Reading IOTA into pyORBIT

Runze Li

August 01 2020

Abstract

Currently, the sequence of IOTA is generated by MADX saving command as a sequence file, and then it is read into pyORBIT. However, there are many differences between pyORBIT and MADX which requires some special treatments in order to make this sequence work as expected. In this report all these special treatments and their necessities are explained.

1 Adding Dipole Edge Element in pyORBIT

In pyORBIT, dipole edge element is currently not supported, so when reading a sequence from MADX with dipole edge pyORBIT will show an error. In order to add dipole edge effect to pyORBIT, changes in 3 parts were made: parser, tracking method, and wrapper.

For the parser part, in teapot.py file a new class named DipedgeTEAPOT is created which represents the dipole edge element. When a line in MADX sequence with type "dipedge" is read by pyORBIT parser, a new DipedgeTEAPOT element is created which takes the input $e1$, h, $hgap$, and $fint$. These names correspond to the dipole edge parameters in MADX, where $e1$ is the rotation angle for the pole face, h is $\frac{1}{\rho}$, $hgap$ is the half gap length of bending magnet, and $fint$ is field integral. After that this element is put into lattice, and during tracking its own tracking method is callled to specify the action.

The tracking method is defined in teapotbase.cc using c++. For the dipole edge it considers the first order linear terms only, according to the transfer matrix[1]:

$$\begin{bmatrix} x \\ x' \end{bmatrix} = M = \begin{bmatrix} 1 & 0 \\ h * tan(e1) & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x'_0 \end{bmatrix}$$
$$\begin{bmatrix} y \\ y' \end{bmatrix} = M = \begin{bmatrix} 1 & 0 \\ -h * tan(e1 - \psi) & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y'_0 \end{bmatrix}$$
$$\psi = fint * 2 * hgap * h * \frac{1+sin^2(e1)}{cos(e1)}$$

So for every particle in the bunch, this tracking method changes its coordinate (x_0, x'_0, y_0, y'_0) according to this transformation. Since for the first order linear approximation, there is no difference except e1 between entrance and exit dipole edge, this transfer matrix is general.

After this, the wrapper file wrap_teapotbase.cc is changed since pyORBIT uses both python and c++. This change passes the parameters of dipole edge read from teapot.py into tracking method in teapotbase.cc during tracking.

As a result, now pyORBIT can handle the dipole edge element in MADX sequence. Without dipole edge, the tune Q_y calculated by pyORBIT is 5.42, and now it is 5.3 which is the same as result from MADX.

2 RF Cavity Issue

The treatment of RF cavity in pyORBIT and MADX is different, and in order to maintain a longitudinal stable bunch the sequence produced by MADX needs to be modified.

According to the design of IOTA[2], the RF Cavity of IOTA has 400 Volt and 0 phase lag, which creates a synchrotron tune of 0.007. In MADX sequence, it is defined as:

$rfc : rfcavity, l := 0.05, volt := 0.0004, lag := 0, harmon := 4;$

In order to get the longitudinal bucket generated by this rf cavity, 100 particles are initialized at (σ_x, 0, σ_y, 0, z[i], 0) where z[i] ranges from 0 to 5*1.7 meters where 1.7 meter is σ_z as defined in IOTA parameters. These particles are tracked for 5000 turns and their z coordinates every time this bunch finishes one turn is used to perform an fft to get longitudinal tune. The relation of longitudinal tune vs initial amplitude is shown in figure 1.

As we see for initial z close to 0, the tune is close to synchrotron tune which is 0.007, then as initial z deviates further from 0 the longitudinal tune decreases to 0. At some point the longitudinal tune begins increasing again, which means that one bucket ends and another bucket begins. As a result, we determine that the size of longitudinal bucket is roughly 3 σ_z.

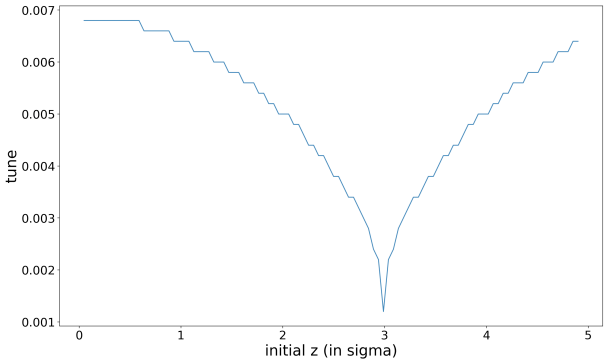

Figure 1: initial z amplitude vs longitudinal tune for MADX

However, when this sequence is read into pyORBIT and the same test is performed, the result becomes:

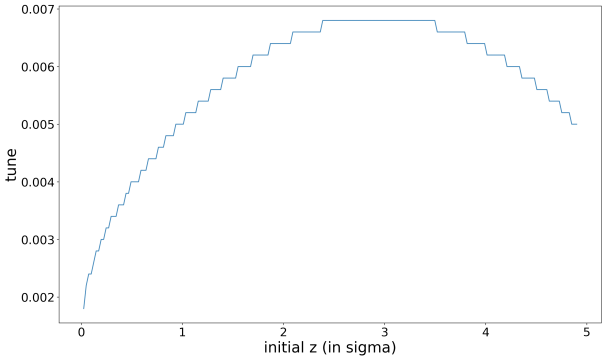

Figure 2: initial z amplitude vs longitudinal tune for pyORBIT

This plot seems to have a 90 degree phase shift. In order to remove this shift, in MADX sequence file either $volt$ needs to be set as -400V or lag needs to be 0.5, which corresponds to a 90 degree phase shift. However, if we want to remove the rf cavity from MADX sequence and manually add it in pyORBIT, no such change should be made and the original set up (i.e. $volt = 400V$ and $lag = 0$) is correct. It seems that the parser of pyORBIT performs a sign flip when reading rf cavity parameters in MADX sequence, while the definition of rf cavity in pyORBIT maintains the same as that of MADX.

Another noteworthy fact is that in pyORBIT, the unit of rf voltage is

Giga volt, while in MADX it is Million volt. So if the rf cavity is removed from sequence and added separately in pyORBIT, we need $volt = 4e^{-7}$

3 Nonlinear Effect of Magnet Edge

The tracking method used by MADX and pyORBIT is different. In MADX the tracking algorithm we use is called thin lens tracking, which firstly converts every elements into thin lens according to an algorithm that minimize the error and then do the tracking. In pyORBIT, however, elements are treated as thick lens and each of them has a pre defined variable called nparts, which specifies how many pieces this element is divided into. For example, for a dipole with nparts = n, it is treated in pyORBIT as:

Fringe In, Entrance, Dipole_1, ..., Dipole_n-2, Exit, Fringe Out

The fringe in and out creates the edge effect of magnets, and the other n parts (including entrance and exit) are main body of this dipole element. The length (and bending angle for dipole) will be distributed among each of these n parts, with $l_{entrance} = l_{exit} = \frac{l_{others}}{2} = \frac{L}{2*(n-1)}$. The entrance contains only linear transformation and other elements has both linear and nonlinear part. Also, this nparts value can at least be 2, since every element needs to have an entrance and an exit.

However, when tracking a test particle starting at (σ_x, 0, σ_y, 0, 0, 0) for 5000 turns and comparing the result of MADX and pyORBIT, we see two very different results. As shown in figure 3:

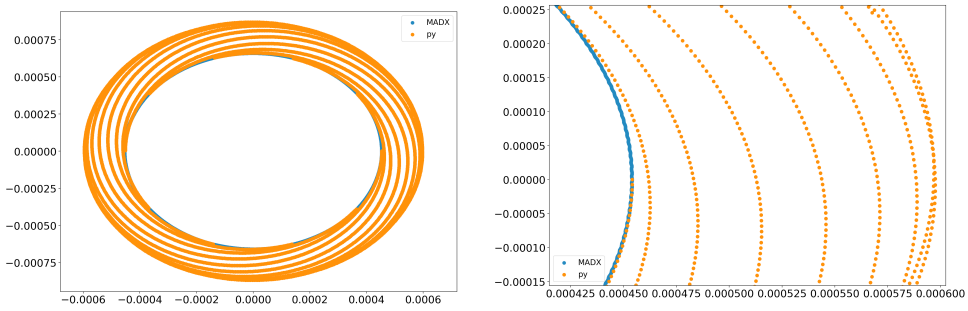

Figure 3: tracking result in x-px space from MADX and pyORBIT

Since $\sigma_x = 0.000454$, we see in the left zoom in plot that the maximum difference on x between MADX and pyORBIT result is larger than 30%. Due to the shape of phase space ellipse produced by pyorbit, there are some nonlinear terms that exists in pyORBIT but not in MADX, and since we have $Q_x = Q_y = 5.3$ here in IOTA, this nonlinear term creates some resonance effects.

After checking the code of pyORBIT and removing all sources of nonlinear effects, it is found that the fringe field in and out at every dipole and quadrupoles is the source of such resonance. After removing their effect, the same tracking result becomes:

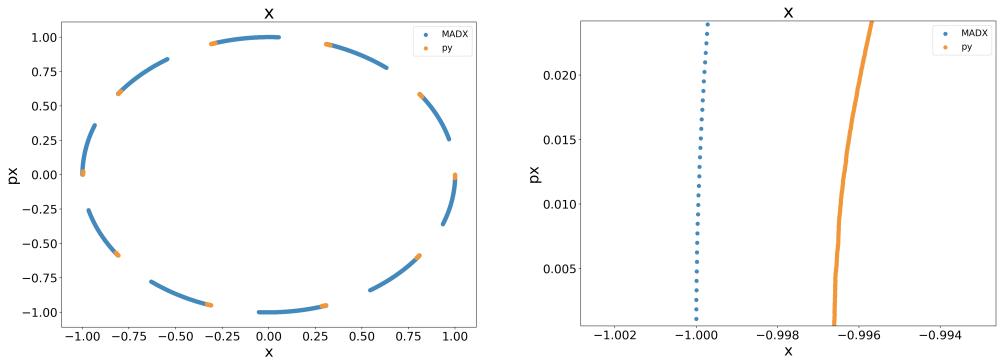

Figure 4: tracking result in x-px space from MADX and pyORBIT without the effect of fringe field, with axis scaled by σ

As a result, the nonlinear resonance in pyORBIT is removed and the difference between result of MADX and pyORBIT is less than 1%. However, although MADX does not consider this fringe nonlinear effect, for small ring like IOTA it might not be neglectable. Currently this effect is ignored in order to make result of MADX and pyORBIT consistant, however, more tests may need to be done to confirm the edge effect of magnets in IOTA.

4 Using Mpirun Features in pyORBIT

Instead of pointing any bugs in pyORBIT, this part serves as an instruction of using mpirun in pyORBIT. Since tracking a bunch usually involves the simulation of millions of macro particles, multiple cpus are usually used to divide the bunch into many parts and track them in parallel. If coherent

effects such space charge effect need to be considered, these parts will be put together. All these features are built in pyORBIT, and in order to run a program in parallel, the command:

mpirun -np #cpus pyORBIT mycode.py

needs to be used. This command will run the program "mycode.py" in parallel on multiple cpus defined by #cpus. With this command, each process will only deal with its own bunch when tracking through elements without coherent effect. For example, if in the python program 1,000 macro particles are added to the bunch and this program is run in parallel on 10 cpus, the total bunch will contains 10,000 macro particles. However, since these processes only communicate when calculating coherent effects, the program will run much faster. Also, pyORBIT has methods that consider every bunch in different processes. For example, in dumpBunch() bunches in all processes will be dumped together into one file, and in readBunch() the whole bunch read from file will be evenly distributed on multiple cpus as individual bunches.

5 Other Changes

There are also several bugs in pyORBIT, and here they are listed and corrected.

First, in IOTA sequence the strength of solenoid is defined to be 0. However, this will causes a division by 0 error in pyORBIT. In order to fix this the strength of solenoid is set to be a very small value, 10^{-10}, instead.

Second, in the py-orbit/py/orbit/teapot/teapot.py file, the function initialize() of class ApertureTEAPOT initializes the Aperture object with Aperture(shape, dim[0], dim[1], 0.0, 0.0), however the Aperture object is created by Aperture(int shape, double a, double b, double c, double d, double pos) so totaly 6 parameters are needed. Here a dummy variable 0 is added in the end to fix this. However based on current output, the treatment of aperture in pyORBIT still has more problems, since it seems to try to find the element of type "Aperture" in sequence, however, in IOTA sequence the aperture is defined using marker with attribute "aperture". As the result, the current solution is either changing in MADX sequence all markers with aperture attribute into type: aperture, or removing them and manually adding them in pyORBIT.

Also, in py-orbit/py/orbit/matrix-lattice/Matrix-Lattice.py, the method trackTwissData(), which calculates phase advance and twiss functions, is modified. This method calculates the phase advance in each turn as delta-phi and sum them up as 2π*tune, however the tune on y calculated in this way for IOTA is q2 = 4.42, which is smaller than what is calculated by MADX. The phase advance plot is shown in the left subplot of Figure 5, comparing to the right side plot calculated by MADX which is mono-increasing.

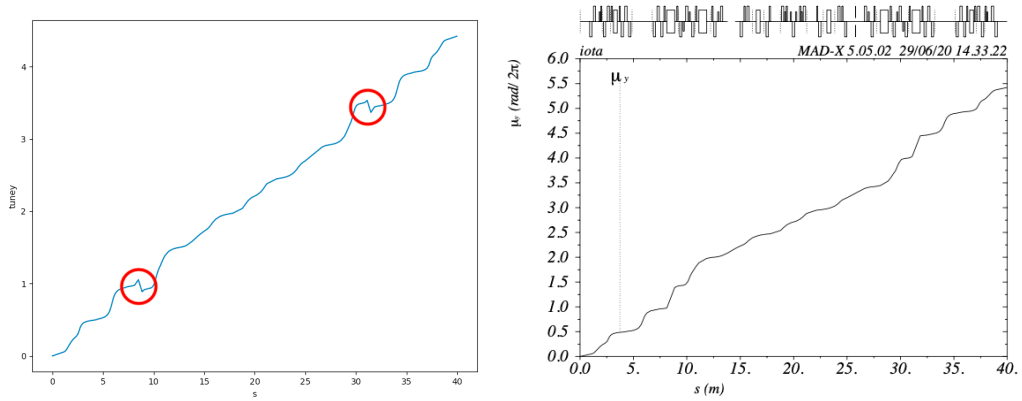

Figure 5: phase advance in y

After printing out the delta-phi value, some of them are found to be negative, which is impossible since they represent the phase advance in each iteration. As a result a condition check is added such that if delta-phi is less than 0, it is increased by π. After that the both phase advance and phase advance plot agree with the result from MADX.

Also, the default number of parts every lattice node is divided into is modified. Since IOTA is a small ring with short radius, the length of dipoles and quadrupoles is small and the strength is large, every element need to be divided into more slices to make sure the accuracy in twiss function calculation. Here the default set up is using npart = 2 for dipole, quadrupole, multipole, and kickers. This value is now changed to 8. Actually the negative delta-phi described in last paragraph is also caused by the inaccuracy in small nparts. Either changing nparts = 8 or adding π to negative delta-phi fixes this issue, as shown in the corrected phase advance plot in y. All 3 methods give the same result line up with each other in figure 6, and it is also the same as MADX result in Figure 5.

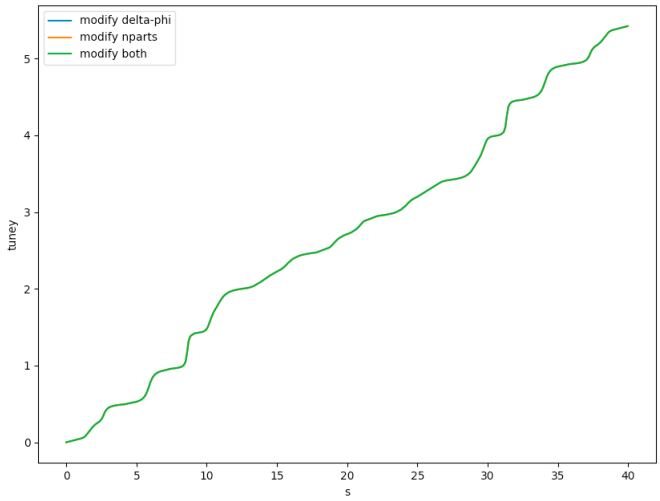

Figure 6: modified phase advance in y

References

[1] Karl L. Brown. A First- and Second-Order Matrix Theory for the Design of Beam Transport Systems and Charged Particle Spectrometers. SLAC Report-75

[2] Sergei Antipov, Daniel Broemmelsiek, David Bruhwiler, and et al. IOTA (Integrable Optics Test Accelerator): Facility and Experimental Beam Physics Program