apace

Release 0.1.0

Felix Andreas

Dec 20, 2021

CONTENTS

apace is yet **a**nother **p**article **a**ccelerator **c**od**e** designed for the optimization of beam optics. It is available as Python package and aims to provide a convenient and straightforward API to make use of Python's numerous scientific libraries.

CHAPTER

ONE

INSTALLATION GUIDE

apace runs on Python 3.7 or above under CPython and PyPy.

1.1 Install from PyPI

You can install/update **apace** and its dependencies from PyPI using pip or pipenv:

```
pip install -U apace
```

1.2 Build from Source

To build from source either clone the public repository from GitHub:

```
git clone https://github.com/andreasfelix/apace.git
```

Or, download and extract the tarball:

```
curl -OL https://github.com/andreasfelix/apace/tarball/master
```

Then change into the top level source tree and install the **apace** and its dependencies:

```
cd apace
pip install .
```

CHAPTER

TWO

QUICKSTART

A simple example on how to calculate the Twiss parameter for a FODO lattice.

Import apace:

```
import apace as ap
```

Create a ring consisting of 8 FODO cells:

```
d1 = ap.Drift('D1', length=0.55)
b1 = ap.Dipole('B1', length=1.5, angle=0.392701, e1=0.1963505, e2=0.1963505)
q1 = ap.Quadrupole('Q1', length=0.2, k1=1.2)
q2 = ap.Quadrupole('Q2', length=0.4, k1=-1.2)
fodo_cell = ap.Lattice('FODO', [q1, d1, b1, d1, q2, d1, b1, d1, q1])
fodo_ring = ap.Lattice('RING', [fodo_cell] * 8)
```

Calculate the twiss parameters:

```
twiss = ap.Twiss(ring)
```

Plot horizontal and vertical beta functions using matplotlib:

```
import matplotlib.pyplot as plt
plt.plot(twiss.s, twiss.beta_x, twiss.beta_y, twiss.eta_x)
plt.show()
```

Note: A FODO lattice is the simplest possible strong focusing lattice consisting out of a horizontal focusing quadrupole **(F)**, a drift space **(0)**, a horizontal defocusing quadrupole **(D)** and another drift space **(0)**.

CHAPTER

THREE

USER GUIDE

This guide is indented as an informal introduction into basic concepts and features of apace. For a detailed reference of its classes and functions, see *API Reference*.

All data describing the structure and properties of the accelerator is represented by different objects. This sections gives an brief overview of the most important classes.

3.1 Element Classes

All basic components of the magnetic lattice like drift spaces, bending magnets and quadrupole are a subclass of an abstract base class called *Element*. All elements have a name (which should be unique), a length and an optional description.

You can create a new *Drift* space with:

```
drift = ap.Drift(name='Drift', length=2)
```

Check that the `drift` is actually a subclass of *Element*:

```
>>> isinstance(drift, ap.Element)
True
```

To create the more interresting *Quadrupole* object use:

```
quad = ap.Quadrupole(name='Quadrupole', length=1, k1=1)
```

The attributes of elements can also be changed after they a created:

```
>>> quad.k1
1
>>> quad.k1 = 0.8
>>> quad.k1
0.8
```

Note: You can also set up an event listener to whenever an element gets changed, for more information see *Signals and Events*.

When using Python interactively you can get further information on a specific element with the builtin `print()` function:

```
>>> print(quad)
name           : Quadrupole
description    :
parent_lattices  : set()
k1             : 0.8
length         : 1
length_changed: Signal
value_changed : Signal
```

As you can see, the *Quadrupole* object has by default also the *parent_lattices* attribute, which we will discuss in the next subsection.

3.2 Lattice class

The magnetic lattice of modern Particle accelerators is typically more complex than a single quadrupole. Therefore multiple elements can be arranged into a more complex structure using the *Lattice* class.

3.2.1 Creating a Double Dipole Achromat

As we already created a FODO structure in *Quickstart*, let's create a Double Dipole Achromat Lattice this time. In addition to our `drift` and `quad` elements, we need a new *Dipole* object:

```
bend = ap.Dipole('Dipole', length=1, angle=math.pi / 16)
```

Now we can create a DBA lattice:

```
dba_cell = ap.Lattice('DBA_CELL', [drift, bend, drift, quad, drift, bend, drift])
```

As you can see, it is possible for elements to occur multiple times within the same lattice. Elements can even be in different lattices at the same time. What is important to note is, that elements which appear within a lattice multiple times (e.g. all instances of `drift` within the `dba_cell`) correspond to the same underlying object.

You can easily check this by changing the length of the `drift` and displaying the length of the `dba_cell` before and afterwards:

```
>>> dba_cell.length
11
>>> drift.length += 0.25
>>> dba_cell.length
12
```

As the `drift` space appears four times within the `dba_cell` its length increased four-fold.

3.2.2 Parent lattices

You may have also noticed that length of the `dba_cell` was updated automatically without you having to call any update function. This works because apace keeps track of all parent lattices through the *`parent_lattices`* attribute and informs all parents whenever the length of an element changes.

Note: Apace only notifies the lattice that it has to update its length value. The calculation of new length only happens when the attribute is accessed. This may be not that advantageous for a simple length calculation, but (apace uses this system for all its data) makes a difference for more computational expensive properties. For more see *Lazy Evaluation*.

Try to print the contents of *`parent_lattices`* for the `quad` object:

```
>>> quad.parent_lattices
{Lattice}
```

In contrast to the end of *Element Classes* section, where it was empty, `quad.parent_latticess` now has one entry. Note that this is a Python `set`, so it cannot to contain duplicates in case that an element appears multiple times within the same lattice. The set gets updated whenever an element gets added or removed from a *`Lattice`*.

It is also possible to create a lattice out of lattices. For example you could create a DBA ring using the already existing `dba_cell`:

```
dba_ring = ap.Lattice('DBA_RING', [dba_cell] * 16)
```

The `dba_ring` should now be listed as a parent of the `dba_cell`:

```
>>> dba_cell.parent_lattices
{DBA_RING}
```

Its length should be 16 times the length of the `dba_cell`:

```
>>> dba_ring.length
192.0
```

3.2.3 Direct children

The structure which defines the order of elements in our DBA ring can be thought of as a Tree, where `dba_ring` is the root, the `dba_cell` objects are the nodes and the `bend`, `drift` and `quad` elements are the leafes. The attribute which stores the order of objects within a lattice is called `chilldren`. Try to pritn the children for the `dba_ring` and `dba_cell` objects:

```
>>> dba_ring.children
[DBA_CELL, DBA_CELL, DBA_CELL, DBA_CELL, DBA_CELL, DBA_CELL, DBA_CELL, DBA_CELL, DBA_
↪CELL, DBA_CELL, DBA_CELL, DBA_CELL, DBA_CELL, DBA_CELL, DBA_CELL, DBA_CELL]
>>> dba_cell.children
[Drift, Dipole, Drift, Quadrupole, Drift, Dipole, Drift]
```

This can be also visualized by calling the *`Lattice.print_tree()`* method:

```
>>> dba_ring.print_tree()
DBA_RING
├── DBA_CELL
```

(continues on next page)

(continued from previous page)

```
│   ├── Drift
│   ├── Dipole
│   ├── Drift
│   ├── Quadrupole
│   ├── Drift
│   ├── Dipole
│   └── Drift
# ... 14 times more ...
└── DBA_CELL
    ├── Drift
    ├── Dipole
    ├── Drift
    ├── Quadrupole
    ├── Drift
    ├── Dipole
    └── Drift
```

As a nested structure is not always convenient to work with, there are three other representations of the nested *children* attribute:

1. The *sequence* attribute

   To loop over the exact sequence of objects there is the *Lattice.sequence* attribute, which is a list of *Element* objects. It can be thought of a flattened version of *children*. The *sequence* attribute can be used in regular Python `for ... in` loops:

   ```
   >>> sum(element.length for element in dba_ring.sequence)
   192
   ```

   As the `dba_cell` does not contain any other lattices, the *sequence* and *children* attributes should be equal:

   ```
   >>> dba_cell.children == dba_cell.sequence
   True
   ```

   On the other hand, the *sequence* attribute of the `dba_ring` should look different then its *children*:

   ```
   >>> dba_ring.sequence
   [Drift, Dipole, Drift, Quadrupole, Drift, Dipole, Drift, Drift, Dipole, Drift,␣
   ↪Quadrupole, Drift, Dipole, Drift, Drift, Dipole, Drift, Quadrupole, Drift, Dipole,
   ↪ Drift, Drift, Dipole, Drift, Quadrupole, Drift, Dipole, Drift, Drift, Dipole,␣
   ↪Drift, Quadrupole, Drift, Dipole, Drift, Drift, Dipole, Drift, Quadrupole, Drift,␣
   ↪Dipole, Drift, Drift, Dipole, Drift, Quadrupole, Drift, Dipole, Drift, Drift,␣
   ↪Dipole, Drift, Quadrupole, Drift, Dipole, Drift, Drift, Dipole, Drift, Quadrupole,
   ↪ Drift, Dipole, Drift, Drift, Dipole, Drift, Quadrupole, Drift, Dipole, Drift,␣
   ↪Drift, Dipole, Drift, Quadrupole, Drift, Dipole, Drift, Drift, Dipole, Drift,␣
   ↪Quadrupole, Drift, Dipole, Drift, Drift, Dipole, Drift, Quadrupole, Drift, Dipole,
   ↪ Drift, Drift, Dipole, Drift, Quadrupole, Drift, Dipole, Drift, Drift, Dipole,␣
   ↪Drift, Quadrupole, Drift, Dipole, Drift, Drift, Dipole, Drift, Quadrupole, Drift,␣
   ↪Dipole, Drift]
   ```

2. The *elements* attribute

   If the elements are loaded from a lattice file, you do no have individual Python variables to access them. For this purpose you can use the *Lattice.elements* attribute, which is a key value pair of `Element.name` and *Element* objects. You can access a specific element of a lattices with:

```
>>> drift = dba_ring['Drift']
>>> drift.length
2.25
```

To loop over all elements in no specific order use `Lattice.elements.values()`

```
>>> for element in dba_ring.elements.values():
>>>   print(element.name, element.length)
Drift 2.25
Dipole 1
Quadrupole 1
```

Note: In contrary to `lattice` elements to not appear multiple times in `elements.values()`. It can be thought of as a `set` of `lattice`.

---

3. The *sub_lattices* attribute

   This attribute is equivalent to the *elements* attribute but for lattices. It contains all sub-lattices within a given lattice, including grandchildren, great grandchildren, etc. The *sub_lattices* attribute should be empty for the `dba_cell` as it does not contain any other lattices:

   ```
   >>> dba_cell.sub_lattices
   {}
   ```

3.2.4 Adding and Removing Objects

As adding and removing objects from the *children* significantly increased the code complextetiy, so it was decided that *children* cannont be altered after a *Lattice* instance was created. If you needed to add/remove an object just create a new *Lattice* instance or add an *Element* with length zero, which can be altered when needed.

3.2.5 Load and Save Lattice Files

lattices can also be imported from a lattice file. This can be done using the `Lattice.from_file()` method:

```
lattice = ap.Lattice.from_file('/path/to/file')
```

Individual elements and sub-lattices can be accessed through the *elements* and *sub_lattices*, respectively:

```
bend = lattice.elements['bend']
sub_cell = lattice.sub_lattices['sub_cell']
```

A given lattice can be saved to a lattice file using the `save_lattice()` function:

```
ap.Lattice.from_file(lattice, '/path/to/file')
```

3.3 The Twiss class

The *Twiss* class acts as container object for the Twiss parameter. Create a new *Twiss* object for our DBA lattice:

```
twiss = ap.Twiss(dba_ring)
```

The orbit position, horizontal beta and dispersion functions can be accessed through the *s*, *beta_x* and *eta_x* attributes, respectively:

```
beta_x = twiss.beta_x
s = twiss.s
eta_x = twiss.beta_x
```

These are simple numpy.ndarray objects and can simply plotted using matplotlib:

```
import matplotlib.pyplot as plt
plt.plot(s, beta_x, s, eta_x)
```

The tunes and betatron phase are available via *tune_x* and *psi_x*. To view the complete list of all attributes click *Twiss* .

3.4 The Tracking class

Similar to the *Twiss* class the Tracking class acts as container for the tracking data. Before creating a new Tracking object we need to create an initial particle distribution:

```
n_particles = 100
dist = create_particle_dist(n_particles, x_dist='uniform', x_width=0.001)
```

Now a Tracking object for dba_ring can be created with:

```
track = ap.Tracking(dba_ring, dist, n_turns=2)
```

Now one could either plot horizontal particle trajectory:

```
plt.plot(track.s, track.x)
```

Or, picture the particle movement within the horizontal phase space:

```
plt.plot(track.x, track.x_dds)
```

3.5 Lattice File Format

The layout and order of elements within an accelerator is usually stored in a so-called "lattice file". There are a variety of different lattice files and different attempts to unify them:

- MAD and elegant have relatively human readable lattice files but are difficult to parse and also not commonly used in other areas.
- The Accelerator Markup Language (AML) is based on XML, which is practical to describe the hierarchical data structure elements within an accelerator lattice, and can be parsed by different languages. XML's main drawback is that it is fairly verbose, hence less human readable and has become less common recently.

apace tries to get the best out of both worlds and uses a JSON based lattice file. JSON is able to describe complex data structures, has a simple syntax and is available in all common programming language.

apace lattice file for a simple fodo cell:

```
{
  "version": "2.0",
  "title": "FODO Lattice",
  "info": "This is the simplest possible strong focusing lattice. (from Klaus Wille␣
↪Chapter 3.13.3)",
  "root": "FODO",
  "elements": {
    "D1": ["Drift", {"length": 0.55}],
    "Q1": ["Quadrupole", {"length": 0.2, "k1": 1.2}],
    "Q2": ["Quadrupole", {"length": 0.4, "k1": -1.2}],
    "B1": ["Dipole", {"length": 1.5, "angle": 0.392701, "e1": 0.1963505, "e2": 0.1963505}
↪]
  },
  "lattices": {
    "FODO": ["Q1", "D1", "B1", "D1", "Q2", "D1", "B1", "D1", "Q1"]
  }
}
```

3.6 Implementation Details

3.6.1 Signals and Events

As we have already seen in the *Parent lattices* section, the *length* of of a *Lattice* gets updated whenever the length of one of its *Element* objects changes. The same happens for the transfer matrices of the *Twiss* object. This is not only convenient - as one does not have to call an `update()` function every time an attribute changes - but is also more efficient, because apace has internal knowledge about which elements have changed and can accordingly only update the transfer matrices which have actually changed.

This is achieved by a so called Observer Pattern, where an **subject** emits an **event** to all its **observers** whenever its state changes.

These events are implemented by the *Signal* class. A callback can be connected to a given *Signal* through the *connect()* method. Calling an instance of the *Signal* will have the same effect as calling all connected callbacks.

Example: Each *Element* has a `length_changed` signal, which gets emitted whenever the length of the element changes. You can check this yourself by connecting your own callback to the `length_changed` signal:

```
>>> callback = lambda: print("This is a callback")
>>> drift = ap.Drift('Drift', length=2)
>>> drift.length_changed.connect(callback)
>>> drift.length += 1
This is a callback
```

This may not seem useful at first, but can be handy for different optimization tasks. Also apace internally heavily relies on this event system.

3.6.2 Lazy Evaluation

In addition to the event system apace also makes use of Lazy Evaluation. This means that whenever an object changes its state, it will only notify its dependents that an updated is needed. The recalculation of the dependents's new attribute will be delayed until the next time it is accessed.

This lazy evaluation scheme is especially important in combination with the signal system as it can prevent unnecessary calculations: Without the lazy evaluation scheme computational expensive properties will get recalculated whenever one of its dependents changes. With the lazy evaluation scheme they are only calculated if they are actually accessed.

To check if a property needs to be updated one can log the private variable `_needs_update` variables:

```
>>> drift = ap.Drift("Drift", length=2)
>>> lattice = ap.Lattice('Lattice', drift)
>>> drift.length = 1
>>> lattice._length_needs_update
True
```

Warning: The `_needs_update` variables are meant for internal use only!

CHAPTER

FOUR

TUTORIALS

This sections contains various tutorials on how to use the apace library. All those examples can also be found here. At the end of each tutorial are also separate links to download the particular tutorial as Python script/Jupyter notebook or view it directly in the Jupyter nbviewer.

The examples have to be executed from the `examples` directory to correclty resolve the paths.

4.1 Necktie Plot

Plot the lattice stability in dependence of the quadrupole strengths.

Create a new FODO lattice

```
import apace as ap

D1 = ap.Drift("D1", length=0.55)
d1 = ap.Drift("D1", length=0.55)
b1 = ap.Drift("B1", length=1.5)
b1 = ap.Dipole("B1", length=1.5, angle=0.392701, e1=0.1963505, e2=0.1963505)
q1 = ap.Quadrupole("Q1", length=0.2, k1=1.2)
q2 = ap.Quadrupole("Q2", length=0.4, k1=-1.2)
fodo_cell = ap.Lattice("FODO_CELL", [q1, d1, b1, d1, q2, d1, b1, d1, q1])
fodo_ring = ap.Lattice("FODO_RING", [fodo_cell] * 8)
```

Scan the quadrupole strength and record lattice stability

```
import numpy as np

n_steps = 100
k1_start = 0
k1_end = 2

q1_values = np.linspace(k1_start, k1_end, n_steps)
q2_values = np.linspace(k1_start, -k1_end, n_steps)
stable = np.empty((n_steps, n_steps), dtype=bool)

twiss = ap.Twiss(fodo_ring)

for i, q1.k1 in enumerate(q1_values):
    for j, q2.k1 in enumerate(q2_values):
        stable[i, j] = twiss.stable
```

Plot results using matplotlib

```
import matplotlib.pyplot as plt

x, y = np.meshgrid(q1_values, -q2_values)
CS = plt.contour(x, y, stable)
plt.xlabel(f"{q1.name} k1 / m$^{{-1}}$")
plt.ylabel(f"{q2.name} -k1 / m$^{{-1}}$")
plt.show()

# TODO change Bend to Drift -> changes necktieplot to center
```

Total running time of the script: (0 minutes 0.000 seconds)

4.2 Twiss parameter of a FODO lattice

This example shows how to calulate and plot the Twiss parameter of a FOOD lattice.

```
import numpy as np
import apace as ap

D1 = ap.Drift("D1", length=0.55)
d1 = ap.Drift("D1", length=0.55)
b1 = ap.Dipole("B1", length=1.5, angle=0.392701, e1=0.196350, e2=0.196350)
q1 = ap.Quadrupole("Q1", length=0.2, k1=1.2)
q2 = ap.Quadrupole("Q2", length=0.4, k1=-1.2)
fodo_cell = ap.Lattice("FODO_CELL", [q1, d1, b1, d1, q2, d1, b1, d1, q1])
fodo_ring = ap.Lattice("FODO_RING", [fodo_cell] * 8)
```

Output some info on the FODO lattice

```
print(
    f"Overview of {fodo_ring.name}",
    f"Num of elements: {len(fodo_ring.sequence)}",
    f"Lattice Length : {fodo_ring.length}",
    f"Cell Length    : {fodo_cell.length}",
    sep="\n",
)
```

Create a new `Twiss` object to calculate the Twiss parameter

```
twiss = ap.Twiss(fodo_ring)

print(
    f"Twiss parameter of {fodo_ring.name}",
    f"Stable in x-plane: {twiss.stable_x}",
    f"Stable in y-plane: {twiss.stable_y}",
    f"Horizontal tune  : {twiss.tune_x:.3f}",
    f"Vertical tune    : {twiss.tune_y:.3f}",
    f"Max beta x       : {np.max(twiss.beta_x):.3f}",
    f"Max beta y       : {np.max(twiss.beta_y):.3f}",
    sep="\n",
)
```

Use the builtin *plot_lattice* utility function to plot the Twiss parameter

```
from apace.plot import TwissPlot

fig = TwissPlot(twiss, fodo_ring).fig
```

Total running time of the script: (0 minutes 0.000 seconds)

4.3 Tune Measurement

Get tune from Fourier-Transform of transversal particle oscillation at fixed position

Lattice file:

```
{
  "version": "2.0",
  "title": "FODO Lattice",
  "info": "This is the simplest possible strong focusing lattice. (from Klaus Wille␣
↪Chapter 3.13.3)",
  "root": "FODO",
  "elements": {
    "D1": ["Drift", {"length": 0.55}],
    "Q1": ["Quadrupole", {"length": 0.2, "k1": 1.2}],
    "Q2": ["Quadrupole", {"length": 0.4, "k1": -1.2}],
    "B1": ["Dipole", {"length": 1.5, "angle": 0.392701, "e1": 0.1963505, "e2": 0.1963505}
↪]
  },
  "lattices": {
    "FODO": ["Q1", "D1", "B1", "D1", "Q2", "D1", "B1", "D1", "Q1"]
  }
}
```

Some imports ...

```
import numpy as np
from pathlib import Path
from scipy.fftpack import fft
import apace as ap
import matplotlib.pyplot as plt
from math import sqrt
```

Load FODO lattice from file

```
fodo = ap.Lattice.from_file("../data/lattices/fodo_cell.json")
```

Create particle distribution

```
n_particles = 5
n_turns = 50
position = 0
dist = ap.distribution(n_particles, x_dist="uniform", x_width=0.002, x_center=0.001)

matrix_tracking = ap.TrackingMatrix(fodo, dist, turns=n_turns, watch_points=[0])
```

Plot x-x' phase space

```
plt.subplot(2, 2, 1)

for i in range(n_particles):
    plt.plot(matrix_tracking.x[:, i], matrix_tracking.x_dds[:, i], "o")

plt.xlabel("x / m")
plt.ylabel("x'")

freq = np.linspace(0.0, 1.0 / (2.0 * fodo.length / 299_792_458), n_turns // 2)
fft_tracking = 2.0 / n_turns * np.abs(fft(matrix_tracking.x[:, -1])[: n_turns // 2])
main_freq = freq[np.argmax(fft_tracking)]

# Plot horizontal frequency spectrum
plt.subplot(2, 2, 2)
plt.plot(freq, fft_tracking)
plt.xlabel("Freq / Hz")
plt.ylabel("Fourier transform")
plt.axvline(x=main_freq, color="k")

# Plot horizontal offset for fixed position
plt.subplot(2, 2, 3)
plt.plot(matrix_tracking.orbit_position, matrix_tracking.x[:, -1], "rx")
plt.xlabel(f"orbit position / s")
plt.ylabel(f"horizontal offset x at fixed position {position} / m")

# Plot horizontal offset for multiple positions
matrix_tracking_all_positions = ap.TrackingMatrix(fodo, dist, turns=n_turns)

plt.subplot(2, 2, 4)
plt.plot(
    matrix_tracking_all_positions.orbit_position,
    matrix_tracking_all_positions.x[:, -1],
    linewidth=0.5,
)
plt.plot(matrix_tracking.orbit_position, matrix_tracking.x[:, -1], "rx")
plt.xlabel("orbit position / s")
plt.ylabel("horizontal offset for all positions / m")


plt.gcf().set_size_inches(16, 8)
```

Total running time of the script: (0 minutes 0.000 seconds)

CHAPTER

FIVE

COMMAND LINE TOOL

apace also has a simple command line tool which gets automatically installed when installing with pip. This tool is currently work in progress, but the calculation of the Twiss parameter should already be functioning.

5.1 Installing the CLI

The **apace-cli** should be already available if apace was installed using pip. It can be invoked from the command line via:

```
apace
```

5.2 Getting Help

To get help use the `--help` flag,

```
apace --help
```

which should output something like this:

```
usage: apace [-h] [--version] {help,twiss,convert} ...

This is the apace CLI.

positional arguments:
  {help,twiss,convert}
    help                Get help
    twiss               plot or save twiss functions to file
    convert             convert lattice files.

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
```

5.3 The twiss subcommand

Plot Twiss parameter for a given lattice:

```
apace twiss path/to/lattice.json
```

Other options:

```
usage: apace twiss [-h] [-o OUTPUT_PATH] [-v] [-q] [-show]
                   [-ref REF_LATTICE_PATH] [-y_min Y_MIN] [-y_max Y_MAX]
                   [-s SECTIONS] [-pos POSITIONS] [-m MULTI_KNOB]
                   path [path ...]

positional arguments:
  path                  Path to lattice file or directory with lattice files.

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT_PATH, --output_path OUTPUT_PATH
                        Output path for plot
  -v, --verbose         Verbose
  -q, --quiet           Quiet
  -show, --show_plot    show interactive plot
  -ref REF_LATTICE_PATH, --ref_lattice_path REF_LATTICE_PATH
                        Path to reference lattice
  -y_min Y_MIN          Min Y-value
  -y_max Y_MAX          Max Y-value
  -s SECTIONS, --sections SECTIONS
                        Plot Twiss parameter at given sections. Can be a
                        2-tuple (START, END), the name of the section or
                        sequence those '[(START, END), SECTION_NAME, ...]'.
  -pos POSITIONS, --positions POSITIONS
                        Print Twiss parameter at given positions. Can be a
                        number, a 2-tuple (START, END), a section name or
                        sequence of those.
  -m MULTI_KNOB, --multi_knob MULTI_KNOB
                        Multi-knob (Assumes plot)
```

CHAPTER

SIX

API REFERENCE

This is the API reference. To learn how to use and work with apace, see *User Guide*.

- `__version__ = 0.1.0`
- `__license__ = GNU General Public License v3.0`

6.1 Classes

- *Base* - Abstract base for all element and lattice classes.
- *Element* - Abstract base for all element classes.
- *Drift* - A drift space element.
- *Dipole* - A dipole element.
- *Quadrupole* - A quadrupole element.
- *Sextupole* - A sextupole element.
- *Octupole* - An octupole element.
- *Lattice* - Defines the order of elements in the accelerator.
- *MatrixMethod* - The transfer matrix method.
- *Twiss* - Calculate the Twiss parameter for a given lattice.
- *TrackingMatrix* - Particle tracking using the transfer matrix method.
- *Signal* - A callable signal class to which callbacks can be registered.

6.2 Functions

- *distribution()* - Create a particle distribution array (6, N).

6.3 Exceptions

- *AmbiguousNameError* - Raised if multiple elements or lattices have the same name.
- *UnstableLatticeError* - Raised if no stable solution exists for the lattice.

6.4 Detailed Overview

class Base(*name*, *length*, *info=''*)

Abstract base for all element and lattice classes.

Parameters

- **name** (*str*) – The name of the object.
- **info** (*str, optional*) – Additional information about the object.

Attributes

name :str
The name of the object.

info :str
Additional information about the object

parent_lattices :Set[Lattice]
All lattices which contain the object.

length
Length of the object (m).

Methods

__repr__()
Return repr(self).

__str__()
Return str(self).

class Element(*name*, *length*, *info=''*)

Inherits: *Base*

Abstract base for all element classes.

Parameters

- **name** (*str*) – The name of the element.
- **length** (*float*) – The length of the element (m).
- **info** (*str, optional*) – Additional information about the element.

Attributes

attribute_changed :apace.utils.Signal
Gets emitted when one of the attributes changes.

length
Length of the element (m).

Methods

_on_attribute_changed(*element*, *attribute*)

__repr__()
Return repr(self).

__str__()
Return str(self).

class Drift(*name*, *length*, *info=''*)
Inherits: *Element*

A drift space element.

Parameters

- **name** (*str*) – The name of the element.
- **length** (*float*) – The length of the element (m).
- **info** (*str, optional*) – Additional information about the element.

Attributes

attribute_changed :apace.utils.Signal
Gets emitted when one of the attributes changes.

length
Length of the element (m).

Methods

_on_attribute_changed(*element*, *attribute*)

__repr__()
Return repr(self).

__str__()
Return str(self).

class Dipole(*name*, *length*, *angle*, *e1=0*, *e2=0*, *info=''*)
Inherits: *Element*

A dipole element.

Parameters

- **name** (*str*) – Name of the element.
- **length** (*float*) – Length of the element (m).
- **angle** (*float*) – Deflection angle in rad.
- **e1** (*float, optional*) – Entrance angle in rad.
- **e2** (*float, optional*) – Exit angle in rad.
- **info** (*str, optional*) – Additional information about the element.

Attributes

angle
Deflection angle (rad).

e1
Entrance angle (rad).

e2
Exit angle (rad).

radius
Radius of curvature (m).

k0
Geometric dipole strength or curvature of radius (m).

length
Length of the element (m).

Methods

_on_attribute_changed(*element*, *attribute*)

__repr__()
Return repr(self).

__str__()
Return str(self).

class Quadrupole(*name*, *length*, *k1*, *info=''*)
Inherits: *Element*

A quadrupole element.

Parameters

- **name** (*str*) – Name of the element.
- **length** (*float*) – Length of the element (m).
- **k1** (*float*) – Geometric quadrupole strength (m^-2).
- **info** (*str, optional*) – Additional information about the element.

Attributes

k1
Geometric quadrupole strength (m^-2).

length
Length of the element (m).

Methods

_on_attribute_changed(*element*, *attribute*)

__repr__()
Return repr(self).

__str__()
Return str(self).

class Sextupole(*name*, *length*, *k2*, *info=''*)
Inherits: *Element*

A sextupole element.

Parameters

- **name** (*str*) – Name of the element.
- **length** (*float*) – Length of the element (m).
- **k1** (*float*) – Geometric quadrupole strength (m^-3).
- **info** (*str, optional*) – Additional information about the element.

Attributes

k2
Geometric sextupole strength (m^-3).

length
Length of the element (m).

Methods

_on_attribute_changed(*element*, *attribute*)

__repr__()
Return repr(self).

__str__()
Return str(self).

class Octupole(*name*, *length*, *k3*, *info=''*)
Inherits: *Element*

An octupole element.

Parameters

- **name** (*str*) – Name of the element.
- **length** (*float*) – Length of the element (m).
- **k3** (*float*) – Geometric quadrupole strength (m^-4).
- **info** (*str, optional*) – Additional information about the element.

Attributes

k3
Geometric sextupole strength (m^-1).

length
Length of the element (m).

Methods

_on_attribute_changed(*element*, *attribute*)

__repr__()
Return repr(self).

__str__()
Return str(self).

class Lattice(*name*, *children*, *info=''*)
Inherits: *Base*

Defines the order of elements in the accelerator.

Parameters

- **name** (*str*) – Name of the lattice.
- **children** (*List[Union[Element, Lattice]]*) – List of elements and sub-lattices.
- **info** (*str*) – Additional information about the lattice.

Attributes

length_changed :apace.utils.Signal
Gets emitted when the length of lattice changes.

element_changed :apace.utils.Signal
Gets emitted when an attribute of an element within this lattice changes.

n_elements
The number of elements within this lattice.

length
Length of the lattice.

children
List of direct children (elements or sub-lattices) in physical order.

sequence
List of elements in physical order. (Flattend *children*)

indices
A dict which contains the a *List* of indices for each element. Can be thought of as inverse of sequence. Sub-lattices are associated with the list of indices of their first element.

objects
A Mapping from names to the given *Element* or *Lattice* object.

elements
Unordered set of all elements within this lattice.

sub_lattices
Unordered set of all sub-lattices within this lattice.

Methods

traverse_children() :staticmethod:
Returns iterator which traverses all children of a lattice.

_init_properties()
A recursive helper function to initialize the properties.

__getitem__(*key*)

__del__()

update_length()
Manually update the Length of the lattice (m).

_on_length_changed()

_on_element_changed(*element*, *attribute*)

print_tree()
Print the lattice as tree of objects. (Similar to unix tree command)

_print_tree(prefix='') :staticmethod:

from_file(location, file_format=None) :classmethod:
Creates a new *Lattice* from file at *location* (path or url). :param location: path-like or url-like string which locates the lattice file :type location: Union[AnyStr, Path] :param file_format str: File format of the lattice file :type file_format: str, optional (use file extension) :rtype Lattice

from_dict(data) :classmethod:
Creates a new *Lattice* object from a latticeJSON compliant dictionary.

as_file(*path*, *file_format=None*)

as_dict()
Serializes the *Lattice* object into a latticeJSON compliant dictionary.

__repr__()
Return repr(self).

__str__()
Return str(self).

class MatrixMethod(*lattice*, *steps_per_element=10*, *steps_per_meter=None*, *start_index=None*, *start_position=None*, *energy=None*)
The transfer matrix method.

Parameters

- **lattice** – Lattice which transfer matrices gets calculated for.
- **steps_per_element** (*int*) – Fixed number of steps per element. (ignored if steps_per_meter is passed)
- **steps_per_meter** (*number*) – Fixed number of steps per meter.
- **start_index** (*int*) – Start index for the one-turn matrix and for the accumulated transfer matrices.
- **start_position** (*number*) – Same as start_index but uses position instead of index of the position. Is ignored if start_index is set.
- **energy** (*number*) – Total energy per particle in MeV.

Attributes

energy

gamma

velocity

n_steps
Total number of steps.

element_indices
Contains the indices of each element within the transfer_matrices.

step_size
Contains the step_size for each point. Has length of *n_kicks*

s
Contains the orbit position s for each point. Has length of *n_kicks + 1*.

matrices
Array of transfer matrices with shape (6, 6, n_kicks)

k0
Array of deflections angles with shape (n_kicks).

k1
Array of geometric quadruole strenghts with shape (n_kicks).

start_index
Start index of the one-turn matrix and the accumulated transfer matrices.

start_position
Same as start_index, but position in meter instead of index.

matrices_acc
The accumulated transfer matrices starting from start_index.

Methods

_on_element_changed(*element*, *attribute*)

update_n_steps()
Manually update the total number of kicks.

_on_n_steps_changed()

update_element_indices()
Manually update the indices of each element.

_on_element_indices_changed()

update_step_size()
Manually update the step_size array.

_on_step_size_changed()

update_s()
Manually update the orbit position array s.

_on_s_changed()

update_matrices()
Manually update the transfer_matrices.

update_matrices_acc()

_on_matrices_accumulated_changed()

class Twiss(*lattice*, *, *initial=None*, *start_idx=0*, ***kwargs*)
Inherits: `apace.matrixmethod.MatrixMethod`

Calculate the Twiss parameter for a given lattice.

Parameters

- **lattice** (*Lattice*) – Lattice to calculate the Twiss parameter for.
- **start_idx** (*int, optional*) – Index from which the accumulated array is calculated. This index is also used to calculated the initial twiss parameter using the periodicity condition.
- **initial** (*nd.ndarray, optional*) – Initial Twiss parameter, otherwise periodic solution is used.
- **energy** (*float, optional*) – Energy of the beam in mev

Attributes

start_idx_changed
Gets emitted when the start index changes

one_turn_matrix_changed
Gets emitted when the one turn matrix changes.

twiss_array_changed
Gets emitted when the twiss functions change.

psi_changed
Gets emitted when the betatron phase changes.

tune_fractional_changed
Gets emitted when the fractional tune changes.

start_idx
Index from which the accumulated array is calculated. This index is also used to calculated the initial twiss parameter using the periodicity condition.

accumulated_array
Contains accumulated transfer matrices.

one_turn_matrix
The transfer matrix for a full turn.

term_x
Corresponds to $2 - m_{11}^2 - 2m_{12}m_{21} - m_{22}^2$, where m is the one turn matrix. Can be used to calculate the initial *beta_x* value $\beta_{x0} = |2m_{12}|/\sqrt{term_x}$. If *term_x* > 0, this means that there exists a periodic solution within the horizontal plane.

term_y
Corresponds to $2 - m_{33}^2 - 2m_{34}m_{43} - m_{44}^2$, where m is the one turn matrix. Can be used to calculate the initial *beta_y* value $\beta_{y0} = |2m_{12}|/\sqrt{term_y}$. If *term_y* > 0, this means that there exists a periodic solution within the vertical plane.

stable_x
Periodicity condition *term_x* > 0 for a stable solution in the horizontal plane.

stable_y
Periodicity condition *term_y* > 0 for a stable solution in the vertical plane.

stable
Periodicity condition *term_x* > 0 and *term_y* > 0 for a stable solution in both planes.

initial_twiss
Array containing the initial twiss parameter.

twiss_array
Contains the twiss parameter.

beta_x
Horizontal beta function.

beta_y
Vertical beta function.

alpha_x
Horizontal alpha function.

alpha_y
Vertical alpha function.

gamma_x
Horizontal gamma function.

gamma_y
Vertical gamma function.

eta_x
Horizontal dispersion function.

eta_x_dds
Derivative of the horizontal dispersion with respect to s.

psi_x
Horizontal betatron phase.

psi_y
Vertical betatron phase.

tune_x
Horizontal tune. Corresponds to psi_x[-1] / 2 pi. Strongly depends on the selected step size.

tune_y
Vertical tune. Corresponds to psi_y[-1] / 2 pi. Strongly depends on the selected step size.

tune_x_fractional
Fractional part of the horizontal tune (Calculated from one-turn matrix).

tune_y_fractional
Fractional part of the vertical tune (Calculated from one-turn matrix).

chromaticity_x
Natural Horizontal Chromaticity. Depends on *n_kicks*

chromaticity_y
Natural Vertical Chromaticity. Depends on *n_kicks*

curly_h
The curly H function.

i1
The first synchrotron radiation integral.

i2
The second synchrotron radiation integral.

i3
The third synchrotron radiation integral.

i4
The fourth synchrotron radiation integral.

i5
The fifth synchrotron radiation integral.

alpha_c
Momentum Compaction Factor. Depends on *n_kicks*

gamma

emittance_x

energy

velocity

n_steps
Total number of steps.

element_indices
Contains the indices of each element within the transfer_matrices.

step_size
Contains the step_size for each point. Has length of *n_kicks*

s
Contains the orbit position s for each point. Has length of *n_kicks + 1*.

matrices
Array of transfer matrices with shape (6, 6, n_kicks)

k0
Array of deflections angles with shape (n_kicks).

k1
Array of geometric quadruole strenghts with shape (n_kicks).

start_index
Start index of the one-turn matrix and the accumulated transfer matrices.

start_position
Same as start_index, but position in meter instead of index.

matrices_acc
The accumulated transfer matrices starting from start_index.

Methods

update_one_turn_matrix()
Manually update the one turn matrix and the accumulated array.

_on_one_turn_matrix_changed()

update_twiss_array()
Manually update the twiss_array.

_on_twiss_array_changed()

update_betatron_phase()
Manually update the betatron phase psi and the tune.

_on_psi_changed()

update_fractional_tune()
Manually update the fractional tune.

_on_tune_fractional_changed()

update_chromaticity()
Manually update the natural chromaticity.

_on_chromaticity_changed()

_on_curly_h_changed()

_on_i1_changed()

_on_i2_changed()

_on_i3_changed()

_on_i4_changed()

_on_i5_changed()

_on_alpha_c_changed()

_on_emittance_changed()

_on_element_changed(*element*, *attribute*)

update_n_steps()
Manually update the total number of kicks.

_on_n_steps_changed()

update_element_indices()
Manually update the indices of each element.

_on_element_indices_changed()

update_step_size()
Manually update the step_size array.

_on_step_size_changed()

update_s()
Manually update the orbit position array s.

_on_s_changed()

update_matrices()
Manually update the transfer_matrices.

update_matrices_acc()

_on_matrices_accumulated_changed()

class **TrackingMatrix**(*lattice*, *initial_distribution*, *turns=1*, *watch_points=None*, *start_point=0*, ***kwargs*)
Inherits: apace.matrixmethod.MatrixMethod

Particle tracking using the transfer matrix method.

Parameters

- **lattice** (Lattice) – Lattice which particles will be tracked through.
- **initial_distribution** (*np.ndarray*) – Initial particle distribution.
- **turns** (*int*) – Number of turns.
- **watch_points** (*array-like, optional*) – List of watch points. If unset all particle trajectory will be saved for all positions. Indices correspont to orbit_positions.
- **start_point** (*int*) – Point at which the particle tracking begins.

Attributes

watch_points

initial_distribution

particle_trajectories
Contains the 6D particle trajectories.

orbit_position

x

x_dds

y

y_dds

lon

delta

energy

gamma

velocity

n_steps
Total number of steps.

element_indices
Contains the indices of each element within the transfer_matrices.

step_size
Contains the step_size for each point. Has length of *n_kicks*

s
Contains the orbit position s for each point. Has length of *n_kicks + 1*.

matrices
Array of transfer matrices with shape (6, 6, n_kicks)

k0
Array of deflections angles with shape (n_kicks).

k1
Array of geometric quadruole strenghts with shape (n_kicks).

start_index
Start index of the one-turn matrix and the accumulated transfer matrices.

start_position
Same as start_index, but position in meter instead of index.

matrices_acc
The accumulated transfer matrices starting from start_index.

Methods

update_particle_trajectories()
Manually update the 6D particle trajectories

_on_particle_trajectories_changed()

_on_element_changed(*element*, *attribute*)

update_n_steps()
Manually update the total number of kicks.

_on_n_steps_changed()

update_element_indices()
Manually update the indices of each element.

_on_element_indices_changed()

update_step_size()
Manually update the step_size array.

_on_step_size_changed()

update_s()
Manually update the orbit position array s.

_on_s_changed()

update_matrices()
Manually update the transfer_matrices.

update_matrices_acc()

_on_matrices_accumulated_changed()

class Signal(**signals*)
A callable signal class to which callbacks can be registered.

When ever the signal is emitted all registered functions are called.

Parameters signals (Signal, *optional*) – Signals which this signal gets registered to.

Attributes

callbacks
Functions called when the signal is emitted.

__repr__

Methods

__call__(**args*, ***kwargs*)
Emit signal and call registered functions.

__str__()
Return str(self).

connect(*callback*)
Connect a callback to this signal.

Parameters **callback** (*function*) – Function which gets called when the signal is emitted.

distribution(*n_particles*, *x_dist=None*, *x_center=0*, *x_width=0*, *y_dist=None*, *y_center=0*, *y_width=0*, *x_dds_dist=None*, *x_dds_center=0*, *x_dds_width=0*, *y_dds_dist=None*, *y_dds_center=0*, *y_dds_width=0*, *l_dist=None*, *l_center=0*, *l_width=0*, *delta_dist=None*, *delta_center=0*, *delta_width=None*)
Create a particle distribution array (6, N).

Parameters

- **n_particles** (*int*) – Number of particles.
- **x_dist** (*str*) – Type of distribution in horizontal phase space.
- **x_center** (*float*) – Center of distribution.
- **x_width** (*float*) – Width of distribution.
- **y_dist** (*str*) – Type of distribution in vertical phase space.
- **y_center** (*float*) – Center of distribution.
- **y_width** (*float*) – Width of distribution.
- **x_dds_dist** (*str*) – Type of distribution in horizontal slope phase space.
- **x_dds_center** (*float*) – Center of distribution.
- **x_dds_width** (*float*) – Width of distribution.
- **y_dds_dist** (*str*) – Type of distribution in vertical slope phase space.
- **y_dds_center** (*float*) – Center of distribution.
- **y_dds_width** (*float*) – Width of distribution.
- **l_dist** (*str*) – Type of distribution in longitudinal phase space.
- **l_center** (*float*) – Center of distribution.
- **l_width** (*float*) – Width of distribution.
- **delta_dist** (*str*) – Type of distribution in momentum phase space.
- **delta_center** (*float*) – Center of distribution.
- **delta_width** (*float*) – Width of distribution.

Returns Array of shape (6, n_particles)

Return type numpy.ndarray

exception AmbiguousNameError(*name*)

Inherits: Exception

Raised if multiple elements or lattices have the same name.

Parameters **name** (*str*) – The ambiguous name.

class __cause__
exception cause

class __context__
exception context

class __suppress_context__

class __traceback__

class args

Methods

__delattr__()
Implement delattr(self, name).

__dir__()
Default dir() implementation.

__eq__()
Return self==value.

__format__()
Default object formatter.

__ge__()
Return self>=value.

__getattribute__()
Return getattr(self, name).

__gt__()
Return self>value.

__hash__()
Return hash(self).

__le__()
Return self<=value.

__lt__()
Return self<value.

__ne__()
Return self!=value.

__reduce__()
Helper for pickle.

__reduce_ex__()
Helper for pickle.

__repr__()
Return repr(self).

__setattr__()
Implement setattr(self, name, value).

__setstate__()

__sizeof__()
Size of object in memory, in bytes.

__str__()
Return str(self).

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception UnstableLatticeError(*twiss*)
Inherits: Exception

Raised if no stable solution exists for the lattice.

class __cause__
exception cause

class __context__
exception context

class __suppress_context__

class __traceback__

class args

Methods

__delattr__()
Implement delattr(self, name).

__dir__()
Default dir() implementation.

__eq__()
Return self==value.

__format__()
Default object formatter.

__ge__()
Return self>=value.

__getattribute__()
Return getattr(self, name).

__gt__()
Return self>value.

__hash__()
Return hash(self).

__le__()
Return self<=value.

__lt__()
Return self<value.

__ne__()
Return self!=value.

__reduce__()
Helper for pickle.

__reduce_ex__()
Helper for pickle.

__repr__()
Return repr(self).

__setattr__()
Implement setattr(self, name, value).

__setstate__()

__sizeof__()
Size of object in memory, in bytes.

__str__()
Return str(self).

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

6.5 Submodules

6.5.1 apace.plot

Module Contents

Classes

Color	
TwissPlot	Convenience class to plot twiss parameters

Functions

draw_elements(ax, lattice, *, labels = True, location = 'top')	Draw the elements of a lattice onto a matplotlib axes.
draw_sub_lattices(ax, lattice, *, labels = True, location = 'top')	
plot_twiss(ax, twiss, *, twiss_functions=('beta_x', 'beta_y', 'eta_x'), scales={'eta_x': 10, 'eta_x_dds': 10}, line_style='solid', line_width=1.3, alpha=1.0, show_ylabels=False)	
_twiss_plot_section(ax, twiss, *, x_min=-inf, x_max=inf, y_min=None, y_max=None, annotate_elements=True, annotate_lattices=True, line_style='solid', line_width=1.3, twiss_ref=None, scales={'eta_x': 10}, overwrite=False)	
find_optimal_grid(n)	
floor_plan(ax, lattice, *, start_angle = 0, labels = True)	

Attributes

FONT_SIZE
ELEMENT_COLOR
OPTICAL_FUNCTIONS

apace.plot.**FONT_SIZE = 8**

class apace.plot.**Color**

Attributes

RED = #EF4444

YELLOW = #FBBF24

GREEN = #10B981

BLUE = #3B82F6

ORANGE = #F97316

PURPLE = #8B5CF6

CYAN = #06B6D4

WHITE = white

BLACK = black

LIGHT_GRAY = #D1D5DB

apace.plot.**ELEMENT_COLOR :Dict[type, str]**

apace.plot.**OPTICAL_FUNCTIONS**

apace.plot.**draw_elements**(*ax*, *lattice*, *, *labels=True*, *location='top'*)
Draw the elements of a lattice onto a matplotlib axes.

Parameters

- **ax** (*matplotlib.axes.Axes*) –
- **lattice** (*apace.classes.Lattice*) –
- **labels** (*bool*) –
- **location** (*str*) –

apace.plot.**draw_sub_lattices**(*ax*, *lattice*, *, *labels=True*, *location='top'*)

Parameters

- **ax** (*matplotlib.axes.Axes*) –
- **lattice** (*apace.classes.Lattice*) –
- **labels** (*bool*) –
- **location** (*str*) –

apace.plot.**plot_twiss**(*ax*, *twiss*, *, *twiss_functions=('beta_x', 'beta_y', 'eta_x')*, *scales={'eta_x': 10, 'eta_x_dds': 10}*, *line_style='solid'*, *line_width=1.3*, *alpha=1.0*, *show_ylabels=False*)

apace.plot.**_twiss_plot_section**(*ax*, *twiss*, *, *x_min=- inf*, *x_max=inf*, *y_min=None*, *y_max=None*, *annotate_elements=True*, *annotate_lattices=True*, *line_style='solid'*, *line_width=1.3*, *twiss_ref=None*, *scales={'eta_x': 10}*, *overwrite=False*)

class apace.plot.**TwissPlot**(*twiss*, *twiss_functions=('beta_x', 'beta_y', 'eta_x')*, *, *sections=None*, *y_min=None*, *y_max=None*, *main=True*, *scales={'eta_x': 10}*, *twiss_ref=None*, *title=None*, *pairs=None*)
Convenience class to plot twiss parameters

Parameters

- **twiss** (Twiss) – The name of the object.
- **tuple** (*List[Union[Tuple[float, float], str, Base]*) – List of sections to plot. Can be either (min, max), "name" or object.
- **float** (*y_min*) – Maximum y-limit
- **float** – Minimum y-limit
- **bool** (*main*) – Wheter to plot whole ring or only given sections
- **int]** (*scales Dict[str,*) – Optional scaling factors for optical functions
- **twiss_ref** (Twiss) – Reference twiss values. Will be plotted as dashed lines.
- **pairs** (*List[Tuple[*Element, *str]]*) – List of (element, attribute)-pairs to create interactice sliders for.

Methods

update()

apace.plot.**find_optimal_grid**(*n*)

apace.plot.**floor_plan**(*ax*, *lattice*, *, *start_angle=0*, *labels=True*)

Parameters

- **ax** (*matplotlib.axes.Axes*) –
- **lattice** (*apace.classes.Lattice*) –
- **start_angle** (*float*) –
- **labels** (*bool*) –

- genindex

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y