Py RFQ Helper Module for WARP v1

Jared Hwang
Summer 2018

Introduction

Py RFQ Module is a python module designed to provide a convenient and flexible Radio Frequency Quadrupole (RFQ) object that one can place into a WARP simulation, given any location, size, vane shape, etc..

The necessity for this module stems from the Isotope Decay at Rest (IsoDAR) experiment, and lack of an effective and accurate way to easily simulate an RFQ. IsoDAR is an experiment wherein antineutrinos are produced by accelerating a beam of H_2^+ through a cyclotron which then impinges onto a ^{9}Be target, which produces a large flux of neutrons which then pass through a ^{7}Li sleeve. The ^{7}Li captures the neutrons to produce ^{8}Li, which then beta decays and produces $\bar{v}_e$.

An RFQ is used for several reasons: it bunches particles more effectively, thus improving the injection efficiency, reduces the size of the overall device, and accelerates the bunches as well, reducing the load on the cyclotron and the other accelerator elements.

Previously, PARMTEQ was used as the simulation software for its ease in establishing and defining initial parameters for an RFQ simulation. However, its lack of accurate space-charge calculations and flexibility drastically reduce its long term feasibility for accurate RFQ simulations. Warp, on the other hand, provides the sandbox for use defined and generated simulations, with accurate calculations and flexibility. As a result, a solution that allows the structure of PARMTEQ with the versatility of WARP would be optimal for both ease of use and accurate simulations.

Overview and Structure

The Py RFQ Module is structured in an object oriented manner. There are three classes to represent the RFQ itself: PyRFQCell, PyRFQVane, and the PyRFQ itself. In addition, there are three classes that perform side and helper functions to the RFQ class: FieldGenerator, Field, and PyRFQUtils.

The PyRFQ class uses objects of the above classes to perform vane and field calculations, and generally is the only class that the user needs to directly interact with to create an RFQ object in a WARP simulation. For function documentation, see the Appendix section 0.1.

The Field object is used to store the field data after it has been calculated or imported. A Field object is contained within the RFQ class for access to field information. The FieldGenerator, Cell, and Vane classes are used within the RFQ class in certain functions to assist in calculating the field when the user provides cell parameters from PARMTEQ. The user can add Cells manually, or import them from a file. The PyRFQUtils class is not necessary to simulate an RFQ, but provides helpful diagnostic data. Usage of the above classes is described in further detail below.

Usage

0.1 RFQ instantiation and variables

To place an RFQ into a WARP simulation, it is as simple as creating an instance of an RFQ object.

```
rfq = PyRFQ(filename=filename, from_cells=False, twoterm=True, boundarymethod=False)
```

The user must provide a filename that contains the field or cell parameters, along with the corresponding boolean `from_cells`: TRUE if the file contains PARMTEQ cell data, FALSE if it is field data. The structure of the field file must be as follows: the first line is ignored. Then, each proceeding line is: x position, y position, z position, x efield, y efield, z efield, in that order, delimited by spaces. For example:

```
x, y, z, ex, ey, ez
-1.5e-02  -1.5e-02  0.0e+0  4.9e+05  -4.9e+05  0.0e+00
...
```

`twoterm` and `boundarymethod` indicate whether to use the two term potential method to calculate the electric field, or the boundary method using BEMPP. Both cannot be TRUE simultaneously. It should be noted that using the two term potential method is faster than the boundary method by a significant degree. One can expect the boundary method to take on the scale of hour(s) to calculate a field sufficiently, depending on resolution and size.

Several variables must be set outside of the initial object creation. They are as follows:

```
    rfq.simple_rods    = True  # simple rods
    rfq.vane_radius    = None  # radius of simple rods
    rfq.vane_distance  = None  # vane center distance from central axis
    rfq.rf_freq        = None  # RF frequency
    rfq.zstart         = 0.0   # start of the RFQ
    rfq.sim_start      = 0.0   # start of the simulation
    rfq.sim_end_buffer = 0.0   # added distance to the end of the RFQ beyond vanes
    rfq.resolution     = 0.002 # in meters
```

Above indicate the default values. The program will print an error and exit if any of the variables with default value `None` are not set. `simple_rods` indicates whether to use a simple cylindrical vane shape to simulate the vanes. Currently, there is no way to have any other vane geometry, so it is undefined behavior if this variable is set to false. More options may be added in future versions. The rest of the variables are fairly self explanatory. There is no RFQ length variable, as that is determined by the field length.

The following variables are required if the field is to be calculated from cell parameters:

```
rfq.xy_limits     = None   # X and Y limits in the field calculation
rfq.z_limits      = None   # Z limits for the field calculation
rfq.ignore_rms    = False  # Whether to ignore cells of type RMS

# Two term variables
rfq.tt_a_init     = None   # initial aperture size for two term potential

# Bempp variables
rfq.add_endplates  = True
rfq.cyl_id         = None
rfq.grid_res_bempp = None
rfq.pot_shift      = None
```

The following variables are optional:

```
rfq.endplates                       = False
rfq.endplates_outer_diameter        = 0.2 # in meters...
rfq.endplates_inner_diameter        = 0.1
rfq.endplates_distance_from_vanes = 0.1
rfq.endplates_thickness             = 0.1
```

`endplates` is an option to add cylindrical endplates with holes in the center at either end of the rfq. The following variables must be set accordingly if the first is set to TRUE.

The following variables are available for diagnostics or whatnot:

```
rfq._conductors   # WARP conductors to construct the vanes
rfq._field        # Field object
rfq._sim_end      # End of the RFQ simulation
rfq._length       # Length of the simulation
rfq._fieldzmax    # Z maximum of the field
```

However, it is inadvisable to alter these variables in any way, other than potentially the conductors variable, as it could potentially disturb RFQ calculations and results in undefined behavior.

0.2 RFQ Setup and installation

Once the PyRFQ object has been instantiated and the requisite variables have been set, the user must call the `setup()` and `install()` functions. The reason for the distinction is that `setup()` evaluates that the proper variables are in place, and loads the cells or field from the file into memory. `install()` on the other hand calculates the field and loads it into the WARP simulation. This distinction allows the user to include extra cells at the end of the cells loaded from the file using the `add_cell()` routine. This must occur between the calls to `setup()` and `install()`, as such:

```
        rfq.setup()

        rfq.add_cell(cell_type="TCS",
                     aperture=0.011255045027294745,
                     modulation=1.6686390559337798,
                     length=0.0427)
        rfq.add_cell(cell_type="DCS",
                     aperture=0.015017826368066015,
                     modulation=1.0,
                     length=0.13)

        rfq.install()
```

0.3 Diagnostics

The class PyRfqUtils provides several diagnostic functions outside of the typical WARP diagnostics that the user can call during or after the WARP simulation. To use them, the user must instantiate a PyRfqUtils with a PyRFQ object, and the beam object from the WARP simulation.

`beamplots()` produces plots of the particles X vs Z, Y vs Z, as well as the X vs X' and Y vs Y' plots natively using WARP's gist methods. `makeplots()` is a method that can be installed conveniently to run every time step, and you can plot the particles every certain amount of steps. See the appendix for details.

`plot_rms_graph()` plots a graph of the RMS speed of the particles along the X and Y axis for every small step or so along the RFQ. It can be called at any point during the simulation.

`find_bunch()` will run extra steps in the simulation, and will exit once it has found and isolated a bunch that has come out of the RFQ. It then returns a bunch dictionary, which contains all the data that WARP can provide for the particles in said bunch. These are: x positions, y positions, z positions, r and theta positions, x y and z velocities, x y and z momenta, x y and r phase, and gamma inverse. The names you can access this data from the dictionary with are:

```
"x"
"y"
"z"
"r"
"theta"
"vx"
"vz"
"ux"
"uy"
"uz"
"xp"
"yp"
"rp"
"gaminv"
```

The function can be called at any point after the particles have reached the end of the RFQ. Otherwise, the function will return nothing and print an error.

Example

See `testsim.py` for an example of a simple WARP simulation designed to simulate an RFQ with no endplates, simple rods, reading in a file containing PARMTEQ cell parameters.

Future Steps

While this module is in a functioning state, and can simulate an RFQ in a WARP simulation, there are several areas that should be improved and added to make it a fully functioning RFQ module. Firstly, currently there is only one option in how to calculate the field from cell parameters (two term), in spite of the ability to indicate to use the second option (boundary method). Adding the functionality of using the boundary method would be the first step from now: it would allow for more accurate results, at a slower pace.

Second would be more varied vane geometry. Currently, the only available option is the simple cylindrical vane shape, which is not an accurate representation of most RFQ's. The options being explored are to manually generate sinusoidal conducting object in Warp, and directly importing a CAD file containing any vane geometry as a conductor into the Warp simulation.

Third would be refinement of the general structure of the RFQ placed into the Warp simulation. As of right now, the surrounding cylinder around the entirety of the RFQ is at a predefined few millimeters above the vanes. The module should allow for customization of this. Furthermore, currently, the vanes begin at the start of the field, and end at the start of the field. This is not always the case, and should be a configurable user parameter.

Fourth would be general usability and adding functionality for the user – to make the module easier to use, configure, and more flexible in terms of such things as vane geometry, structure of the RFQ, diagnostics at the end, and more small features. Overall, to make it a more polished product.

Overall, the current state of the module is very much a working skeleton model, and there is much potential and room for expansion for the future.

Appendix

PyRFQ Function Documentation

```
rfq.install()
        Installs the field and conductors into the Warp simulation
        Must be used before rfq.setup()
        Returns: None

rfq.setup()
        Evaluates user parameters and loads field into Warp simulation.
        Can be used after call to rfq.install()
        Returns: None

rfq.plot_efield()
        Plots the e-field along the z axis
        Returns: None

rfq.add_cell(cell_type, aperture, modulation, length, flip_z=False, shift_cell_no=False)
        Adds a cell to the end of the PARMTEQ cell list, after setup() and before
        install().
        Returns: None
        • cell_type (string): type of the cell. Must be one of one of STA, RMS, NCS, TCS,
        DCS.
        • aperture (float): aperture of the cell
        • modulation (float): modulation of the cell
        • length (float): length of the cell
        • flip_z=False (bool)
        • shift_cell_no=False
```

PyRfqUtils Function Documentation

```
utils.find_bunch(max_steps)
        Runs extra steps to find and give data about a bunch.
        Returns: Bunch dictionary. None if particles have not reached end of RFQ.
        Bunch data is also dumped into a file with the form "bunch_particles.%s.dump"
        • max_steps (int): maximum number of steps allocated to find a bunch

utils.plotXZparticles(view=1)
        Plots X by Z position of the particles in a gist window
        • view=1: gist window

utils.plotYZparticles(view=1):
        Plots Y by Z position of the particles in a gist window
        • view=1: gist window
```

utils.plotXphase(view=1):

Plots X phase space

- view=1: gist window

utils.plotYphase(view=1):

Plots Y phase space

- view=1: gist window

utils.beamplots()

Plots X by Z, Y by Z, X phase, and Y phase in four different gist windows

utils.make_plots(rate=10)

Calls beamplots() if top.it (current step number) % rate is zero.

- rate=10 (int): rate at which plots are made

utils.plot_rms_graph(start, end, bucketsize=0.0001):

Produces a plot with the average RMS for x and y in each bucket along the z axis.

- start (float): where to start the plot along the z axis
- end (float): where to end the plot along the z axis
- bucketsize (float): how large the buckets along the z axis are