ParticleFlow

Release 0.1.dev7+gf600fed

Dominik Vilsmeier

Sep 15, 2019

CONTENTS:

CHAPTER

ONE

INSTALLATION

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

On Windows it is recommended to use Anaconda or Miniconda for the Pyton 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 ParticeFlow (PF) 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* module attributes. Please refer to the module's documentation for more information. The MADX parser is fully contained in the *madx* module. 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 ParticleFlow will likely differ. For that reason is it important, when comparing results obtained with PF and MADX, to always generate a new MADX script from the particular PF lattice to ensure the same numerical values from random functions. If the original MADX script (from which the PF 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 PF 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 ParticleFlow 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,
- 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 particleflow.build import from_file, from_script
>>>
>>> lattice = from_file('example.madx')
>>>
>>> with open('example.madx') as fh:  # alternatively
...     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 the corresponding PyTorch backend modules.

```
>>> from pprint import pprint
>>> from particleflow.elements import elements
>>>
>>> pprint(elements)
{'dipedge': <class 'particleflow.elements.Dipedge'>,
 'drift': <class 'particleflow.elements.Drift'>,
```

(continues on next page)

(continued from previous page)

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

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

```
>>> from particleflow.elements import alignment_errors, aperture_types
>>>
>>> pprint(alignment_errors)
{'dpsi': <class 'particleflow.elements.LongitudinalRoll'>,
 'dx': <class 'particleflow.elements.Offset'>,
 'dy': <class 'particleflow.elements.Offset'>,
 'mrex': <class 'particleflow.elements.BPMError'>,
 'mrey': <class 'particleflow.elements.BPMError'>,
 'mscalx': <class 'particleflow.elements.BPMError'>,
 'mscaly': <class 'particleflow.elements.BPMError'>,
 'tilt': <class 'particleflow.elements.Tilt'>}
>>>
>>> pprint(aperture_types)
{'circle': <class 'particleflow.elements.ApertureCircle'>,
 'ellipse': <class 'particleflow.elements.ApertureEllipse'>,
 'rectangle': <class 'particleflow.elements.ApertureRectangle'>,
 'rectellipse': <class 'particleflow.elements.ApertureRectEllipse'>}
```

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

```
>>> from importlib import resources
>>> from particleflow.build import from_file
>>> import particleflow.test.sequences
>>>
>>> with resources.path(particleflow.test.sequences, 'hades.seq') as path:
...     lattice = from_file(path)
...
```

This will issue a few warnings of the following form:

```
.../particleflow/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(particleflow.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:

```
>>> from particleflow.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:

```
>>> print(lattice)
[    0.000000]  Drift(l=tensor(2.), label=None)
[    2.000000]  Quadrupole(l=tensor(1.), k1=tensor(0.2500), 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:

```
>>> 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:

```
>>> 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 *from_file* or *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 out lattice to a Segment as follows:

```
>>> from particleflow.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.

```
>>> from particleflow.build import Beam
>>> import particleflow.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:

```
>>> 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}
```

This work of 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:

```
>>> from importlib import resources
>>> from particleflow.build import from_file
>>> from particleflow.elements import Quadrupole, HKicker, SBend, Offset
>>> import particleflow.test.sequences
>>>
>>> with resources.path(particleflow.test.sequences, 'hades.seq') as path:
...     lattice = from_file(path)
...
>>> 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:

```
>>> for kicker in lattice[HKicker]:
...     print(kicker)
...
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')
>>>
>>> for sbend in lattice[SBend]:
...     print(sbend)
...
Tilt(psi=tensor(0.3795), target=Dipedge(l=tensor(0.), h=tensor(-0.0888), e1=tensor(0.
↪), fint=tensor(0.), hgap=tensor(0.), label=None) | SBend(l=tensor(1.4726),␣
↪angle=tensor(-0.1308), e1=tensor(0.), e2=tensor(0.), fint=tensor(0.),␣
↪fintx=tensor(0.), hgap=tensor(0.), label='ghadmu1') | 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=Dipedge(l=tensor(0.), h=tensor(-0.0891), e1=tensor(0.
↪), fint=tensor(0.), hgap=tensor(0.), label=None) | SBend(l=tensor(1.4726),␣
↪angle=tensor(-0.1311), e1=tensor(0.), e2=tensor(0.), fint=tensor(0.),␣
↪fintx=tensor(0.), hgap=tensor(0.), label='ghadmu2') | 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:

```
>>> lattice[Quadrupole, 5]
Quadrupole(l=tensor(1.), k1=Parameter containing: tensor(0.0298, requires_grad=True),␣
↪label='gth1qd11')
>>> lattice[Quadrupole, 5] is lattice[Quadrupole][5]
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:

```
>>> from particleflow.elements import Tilt
>>>
```

(continues on next page)

(continued from previous page)

```
>>> 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*:

```
>>> lattice[Offset]
[]
>>> for element in lattice[Tilt]:
...     print(element)
...
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=Dipedge(l=tensor(0.), h=tensor(-0.0888), e1=tensor(0.
↪), fint=tensor(0.), hgap=tensor(0.), label=None) | SBend(l=tensor(1.4726),␣
↪angle=tensor(-0.1308), e1=tensor(0.), e2=tensor(0.), fint=tensor(0.),␣
↪fintx=tensor(0.), hgap=tensor(0.), label='ghadmu1') | 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=Dipedge(l=tensor(0.), h=tensor(-0.0891), e1=tensor(0.
↪), fint=tensor(0.), hgap=tensor(0.), label=None) | SBend(l=tensor(1.4726),␣
↪angle=tensor(-0.1311), e1=tensor(0.), e2=tensor(0.), fint=tensor(0.),␣
↪fintx=tensor(0.), hgap=tensor(0.), label='ghadmu2') | 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:

```
>>> lattice['gth1kx1']
Tilt(psi=tensor(0.5000), target=HKicker(l=tensor(0.), hkick=tensor(0.),␣
↪vkick=tensor(0.), kick=tensor(0.), label='gth1kx1'))
>>> lattice['gth1kx1'] is lattice[HKicker, 1]
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:

```
>>> from particleflow.elements import Drift
>>>
>>> lattice[Drift, 0].label
'pad_drift_0'
>>> lattice[Drift, 1].label = 'pad_drift_0'
>>> lattice['pad_drift_0']
[Drift(l=tensor(3.9057), label='pad_drift_0'), Drift(l=tensor(0.8420), label='pad_
↪drift_0')]
>>> lattice['pad_drift_0', 1]
```

(continues on next page)

(continued from previous page)

```
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:

```
>>> 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:

```
>>> lattice[4]  # Selecting the 5-th element.
Drift(l=tensor(0.3370), label='pad_drift_2')
>>> lattice[19]  # Selecting the 20-th element.
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)`).

```
>>> lattice[:6]
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_1'),
 Quadrupole(l=tensor(0.6660), k1=Parameter containing: tensor(0.5668, requires_
↪grad=True), label='gte1qd11'),
 Drift(l=tensor(0.3370), label='pad_drift_2'),
 Monitor(l=tensor(0.), label='gte1dg1')])
>>>
>>> lattice[(Drift, 1):'gte1dg1']
Segment(elements=[Drift(l=tensor(0.8420), label='pad_drift_1'),
 Quadrupole(l=tensor(0.6660), k1=Parameter containing: tensor(0.5668, requires_
↪grad=True), 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*:

```
>>> from importlib import resources
>>> from particleflow.build import from_file
>>> from particleflow.elements import Quadrupole, HKicker, Offset, Tilt
>>> import particleflow.test.sequences
>>>
>>> with resources.path(particleflow.test.sequences, 'hades.seq') as path:
...     lattice = from_file(path)  # Load a fresh lattice.
...
>>> lattice['gte1qd11']  # Returns a single element, good.
Quadrupole(l=tensor(0.6660), k1=Parameter containing: tensor(0.5668, requires_
↪grad=True), label='gte1qd11')
>>> lattice['gte1qd11'] = Offset(lattice['gte1qd11'], dx=0.25, dy=0.50)
>>> lattice['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), label='gte1qd11'))
>>>
>>> lattice[HKicker, 1]  # Returns a single element, good.
HKicker(l=tensor(0.), hkick=tensor(0.), vkick=tensor(0.), kick=tensor(0.), label=
↪'gth1kx1')
>>> lattice[HKicker, 1] = Tilt(lattice[HKicker, 1], psi=1.0)
>>> lattice[HKicker, 1]
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:

```
>>> import numpy as np
>>>
>>> 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:

```
>>> from particleflow.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
>>> kicker
HKicker(l=tensor(1.), hkick=tensor(0.5000), vkick=tensor(0.), kick=tensor(0.5000),␣
↪label='hk1')
>>> kicker.makethin(5)
ThinSegment(elements=[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.ThinSegment` object, a special version of a more general `Segment`. This *ThinSegment* 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*[1] style is used. Other available slicing styles include *SIMPLE* and *EDGE*. For more details please consider the documentation of the *elements.Element.create_thin_sequence* method.

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

```
>>> quad.makethin(5, style='edge')
ThinSegment(elements=[Drift(l=tensor(0.), label='q1__d0'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), label='q1__0'),
 Drift(l=tensor(1.2500), label='q1__d1'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), label='q1__1'),
 Drift(l=tensor(1.2500), label='q1__d2'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), label='q1__2'),
 Drift(l=tensor(1.2500), label='q1__d3'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), label='q1__3'),
 Drift(l=tensor(1.2500), label='q1__d4'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), label='q1__4'),
 Drift(l=tensor(0.), label='q1__d5')])
```

(continues on next page)

[1] H. Burkhardt, R. De Maria, M. Giovannozzi, and T. Risselada, "Improved TEAPOT Method and Tracking with Thick Quadrupoles for the LHC and its Upgrade", in Proc. IPAC'13, Shanghai, China, May 2013, paper MOPWO027, pp. 945-947.

(continued from previous page)

```
>>>
>>> quad.makethin(5, style='simple')
ThinSegment(elements=[Drift(l=tensor(0.5000), label='q1__d0'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), label='q1__0'),
 Drift(l=tensor(1.), label='q1__d1'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), label='q1__1'),
 Drift(l=tensor(1.), label='q1__d2'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), label='q1__2'),
 Drift(l=tensor(1.), label='q1__d3'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), label='q1__3'),
 Drift(l=tensor(1.), label='q1__d4'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), label='q1__4'),
 Drift(l=tensor(0.5000), label='q1__d5')])
```

EDGE places the outermost slices right at the edge 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* to thin elements. Here we can choose the number of slices as well as the style via a dict which maps 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 the previous sections).

```
>>> from particleflow.elements import HKicker, Quadrupole, Segment
>>>
>>> lattice = Segment(lattice)
>>> thin = lattice.makethin({HKicker: 2, 'q1': 5}, style={'hk1': 'edge', Quadrupole:
↪'simple'})
>>> thin
Segment(elements=[ThinSegment(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')]),
 ThinSegment(elements=[Drift(l=tensor(0.5000), label='q1__d0'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), label='q1__0'),
 Drift(l=tensor(1.), label='q1__d1'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), label='q1__1'),
 Drift(l=tensor(1.), label='q1__d2'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), label='q1__2'),
 Drift(l=tensor(1.), label='q1__d3'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), label='q1__3'),
 Drift(l=tensor(1.), label='q1__d4'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), label='q1__4'),
 Drift(l=tensor(0.5000), label='q1__d5')])])
```

The *ThinSegment's represent their thick counterparts which are still accessible via the* '.base'' attribute. Also the base label is inherited (element access works as explained in the previous sections):

```
>>> thin['q1'].base
Quadrupole(l=tensor(5.), k1=tensor(0.6250), label='q1')
>>> thin[1].label
'q1'
>>> for drift in thin['q1']['q1__d*']:
...     print(drift)
```

(continues on next page)

(continued from previous page)

```
...
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 *ThinSegment's, using the 'flat* (or *flatten*) method:

```
>>> 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), label='q1__0'),
 Drift(l=tensor(1.), label='q1__d1'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), label='q1__1'),
 Drift(l=tensor(1.), label='q1__d2'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), label='q1__2'),
 Drift(l=tensor(1.), label='q1__d3'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), label='q1__3'),
 Drift(l=tensor(1.), label='q1__d4'),
 ThinQuadrupole(l=tensor(0.), k1l=tensor(0.6250), 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:

```
>>> from importlib import resources
>>> from pprint import pprint
>>> import particleflow.test.sequences
>>>
>>> script = resources.read_text(particleflow.test.sequences, 'hades.seq')
>>> script = script.splitlines()
>>> pprint(script[:4])
['beam, particle=ion, charge=6, energy=28.5779291448, mass=11.1779291448;',
```

(continues on next page)

(continued from previous page)

```
 '',
 '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:

```
>>> from particleflow.build import from_script
>>>
>>> lattice = from_script('\n'.join(script))
>>> lattice['gte1qd11']
Quadrupole(l=tensor(0.6660), k1=Parameter containing: tensor(0.5668, requires_
↪grad=True), label='gte1qd11')
>>> lattice['gte1qd12']
Quadrupole(l=tensor(0.6660), k1=Parameter containing: tensor(-0.5394, requires_
↪grad=True), label='gte1qd12')
>>> type(lattice['gte1qd11'].k1)
<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:

```
>>> script[2] = script[2].split(';')[0] + ';'
>>> pprint(script[:4])
['beam, particle=ion, charge=6, energy=28.5779291448, mass=11.1779291448;',
 '',
 'k1l_GTE1QD11 := 0.3774561583995819;',
 'k1l_GTE1QD12 := -0.35923901200294495;     // [flow] variable']
>>> lattice = from_script('\n'.join(script))
>>> lattice['gte1qd11']
Quadrupole(l=tensor(0.6660), k1=tensor(0.5668), label='gte1qd11')
>>> lattice['gte1qd12']
Quadrupole(l=tensor(0.6660), k1=Parameter containing: tensor(-0.5394, requires_
↪grad=True), 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:

```
>>> import torch
>>>
>>> lattice['gte1qd11'].k1 = torch.nn.Parameter(lattice['gte1qd11'].k1)
>>> lattice['gte1qd11']
Quadrupole(l=tensor(0.6660), k1=Parameter containing: tensor(0.5668, requires_
↪grad=True), label='gte1qd11')
```

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

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

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

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` method:

```
>>> from importlib import resources
>>> from particleflow.build import from_file
>>> from particleflow.elements import Kicker
>>> import particleflow.test.sequences
>>> import torch
>>>
>>> with resources.path(particleflow.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
>>> particles[[0, 2], :].std(dim=1)
tensor([0.0010, 0.0010])
>>>
>>> tracked = lattice.linear(particles)
>>> tracked[[0, 2], :].std(dim=1)
tensor([0.0086, 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`:

```
>>> particles = 0.01 * torch.randn(6, 1000)
>>> tracked = lattice.linear(particles, aperture=True)
>>> tracked.shape
torch.Size([6, 51])
```

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:

```
 >>> tracked, loss = lattice.linear(particles, aperture=True, recloss=True)
>>> tracked.shape
torch.Size([6, 51])
```

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:

```
>>> for label, loss_val in list(loss.items())[:10]:
...     print(f'{label}: {len(loss_val)}')
...
p_0: 1000
drift_0: 1000
p_lp2end: 999
drift_1: 999
yr01lb3: 999
drift_2: 999
yr01lb4: 999
drift_3: 999
```

(continues on next page)

(continued from previous page)

```
yr01df3: 992
drift_4: 992
```

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 999 particles arrive, this means we lost one particle at the previous element. We can confirm that by checking the loss values greater than zero:

```
>>> l_drift_0 = loss['drift_0']
>>> l_drift_0[l_drift_0 > 0]
tensor([0.0018])
```

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:

```
>>> tracked, loss = lattice.linear(particles, aperture=True, recloss='sum')
>>> tracked.shape
torch.Size([6, 51])
>>> loss
tensor(156.6134)
```

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).

```
>>> from particleflow.elements import SBend
>>>
>>> tracked, loss = lattice.linear(particles, aperture=True, recloss=SBend)
>>> for k, v in loss.items():
...     print(k, len(v))
...
yr01mh 967
yr02mh 361
yr03mh 127
yr04mh 115
yr05mh 113
yr06mh 110
yr07mh 92
yr08mh 89
yr09mh 89
yr10mh 85
yr11mh 85
yr12mh 85
```

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:

```
>>> tracked, loss = lattice.linear(particles, aperture=True, recloss='yr*qs*')
>>> len(loss)
18
>>> set(type(lattice[label]) for label in loss)
{<class 'particleflow.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):

```
>>> tracked, locations = lattice.linear(particles, aperture=True, observe='yr*qs*')
>>> len(locations)
18
>>> set(type(lattice[label]) for label in locations)
{<class 'particleflow.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:

```
>>> tracked, locations, loss = lattice.linear(particles, aperture=True, observe=
↪'yr*qs*', recloss='yr*qs*')
>>> loss['yr02qs1'].shape[-1]
730
>>> len(loss['yr02qs1'][loss['yr02qs1'] > 0])
91
>>> locations['yr02qs1'].shape[-1]
639
>>> loss['yr02qs1'].shape[-1] - len(loss['yr02qs1'][loss['yr02qs1'] > 0]) ==␣
↪locations['yr02qs1'].shape[-1]
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`.

```
>>> lattice.linear(particles, exact_drift=False).std(dim=1)
tensor([0.0847, 0.0142, 0.0140, 0.0086, 3.1779, 0.0100])
>>> lattice.linear(particles, exact_drift=True).std(dim=1)
tensor([0.0850, 0.0141, 0.0139, 0.0086, 3.1852, 0.0100])
```

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:

```
>>> from importlib import resources
>>> from particleflow.build import from_file
>>> from particleflow.compute import closed_orbit, linear_closed_orbit
>>> from particleflow.elements import Kicker, HKicker, VKicker
>>> import particleflow.test.sequences
>>>
>>> with resources.path(particleflow.test.sequences, 'cryring.seq') as path:
...     lattice = from_file(path)
```

(continues on next page)

(continued from previous page)

```
...
>>> thin = lattice.makethin({Kicker: 1})
>>> closed_orbit(thin, order=1)
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 through the lattice. Since all kickers are off (*kick = 0*) the closed orbit is just zero. Let's add some kicks:

```
>>> import random
>>>
>>> for kicker in lattice[HKicker] + lattice[VKicker]:
...     kicker.kick = random.uniform(-0.001, 0.001)
...
>>> thin = lattice.makethin({Kicker: 1})
>>> closed_orbit(thin, order=1).flatten()
tensor([-0.0002, -0.0020, -0.0028, -0.0004,  0.0000,  0.0000])
>>> linear_closed_orbit(thin).flatten()
tensor([-0.0002, -0.0020, -0.0028, -0.0004,  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 Transfer maps (sector maps)

By using *compute.sectormaps* we can compute the transfer maps along the lattice. The parameters `accumulate` let's us specify whether we want the local maps (`accumulate=False`) or the cumulative transfer maps w.r.t. to the start of the lattice (`accumulate=True`).

```
>>> from particleflow.compute import sectormaps
>>>
>>> maps = dict(sectormaps(lattice))
>>> maps[lattice[HKicker, 0].label]
tensor([[1.0000, 0.4573, 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, 0.4573, 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, 3.3648],
        [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 1.0000]])
```

Since by default the cumulative transfer maps are computed, the first order map for the first kicker is not the identity matrix. If we want the local maps instead we can use `accumulate=False`:

```
>>> maps = dict(sectormaps(lattice, accumulate=False))
>>> maps[lattice[HKicker, 0].label]
tensor([[1., 0., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0., 0.],
        [0., 0., 1., 0., 0., 0.],
        [0., 0., 0., 1., 0., 0.],
        [0., 0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 0., 1.]])
```

4.7.3 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:

```
>>> from importlib import resources
>>> from particleflow.build import from_file
>>> from particleflow.compute import orm
>>> from particleflow.elements import HKicker, HMonitor
>>> import particleflow.test.sequences
>>>
>>> with resources.path(particleflow.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).

```
>>> 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],
        [ 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,
```

(continues on next page)

(continued from previous page)

```
          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.]])
>>>
>>> orm_x.shape
torch.Size([12, 9])
>>> len(lattice[HKicker]), len(lattice[HMonitor])
(12, 9)
```

The ORMs rows correspond to kickers and the columns to monitors, as can be seen from the column and row count. Since we didn't specify to include the vertical kickers and the lattice does not contain any coupling the vertical ORM is just zero.

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 from the various sections which can be serialized with the following functions.
- `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:

```
>>> from particleflow.build import Lattice, create_script, sequence_script, track_
↪script, error_script
>>> from particleflow.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;
```

(continues on next page)

(continued from previous page)

```
    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, hgap = 0.0,␣
↪l = 6.0, at = 1.0;
    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, hgap = 0.0,␣
↪l = 6.0, at = 8.0;
endsequence;
```

Now let's create the *TRACK* block:

```
>>> 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 = 0.48601598728336326, px = 0.956650219837559, y = 0.9888598467877321,␣
↪py = 0.3745992567587124, t = 0.3191404673565723, pt = 0.8985300221575933;
    start, x = 0.8149549749594972, px = 0.4270200749373768, y = 0.635515919161222, py␣
↪= 0.9159877238895374, t = 0.5733268321302711, pt = 0.11705232703701407;
    start, x = 0.7422944246905987, px = 0.13460973177940772, y = 0.9418764774837417,␣
↪py = 0.8472652201346015, t = 0.9719634898137948, pt = 0.7171852345233938;
    start, x = 0.9364680812918479, px = 0.1542459780825013, y = 0.02137962413448624,␣
↪py = 0.49794824430220497, t = 0.6737652915481176, pt = 0.5786539141858486;
    start, x = 0.015436083161574854, px = 0.25469968297162326, y = 0.
↪02350832613352727, py = 0.47928000031158846, t = 0.1048411358107898, pt = 0.
↪3429838534958689;
    start, x = 0.739526624178238, px = 0.4996576568944264, y = 0.2589179375980676, py␣
↪= 0.19189491120024604, t = 0.9324930913073399, pt = 0.9043012884299407;
    start, x = 0.04371970050064822, px = 0.6079197617709056, y = 0.8672451455425605,␣
↪py = 0.38988876792841975, t = 0.5096824735970663, pt = 0.8265864954728698;
    start, x = 0.4328754778996823, px = 0.45375639994207795, y = 0.7022223115273295,␣
↪py = 0.050432233997635745, t = 0.26510847661524306, pt = 0.9823342991489998;
    start, x = 0.17927247316362627, px = 0.054034462012613305, y = 0.8949700770767757,
↪ py = 0.1963458427134087, t = 0.9191047099541068, pt = 0.8805875095812568;
    start, x = 0.5757753000336506, px = 0.06749373808823422, y = 0.11149076842268257,␣
↪py = 0.6651908216706869, t = 0.13855791459225308, pt = 0.46272792905279236;

    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:

```
>>> from particleflow.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;
```

(continues on next page)

(continued from previous page)

```
    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, 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, hgap = 0.0,␣
↪l = 6.0, at = 8.0;
endsequence;
>>>
>>> print(error_script(lattice))
eoption, add = true;
select, flag = error, clear = true;
select, flag = error, range = "qd";
ealign, dx = 0.01, dy = 0.02;
ealign, dpsi = 0.35;
```

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:

```
>>> print(create_script(
...     dict(particle='proton', gamma=1.25),
...     sequence=sequence_script(lattice, label='testseq'),
...     track=track_script(particles, ['qf', 'qd']),
...     errors=error_script(lattice)
... ))
beam, particle = proton, gamma = 1.25;

testseq: 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, 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, hgap = 0.0,␣
↪l = 6.0, at = 8.0;
endsequence;

use, sequence = testseq;

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 = 0.48601598728336326, px = 0.956650219837559, y = 0.9888598467877321,␣
↪py = 0.3745992567587124, t = 0.3191404673565723, pt = 0.8985300221575933;
    start, x = 0.8149549749594972, px = 0.4270200749373768, y = 0.635515919161222, py␣
↪= 0.9159877238895374, t = 0.5733268321302711, pt = 0.11705232703701407;
    start, x = 0.7422944246905987, px = 0.13460973177940772, y = 0.9418764774837417,␣
↪py = 0.8472652201346015, t = 0.9719634898137948, pt = 0.7171852345233938;
    start, x = 0.9364680812918479, px = 0.1542459780825013, y = 0.02137962413448624,␣
↪py = 0.49794824430220497, t = 0.6737652915481176, pt = 0.5786539141858486;
    start, x = 0.015436083161574854, px = 0.25469968297162326, y = 0.
↪02350832613352727, py = 0.47928000031158846, t = 0.1048411358107898, pt = 0.
↪3429838534958689;
```

(continues on next page)

(continued from previous page)

```
    start, x = 0.739526624178238, px = 0.4996576568944264, y = 0.2589179375980676, py␣
↪= 0.19189491120024604, t = 0.9324930913073399, pt = 0.9043012884299407;
    start, x = 0.04371970050064822, px = 0.6079197617709056, y = 0.8672451455425605,␣
↪py = 0.38988876792841975, t = 0.5096824735970663, pt = 0.8265864954728698;
    start, x = 0.4328754778996823, px = 0.45375639994207795, y = 0.7022223115273295,␣
↪py = 0.050432233997635745, t = 0.26510847661524306, pt = 0.9823342991489998;
    start, x = 0.17927247316362627, px = 0.054034462012613305, y = 0.8949700770767757,
↪ py = 0.1963458427134087, t = 0.9191047099541068, pt = 0.8805875095812568;
    start, x = 0.5757753000336506, px = 0.06749373808823422, y = 0.11149076842268257,␣
↪py = 0.6651908216706869, t = 0.13855791459225308, pt = 0.46272792905279236;

    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:

```
>>> 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, 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, 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 *particleflow.madx* sub-package directly.

4.10 Running MADX scripts

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

```
>>> from importlib import resources
>>> import os.path
>>> from particleflow.madx import run, run_script, run_orm
>>> import particleflow.test.sequences
>>>
>>> with resources.path(particleflow.test.sequences, 'cryring.seq') as path:
...     result = run_file(path, [], madx=os.path.expanduser('~/bin/madx'))
...
>>> list(result)
['stdout', 'stderr']
>>> result['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:

```
>>> script = resources.read_text(particleflow.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:

```
>>> list(result)
['stdout', 'stderr', 'twiss']
>>> type(result['twiss'])
<class 'pandas.core.frame.DataFrame'>
>>> result['twiss'].columns
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.

4.11 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:

```
>>> result = run_script(script, {'twiss': True}, madx=os.path.expanduser('~/bin/madx
↪'))
>>> type(result['twiss'])
<class 'tuple'>
>>> type(result['twiss'][0])
<class 'pandas.core.frame.DataFrame'>
>>> type(result['twiss'][1])
<class 'dict'>
>>>
>>> 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': '05/09/19',
 '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,
 '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': '13.48.43',
 '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.

4.12 Configure scripts before running

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

```
>>> twiss = result['twiss'][0]
>>> 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`.

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

4.13 Compute the Orbit Resonse 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:

```
>>> kickers = ['yr04kh', 'yr06kh', 'yr08kh', 'yr10kh']
>>> monitors = ['yr02dx1', 'yr03dx1', 'yr03dx4']
>>> orm = run_orm(script, kickers=kickers, monitors=monitors, madx=os.path.expanduser(
↪'~/bin/madx'))
>>> orm
               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
```

(continues on next page)

(continued from previous page)

```
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
>>>
>>> orm['X']
         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).

4.14 Visualizing lattices

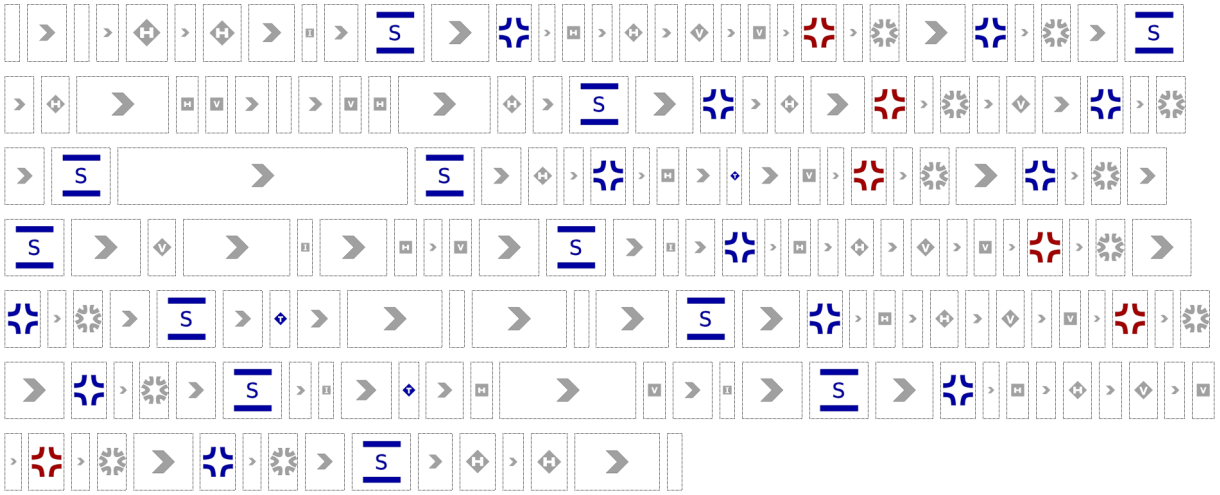
Lattices can be visualized by serializing them into an HTML file. This can be done also via *build.sequence_script* by supplying the argument `markup='html'`. The resulting HTML sequence file can be viewed in any modern browser and elements can be inspected by using the browser's inspector tool (e.g.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

Let's visualize one of the example sequences:

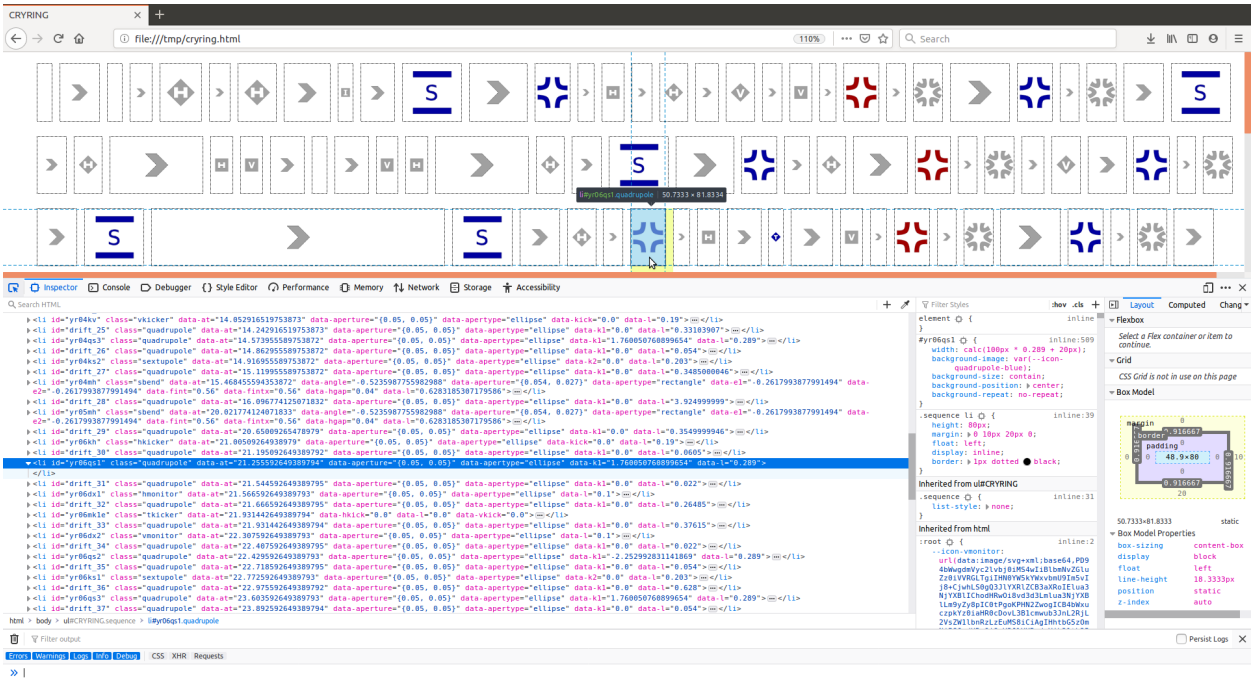
```
>>> from importlib import resources
>>> from particleflow.build import from_file, sequence_script
>>> import particleflow.test.sequences
>>>
>>> with resources.path(particleflow.test.sequences, 'cryring.seq') as path:
...     lattice = from_file(path)
...
>>> with open('cryring.html', 'w') as fh: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

PARTICLEFLOW API

5.1 particleflow package

5.1.1 Subpackages

particleflow.madx package

Submodules

particleflow.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.

particleflow.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

particleflow.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

particleflow.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

particleflow.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

particleflow.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

particleflow.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

particleflow.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

particleflow.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

particleflow.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

particleflow.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

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

Type str

particleflow.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

particleflow.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

particleflow.madx.parser.**parse_file**(*f_name: str*) → particleflow.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()*.

particleflow.madx.parser.**parse_script**(*script: str*) → particleflow.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.

particleflow.madx.utils module

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

particleflow.madx.utils.**run_file**(*scripts: Union[str, Sequence[str]], results: Union[Sequence[str], Dict[str, bool]], *, variables: Dict[str, Dict[str, Any]] = None, format: Dict[str, Any] = None, madx: 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 exectutable. 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*) – See *run_script()*.
- **variables** (*dict*) – Variables configuration for each of the scripts in *scripts*. See *run_script()* for more details.
- **format** (*dict*) – Format values for each of the scripts in *scripts*. See *run_script()* for more details.
- **madx** (*str*) – See *run_script()*.

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.

particleflow.madx.utils.**run_script**(*script: str, results: Union[Sequence[str], Dict[str, bool]], *, variables: Dict[str, Any] = None, format: Dict[str, Any] = None, madx: 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*) – 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. For more information about file types see *convert()*.
- **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`.
- **madx** (*str, optional*) – File name pointing to the MADX executable. If the *MADX* environment variable is set it takes precedence.

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.

particleflow.madx.utils.**run_orm**(*script: str, kickers: Sequence[str], monitors: Sequence[str], *, kicks: Tuple[float, float] = (-0.001, 0.001), variables: Dict[str, Any] = None, madx: str = None*) → <MagicMock name='mock.DataFrame' id='139943556480416'>

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()*.
- **madx** (*str*) – See *run_script()*.

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

Return type pd.DataFrame

particleflow.madx.utils.**convert**(*f_name: str, meta: bool = False*) → Union[<MagicMock name='mock.DataFrame' id='139943556480416'>, str, particleflow.madx.parser.Script, Tuple[<MagicMock name='mock.DataFrame' id='139943556480416'>, 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.
- **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*).

particleflow.madx.utils.**convert_tfs**(*f_name: str, meta: bool = False*) → Union[<MagicMock name='mock.DataFrame' id='139943556480416'>, Tuple[<MagicMock name='mock.DataFrame' id='139943556480416'>, 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.

particleflow.madx.utils.**convert_trackone**(*f_name: str*, *meta: bool = False*) → Union[<MagicMock name='mock.DataFrame' id='139943556480416'>, Tuple[<MagicMock name='mock.DataFrame' id='139943556480416'>, 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

particleflow.tools package

Submodules

particleflow.tools.madx_to_html module

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

particleflow.tools.madx_to_html.**main**()

Module contents

5.1.2 Submodules

particleflow.build module

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

particleflow.build.**from_file**(*f_name: str*, ***, *beam: dict = None*, *errors: Union[bool, <MagicMock name='mock.DataFrame' id='139943556480416'>] = True*, *padding: Union[float, Tuple[float, ...], Dict[Union[str, Pattern[AnyStr], Type[Element], None], Union[float, Tuple[float, ...]]]] = None*) → particleflow.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*) –

  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>`.
- **padding** (*float or tuple or dict*) – Additional padding applied to lattice elements. See `elements.Aperture`.

Returns lattice

Return type *Segment*

particleflow.build.**from_script**(*script: str, *, beam: dict = None, errors: Union[bool, <MagicMock name='mock.DataFrame' id='139943556480416'>] = True, padding: Union[float, Tuple[float, ...], Dict[Union[str, Pattern[AnyStr], Type[Element], None], Union[float, Tuple[float, ...]]]] = None*) → particleflow.elements.Segment

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

particleflow.build.**create_script**(*beam: dict, *, sequence: Union[str, particleflow.elements.Segment], track: str = '', errors: 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.
- **track** (*str*) – Part of the script describing the tracking.
- **errors** (*str*) – Part of the script describing error definitions.

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

Return type str

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

particleflow.build.**sequence_script**(*lattice: particleflow.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

particleflow.build.**error_script**(*lattice: particleflow.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.

particleflow.build.**track_script**(*particles: Union[<MagicMock name='mock.ndarray' id='139943557452856'>, <MagicMock name='mock.Tensor' id='139943556025088'>], 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

particleflow.compute module

Convenience functions for computing various simulation related quantities.

particleflow.compute.**orm**(*lattice: particleflow.elements.Segment, *, kickers: Union[str, Pattern, Type[Union[Element, AlignmentError]], Sequence[Union[int, str, Element, AlignmentError, Tuple[Union[str, Pattern, Type[Union[Element, AlignmentError]]], int]]]], monitors: Union[str, Pattern, Type[Union[Element, AlignmentError]], Sequence[Union[int, str, Element, AlignmentError, Tuple[Union[str, Pattern, Type[Union[Element, AlignmentError]]], int]]]], kicks: Tuple[float, float] = (-0.001, 0.001)*) → Tuple[<MagicMock name='mock.Tensor' id='139943556025088'>, <MagicMock name='mock.Tensor' id='139943556025088'>]

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.

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

Return type torch.Tensor

particleflow.compute.**closed_orbit**(*lattice: particleflow.elements.Segment*, *, *order: int = 1*, *max_iter: int = None*, *tolerance: float = 1e-09*) → <MagicMock name='mock.Tensor' id='139943556025088'>

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*) – The order at which transport maps are truncated (e.g. linear = 1).

- **max_iter** (*int, optional*) – Maximum number of iterations.
- **tolerance** (*float, optional*) – Maximum L2 distance between initial orbit and tracked orbit after one turn for convergence.

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

Return type torch.Tensor

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

particleflow.compute.**linear_closed_orbit**(*lattice: particleflow.elements.Segment*) → <MagicMock name='mock.Tensor' id='139943556025088'>

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)):

$$\left[\not\vdash - \bar{R}_0\right]\ x_{co} = \sum_{i=1}^{n} \bar{R}_i\, d_i$$

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 torch.Tensor

particleflow.compute.**sectormaps**(*lattice: particleflow.elements.Segment, *, accumulate: bool = True*) → Iterator[Tuple[str, <MagicMock name='mock.Tensor' id='139943556025088'>]]

Compute the transfer maps along the lattice.

Parameters

- **lattice** (Segment) –
- **accumulate** (*bool, optional*) – If true then the transfer maps are accumulated along the lattice (i.e. representing the compound transfer maps up to respective point along the lattice), otherwise the transfer maps belong to each single element.

Yields

- **label** (*str*)
- **transfer_map** (*torch.Tensor*)

particleflow.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.

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

Type dict

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

Type dict

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

Type dict

class particleflow.elements.**ApertureCircle**(*aperture: float = <MagicMock name='mock.inf' id='139943557067832'>, offset: Tuple[float, float] = (0.0, 0.0), padding: Union[float, Tuple] = 0.0*)
Bases: *particleflow.elements.ApertureEllipse*

Circular aperture.

Parameters **aperture** (*float*) – Radius of the circle.

class particleflow.elements.**ApertureEllipse**(*aperture: Tuple[float, float] = (<MagicMock name='mock.inf' id='139943557067832'>, <MagicMock name='mock.inf' id='139943557067832'>), offset: Tuple[float, float] = (0.0, 0.0), padding: Union[float, Tuple] = 0.0*)
Bases: particleflow.elements.Aperture

Elliptical aperture.

Parameters **aperture** (*tuple*) – Horizontal and vertical semi-axes of the ellipse.

loss(*xy: <MagicMock name='mock.Tensor' id='139943556025088'>*) → <MagicMock name='mock.Tensor' id='139943556025088'>
Compute loss values for the given xy-positions.

Parameters **xy** (*torch.Tensor*) – Tensor of shape *(2, N)* where *N* is the number of particles and rows are x- and y-positions respectively.

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 torch.Tensor

class particleflow.elements.**ApertureRectangle**(*aperture: Tuple[float, float] = (<MagicMock name='mock.inf' id='139943557067832'>, <MagicMock name='mock.inf' id='139943557067832'>), offset: Tuple[float, float] = (0.0, 0.0), padding: Union[float, Tuple] = 0.0*)
Bases: particleflow.elements.Aperture

Rectangular aperture.

Parameters **aperture** (*tuple*) – Half width (horizontal) and half height (vertical) of the rectangle.

loss(*xy:* <*MagicMock name='mock.Tensor' id='139943556025088'*>) → <MagicMock name='mock.Tensor' id='139943556025088'>
Compute loss values for the given xy-positions.

Parameters **xy** (*torch.Tensor*) – Tensor of shape *(2, N)* where *N* is the number of particles and rows are x- and y-positions respectively.

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 torch.Tensor

class particleflow.elements.**ApertureRectEllipse**(*aperture: Tuple[float, float, float, float] = (<MagicMock name='mock.inf' id='139943557067832'>, <MagicMock name='mock.inf' id='139943557067832'>, <MagicMock name='mock.inf' id='139943557067832'>, <MagicMock name='mock.inf' id='139943557067832'>)*, *offset: Tuple[float, float] = (0.0, 0.0)*, *padding: Union[float, Tuple] = 0.0*)

Bases: particleflow.elements.Aperture

Rectangular and elliptical aperture overlaid.

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

loss(*xy:* <*MagicMock name='mock.Tensor' id='139943556025088'*>) → <MagicMock name='mock.Tensor' id='139943556025088'>
Compute loss values for the given xy-positions.

Parameters **xy** (*torch.Tensor*) – Tensor of shape *(2, N)* where *N* is the number of particles and rows are x- and y-positions respectively.

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 torch.Tensor

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

Bases: particleflow.elements.Element

Marker element.

exact(*x:* <*MagicMock name='mock.Tensor' id='139943556025088'*>) → <MagicMock name='mock.Tensor' id='139943556025088'>
Exact analytic solution for tracking through the element.

linear(*x:* <*MagicMock name='mock.Tensor' id='139943556025088'*>) → <MagicMock name='mock.Tensor' id='139943556025088'>
Linear tracking through the element.

class particleflow.elements.**Drift**(*l: float*, *beam: dict*, ***kwargs*)

Bases: particleflow.elements.Element

Drift space.

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

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

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

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

class particleflow.elements.**Instrument**(*l: float*, *beam: dict*, ***kwargs*)
Bases: *particleflow.elements.Drift*

A place holder for any type of beam instrumentation.

class particleflow.elements.**Placeholder**(*l: float*, *beam: dict*, ***kwargs*)
Bases: *particleflow.elements.Drift*

A place holder for any type of element.

class particleflow.elements.**Monitor**(*l: float*, *beam: dict*, ***kwargs*)
Bases: *particleflow.elements.Drift*

Beam position monitor.

class particleflow.elements.**HMonitor**(*l: float*, *beam: dict*, ***kwargs*)
Bases: *particleflow.elements.Monitor*

Beam position monitor for measuring horizontal beam position.

class particleflow.elements.**VMonitor**(*l: float*, *beam: dict*, ***kwargs*)
Bases: *particleflow.elements.Monitor*

Beam position monitor for measuring horizontal beam position.

class particleflow.elements.**Kicker**(*hkick: Union[float, <MagicMock name='mock.nn.Parameter' id='139943557496448'>] = 0*, *vkick: Union[float, <MagicMock name='mock.nn.Parameter' id='139943557496448'>] = 0*, ***kwargs*)
Bases: particleflow.elements.Element

Combined horizontal and vertical kicker magnet.

Parameters

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

property d

linear(*x: <MagicMock name='mock.Tensor' id='139943556025088'>*) → <MagicMock name='mock.Tensor' id='139943556025088'>
Linear tracking through the element.

class particleflow.elements.**HKicker**(*kick: Union[float, <MagicMock name='mock.nn.Parameter' id='139943557496448'>], **kwargs*)
Bases: *particleflow.elements.Kicker*

Horizontal kicker magnet.

property kick

class particleflow.elements.**VKicker**(*kick: Union[float, <MagicMock name='mock.nn.Parameter' id='139943557496448'>], **kwargs*)

Bases: *particleflow.elements.Kicker*

Vertical kicker magnet.

property kick

class particleflow.elements.**TKicker**(*hkick: Union[float, <MagicMock name='mock.nn.Parameter' id='139943557496448'>] = 0, vkick: Union[float, <MagicMock name='mock.nn.Parameter' id='139943557496448'>] = 0, **kwargs*)

Bases: *particleflow.elements.Kicker*

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

class particleflow.elements.**Quadrupole**(*k1: Union[float, <MagicMock name='mock.nn.Parameter' id='139943557496448'>], l: float, beam: dict, **kwargs*)

Bases: particleflow.elements.Element

Quadrupole magnet.

Whether this is a (horizontally) focusing or defocusing magnet will be 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.

k1
Normalized quadrupole gradient [1/m^2].

Type torch.Tensor or Parameter

property R

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

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

Return type *Segment*

See also:

`Element.create_thin_sequence()`

class particleflow.elements.**Sextupole**(*k2: Union[float, <MagicMock name='mock.nn.Parameter' id='139943557496448'>], l: float, beam: dict, **kwargs*)

Bases: `particleflow.elements.Element`

Sextupole magnet.

k2

Normalized sextupole coefficient [1/m^3].

Type torch.Tensor or Parameter

class particleflow.elements.**SBend**(*angle: float*, *l: float*, *beam: dict*, *e1: float = 0.0*, *e2: float = 0.0*, *fint: Union[float*, *bool] = 0.0*, *fintx: Optional[float] = None*, *hgap: float = 0.0*, ***kwargs*)

Bases: `particleflow.elements.Element`

Sector bending magnet.

Parameters

- **angle** (*float*) – Bending angle of the dipole [rad].
- **e1** (*float*) – Rotation angle for the entrance pole face [rad]. `e1 = e2 = angle/2` turns an *SBEND* into a *RBEND*.
- **e2** (*float*) – Rotation angle for the exit pole face [rad].
- **fint** (*float*) – Fringing field integral at entrance. If *fintx* is not specified then *fint* is also used at the exit.
- **fintx** (*float*) – Fringing field integral at exit.
- **hgap** (*float*) – Half gap of the magnet [m].

makethin(*n: int*, ***, *style: Optional[str] = None*) → particleflow.elements.Segment

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 `create_thin_sequence()`.

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

Return type *Segment*

See also:

`Element.create_thin_sequence()`

class particleflow.elements.**RBend**(*angle: float*, *l: float*, *beam: dict*, *e1: float = 0.0*, *e2: float = 0.0*, *fint: float = 0.0*, *fintx: Optional[float] = None*, *hgap: float = 0.0*, ***kwargs*)

Bases: *particleflow.elements.SBend*

class particleflow.elements.**Dipedge**(*h: float*, *e1: float*, *fint: float*, *hgap: float*, ***kwargs*)

Bases: particleflow.elements.Element

Fringing fields at the entrance and exit of dipole magnets.

Parameters

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

second_order(*x: <MagicMock name='mock.Tensor' id='139943556025088'>*) → <MagicMock name='mock.Tensor' id='139943556025088'>

Second order tracking through the element.

class particleflow.elements.**ThinQuadrupole**(*k1l: Union[float, <MagicMock name='mock.nn.Parameter' id='139943557496448'>], **kwargs*)

Bases: particleflow.elements.Element

Thin lens representation of a quadrupole magnet.

Parameters **k1l** (*torch.Tensor or torch.nn.Parameter*) –

property R

makethin(*n: int*, ***, *style: Optional[str] = None*) → List[particleflow.elements.Element]

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 create_thin_sequence().

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

Return type *Segment*

See also:

Element.create_thin_sequence()

class particleflow.elements.**ThinSextupole**(*k2l: Union[float, <MagicMock name='mock.nn.Parameter' id='139943557496448'>], **kwargs*)

Bases: particleflow.elements.Element

Thin lens representation of a sextupole magnet.

Parameters **k2l** (*torch.Tensor or torch.nn.Parameter*) –

makethin(*n: int*, *, *style: Optional[str] = None*) → List[particleflow.elements.Element]
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 `create_thin_sequence()`.

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

Return type *Segment*

See also:

`Element.create_thin_sequence()`

class particleflow.elements.**Tilt**(*target: Union[particleflow.elements.Element, particleflow.elements.AlignmentError], psi: Union[float, <MagicMock name='mock.nn.Parameter' id='139943557496448'>] = 0.0*)
Bases: *particleflow.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 torch.Tensor 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 particleflow.elements.**Offset**(*target: Union[particleflow.elements.Element, particleflow.elements.AlignmentError], dx: Union[float, <MagicMock name='mock.nn.Parameter' id='139943557496448'>] = 0.0, dy: Union[float, <MagicMock name='mock.nn.Parameter' id='139943557496448'>] = 0.0*)
Bases: particleflow.elements.AlignmentError

AlignmentError representing the xy-offset of an element.

dx
Horizontal offset.

Type torch.Tensor or Parameter

dy
Vertical offset.

Type torch.Tensor or Parameter

property d

property d_inv

triggers = ('dx', 'dy')

class particleflow.elements.**LongitudinalRoll**(*target: Union[particleflow.elements.Element, particleflow.elements.AlignmentError], psi: Union[float, <MagicMock name='mock.nn.Parameter' id='139943557496448'>] = 0.0*)

Bases: particleflow.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 torch.Tensor or Parameter

property R

property R_inv

triggers = ('dpsi',)

class particleflow.elements.**BPMError**(*target: Union[particleflow.elements.Element, particleflow.elements.AlignmentError], ax: Union[float, <MagicMock name='mock.nn.Parameter' id='139943557496448'>] = 0, ay: Union[float, <MagicMock name='mock.nn.Parameter' id='139943557496448'>] = 0, rx: Union[float, <MagicMock name='mock.nn.Parameter' id='139943557496448'>] = 0, ry: Union[float, <MagicMock name='mock.nn.Parameter' id='139943557496448'>] = 0*)

Bases: particleflow.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 torch.Tensor or Parameter

ay
Vertical absolute read error.

Type torch.Tensor or Parameter

rx
Horizontal relative read error.

Type torch.Tensor or Parameter

ry
Vertical relative read error.

Type torch.Tensor or Parameter

enter(*x: <MagicMock name='mock.Tensor' id='139943556025088'>*) → <MagicMock name='mock.Tensor' id='139943556025088'>
Applies linear coordinate transformation at the entrance of the wrapped element.

exit(*x: <MagicMock name='mock.Tensor' id='139943556025088'>*) → <MagicMock name='mock.Tensor' id='139943556025088'>
Applies linear coordinate transformation at the exit of the wrapped element.

readout(*x: <MagicMock name='mock.Tensor' id='139943556025088'>*) → <MagicMock name='mock.Tensor' id='139943556025088'>
Return BPM readings for the given coordinates.

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

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

Return type torch.Tensor

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

class particleflow.elements.**Segment**(*elements: Sequence[Union[particleflow.elements.Element, particleflow.elements.AlignmentError, particleflow.elements.Segment]]*)
Bases: particleflow.elements.Module

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.
- 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 Element or AlignmentError or *Segment*

property R

property d

flat() → particleflow.elements.Segment
Convenience function wrapping *Segment.flatten*.

flatten() → Iterator[Union[particleflow.elements.Element, particleflow.elements.AlignmentError]]
Retrieve a flat representation of the segment (with sub-segments flattened as well).

forward(*x: <MagicMock name='mock.Tensor' id='139943556025088'>, method='forward', *, aperture: bool = False, exact_drift: bool = True, observe: Union[str, Pattern, Type[Union[Element, AlignmentError]], Sequence[Union[str, Pattern, Type[Union[Element, AlignmentError]]]], None] = None, recloss: Union[bool, str, Pattern, Type[Union[Element, AlignmentError]], Sequence[Union[str, Pattern, Type[Union[Element, AlignmentError]]]], None] = None, loss_func: Optional[Callable[[<MagicMock name='mock.Tensor' id='139943556025088'>], <MagicMock name='mock.Tensor' id='139943556025088'>]] = None*) → Union[<MagicMock name='mock.Tensor' id='139943556025088'>, Tuple]
Track the given particles through the segment.

Parameters

- **x** (*torch.Tensor*) – Shape *(6, N)* where *N* is the number of particles.
- **method** (*str*) – 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** (*torch.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** (*torch.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, Element, AlignmentError, Tuple[Union[str, Pattern, Type[Union[Element, AlignmentError]]], int]]*) → int

makethin(*n: Union[int, Dict[Union[str, Pattern[AnyStr], Type[Element], None], int]], *, style: Union[str, Dict[Union[str, Pattern[AnyStr], Type[Element], None], str], None] = None*) → particleflow.elements.Segment

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

Parameters

- **n** (*int or dict*) – Number of slices (thin elements). If *int* this applies to all elements. If *dict* then keys should be either element classes (subclasses of *Element*) or strings (indicating element names) or regex patterns (indicating element name patterns). Values should be the corresponding number of slices that are applied to a matching element. The first matching criterion 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 options 2. and 3. these should come at the very end of the dict, otherwise they will override any subsequent definitions.

- **style** (*str or dict*) – Slicing style per element. Works similar to *n*. See `Element.makethin()` for more information about available styles.

Returns Containing thin elements (or sub-segments).

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.

unstack() → Iterator[particleflow.elements.Segment]

Yields the segment itself (this method exists for consistency with *AlignmentError*).

particleflow.utils module

General purpose utility functions.

particleflow.utils.**copy_doc**(*source*)
Copy the docstring of one object to another.

particleflow.utils.**format_doc**(***kwargs*)
Format the doc string of an object according to *str.format* rules.

particleflow.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).

particleflow.utils.**pad_max_shape**(**arrays*, *before=None*, *after=1*, *value=0*, *tie_break=<MagicMock name='mock.floor' id='139943557436192'>*) → List[<MagicMock name='mock.ndarray' id='139943557452856'>]
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

particleflow.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])
```

(continues on next page)

(continued from previous page)

```
[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)]
```

particleflow.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
```

particleflow.utils.**setattr_multi**(*obj*, *names: Sequence[str]*, *values: Union[Sequence, Dict[str, Any]]*, ***, *convert: Callable[[Any], Any] = None*) → None

Set multiple attributes at once.

Parameters

- **obj** (*object*) –
- **names** (*sequence*) –
- **values** (*sequence or dict*) – If *dict* then it must map *names* to values.
- **convert** (*callable, optional*) – If not *None* then will be used to convert each value by calling it before setting as an attribute.

class particleflow.utils.**singledispatchmethod**(*func*)

Bases: object

register(*cls*, *method=None*)

5.1.3 Module contents

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

M

N

O

P

Q

R

S

T

U

V