PyAT: Getting Started

Lee Carver, Simon White, ESRF, Grenoble, France

3rd November 2020

Foreword

This guide will not discuss the physics of AT. This is already very well documented in http://atcollab.sourceforge.net/docs.html and various other locations. Instead, we will simply describe the syntax of the most common features of PyAT to get you started and give you a feeling for how to debug and troubleshoot.

Installation

```
git clone https://github.com/atcollab/at
cd at/pyat
python setup.py install
```

or

```
python setup.py develop
```

Then the at/pyat directory must be added to the python path (in the bash_profile or bashrc).

Basics

Load the .mat file. (The examples used here are good for the EBS lattice).

```
>>> import at
>>> ring = at.load_mat('./S28F.mat', mat_key='LOW_EMIT_RING_INJ')
```

To see the functions or parameters that are accessible by the ring variable (or element)

```
>>> print(dir(ring))
```

To make this variable inherit directly many useful functions, you should convert it to a Lattice object

```
>>> ring = at.Lattice(ring)
```

All the functions have a short description which can be seen in python with

```
>>> help(function_name)
```

or in ipython with

```
>>> function_name?
```

For general lattice manipulations the standard list commands work (pop, insert, delete, index, append, prepend). The get_refpts function can be used to search for wildcards in the FamName and also pull out indexes of certain elements.

```
>>> from at import get_refpts
>>> bpm_indexes = get_refpts(ring, 'BPM*')
```

This scans through each element in the ring lattice and returns the index of all elements whose name starts with 'BPM'.

Or also to search for all elements of a certain type.

```
>>> from at import get_refpts
>>> from at.lattice.elements import Sextupole
>>> sext_indexes = get_refpts(ring, Sextupole)
```

This performs the same action as above but now searches for any Sextupole element in the lattice.

Once you have the indexes you can use standard python iteration to read or write values.

```
>>> from at import get_refpts
>>> from at.lattice.elements import Quadrupole
>>> quad_indexes = get_refpts(ring, Quadrupole)
>>> quad_strengths = np.array([ring[ind].K for ind in quad_indexes])
>>> for ind in quad_indexes:
>>>     ring[ind].K *= 1.01
```

Radiation can be switched on or off using the call directly on the lattice. This performs the same action as in matlab, where the pass methods are switched between *RadPass and *Pass

```
>>> ring.radiation_off()
```

or

```
>>> ring.radiation_on()
```

Orbits and Optics

```
>>> from at import get_refpts
>>> bpm_indexes = get_refpts(ring, 'BPM*')
>>> orb0, orb = ring.find_orbit4(dp=0, refpts=bpm_indexes)
```

This runs find_orbit4 and returns the orbit at the bpm locations. Similarly for linopt:

```
>>> from at import get_refpts
>>> bpm_indexes = get_refpts(ring, 'BPM*')
>>> lindata0, tune, chrom, lindata =
        ring.linopt(get_chrom=True, refpts=bpm_indexes)
>>> print(ld.dtype.names)
>>> betax = ld['beta'][:,0]
```

There are two very versatile functions for tune and chromaticity called get_tune and get_chrom. Both of these functions have several possible methods, by default they will both get the tune or chromaticity from linopt, simple usage as below

```
>>> ring.get_tune()
>>> ring.get_chrom()
```

There is also a method to generate a single particle centroid and compute the tune from either the fft or by using a laskar method. All of the necessary variables can be given as kwargs and the descriptions are given in the help. This centroid can also be used to compute the detuning with amplitude.

```
>>> qx0, qy0 = ring.get_tune(method='laskar', nturns=512, ampl=1e-6)
>>> qx, qy = ring.get_tune(method='laskar', nturns=512, ampl=1e-3)
```

windows for the tune computation can also be taken into account. These arguments can also be provided to get_chrom and will be taken into account in the same way.

For higher order chromaticity, the function chromaticity can be used.

```
>>> (fitx, fity), dpa, qz = at.chromaticity(ring,
        method='linopt', dpm=0.02, npoints=11, order=3, dp=0)
```

This uses get_tune so you can also provide the kwargs for the harmonic analysis (as above) in the same way.

To do a quick fit of the tune and adjust

```
>>> from at.matching import fit_tune
>>> from at import get_refpts
>>> fit_tune(ring, get_refpts(ring, 'QF1*'),
        get_refpts(ring, 'QD2*'), [0.1,0.25])
```

This computes the tune response from the QF1 family and QD2 family (names relevant for the ESRF lattice) and then sets the tune to 0,1 0.25.

And likewise

```
>>> from at.matching import fit_chrom
>>> from at import get_refpts
>>> fit_chrom(ring, get_refpts(ring, 'SD1*'),
        get_refpts(ring, 'SF2*'), [10,10])
```

For emittance, synchronous phase, synchrotron frequency, zero current bunch length and energy spread. You can use the function envelope_parameters

```
>>> env = ring.envelope_parameters()
>>> dir(env)
```

AT Plot

There is of course a python version of atplot. It is accessible within the lattice object but only if you import the at.plot module.

```
>>> import at.plot
>>> ring.plot_beta()
```

Fast Ring

There is a fast_ring function that is identical to the fastring function in matlab. It can be called by the following

```
>>> lin_ring, lin_ring_rad = at.fast_ring(ring, split_inds=[])
```

This function reduces the full lattice to a few 6x6 matrices. It moves the cavities to the beginning and replaces the cavities with drifts. It computes a linear matrix, non-linear matrix and a diffusion matrix. The ring without radiation does not include the diffusion matrix. This function is not give the exact same diffusion matrices as matlab, but the difference is very small and under investigation. If split_inds is provided then the lattice is split at the index given and it returns the fast ring for each section.

Tracking

First, particles need to be generated. This is done with a combination of the beam and sigma_matrix functions.

First generate a sigma matrix, then pass this matrix to at.beam

```
>>> sig = at.sigma_matrix([10, 1, 135e-12])
>>> pin = at.beam(100, sig)
```

For the generation of the sigma matrix, the help has all of the details. Basically there is a unique number of values to provide for each case (2x2, 4x4, 6x6, 2x2 longitudinal). You can also provide a ring object i.e.

```
>>> sig = at.sigma_matrix(ring)
```

To do some actual tracking

```
>>> pout = at.track.lattice_pass(ring, pin.copy(), nturns=100, refpts=refpts)
```

The output of the particles is (6, N, R, T) which corresponds to 6 degrees of freedom, N number of particles, R reference points and T number of turns.