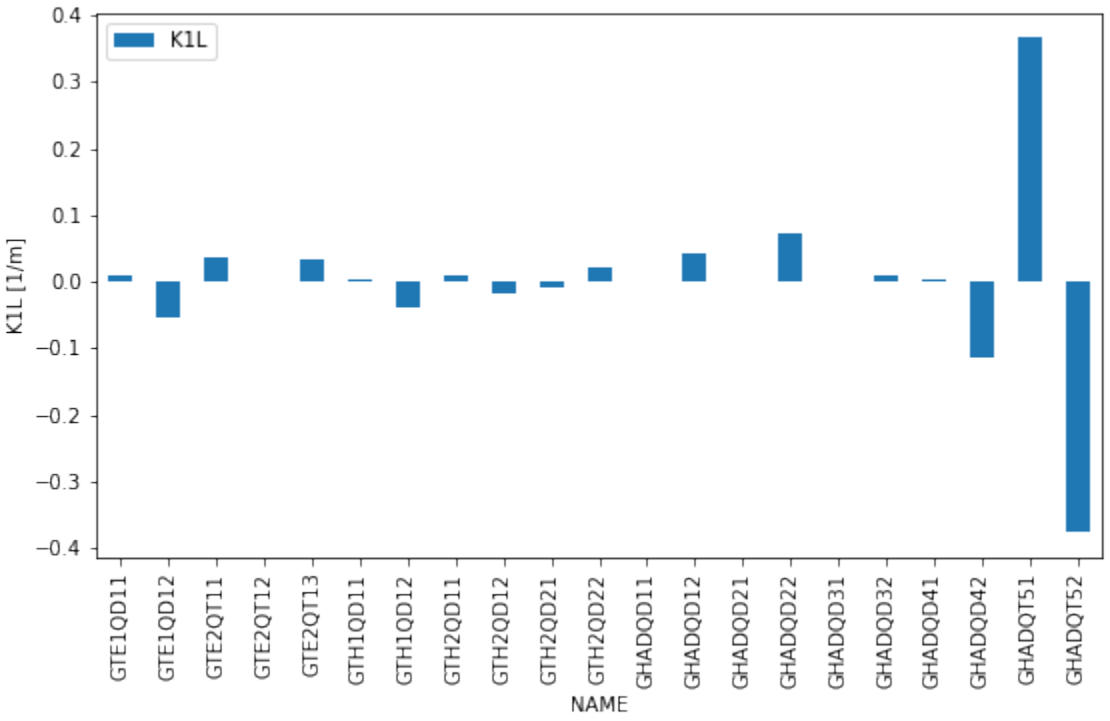
```
Crosscheck with MADX:
        Fraction of particles lost:  0.006
        Beam spot size at target:    [0.00051271 0.00050737]
        Beam spot size at beam dump: [0.01056864 0.00879908]
```

There's a small deviation in the results due to the fact that we used linear optics for the tracking while MADX uses non-linear update formulas.

Finally let's plot the quadrupole gradients along the beamline:

[20]:
```
ax = results['twiss'][0].set_index('NAME').loc[[q.label.upper() for q in quadrupoles],
↪ ['K1L']].plot(kind='bar', figsize=(9, 5))
ax.set_ylabel('K1L [1/m]')
```

[20]:
```
Text(0, 0.5, 'K1L [1/m]')
```



[ ]:

CHAPTER

SIX

DIPAS API

6.1 dipas package

6.1.1 Subpackages

dipas.madx package

Submodules

dipas.madx.builder module

class dipas.madx.builder.**LiteralString**(*seq*)
Bases: collections.UserString

dipas.madx.builder.**write_attribute_assignment**(*label: str*, *attribute: str*, *value: Any*) → str

dipas.madx.builder.**write_attribute_increment**(*label: str*, *attribute: str*, *value: numbers.Real*) → str

dipas.madx.builder.**write_attribute_value**(*x*) → str
Convert an attribute value to its corresponding MADX representation.

dipas.madx.builder.**write_attributes**(*attributes: dict*) → str
Convert the given attributes to a corresponding MADX attribute string.

dipas.madx.builder.**write_command**(*keyword: str*, *attributes: Optional[dict] = None*, *label: Optional[str] = None*) → str
Convert the given keyword and attributes to a corresponding MADX command string.

dipas.madx.parser module

Functionality for parsing MADX files.

Some details of the parsing procedure are configured (and may be altered) by the following module-level attributes.

dipas.madx.parser.**replacement_string_for_dots_in_variable_names**
The parser does not support dots in variable names (it only supports variable names that are valid Python identifiers) and so each dot in a variable name will be replaced by the string indicated by this attribute.

Type str

dipas.madx.parser.**negative_offset_tolerance**
If during sequence expansion (*pad_sequence* more specifically) a negative offset between two elements is encountered the parser will raise a *ParserError* if this offset is smaller than this attribute.

Type float

dipas.madx.parser.**minimum_offset_for_drift**
During sequence expansion (*pad_sequence* more specifically) an implicit drift space will be padded by an explicit drift space only if the offset between the two involved elements is greater than this attribute.

Type float

dipas.madx.parser.**allow_popup_variables**
If an expression consists of a single variable name which cannot be resolved (i.e. that variable was not defined before), then the parser will fallback on (float) zero. If this parameter is false a *ParserError* will be raised instead. Example for a "popup variable": `quadrupole, k1 = xyz` where `xyz` was not defined before. This will either fall back on `0.0` or raise an error, depending on this parameter.

Type bool, default = True

dipas.madx.parser.**rng_default_seed**
The default seed for the random number generator used for evaluating calls to functions such as "ranf" (np.random.random) or "gauss" (np.random.normal).

Type int

dipas.madx.parser.**command_str_attributes**
Command attributes with names that are part of this set are assumed to be strings and will hence not be evaluated (i.e. not name resolution, evaluation of formulas, etc). By default this only lists argument names that are strings by MADX definitions.

Type set

dipas.madx.parser.**special_names**
During evaluation of expressions this dict will be used for resolving any names that are encountered. By default this contains functions such as `"asin": np.arcsin` or constants like `"pi": np.pi`.

Type dict

dipas.madx.parser.**particle_dict**
Given a *BEAM* command the parser computes the relativistic beta and gamma factors from the given quantities (actually it augments the BEAM command by all other quantities). This dict is used for resolving particles names given by `BEAM, particle = xyz;` (i.e. selects the corresponding *charge* and *mass*).

Type dict

dipas.madx.parser.**patterns**
Contains regex patterns for MADX statements (e.g. comments, variable definitions, etc) mapped to by a corresponding (descriptive) name. If a pattern is matched a corresponding statement handler will be invoked which must have been previously registered via *register_handler* (or inserted into *statement_parsers*).

Type dict

dipas.madx.parser.**statement_handlers**
This dict maps pattern names (keys of the `patterns` dict) to corresponding statement handlers. A handler will be invoked when the corresponding statement pattern matched a statement. For more details on the signature of such a handler, see `register_handler()`.

Type dict

dipas.madx.parser.**VARIABLE_INDICATOR**
The string which is used to indicate *flow variables* in comments (by default this is `[flow] variable`).

Type str

dipas.madx.parser.**prepare_script**
Contains functions that that perform preparation steps prior to parsing the script. All the listed preparation steps are performed in order on the particular previous result. This signature of such preparation step function should be `(str) -> str`, i.e. accept the current version of the (partially) prepared script and return a new version. The last function must return a list of single statements when given the prepared script (i.e. `(str) -> List[str]`).

Type list

dipas.madx.parser.**prepare_statement**
Contains functions that perform preparation steps for each single statement in order, prior to parsing it. The signature of these functions should be `(str) -> str`, i.e. accepting the current version of the (partially) prepared statement and return a new version.

Type list

dipas.madx.parser.**parse_file**(*f_name: str*) → dipas.madx.parser.Script

Auxiliary function for *parse_script* working on file names which also resolves references to other scripts via `CALL, file = ...`.

Parameters **f_name** (*str*) – File name pointing to the MADX script.

Returns

Return type See *parse_script()*.

dipas.madx.parser.**parse_script**(*script: str*) → dipas.madx.parser.Script
Parses a MADX script and returns the relevant commands list as well as command variable definitions.

Flow variables should be declared on a separate statement and indicated using one of the following syntax options:

```
q1_k1 = 0;  // < optional text goes here > [flow] variable
q1: quadrupole, l=1, k1=q1_k1;

// < optional text goes here > [flow] variable
q1_k1 = 0;
q1: quadrupole, l=1, k1=q1_k1;

q1: quadrupole, l=1, k1=0;
q1->k1 = 0;  // < optional text goes here > [flow] variable
```

Important: If the script contains any `CALL, file = ...` commands these will not be resolved (just parsed as such). Use *parse_file* for that purpose.

Parameters **script** (*str*) – The MADX script's content.

Returns **script**

Return type Script

Raises **ParserError** – If a misplaced variable indicator is encountered, e.g. not followed by a variable definition.

dipas.madx.utils module

Utilities for interfacing the MADX program and parsing MADX generated output files.

dipas.madx.utils.**run_file**(*scripts: Union[str, Sequence[str]], results: Union[Sequence[str], Dict[str, bool]] = None, *, variables: Dict[str, Any] = None, format: Dict[str, Any] = None, parameters: Dict[str, Any] = None, twiss: Dict[str, Any] = None, madx: str = None, wdir: Optional[str] = None*) → Dict

Runs a single script or a bunch of dependent scripts, with optional configuration.

The first script specified in *scripts* is considered the entry point and is passed to the MADX executable. Other scripts should be invoked implicitly via `call, file = "...";`. If there is only a single script it can be used directly as the first argument.

Parameters

- **scripts** (*str or list of str*) – A single script or a list of MADX script file names. The first item is used as the entry point. Actually the list can contain any file names, relative to the current working directory. These files will be copied to the new, temporary working directory where the script will be run (regardless of whether they are used or not). For example one can include error definitions that way, which are then loaded in the main script. Note that files within the main script which are referred to via `call, file = ...` or `readtable, file = ...` are auto-discovered and appended to the list of required scripts (if not already present).
- **results** (*list or dict, optional*) – See *run_script()*.
- **variables** (*dict, optional*) – Variables configuration for each of the scripts in *scripts*. See *run_script()* for more details.
- **format** (*dict, optional*) – Format values for each of the scripts in *scripts*. See *run_script()* for more details.
- **parameters** (*dict, optional*) – Parameter definitions for each of the scripts in *scripts*. See *run_script()* for more details.
- **twiss** (*dict, optional*) – Twiss command specification for each of the scripts. See *run_script()* for more details.
- **madx** (*str, optional*) – See *run_script()*.
- **wdir** (*str, optional*) – The working directory in which the *scripts* will be run. Note that any existing files with similar names as the ones in *scripts* will be overwritten. If not specified then a temporary working directory is created.

Returns **output** – Containing the stdout and stderr at keys "stdout" and "stderr" respectively as well as any output files specified in *results*, converted by *convert()*.

Return type dict

Raises

- **ValueError** – If the MADX executable cannot be resolved either via *madx* or the *MADX* environment variable.
- **MADXError** – If the MADX executable returns a non-zero exit code. The raised error has additional attributes *script*, which is the content of the script that caused the error, as well as *stdout* and *stderr* which contain the MADX output. The `__cause__` of that error is set to the original *subprocess.CalledProcessError*.

See also:

convert() Used for converting specified output files.

dipas.madx.utils.**run_script**(*script: str, results: Union[Sequence[str], Dict[str, bool]] = None, *, variables: Dict[str, Any] = None, format: Dict[str, Any] = None, parameters: Dict[str, Dict[str, Any]] = None, twiss: Union[typing_extensions.Literal[True][True], Dict[str, Any]] = None, madx: str = None, wdir: Optional[str] = None*) → Dict

Run the given MADX script through the MADX program.

Parameters

- **script** (*str*) – The MADX script to be run.
- **results** (*list or dict, optional*) – File names of generated output files that should be returned. The content of these files will be automatically converted based on the chosen filename. For TFS-style files this does not include the header meta data (prefixed by "@") by default. If the header meta data should be returned as well, a *dict* can be used, mapping file names to *bool* flags that indicate whether meta data for this file is requested or not. More generally one can also provide a dict per file name that represents the keyword arguments that will be passed to *convert()* for that particular file. One can also use a special syntax for requesting meta data or indicating a specific file type. The syntax for each file name is `<file_name>+meta;<file_type>` where `+meta` and `;<file_type>` are optional. For example `example.tfs` would be parsed as a TFS file without parsing meta data. Using `example.tfs+meta` also returns the meta data. Or `example.tfs;raw` would return the raw content of the file rather than parsing it. For more information about file types see *convert()*. If the *twiss* argument is given, and a file name is specified, then this will automatically be added to the *results* list (similar for `twiss=True`).
- **variables** (*dict, optional*) – Used for replacing statements of the form `key = old;` with `key = value;`.
- **format** (*dict, optional*) – Used for filling in format specifiers of the form `%(key)s`.
- **parameters** (*dict, optional*) – Parameters of lattice elements. Keys should be element labels and values dicts that map attribute names to their values. These definitions will be inserted after the last sequence definition. For example `{"qh1": {"k1": 0.1}}` will be inserted as `qh1->k1 = 0.1;`.
- **twiss** (*True or dict, optional*) – Parameters for the *TWISS* command. If this argument is given, then a *TWISS* command is appended at the end of the script. The dict keys should be parameter names, such as `tolerance` with corresponding values. The key `"select"`, if present, should map to another dict that specifies the parameters for a preceding select statement. The `flag = twiss` parameter does not need to be specified as it will be added automatically. If the `file` parameter is specified, the file name does not need to be specified in *results*, it will be added automatically. If meta information of the resulting TWISS file is required, an additional `'meta': True` entry in the *twiss* dict can be provided. If *True* then the *TWISS* command is run with MADX default parameters while saving to a file named "twiss". The corresponding data is returned together with the meta data. Thus specifying `twiss=True` is equivalent to `twiss=dict(file='twiss', meta=True)`.
- **madx** (*str, optional*) – File name pointing to the MADX executable. If the *MADX* environment variable is set it takes precedence.
- **wdir** (*str, optional*) – The working directory in which the script will be run. Note that a file named "main.madx" will be created in that directory, overwriting any existing file with that name. If not specified then a temporary working directory is created.

Returns **output** – Containing the stdout and stderr at keys "stdout" and "stderr" respectively as well as any output files specified in *results*, converted by `convert()`.

Return type dict

Raises

- **ValueError** – If the MADX executable cannot be resolved either via *madx* or the *MADX* environment variable.
- **subprocess.CalledProcessError** – If the MADX executable returns a non-zero exit code.

See also:

convert() Used for converting specified output files.

dipas.madx.utils.**run_orm**(*script: str, kickers: Sequence[str], monitors: Sequence[str], *, kicks: Tuple[float, float] = (-0.001, 0.001), variables: Optional[Dict[str, Any]] = None, parameters: Dict[str, Dict[str, Any]] = None, twiss_args: Optional[Dict[str, Any]] = None, madx: str = None*) → <MagicMock name='mock.DataFrame' id='140260823939224'>

Compute the Orbit Response Matrix (ORM) for the given sequence script, kickers and monitors.

Parameters

- **script** (*str*) – Either the file name of the script or the script itself. The script must contain the beam and the sequence definition.
- **kickers** (*list of str*) – Kicker labels.
- **monitors** (*list of str*) – Monitor labels.
- **kicks** (*2-tuple of float*) – The kick strengths to be used for measuring the orbit response.
- **variables** (*dict*) – See *run_script()*.
- **parameters** (*dict*) – See *run_script()*.
- **twiss_args** (*dict*) – Additional parameters for the *TWISS* command.
- **madx** (*str*) – See *run_script()*.

Returns **orm** – Index == kickers, columns == monitors.

Return type pd.DataFrame

dipas.madx.utils.**convert**(*f_name: str, f_type: Optional[typing_extensions.Literal['madx', 'raw', 'tfs', 'trackone']][madx, raw, tfs, trackone]] = None, meta: bool = False*) → Union[<MagicMock name='mock.DataFrame' id='140260823939224'>, str, dipas.madx.parser.Script, Tuple[<MagicMock name='mock.DataFrame' id='140260823939224'>, Dict[str, str]]]

Convert MADX output file by automatically choosing the appropriate conversion method based on the file name.

Parameters

- **f_name** (*str*) – If ends with "one" then a `TRACK, ONETABLE = true` is assumed and a *pd.DataFrame* is returned. If suffix is one of *{".madx", ".seq"}* then a tuple according to *madx.parse_file* is returned. Otherwise a TFS file is assumed and converted to a *pd.DataFrame*. If this fails the raw string content is returned. Raw string content can also be enforced by using the suffix *".raw"*; this supersedes the other cases.

- **f_type** (*str*) –

  Determines how the file should be loaded. The following types can be used:

  - "madx" – parses the file as a MADX script, using `parse_file()`.
  - "raw" – loads the raw content of the file.
  - "tfs" – parses the file as a TFS file, using *convert_tfs()*.
  - "trackone" – parses the file as a TRACKONE file, using *convert_trackone()*.

- **meta** (*bool*) – Indicates whether TFS meta data (prefixed with "@") should be returned in form of a dict. This is only possible for *trackone* and *tfs* tables.

Returns

Return type The return value depends on the choice of the file name (see *f_name*).

dipas.madx.utils.**convert_tfs**(*f_name: str*, *meta: bool = False*) → Union[<MagicMock name='mock.DataFrame' id='140260823939224'>, Tuple[<MagicMock name='mock.DataFrame' id='140260823939224'>, Dict[str, str]]]

Convert table in TFS (Table File System) format to pandas data frame.

Parameters

- **f_name** (*str*) – File name pointing to the TFS file.
- **meta** (*bool, optional*) – If *True*, return meta information prefixed by "@ " in form of a *dict*.

Returns **df** – The corresponding data frame. If *meta* is *True* then a tuple containing the data frame and the meta data in form of a *dict* is returned.

Return type pd.DataFrame

Raises **ValueError** – If the given table is incomplete or if it's not presented in TFS format.

dipas.madx.utils.**convert_trackone**(*f_name: str*, *meta: bool = False*) → Union[<MagicMock name='mock.DataFrame' id='140260823939224'>, Tuple[<MagicMock name='mock.DataFrame' id='140260823939224'>, Dict[str, str]]]

Convert "trackone" table (generated by `TRACK, onetable = true`) to pandas data frame.

Parameters

- **f_name** (*str*) – File name pointing to the "trackone" file.
- **meta** (*bool, optional*) – If *True*, return meta information prefixed by "@ " in form of a *dict*.

Returns **df** – The corresponding data frame, augmented by two columns "PLACE" and "LABEL" indicating the observation places' *number* and *label* respectively. The columns *[LABEL, PLACE, NUMBER, TURN]* are set as the data frame's index. If *meta* is *True* then a tuple containing the data frame and the meta data in form of a *dict* is returned.

Return type pd.DataFrame

Raises **ValueError** – If the given table is incomplete or if it's not presented in TFS format.

Module contents

dipas.tools package

Submodules

dipas.tools.madx_to_html module

Convert a MADX sequence script to a corresponding HTML version (for display in a web browser).

dipas.tools.madx_to_html.**main**()

Module contents

6.1.2 Submodules

dipas.build module

Functionality for building an accelerator lattice using PyTorch as a backend.

dipas.build.**from_file**(*f_name: str, *, beam: dict = None, errors: Union[bool, <MagicMock name='mock.DataFrame' id='140260823939224'>] = True, paramodi: Union[str, <MagicMock name='mock.DataFrame' id='140260823939224'>] = None, padding: Union[float, Tuple[float, ...], Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], int, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], None], Union[float, Tuple[float, ...]]], List[Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], int, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], None], Union[float, Tuple[float, ...]]]], Dict[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], int, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], None], Union[float, Tuple[float, ...]]]] = None*) → dipas.elements.Segment

Build lattice from MADX script file.

Uses the first beam command and the first sequence encountered via *USE* when parsing the script. If no *USE* command is found then the first *SEQUENCE* in the script is considered.

Parameters

- **f_name** (*str*) – File path pointing to the MADX script.
- **beam** (*dict, optional*) – Beam specification similar to the MADX command *beam*. If not provided then the script will be searched for a *beam* command instead. Otherwise the user provided beam specification will override any specification in the script.
- **errors** (*bool or str, optional*) –

Whether and how to assign alignment errors to lattice elements. The following options are available:

- *False* - Ignore error specifications in the script.
- *True* - Apply error specifications from the script, interpreting any involved expressions. In case no random functions are involved in error specification the final values will be the same (when comparing the thus built lattice and MADX). However if random functions are involved then, even if the same seed for the random number generator (RNG) is used, the final values are likely to differ because MADX uses a different RNG than the present parser. Hence this option will result in alignment errors for the same elements and values from the same random variates, but not exactly the same values.
- ***pd.DataFrame* - For details about the structure see `apply_errors()`. Using a data frame the exact sa** values (from MADX) will be assigned. In order to ensure compatibility across multiple runs of the script, make sure to also set `eoption, seed = <rng_seed>`.

- **`paramodi`** (*`str or pd.DataFrame, optional`*) – Device settings to be used when building the lattice. Can be either the file path pointing to a paramodi file or a corresponding data frame, as returned by `external.Paramodi.parse()`. Note that the first encountered *purpose* is used, if that is undesired, the data frame should be filtered accordingly. For details see *`update_from_paramodi()`*.
- **`padding`** (*`float or tuple or dict, optional`*) – Additional padding applied to lattice elements. See `elements.Aperture`.

Returns lattice

Return type *Segment*

See also:

`assign_errors()`, *`update_from_paramodi()`*

dipas.build.**from_script**(*script: str, *, beam: dict = None, errors: Union[bool, <MagicMock name='mock.DataFrame' id='140260823939224'>] = True, paramodi: Union[str, <MagicMock name='mock.DataFrame' id='140260823939224'>] = None, padding: Union[float, Tuple[float, ...], Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], int, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], None], Union[float, Tuple[float, ...]]], List[Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], int, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], None], Union[float, Tuple[float, ...]]]], Dict[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], int, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], None], Union[float, Tuple[float, ...]]]] = None*) → dipas.elements.Segment

Build lattice from MADX script (for details see *from_file()*).

dipas.build.**from_twiss**(*twiss: Union[str, <MagicMock name='mock.DataFrame' id='140260823939224'>], *, beam: dict, center: bool = False, errors: Optional[<MagicMock name='mock.DataFrame' id='140260823939224'>] = None, paramodi: Optional[<MagicMock name='mock.DataFrame' id='140260823939224'>] = None, padding: Union[float, Tuple[float, ...], Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], int, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], None], Union[float, Tuple[float, ...]]], List[Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], int, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], None], Union[float, Tuple[float, ...]]]], Dict[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], int, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], None], Union[float, Tuple[float, ...]]], None] = None*) → dipas.elements.Segment

Build lattice from the given device data (optionally applying errors and/or paramodi specifications).

Parameters

- **twiss** (*str or pd.DataFrame*) – File path pointing to a TWISS file or corresponding data frame such as returned by *dipas.madx.utils.convert_tfs()*. Must contain the following columns: `NAME, KEYWORD, S, ... any required element attributes ...`.
- **errors** (*pd.DataFrame*) – See *from_file()*.
- **paramodi** (*pd.DataFrame*) – See *from_file()*.
- **padding** (*PaddingSpec*) – See *from_file()*.

Returns lattice

Return type *Segment*

dipas.build.**from_device_data**(*devices: <MagicMock name='mock.DataFrame' id='140260823939224'>, *, beam: dict, errors: Optional[<MagicMock name='mock.DataFrame' id='140260823939224'>] = None, paramodi: Optional[<MagicMock name='mock.DataFrame' id='140260823939224'>] = None, padding: Union[float, Tuple[float, ...], Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], int, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], None], Union[float, Tuple[float, ...]]], List[Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], int, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], None], Union[float, Tuple[float, ...]]]], Dict[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], int, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], None], Union[float, Tuple[float, ...]]], None] = None*) → dipas.elements.Segment

Build lattice from the given device data (optionally applying errors and/or paramodi specifications).

Parameters

- **devices** (*pd.DataFrame*) – Indices are element labels and columns are attributes. For details see `collect_device_data()`.
- **errors** (*pd.DataFrame*) – See *from_file()*.
- **paramodi** (*pd.DataFrame*) – See *from_file()*.
- **padding** (*PaddingSpec*) – See *from_file()*.

Returns lattice

Return type *Segment*

dipas.build.**update_from_twiss**(*lattice: dipas.elements.Segment, twiss: Union[str, <MagicMock name='mock.DataFrame' id='140260823939224'>]*) → None

Update the given lattice with the attributes specified in the given TWISS data.

Parameters

- **lattice** (Segment) –
- **twiss** (*str or pd.DataFrame*) – File name pointing to the TWISS file or equivalent data frame (such as returned by *dipas.madx.utils.convert_tfs()*).

dipas.build.**update_from_paramodi**(*lattice: dipas.elements.Segment, paramodi: Union[str, <MagicMock name='mock.DataFrame' id='140260823939224'>]*) → None

Apply the specified paramodi definitions to the given lattice in-place.

The following paramodi specifications are currently supported (others are ignored):

- **[SBend]** * hkick - Increases/Decreases the SBend's angle attribute.
- **[Quadrupole]** * kl - Replaces the Quadrupole's k1 attribute.
- **[HKicker]** * hkick - Replaces the HKicker's kick attribute.
- **[VKicker]** * vkick - Replaces the VKicker's kick attribute.

Parameters

- **lattice** (Segment) –
- **paramodi** (*str or pd.DataFrame*) – File name pointing to the paramodi file or data frame with layout corresponding to external.Paramodi.parse().

dipas.build.**create_script**(*beam: dict, *, sequence: Union[str, dipas.elements.Segment], errors: Union[typing_extensions.Literal[True][True], dipas.elements.Segment, str] = '', track: str = ''*) → str

Create a MADX script that can be used for particle tracking in the given sequence.

Note: The *sequence* string must start with the sequence's label.

Parameters

- **beam** (*dict*) – Beam configuration that will be transformed to the "beam" command.
- **sequence** (str or Segment) – Part of the script describing the sequence.
- **errors** (True or Segment or str) – Part of the script describing error definitions. If *True* then *sequence* must be a Segment and it will be used to generate the error specifications.
- **track** (*str*) – Part of the script describing the tracking.

Returns **script** – The compound MADX script.

Return type str

Raises **SerializerError** – If the *sequence* string does not start with a label.

dipas.build.**sequence_script**(*lattice: dipas.elements.Segment*, *label: str = 'seq'*, *, *markup: str = 'madx'*) → str

Convert the given lattice to a corresponding MADX sequence script or HTML file.

Important: The sequence must not assume implicit drift spaces; elements are laid out as presented.

Parameters

- **lattice** (Segment) – The lattice to be converted. Elements are placed one after another (no implicit drifts).
- **label** (*str, optional*) – The label of the sequence to be used in the script.
- **markup** (*str, optional*) – The markup language which is used for dumping the sequence; one of {"madx", "html"}.

Returns **script**

Return type str

dipas.build.**error_script**(*lattice: dipas.elements.Segment*) → str

Convert error definitions in form of *AlignmentError* to a corresponding MADX script.

Important: Elements which have associated errors must have a (unique) label (uniqueness is not checked for).

Parameters **lattice** (Segment) –

Returns **script** – The corresponding MADX statements for assigning the associated errors.

Return type str

Raises **SerializerError** – If an element with associated errors has no label.

dipas.build.**track_script**(*particles: Union[<MagicMock name='mock.ndarray' id='140260824044432'>, <MagicMock name='mock.Tensor' id='140260824000440'>], observe: Sequence[str], aperture: bool = True, recloss: bool = True, turns: int = 1, maxaper: Union[tuple, list] = (0.1, 0.01, 0.1, 0.01, 1.0, 0.1)*) → str

Convert particle array / tensor to corresponding MADX track script.

Uses `onetable = true` and hence the results will be available at the file "trackone".

Parameters

- **particles** (*array*) – Array / tensor of shape *(6, N)* where *N* is the number of particles.
- **observe** (*list or tuple*) – Labels of places where to observe.
- **aperture** (*bool*) –
- **recloss** (*bool*) –
- **turns** (*int*) –
- **maxaper** (*tuple or list*) –

Returns **script**

Return type str

Raises **ValueError** – If the given particle array has an illegal shape (must be (6,N) where *N* is the number of particles).

dipas.compute module

Convenience functions for computing various simulation related quantities.

dipas.compute.**orm**(*lattice: dipas.elements.Segment, *, kickers: Union[str, Pattern, Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], Sequence[Union[int, str, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern, Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int]]]], monitors: Union[str, Pattern, Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], Sequence[Union[int, str, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern, Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int]]]], kicks: Tuple[float, float] = (-0.001, 0.001), order: typing_extensions.Literal[1, 2][1, 2] = 2, co_args: Optional[dict] = None*) → Tuple[<MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.Tensor' id='140260824000440'>]

Compute the orbit response matrix (ORM) for the given lattice, kickers and monitors.

Parameters

- **lattice** (*Segment*) –
- **kickers** (*MultiElementSelector or list of SingleElementSelector*) – Can be an identifier for selecting multiple kickers, or a list of identifiers each selecting a single kicker.
- **monitors** (*MultiElementSelector or list of SingleElementSelector*) – Can be an identifier for selecting multiple monitors, or a list of identifiers each selecting a single monitor.
- **kicks** (*2-tuple of float*) – The kick strengths to be used for measuring the orbit response.
- **order** (*int*) – See *closed_orbit()*.
- **co_args** (*dict*) – Additional arguments for the closed orbit search.

Returns **orm_x, orm_y** – Shape *len(kickers), len(monitors)*.

Return type Tensor

dipas.compute.**closed_orbit**(*lattice: dipas.elements.Segment*, *, *order: typing_extensions.Literal[1, 2][1, 2] = 2*, *max_iter: Optional[int] = None*, *tolerance: float = 1e-06*, *initial_guess: Optional[<MagicMock name='mock.Tensor' id='140260824000440'>] = None*, *return_transfer_matrix: bool = False*) → Union[<MagicMock name='mock.Tensor' id='140260824000440'>, Tuple[<MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.Tensor' id='140260824000440'>]]

Closed orbit search for a given order on the given lattice.

The given lattice may contain *Element's as well as 'AlignmentError's. Alignment errors are treated as additional elements that wrap the actual element: entrance transformations coming before the element, in order, and exit transformations being placed after, in reverse order. The closed orbit search is a first-order iterative*

procedure with where each update :math:‘x_{Delta} is computed as the solution of the following set of linear equations:

```
.. math:: \left[\mathbb{1} - R\right]\, x_{\Delta} = x_1 - x_0
```

where R is the one-turn transfer matrix and x_1 is the orbit after one turn when starting from x_0. R represents the Jacobian of the orbit w.r.t. itself.

Parameters

- **lattice** (Segment) –
- **order** (*int*) – Transfer maps used for tracking the closed orbit guess are truncated at the specified order (however the linear response *R* does contain second feed-down terms from *T*, if any).
- **max_iter** (*int, optional*) – Maximum number of iterations.
- **tolerance** (*float, optional*) – Maximum L1 distance between initial orbit and tracked orbit after one turn for convergence.
- **initial_guess** (*Tensor*) – Initial guess for the closed orbit search; must be of shape (6,1).
- **return_transfer_matrix** (*bool, default = False*) – If *True* then the one-turn transfer matrix is returned along the computed closed orbit as a tuple: `x, R`. Note that during the closed orbit search, after the deviation reached below the specified *threshold*, one additional update to the closed orbit *x* is performed but not to the transfer matrix *R*. For that reason the matrix *R* might deviate slightly from the matrix obtained by `elements.process_transfer_maps()` when using the returned orbit *x*.

Returns

- **closed_orbit** (*Tensor*) – Tensor of shape (6,) containing the closed orbit in the transverse coordinates and zeros for the longitudinal coordinates.
- **one_turn_transfer_matrix** (*Tensor, optional*) – Tensor of shape (6, 6) representing the one-turn transfer matrix. This is only returned if the argument for *return_transfer_matrix* is set to *True*.

Raises **ConvergenceError** – If the closed orbit search did not converge within the specified number of iterations for the given tolerance.

dipas.compute.**linear_closed_orbit**(*lattice: dipas.elements.Segment*) → <MagicMock name='mock.Tensor' id='140260824000440'>

Compute the linear closed orbit for the given lattice.

The given lattice may contain *Element‘s as well as ‘AlignmentError‘s. Alignment errors are treated as additional elements that wrap the actual element: entrance transformations coming before the element, in order, and exit transformations being placed after, in reverse order. Hence all parts of the lattice can be described as a chain of linear transformations and the linear closed orbit is given as the solution to the following system of equations (in the transverse coordinates, for a total of :math:‘n* elements (actual elements and error transformations)):

$$\left[\mathbb{1} - \bar{R}_0\right]\, x_{co} = \sum_{i=1}^{n} \bar{R}_i\, d_i$$

where $\bar{R}_i$ is given by:

$$\bar{R}_i \equiv \prod_{j=n}$$

$$j \to j - 1^{i+1} R_j$$

and R_k, d_k are, respectively, the first and zero order term of the k-th element.

Parameters **lattice** (Segment) –

Returns **linear_closed_orbit** – Tensor of shape (6,) containing the closed orbit in the transverse coordinates and zeros for the longitudinal coordinates.

Return type Tensor

dipas.compute.**twiss**(*lattice: dipas.elements.Segment*, ***, *order: typing_extensions.Literal[1, 2][1, 2] = 2*, *co_args: Optional[Dict] = None*, *initial: Optional[dipas.compute.InitialLatticeParameters] = None*) → Dict

Compute various lattice functions and parameters.

Parameters

- **lattice** (Segment) –
- **order** (*int*) – Order of truncation for the transfer maps of lattice elements.
- **co_args** (*dict*) – Keyword arguments for *closed_orbit()*.
- **initial** (InitialLatticeParameters) – Initial lattice parameters to be used instead of the values from the periodic solution. If this argument is provided the periodic solution won't be computed; all values are taken from the *initial* specification.

Returns

twiss –

Contains the following key-value pairs:

- "Q1" – first mode tune
- "Q2" – second mode tune
- "coupling_matrix" – the 2x2 coupling matrix
- **"lattice" – a data frame with element labels as indices and the following columns:**
  - "x", "y", "px", "py" – the values of the closed orbit.
  - "bx", "ax", "mx", "by", "ay", "my", "dx", "dpx", "dy", "dpy" – linear lattice functions beta and alpha as well as phase advance and dispersion for the two modes; the phase advance is given in units of [2*pi].

Return type dict

Raises **UnstableLatticeError** –

dipas.elements module

Accelerator lattice elements represented by PyTorch Modules.

The various element classes are made available together with the corresponding MADX command name in the following dicts.

dipas.elements.**elements**

Maps MADX command names to corresponding *Element* backend classes.

Type dict

dipas.elements.**alignment_errors**

Maps MADX EALIGN command attributes to corresponding *AlignmentError* backend classes.

Type dict

dipas.elements.**aperture_types**
Maps MADX `apertypes` definitions to corresponding *Aperture* backend classes.

Type dict

class dipas.elements.**ApertureCircle**(*aperture: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>] = <MagicMock name='mock.tensor()' id='140260823811296'>, padding: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>] = <MagicMock name='mock.tensor()' id='140260823811296'>, offset: Union[List[Union[int, float]], Tuple[Union[int, float], ...], <MagicMock name='mock.Tensor' id='140260824000440'>] = <MagicMock name='mock.zeros()' id='140260823781840'>*)
Bases: dipas.elements.Aperture

Circular aperture.

Parameters **aperture** (*Tensor, scalar (shape [])*) – Radius of the circle.

classmethod loss(*xy: <MagicMock name='mock.Tensor' id='140260824000440'>*, *aperture: <MagicMock name='mock.Tensor' id='140260824000440'>*) → <MagicMock name='mock.Tensor' id='140260824000440'>
Compute the loss values for the given xy-positions.

Parameters

- **xy** (*Tensor*) – Tensor of shape *(2, N)* where *N* is the number of particles and rows are x- and y-positions, respectively, centered within the aperture.
- **aperture** (*Tensor*) – The effective aperture of the element, interpreted by the particular subclass.

Returns **loss_val** – Tensor of shape *(N,)*, zero where positions are inside the aperture (including exactly at the aperture) and greater than zero where positions are outside the aperture.

Return type Tensor

class dipas.elements.**ApertureEllipse**(*aperture: Union[List[Union[int, float]], Tuple[Union[int, float], ...], <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>] = <MagicMock name='mock.tensor()' id='140260823811296'>, padding: Union[List[Union[int, float]], Tuple[Union[int, float], ...], <MagicMock name='mock.Tensor' id='140260824000440'>] = <MagicMock name='mock.zeros()' id='140260823781840'>, offset: Union[List[Union[int, float]], Tuple[Union[int, float], ...], <MagicMock name='mock.Tensor' id='140260824000440'>] = <MagicMock name='mock.zeros()' id='140260823781840'>*)
Bases: dipas.elements.Aperture

Elliptical aperture.

Parameters **aperture** (*Tensor, shape (2,)*) – Horizontal and vertical semi-axes of the ellipse.

classmethod loss(*xy: <MagicMock name='mock.Tensor' id='140260824000440'>*, *aperture: <MagicMock name='mock.Tensor' id='140260824000440'>*) → <MagicMock name='mock.Tensor' id='140260824000440'>

Compute the loss values for the given xy-positions.

Parameters

- **xy** (*Tensor*) – Tensor of shape *(2, N)* where *N* is the number of particles and rows are x- and y-positions, respectively, centered within the aperture.
- **aperture** (*Tensor*) – The effective aperture of the element, interpreted by the particular subclass.

Returns **loss_val** – Tensor of shape *(N,)*, zero where positions are inside the aperture (including exactly at the aperture) and greater than zero where positions are outside the aperture.

Return type Tensor

class dipas.elements.**ApertureRectangle**(*aperture: Union[List[Union[int, float]], Tuple[Union[int, float], ...], <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>] = <MagicMock name='mock.tensor()' id='140260823811296'>, padding: Union[List[Union[int, float]], Tuple[Union[int, float], ...], <MagicMock name='mock.Tensor' id='140260824000440'>] = <MagicMock name='mock.zeros()' id='140260823781840'>, offset: Union[List[Union[int, float]], Tuple[Union[int, float], ...], <MagicMock name='mock.Tensor' id='140260824000440'>] = <MagicMock name='mock.zeros()' id='140260823781840'>*)

Bases: dipas.elements.Aperture

Rectangular aperture.

Parameters **aperture** (*Tensor, shape (2,)*) – Half width (horizontal) and half height (vertical) of the rectangle.

classmethod loss(*xy: <MagicMock name='mock.Tensor' id='140260824000440'>*, *aperture: <MagicMock name='mock.Tensor' id='140260824000440'>*) → <MagicMock name='mock.Tensor' id='140260824000440'>

Compute the loss values for the given xy-positions.

Parameters

- **xy** (*Tensor*) – Tensor of shape *(2, N)* where *N* is the number of particles and rows are x- and y-positions, respectively, centered within the aperture.
- **aperture** (*Tensor*) – The effective aperture of the element, interpreted by the particular subclass.

Returns **loss_val** – Tensor of shape *(N,)*, zero where positions are inside the aperture (including exactly at the aperture) and greater than zero where positions are outside the aperture.

Return type Tensor

class dipas.elements.**ApertureRectEllipse**(*aperture: Union[List[Union[int, float]], Tuple[Union[int, float], ...], <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>] = <MagicMock name='mock.tensor()' id='140260823811296'>, padding: Union[List[Union[int, float]], Tuple[Union[int, float], ...], <MagicMock name='mock.Tensor' id='140260824000440'>] = <MagicMock name='mock.zeros()' id='140260823781840'>, offset: Union[List[Union[int, float]], Tuple[Union[int, float], ...], <MagicMock name='mock.Tensor' id='140260824000440'>] = <MagicMock name='mock.zeros()' id='140260823781840'>*)

Bases: dipas.elements.Aperture

Overlay of rectangular and elliptical aperture.

Parameters **aperture** (*Tensor, shape (4,)*) – Half width (horizontal) and half height (vertical) of the rectangle followed by horizontal and vertical semi-axes of the ellipse.

classmethod loss(*xy: <MagicMock name='mock.Tensor' id='140260824000440'>, aperture: <MagicMock name='mock.Tensor' id='140260824000440'>*) → <MagicMock name='mock.Tensor' id='140260824000440'>

Compute the loss values for the given xy-positions.

Parameters

- **xy** (*Tensor*) – Tensor of shape *(2, N)* where *N* is the number of particles and rows are x- and y-positions, respectively, centered within the aperture.
- **aperture** (*Tensor*) – The effective aperture of the element, interpreted by the particular subclass.

Returns **loss_val** – Tensor of shape *(N,)*, zero where positions are inside the aperture (including exactly at the aperture) and greater than zero where positions are outside the aperture.

Return type Tensor

class dipas.elements.**Marker**(***kwargs*)

Bases: dipas.elements.Element

Marker element.

exact(*x: <MagicMock name='mock.Tensor' id='140260824000440'>*) → <MagicMock name='mock.Tensor' id='140260824000440'>

Exact analytic solution for tracking through the element.

linear(*x: <MagicMock name='mock.Tensor' id='140260824000440'>*) → <MagicMock name='mock.Tensor' id='140260824000440'>

Linear tracking through the element.

class dipas.elements.**Drift**(*l: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>], *, beam: dict, **kwargs*)

Bases: dipas.elements.Element

Drift space.

Note: Unlike in MADX, drift spaces feature aperture checks here.

Parameters **l** (*Number*) – Length of the drift [m].

exact(*x*)
Exact analytic solution for tracking through the element.

makethin(*n: int*, *, *style: Optional[str] = None*) → dipas.elements.Drift
Drift spaces are not affected by *makethin*.

update_transfer_map(*, *reset=False*) → None
Update the element's transfer map coefficients (d, R, T) to correspond to the elements' parameters.

This method should be called after any of the elements' parameters have been modified in order to update the transfer map to correspond to the new values.

Parameters **reset** (*bool*) – If True then the tensors corresponding to transfer map coefficients are replaced by new tensor objects before their values are updated. In this case the resulting tensors guarantee to reflect any change in the elements attribute, parametrized and non-parametrized (see notes below).

Notes

This method guarantees to incorporate the values of all parameters but not for non-parameter attributes (such as the length of an element; these are only guaranteed to be incorporated if `reset=True`). For `reset=False` these non-parameter attributes might be reflected in the transfer map update but this is an implementation detail that should not be relied on. Even though this method is only relevant for elements that cache their transfer map and for others it is a no-op, this caching property also is an implementation detail that should not be relied on. Hence after having modified an element the *update_transfer_map* method should be called.

class dipas.elements.**Instrument**(*l: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>], *, beam: dict, **kwargs*)
Bases: *dipas.elements.Drift*

A place holder for any type of beam instrumentation.

class dipas.elements.**Placeholder**(*l: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>], *, beam: dict, **kwargs*)
Bases: *dipas.elements.Drift*

A place holder for any type of element.

class dipas.elements.**Monitor**(*l: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>], *, beam: dict, **kwargs*)
Bases: *dipas.elements.Drift*

Beam position monitor.

class dipas.elements.**HMonitor**(*l: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>], *, beam: dict, **kwargs*)
Bases: *dipas.elements.Monitor*

Beam position monitor for measuring horizontal beam position.

class dipas.elements.**VMonitor**(*l: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>], *, beam: dict, **kwargs*)
Bases: *dipas.elements.Monitor*

Beam position monitor for measuring horizontal beam position.

class dipas.elements.**Kicker**(*hkick: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>] = 0, vkick: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>] = 0, l: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>] = 0, *, beam: Optional[dict] = None, **kwargs*)

Bases: dipas.elements.Element

Combined horizontal and vertical kicker magnet.

Parameters

- **hkick** (*Number or Parameter*) – Horizontal kick [rad].
- **vkick** (*Number or Parameter*) – Vertical kick [rad].

property d

linear(*x: <MagicMock name='mock.Tensor' id='140260824000440'>*) → <MagicMock name='mock.Tensor' id='140260824000440'>

Linear tracking through the element.

reset_transfer_map()

second_order(*x: <MagicMock name='mock.Tensor' id='140260824000440'>*) → <MagicMock name='mock.Tensor' id='140260824000440'>

Second order tracking through the element.

property transfer_map

class dipas.elements.**HKicker**(*kick: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>], l: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>] = 0, *, beam: Optional[dict] = None, **kwargs*)

Bases: *dipas.elements.Kicker*

Horizontal kicker magnet.

property kick

class dipas.elements.**VKicker**(*kick: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>], l: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>] = 0, *, beam: Optional[dict] = None, **kwargs*)

Bases: *dipas.elements.Kicker*

Vertical kicker magnet.

property kick

class dipas.elements.**TKicker**(*hkick: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>] = 0, vkick: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>] = 0, l: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>] = 0, *, beam: Optional[dict] = None, **kwargs*)

Bases: *dipas.elements.Kicker*

Similar to *Kicker* (see Chapter 10.12, MADX User's Guide).

class dipas.elements.**Quadrupole**(*k1: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>], l: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>], *, beam: dict, dk1: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>] = 0, **kwargs*)

Bases: dipas.elements.Element

Quadrupole magnet.

Whether this is a (horizontally) focusing or defocusing magnet is determined from the value of *k1* (k1 > 0 indicates a horizontally focusing quadrupole). Hence, in case *k1* is a Parameter, it always must be non-zero. For that reason it is convenient to use boundaries e.g. *[eps, k1_max]* with a small number *eps* (e.g. *1e-16*) and clip the value of *k1* accordingly. If *k1* is not a Parameter it can be zero and a corresponding *Drift* transformation will be used.

Parameters

- **k1** (*Number or Parameter*) – Normalized quadrupole gradient strength [1/m^2].
- **dk1** (*Number or Parameter*) – Absolute error of *k1* [1/m^2].

field_errors = {'k1': 'dk1'}

makethin(*n: int, *, style: Optional[str] = None*) → Union[dipas.elements.Quadrupole, dipas.elements.ThinElement]

Transform element to a sequence of *n* thin elements using the requested slicing style.

Important: If optimization of parameters is required then it is important to call *makethin* before *each* forward pass in order to always use the up-to-date parameter values from the original elements. Calling *makethin* only once at the beginning and then reusing the resulting sequence on every iteration would reuse the initial parameter values and hence make the optimization ineffective.

Parameters

- **n** (*int*) – The number of slices (thin elements).
- **style** (*str, optional*) – The slicing style to be used. For available styles see ThinElement.create_thin_sequence().

Returns A segment containing the slices (thin elements) separated by drifts.

Return type ThinElement

See also:

`ThinElement.create_thin_sequence()`

update_transfer_map(*, *reset=False*) → None

Update the element's transfer map coefficients (d, R, T) to correspond to the elements' parameters.

This method should be called after any of the elements' parameters have been modified in order to update the transfer map to correspond to the new values.

Parameters **reset** (*bool*) – If True then the tensors corresponding to transfer map coefficients are replaced by new tensor objects before their values are updated. In this case the resulting tensors guarantee to reflect any change in the elements attribute, parametrized and non-parametrized (see notes below).

Notes

This method guarantees to incorporate the values of all parameters but not for non-parameter attributes (such as the length of an element; these are only guaranteed to be incorporated if `reset=True`). For `reset=False` these non-parameter attributes might be reflected in the transfer map update but this is an implementation detail that should not be relied on. Even though this method is only relevant for elements that cache their transfer map and for others it is a no-op, this caching property also is an implementation detail that should not be relied on. Hence after having modified an element the *update_transfer_map* method should be called.

class dipas.elements.**Sextupole**(*k2: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>], l: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>], *, beam: dict, dk2: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>] = 0, **kwargs*)

Bases: `dipas.elements.Element`

Sextupole magnet.

Parameters

- **k2** (*Number or Parameter*) – Normalized sextupole coefficient [1/m^3].
- **dk2** (*Number or Parameter*) – Absolute error of *k2* [1/m^3].

field_errors = {'k2': 'dk2'}

makethin(*n: int*, *, *style: Optional[str] = None*) → Union[dipas.elements.Sextupole, dipas.elements.ThinElement]

Transform element to a sequence of *n* thin elements using the requested slicing style.

Important: If optimization of parameters is required then it is important to call *makethin* before *each* forward pass in order to always use the up-to-date parameter values from the original elements. Calling *makethin* only once at the beginning and then reusing the resulting sequence on every iteration would reuse the initial parameter values and hence make the optimization ineffective.

Parameters

- **n** (*int*) – The number of slices (thin elements).

- **style** (*str, optional*) – The slicing style to be used. For available styles see `ThinElement.create_thin_sequence()`.

Returns A segment containing the slices (thin elements) separated by drifts.

Return type ThinElement

See also:

`ThinElement.create_thin_sequence()`

update_transfer_map(*, *reset=False*) → None

Update the element's transfer map coefficients (d, R, T) to correspond to the elements' parameters.

This method should be called after any of the elements' parameters have been modified in order to update the transfer map to correspond to the new values.

Parameters **reset** (*bool*) – If True then the tensors corresponding to transfer map coefficients are replaced by new tensor objects before their values are updated. In this case the resulting tensors guarantee to reflect any change in the elements attribute, parametrized and non-parametrized (see notes below).

Notes

This method guarantees to incorporate the values of all parameters but not for non-parameter attributes (such as the length of an element; these are only guaranteed to be incorporated if `reset=True`). For `reset=False` these non-parameter attributes might be reflected in the transfer map update but this is an implementation detail that should not be relied on. Even though this method is only relevant for elements that cache their transfer map and for others it is a no-op, this caching property also is an implementation detail that should not be relied on. Hence after having modified an element the *update_transfer_map* method should be called.

class dipas.elements.**SBend**(*angle: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>], l: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>], e1: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>] = 0, e2: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>] = 0, fint: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, bool] = 0.0, fintx: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, None] = None, hgap: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>] = 0, h1: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>] = 0, h2: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>] = 0, *, beam: dict, dk0: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>] = 0, aperture: Optional[dipas.elements.Aperture] = None, label: Optional[str] = None, **kwargs*)

Bases: `dipas.elements.CompoundElement`

Sector bending magnet.

Parameters

- **angle** (*Number*) – Bending angle of the dipole [rad].
- **e1** (*Number*) – Rotation angle for the entrance pole face [rad]. `e1 = e2 = angle/2` turns an *SBEND* into a *RBEND*.

- **e2** (*Number*) – Rotation angle for the exit pole face [rad].
- **fint** (*Number*) – Fringing field integral at entrance. If *fintx* is not specified then *fint* is also used at the exit.
- **fintx** (*Number*) – Fringing field integral at exit.
- **hgap** (*Number*) – Half gap of the magnet [m].
- **h1** (*Number*) – Curvature of the entrance pole face [1/m].
- **h2** (*Number*) – Curvature of the exit pole face [1/m].

property angle

property dk0

property e1

property e2

field_errors = {'k0': 'dk0'}

property fint

property fintx

flatten() → Iterator[dipas.elements.SBend]

Retrieve a flat representation of the segment (with sub-segments flattened as well).

property h1

property h2

property hgap

property k0

class dipas.elements.**RBend**(*angle: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>], l: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>], e1: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>] = 0, e2: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>] = 0, fint: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, bool] = 0.0, fintx: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, None] = None, hgap: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>] = 0, h1: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>] = 0, h2: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>] = 0, *, beam: dict, dk0: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>] = 0, aperture: Optional[dipas.elements.Aperture] = None, label: Optional[str] = None, **kwargs*)

Bases: *dipas.elements.SBend*

Sector bending magnet with parallel pole faces (see *SBend*).

class dipas.elements.**Dipedge**(*h: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>], e1: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>], fint: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>], hgap: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>], he: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>] = 0, *, entrance: bool = True, **kwargs*)

Bases: dipas.elements.Element

Fringing fields at the entrance and exit of dipole magnets.

Parameters

- **h** (*Number*) – Curvature of the associated dipole magnet body.
- **e1** (*Number*) – The rotation angle of the pole face.
- **fint** (*Number*) – The fringing field integral.
- **hgap** (*Number*) – The half gap height of the associated dipole magnet.
- **he** (*Number*) – The curvature of the pole face.

makethin(*n: int, *, style: Optional[str] = None*) → Union[dipas.elements.Element, dipas.elements.ThinElement]

Transform element to a sequence of *n* thin elements using the requested slicing style.

Important: If optimization of parameters is required then it is important to call *makethin* before *each* forward pass in order to always use the up-to-date parameter values from the original elements. Calling *makethin* only once at the beginning and then reusing the resulting sequence on every iteration would reuse the initial parameter values and hence make the optimization ineffective.

Parameters

- **n** (*int*) – The number of slices (thin elements).
- **style** (*str, optional*) – The slicing style to be used. For available styles see ThinElement.create_thin_sequence().

Returns A segment containing the slices (thin elements) separated by drifts.

Return type ThinElement

See also:

ThinElement.create_thin_sequence()

update_transfer_map(*, *reset=False*) → None

Update the element's transfer map coefficients (d, R, T) to correspond to the elements' parameters.

This method should be called after any of the elements' parameters have been modified in order to update the transfer map to correspond to the new values.

Parameters **reset** (*bool*) – If True then the tensors corresponding to transfer map coefficients are replaced by new tensor objects before their values are updated. In this case the resulting tensors guarantee to reflect any change in the elements attribute, parametrized and non-parametrized (see notes below).

Notes

This method guarantees to incorporate the values of all parameters but not for non-parameter attributes (such as the length of an element; these are only guaranteed to be incorporated if `reset=True`). For `reset=False` these non-parameter attributes might be reflected in the transfer map update but this is an implementation detail that should not be relied on. Even though this method is only relevant for elements that cache their transfer map and for others it is a no-op, this caching property also is an implementation detail that should not be relied on. Hence after having modified an element the *update_transfer_map* method should be called.

class dipas.elements.**ThinQuadrupole**(*k1l: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>], *, dk1l: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>] = 0, **kwargs*)

Bases: `dipas.elements.Element`

Thin lens representation of a quadrupole magnet.

Parameters

- **k1l** (*Number or Parameter*) – Integrated quadrupole gradient strength [1/m].
- **dk1l** (*Number or Parameter*) – Absolute error of *k1l* [1/m].

field_errors = {'k1l': 'dk1l'}

makethin(*n: int*, *, *style: Optional[str] = None*) → NoReturn

Transform element to a sequence of *n* thin elements using the requested slicing style.

Important: If optimization of parameters is required then it is important to call *makethin* before *each* forward pass in order to always use the up-to-date parameter values from the original elements. Calling *makethin* only once at the beginning and then reusing the resulting sequence on every iteration would reuse the initial parameter values and hence make the optimization ineffective.

Parameters

- **n** (*int*) – The number of slices (thin elements).
- **style** (*str, optional*) – The slicing style to be used. For available styles see `ThinElement.create_thin_sequence()`.

Returns A segment containing the slices (thin elements) separated by drifts.

Return type ThinElement

See also:

`ThinElement.create_thin_sequence()`

update_transfer_map(*, *reset=False*) → None

Update the element's transfer map coefficients (d, R, T) to correspond to the elements' parameters.

This method should be called after any of the elements' parameters have been modified in order to update the transfer map to correspond to the new values.

Parameters **reset** (*bool*) – If True then the tensors corresponding to transfer map coefficients are replaced by new tensor objects before their values are updated. In this case the

resulting tensors guarantee to reflect any change in the elements attribute, parametrized and non-parametrized (see notes below).

Notes

This method guarantees to incorporate the values of all parameters but not for non-parameter attributes (such as the length of an element; these are only guaranteed to be incorporated if `reset=True`). For `reset=False` these non-parameter attributes might be reflected in the transfer map update but this is an implementation detail that should not be relied on. Even though this method is only relevant for elements that cache their transfer map and for others it is a no-op, this caching property also is an implementation detail that should not be relied on. Hence after having modified an element the *update_transfer_map* method should be called.

class dipas.elements.**ThinSextupole**(*k2l: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>], *, dk2l: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>] = 0, **kwargs*)

Bases: `dipas.elements.Element`

Thin lens representation of a sextupole magnet.

Parameters

- **k2l** (*Number or Parameter*) – Integrated sextupole coefficient [1/m^2].
- **dk2l** (*Number or Parameter*) – Absolute error of *k2l* [1/m^2].

field_errors = {'k2l': 'dk2l'}

makethin(*n: int*, *, *style: Optional[str] = None*) → NoReturn

Transform element to a sequence of *n* thin elements using the requested slicing style.

Important: If optimization of parameters is required then it is important to call *makethin* before *each* forward pass in order to always use the up-to-date parameter values from the original elements. Calling *makethin* only once at the beginning and then reusing the resulting sequence on every iteration would reuse the initial parameter values and hence make the optimization ineffective.

Parameters

- **n** (*int*) – The number of slices (thin elements).
- **style** (*str, optional*) – The slicing style to be used. For available styles see `ThinElement.create_thin_sequence()`.

Returns A segment containing the slices (thin elements) separated by drifts.

Return type ThinElement

See also:

`ThinElement.create_thin_sequence()`

update_transfer_map(*, *reset=False*) → None

Update the element's transfer map coefficients (d, R, T) to correspond to the elements' parameters.

This method should be called after any of the elements' parameters have been modified in order to update the transfer map to correspond to the new values.

Parameters **reset** (*bool*) – If True then the tensors corresponding to transfer map coefficients are replaced by new tensor objects before their values are updated. In this case the resulting tensors guarantee to reflect any change in the elements attribute, parametrized and non-parametrized (see notes below).

Notes

This method guarantees to incorporate the values of all parameters but not for non-parameter attributes (such as the length of an element; these are only guaranteed to be incorporated if `reset=True`). For `reset=False` these non-parameter attributes might be reflected in the transfer map update but this is an implementation detail that should not be relied on. Even though this method is only relevant for elements that cache their transfer map and for others it is a no-op, this caching property also is an implementation detail that should not be relied on. Hence after having modified an element the *update_transfer_map* method should be called.

class dipas.elements.**Tilt**(*target: Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, dipas.elements.AlignmentError], psi: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>] = 0*)

Bases: *dipas.elements.LongitudinalRoll*

The tilt of an element represents the roll about the longitudinal axis.

Note: MADX uses a right-handed coordinate system (see Fig. 1.1, MADX User's Guide), therefore `x = x*cos(psi) - y*sin(psi)` describes a clockwise rotation of the trajectory.

psi

Rotation angle about s-axis.

Type Number or Parameter

Notes

Tilt is only a subclass of *AlignmentError* for technical reasons and has no meaning beyond that. A *Tilt* is not considered an alignment error from the simulation point of view.

triggers = ('tilt',)

class dipas.elements.**Offset**(*target: Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, dipas.elements.AlignmentError], dx: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>] = 0, dy: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>] = 0*)

Bases: dipas.elements.AlignmentError

AlignmentError representing the xy-offset of an element.

dx

Horizontal offset.

Type Number or Parameter

dy
Vertical offset.

Type Number or Parameter

property d_enter

property d_exit

triggers = ('dx', 'dy')

class dipas.elements.**LongitudinalRoll**(*target: Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, dipas.elements.AlignmentError], psi: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>] = 0*)

Bases: dipas.elements.AlignmentError

AlignmentError representing the roll about the longitudinal axis of an element.

Note: MADX uses a right-handed coordinate system (see Fig. 1.1, MADX User's Guide), therefore `x = x*cos(psi) - y*sin(psi)` describes a clockwise rotation of the trajectory.

psi
Rotation angle about s-axis.

Type Number or Parameter

property R_enter

property R_exit

triggers = ('dpsi',)

class dipas.elements.**BPMError**(*target: Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, dipas.elements.AlignmentError], ax: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>] = 0, ay: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>] = 0, rx: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>] = 0, ry: Union[int, float, <MagicMock name='mock.Tensor' id='140260824000440'>, <MagicMock name='mock.nn.Parameter' id='140260823579000'>] = 0*)

Bases: dipas.elements.AlignmentError

BPM readout errors.

The actual BPM reading is computed as (for both horizontal and vertical plane separately): `rx * (x + ax)`.

ax
Horizontal absolute read error.

Type Tensor or `Parameter`

ay
Vertical absolute read error.

Type Tensor or `Parameter`

rx
Horizontal relative read error.

Type Tensor or `Parameter`

ry
Vertical relative read error.

Type Tensor or `Parameter`

enter(*x: <MagicMock name='mock.Tensor' id='140260824000440'>*) → <MagicMock name='mock.Tensor' id='140260824000440'>
Applies linear coordinate transformation at the entrance of the wrapped element.

exit(*x: <MagicMock name='mock.Tensor' id='140260824000440'>*) → <MagicMock name='mock.Tensor' id='140260824000440'>
Applies linear coordinate transformation at the exit of the wrapped element.

readout(*x: <MagicMock name='mock.Tensor' id='140260824000440'>*) → <MagicMock name='mock.Tensor' id='140260824000440'>
Return BPM readings for the given coordinates.

Parameters **x** (*Tensor*) – 6D phase-space coordinates of shape (6, N).

Returns **xy** – BPM readings in x- and y-dimension of shape (2, N).

Return type Tensor

triggers = ('mrex', 'mrey', 'mscalx', 'mscaly')

class dipas.elements.**Segment**(*elements: Sequence[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]*)

Bases: `object`

Wrapper class representing a sequence of elements (possibly a segment of the lattice).

Elements or sub-segments can be selected via *__getitem__*, i.e. `segment[item]` notation. Here *item* can be one of the following:

- int - indicating the index position in the segment.
- str - will be compared for equality against element labels; if a single element with that label is found it is returned otherwise all elements with that label are returned as a list. An exception are strings containing an asterisk which will be interpreted as a shell-style wildcard and converted to a corresponding regex Pattern.
- re.Pattern - will be matched against element labels; a list of all matching elements is returned.
- instance of `Element` or `AlignmentError` - the element itself (possibly wrapped by other AlignmentError instances is returned.
- subclass of `Element` or `AlignmentError` - a list of elements of that type (possibly wrapped by AlignmentError instances) is returned.
- tuple - must contain two elements, the first being one of the above types and the second an integer; the first element is used to select a list of matching elements and the second integer element is used to select the corresponding element from the resulting list.

- slice - start and stop indices can be any of the above types that selects exactly a single element or None; a corresponding sub-Segment is returned. The *step* parameter of the slice is ignored. In case the stop marker is not None, the corresponding element is included in the selection.

An element of a segment can be updated by using *__setitem__*, i.e. `segment[item] = ...` notation, where *item* can be any of the above types that selects exactly a single element.

elements

Type list of `LatticeElement`

compute_transfer_maps(*method: typing_extensions.Literal['accumulate', 'reduce', 'local'][accumulate, reduce, local], *, order: typing_extensions.Literal[1, 2][1, 2] = 2, index: Optional[typing_extensions.Literal[0, 1, 2][0, 1, 2]] = None, symplectify: bool = True, unfold_alignment_errors: bool = False, d0: Optional[<MagicMock name='mock.Tensor' id='140260824000440'>] = None, R0: Optional[<MagicMock name='mock.Tensor' id='140260824000440'>] = None, T0: Optional[<MagicMock name='mock.Tensor' id='140260824000440'>] = None*) → Iterator[Tuple[Optional[<MagicMock name='mock.Tensor' id='140260824000440'>], ...]]

Compute the transfer maps of the element.

Parameters

- **method** (*str*) – Determines how the transfer maps are computed (for details see implementing class).
- **order** (*int*) – Transfer map order used in computations.
- **index** (*int*) – Only transfer map coefficients up to the specified *index* are stored in the results; others are *None*. If *None* then defaults to *order*.
- **symplectify** (*bool*) – Whether to symplectify the first order coefficients (transfer matrix) after second order feed-down terms have been included.
- **unfold_alignment_errors** (*bool*) – Whether to treat alignment error transformations as separate transfer maps rather than contracting them with the wrapped lattice element.
- **R0, T0** (*d0,*) – Starting values at the beginning of the element (`d0` corresponds to the closed orbit value).

Yields **transfer_map** (`TransferMap`) – Depending on the selected *method*.

exact(*x: <MagicMock name='mock.Tensor' id='140260824000440'>, **kwargs*) → Union[<MagicMock name='mock.Tensor' id='140260824000440'>, Tuple]

Exact tracking through the segment.

flat() → dipas.elements.Segment

Convenience function wrapping *Segment.flatten*.

flatten() → Iterator[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, dipas.elements.AlignmentError]]

Retrieve a flat representation of the segment (with sub-segments flattened as well).

forward(*x: <MagicMock name='mock.Tensor' id='140260824000440'>, *, method: Union[str, Callable[[<MagicMock name='mock.Tensor' id='140260824000440'>], <MagicMock name='mock.Tensor' id='140260824000440'>], Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], int, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], None], Union[str, Callable[[<MagicMock name='mock.Tensor' id='140260824000440'>], <MagicMock name='mock.Tensor' id='140260824000440'>]]], List[Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], int, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], None], Union[str, Callable[[<MagicMock name='mock.Tensor' id='140260824000440'>], <MagicMock name='mock.Tensor' id='140260824000440'>]]]], Dict[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], int, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], None], Union[str, Callable[[<MagicMock name='mock.Tensor' id='140260824000440'>], <MagicMock name='mock.Tensor' id='140260824000440'>]]]], aperture: bool = False, exact_drift: bool = True, observe: Union[bool, int, str, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern, Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], Pattern, Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], List[Union[int, str, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern, Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], Pattern, Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]]], None] = None, recloss: Union[bool, int, str, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern, Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], Pattern, Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], List[Union[int, str, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern, Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], Pattern, Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]]], None] = None, loss_func: Optional[Callable[[<MagicMock name='mock.Tensor' id='140260824000440'>], <MagicMock name='mock.Tensor' id='140260824000440'>]] = None*) → Union[<MagicMock name='mock.Tensor' id='140260824000440'>, Tuple]

Track the given particles through the segment.

Parameters

- **x** (*Tensor*) – Shape *(6, N)* where *N* is the number of particles.
- **method** (SelectionCriteria of str or callable) – Method name which will be used for the lattice elements to perform tracking.
- **aperture** (*bool*) – Determines whether aperture checks are performed (and thus particles marked lost / excluded from tracking).

- **exact_drift** (*bool*) – If true (default) then *Drift's will always be tracked through via 'exact*, no matter what *method* is.
- **observe** (sequence of {str or re.Pattern or subclass of *Element*}) – Indicates relevant observation points; the (intermediary) positions at these places will be returned. Items are matched against element labels (see function *match_element*).
- **recloss** (bool or "sum" or {str or re.Pattern or subclass of *Element*} or a sequence thereof) – If "sum" then the particle loss at each element will be recorded and summed into a single variable which will be returned. If a sequence is given it must contain element labels or regex patterns and the loss will be recorded only at the corresponding elements (similar to *observe* for positions). If *True* then the loss will be recorded at all elements; if *False* the loss is not recorded at all. A true value for this parameter will automatically set *aperture* to *True*.
- **loss_func** (*callable*) – This parameter can be used to supply a function for transforming the returned loss at each element. This can be useful if some variation of the loss is to be accumulated into a single tensor. The function receives the loss tensor as returned by *Aperture.loss* as an input and should return a tensor of any shape. If not supplied this function defaults to *torch.sum* if *recloss == "accumulate"* and the identity if `recloss == "history"`. Note that if a function is supplied it needs to cover all the steps, also the summation in case `recloss == "accumulate"` (the default only applies if no function is given).

Returns

- **x** (*Tensor*) – Shape *(6, M)* where *M* is the number of particles that reached the end of the segment (*M* can be different from *N* in case aperture checks are performed).
- **history** (*dict*) – The (intermediary) positions at the specified observation points. Keys are element labels and values are positions of shape *(6, M_i)* where *M_i* is the number of particles that reached that element. This is only returned if *observe* is true.
- **loss** (*Tensor or dict*) – If `recloss == "accumulate"` the loss value accumulated for each element (i.e. the sum of all individual loss values) is returned. Otherwise, if *recloss* is true, a dict mapping element labels to recorded loss values is returned.

get_element_index(*marker: Union[int, str, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern, Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int]]*) → int

property l

linear(*x: <MagicMock name='mock.Tensor' id='140260824000440'>, **kwargs*) → Union[<MagicMock name='mock.Tensor' id='140260824000440'>, Tuple]

Linear tracking through the segment.

makethin(*n: Union[int, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], int, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], None], int], List[Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], int, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], None], int]], Dict[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], int, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], None], int]], *, style: Union[str, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], int, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], None], str], List[Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], int, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], None], str]], Dict[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]], int, dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError, Tuple[Union[str, Pattern[AnyStr], Type[Union[dipas.elements.CompactElement, dipas.elements.PartitionedElement, AlignmentError]]], int], None], str], None] = None*) → dipas.elements.Segment

Retrieve a thin representation for each element of the segment except for those indicated by *exclude*.

Parameters

- **n** (`SelectionCriteria` of int) – Number of thin slices. If *int* this applies to all elements. Otherwise must key-value pairs. Keys should be element selectors (see `SingleElementSelector` and `MultiElementSelector`) or *None* for providing a default. Values should be the corresponding number of slices that are applied to the elements that match the key selector. Only the first matching key is considered. If a default value is desired it can be provided in various ways:

  1. Using `{None: default_value}`.
  2. Using a regex that matches any label (given that all elements have a *label* different from *None*).
  3. Using the class *Element*.

  Note that for all options these should appear at the very end of the list of criteria, otherwise they will override any subsequent definitions.

- **style** (`SelectionCriteria` of str) – Slicing style per element. Works similar to the *n* parameter. See `Element.makethin()` for more information about available slicing styles.

Returns Containing thin elements.

Return type *Segment*

See also:

find_matching_criterion() For details about how keys in *n* can be used to fine-tune the slice number per element.

Element.makethin() For available slicing styles.

second_order(*x: <MagicMock name='mock.Tensor' id='140260824000440'>, **kwargs*) → Union[<MagicMock name='mock.Tensor' id='140260824000440'>, Tuple]
Second order tracking through the segment.

transfer_maps(*method: typing_extensions.Literal['accumulate', 'reduce', 'local'][accumulate, reduce, local], *, order: typing_extensions.Literal[1, 2][1, 2] = 2, indices: Union[typing_extensions.Literal[0, 1, 2][0, 1, 2], Tuple[typing_extensions.Literal[0, 1, 2][0, 1, 2], ...], None] = None, symplectify: bool = True, labels: bool = False, unfold_alignment_errors: bool = False, d0: Optional[<MagicMock name='mock.Tensor' id='140260824000440'>] = None, R0: Optional[<MagicMock name='mock.Tensor' id='140260824000440'>] = None, T0: Optional[<MagicMock name='mock.Tensor' id='140260824000440'>] = None*) → Union[<MagicMock name='mock.Tensor' id='140260824000440'>, Tuple[Optional[<MagicMock name='mock.Tensor' id='140260824000440'>], ...], List[<MagicMock name='mock.Tensor' id='140260824000440'>], List[Tuple[Optional[<MagicMock name='mock.Tensor' id='140260824000440'>], ...]], List[Tuple[str, <MagicMock name='mock.Tensor' id='140260824000440'>]], List[Tuple[str, Tuple[Optional[<MagicMock name='mock.Tensor' id='140260824000440'>], ...]]]]
Process the transfer maps of the segment's elements according to the specified method.

Note: This does not automatically compute the closed orbit first in order to use it as a starting value for *d0*. In that case, the closed orbit has to be provided manually in form of the parameter *d0*.

Parameters

- **method** (*str in {'accumulate', 'reduce', 'local'}*) – If "accumulate" then the transfer maps are accumulated by contracting subsequent transfer maps. If "reduce" then the contracted transfer map for the complete segment is returned (this is identical to the last transfer map for "accumulate"). If "local" then the local transfer map of each element is computed, taking into consideration the value of the local closed orbit. The result contains the closed orbit at the exit of each element as the zeroth order coefficient.
- **order** (*int*) – The order up to which transfer map coefficients (d, R, T) are taken into account for the contraction of two subsequent transfer maps.
- **indices** (*int or tuple of int*) – Indicates the indices of the processed transfer maps which should be stored in the results. If a single number is given, all indices up to the specified number (inclusive) are considered. E.g. if `indices = 0` and `method = 'reduce'` then the result will be just `d` where `d` is the orbit computed to second order but the resulting first and second order coefficients of the reduced transfer maps are discarded. Discarding coefficients of higher orders in the (intermediary) results can save memory and compute time. Note that `max(indices) <= order` must be fulfilled.
- **symplectify** (*bool*) – Specifies whether the linear term (the transfer matrix) of lattice elements should be symplectified after the addition of second-order feed-down terms. This is only relevant for `order >= 2`.

- **labels** (*bool*) – If True then the element's labels are returned alongside their transfer maps as 2-tuples (for `method = "reduce"` this parameter is ignored). In case *unfold_alignment_errors* is True as well, an element's label is repeated for each alignment error map (at entrance and exit), e.g. for an offset element "e1" it will be `[..., ("e1", entrance_map), ("e1", element_map), ("e1", exit_map), ...]`.
- **unfold_alignment_errors** (*bool*) – If True then the entrance and exit transformations of alignment errors are considered as separate transfer maps rather than being contracted with their wrapped element's map. This increases the number of transfer maps as compared to the length of the segment.
- **d0** (*Tensor, optional*) – The value of the closed orbit at the beginning of the segment.
- **R0** (*Tensor, optional*) – Initial value for the first order terms.
- **T0** (*Tensor, optional*) – Initial value for the second order terms.

Returns

transfer_maps –

Depending on the value of *method* either of the following is returned:

- "accumulate" – a list of the accumulated transfer maps along the lattice is returned.
- "reduce" – the single transfer map, corresponding to the whole segment, is returned.
- "local" – a list of the element-local transfer maps is returned.

Return type `TransferMap` or list of `TransferMap`

update_transfer_maps(*, *reset=False*) → None

Update the transfer map of each element (see `CompactElement.update_transfer_map()`).

dipas.external module

class dipas.external.**Paramodi**

Bases: `object`

classmethod apply_units(*paramodi: <MagicMock name='mock.DataFrame' id='140260823939224'>*) → List[Tuple]

Apply the specified units to the specified values.

If either the value is NaN or the unit is not available (`---`) the original value is used otherwise a corresponding quantity is computed. This uses the `pint` package for assigning the units.

Parameters **paramodi** (*pd.DataFrame*) – Such as from *parse()*.

Returns **quantities** – A list containing the indices and quantities wherever applicable and the original values otherwise for each row of the *paramodi* data frame.

Return type list of (index, pint.Quantity)

column_names = ['device', 'attribute', 'purpose', 'value', 'unit', 'parameter_name', '

classmethod parse(*f_name_or_text: str*) → <MagicMock name='mock.DataFrame' id='140260823939224'>

Parse the given paramodi file and return the data as a pandas data frame.

The resulting data frame has the following columns:

```
device, attribute, purpose, value, unit, parameter_name, original_value
```

where *device* and *attribute* are the original parameter name split at the first forward slash (`/`). *parameter_name* and *original_value* are the original representations of the parameter name and the parameter value. The first three columns serve as an index of the data frame.

Parameters **f_name_or_text** (*str*) – File name pointing to the paramodi file or the content of such a file.

Returns

Return type Data frame with the above described properties.

classmethod parse_line(*line: str*) → Iterator[Tuple[str]]
Parse the given line into the required column values.

classmethod update_madx_device_data(*devices: <MagicMock name='mock.DataFrame' id='140260823939224'>*, *paramodi: <MagicMock name='mock.DataFrame' id='140260823939224'>*) → Tuple[<MagicMock name='mock.DataFrame' id='140260823939224'>, dict]
Update the given MADX device data with values from the paramodi data.

This applies the following conversions:

- **[SBend]** * `angle` is incremented by `HKICK` from the paramodi data.
- **[Quadrupole]** * `k1` is replaced by `KL / q.L` where `KL` is from the paramodi data.
- **[HKicker, VKicker]** * `kick` is replaced by `HKICK` and `VKICK` respectively.

Parameters

- **devices** (*pd.DataFrame*) – Data frame containing the MADX device data. Indices should be device labels and match those in the first index level of the *paramodi* data frame. Columns should be device attributes (NaN where no such attribute is applicable). There must be a `"type"` column which indicates the device type as MADX command keyword (e.g. “quadrupole” or “sbend”).
- **paramodi** (*pd.DataFrame*) – Structure according to *parse()*.

Returns

- **updated_devices** (*pd.DataFrame*) – Data frame similar to *devices* with the relevant columns updated according to the above rules.
- **updates** (*dict*) – A dict containing the applied updates. Keys are element labels and values are dicts that map parameter names to their *new* (updated) value.

dipas.utils module

General purpose utility functions.

dipas.utils.**copy_doc**(*source*, *, *override: bool = False*)
Copy the docstring of one object to another.

dipas.utils.**format_doc**(***kwargs*)
Format the doc string of an object according to *str.format* rules.

dipas.utils.**func_chain**(**funcs*)

Return a partial object that, when called with an argument, will apply the given functions in order, using the output of the previous function as an input for the next (starting with the argument as the initial value).

dipas.utils.**pad_max_shape**(**arrays*, *before=None*, *after=1*, *value=0*, *tie_break=<MagicMock name='mock.floor' id='140260824015480'>*) → List[<MagicMock name='mock.ndarray' id='140260824044432'>]

Pad the given arrays with a constant values such that their new shapes fit the biggest array.

Parameters

- **arrays** (*sequence of arrays of the same rank*) –
- **after** (*before,*) – Similar to *np.pad -> pad_width* but specifies the fraction of values to be padded before and after respectively for each of the arrays. Must be between 0 and 1. If *before* is given then *after* is ignored.
- **value** (*scalar*) – The pad value.
- **tie_break** (*ufunc*) – The actual number of items to be padded _before_ is computed as the total number of elements to be padded times the *before* fraction and the actual number of items to be padded _after_ is the remainder. This function determines how the fractional part of the *before* pad width is treated. The actual *before* pad with is computed as `tie_break(N * before).astype(int)` where `N` is the total pad width. By default *tie_break* just takes the *np.floor* (i.e. attributing the fraction part to the *after* pad width). The after pad width is computed as `total_pad_width - before_pad_width`.

Returns **padded_arrays**

Return type list of arrays

dipas.utils.**remove_duplicates**(*seq: Sequence*, *op=<built-in function eq>*) → List

Remove duplicates from the given sequence according to the given operator.

Examples

```
>>> import operator as op
>>> remove_duplicates([1, 2, 3, 1, 2, 4, 1, 2, 5])
[1, 2, 3, 4, 5]
```

```
>>> from dataclasses import dataclass
>>> @dataclass
... class Foo:
...     n: int
...
>>> a, b, c = Foo(1), Foo(2), Foo(1)
>>> remove_duplicates([c, b, b, c, a, b, a])
[Foo(n=1), Foo(n=2)]
>>> remove_duplicates([c, b, b, c, a, b, a], op=op.is_)
[Foo(n=1), Foo(n=2), Foo(n=1)]
```

dipas.utils.**safe_math_eval**(*expr: str*, *locals_dict: dict = None*) → Any

Safe evaluation of mathematical expressions with name resolution.

The input string is converted to lowercase and any whitespace is removed. The expression is evaluated according to Python's evaluation rules (e.g. ** denotes exponentiation). Any names, again according to Python's naming rules, are resolved via the *locals_dict* parameter.

Parameters

- **expr** (*str*) – The mathematical expression to be evaluated.
- **locals_dict** (*dict*) – Used for name resolution.

Returns

Return type The value of the expression, in the context of names present in *locals_dict*.

Raises

- **TypeError** – If *expr* is not a string.
- **ValueError** – If the evaluation of *expr* is considered unsafe (see the source code for exact rules).
- **NameError** – If a name cannot be resolved via *locals_dict*.

Examples

```
>>> safe_math_eval('2 * 3 ** 4')
162
>>> import math
>>> safe_math_eval('sqrt(2) * sin(pi/4)', {'sqrt': math.sqrt, 'sin': math.sin, 'pi
↪': math.pi})
1.0
>>> safe_math_eval('2.0 * a + b', {'a': 2, 'b': 4.0})
8.0
```

dipas.utils.**setattr_multi**(*obj*, *names: Sequence[str]*, *values: Union[Sequence, Dict[str, Any]]*) → None

Set multiple attributes at once.

Parameters

- **obj** (*object*) –
- **names** (*sequence*) –
- **values** (*sequence or dict*) – If *dict* then it must map *names* to values.

class dipas.utils.**singledispatchmethod**(*func*)

Bases: object

register(*cls*, *method=None*)

6.1.3 Module contents

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W