Dark Current Simulations: User's Guide

Christie Chiu*
Massachusetts Institute of Technology
Cambridge MA, 02135

August 28, 2011

This document contains user's guides to the code `dark_current_tracker.f90`, `particle_generator_cavity7.f90`, and `wall_generator.f90`. It also contains a guide to the Mathematica notebook `wallCreator_datapts.nb`. Together, this software allows a user to simulate dark current in an accelerator lattice.

1 `dark_current_tracker.f90`

This dark current simulation code is the main driver program. It tracks multiple particles backwards or forwards through elements until they either exit the accelerator lattice or hit a wall.

1.1 Input

There are no command line arguments, but the program uses the following variables, defined in the namelist file `dark_current_tracker.in`:

- `lat_name`: lattice file name, set to `lat.bmad` by default.
- `particle_file_name`: particle file name of particles' initial positions and charges, set to `ReferenceParticles.dat` by default.
- `tracking`: Indicates whether particles tracks will be saved (True) or not (False). Default is False.
- `verbose`: Indicates whether extra information will be printed to screen as program runs (True) or not (False), False by default.
- `max_charge_ratio`: Particles will only be tracked if they have charge within 1/`max_charge_ratio` of the maximum charge present. This is set to 10^6 by default.

1.2 Output

The output is a file with the same filename as `particle_file_name`, but with a `.out` extension. It contains the particle tracks for each particle if `tracking` is True, and only the final destinations otherwise. Each data point has time, phase-space coordinates for x, y, and s, angle of impact, and effective charge. The angle of impact is the angle to the normal vector to the surface at the point of impact, and is measured in radians. All coordinates are in the global t-based reference frame.

1.3 Methodology

The dark current simulation code iterates through each particle and tracks it from its starting location to final destination. For each particle, we:

1. Determine in which element it begins
2. Track the particle through that element with our custom time-based tracker (in `dark_current_tracker.f90`). The

tracker queries for the EM field at the particle's current location, then uses Runge-Kutta to calculate where it travels in a time step. It repeats this process until the particle either hits a wall or exits the element at either end. The tracker also uses a refined time step as a rootfinding method to minimize errors in the final time step.

- If the particle exits into another element, it tracks the particle through the subsequent element.
- If the particle exits the lattice or hits a wall, it moves on to the next particle.

`dark_current_tracker.f90` uses the `track1_custom` subroutine in `track1_custom.f90`, which does the following:

1. Checks to see if the current element has length 0. If it does, it continues on to the next element.
2. Converts the particle's global t-based coordinates to local element t-based coordinates. To do this, it must convert:
   a) Global t-based $\rightarrow$ Global s-based
   b) Global s-based $\rightarrow$ Element s-based
   c) Element s-based $\rightarrow$ Element t-based

   We cannot convert directly because we use conversion subroutines in `Bmad`, which are all s-based.
3. Checks that particle is inside of accelerator walls. If not, we say it hit a wall and exit the subroutine.
4. Query the field and use Runge-Kutta to find the particle's position after every time step. This happens until the particle hits a wall or exits.
5. Converts the particle back to global t-based coordinates.

2 particle_generator_cavity7.f90

The particle generator code creates a set of particles that we can then simulate in our driver program. Currently, it only creates particles on cavity walls. The particle locations are evenly spaced on these cavity walls, but given a random angle ϕ about the center s-axis. At each location, multiple particles are created, each with a different phase. These phases are evenly spaced about the RF cavity period. Each particle is also given an effective charge that is weighted according to the likelihood of field emission at that location.

2.1 Input

There are no command line arguments, but the program uses the following variables, defined in the namelist file `particle_generator_cavity7.in`:

- `lat_name`: lattice file name, set to `la.bmad` by default.
- `numParticles`: number of particle locations per cavity, set to 100 by default.
- `numPhases`: number of phases per particle location, set to 1 by default.
- `lattice_part`: we will only create particles for 1/`lattice_part` of the cavities in the entire lattice. This is set to 30 by default.
- `lattice_shift`: we will start creating particles for the first cavity after element number `lattice_shift`, set to 0 by default.
- `buffer`: distance between particle and wall, because if it is too close we may have precision issues. The default buffer is 10^{-8}.
- `beta_fn`: Fowler-Nordheim field enhancement factor, set to 300 by default.
- `A_fn`: Fowler-Nordheim effective emitter area, set to $3 \cdot 10^{-17}$ by default.
- `angles_on`: If True, we will give each location a random angle ϕ as described above. If False, ϕ equals 0 for all locations. This is set to True by default.

- `max_charge_ratio`: Particles will only be created if they have charge within 1/`max_charge_ratio` of the maximum charge present. Its default value is 10^6.

2.2 Output

The output is a file with a filename structure `l#_n#_p#.dat`. The first `#` equals `lattice_part`, the second is `numParticles`, and the third is `numPhases`. The first line in this file contains the total number of particles in the file. After that, each line contains the global phase space coordinates in the form $(x, \beta_x \cdot \gamma_x, y, \beta_y \cdot \gamma_y, z, \beta_z \cdot \gamma_z)$, followed by the phase and effective charge. The effective charge is the current calculated from the Fowler-Nordheim cold emission model, multiplied by the phase resolution, then multiplied by the radius to the center axis. We need the last correction to account for the fact that bands of wall with larger radii will have larger surface area, making it more likely for field emission if all other factors are constant.

2.3 Methodology

During initialization, we count the maximum number of particles we could have and allocate an array to hold these particles' coordinates. To create particles, we start at element `lattice_shift` and look for a cavity until we've looked through 1/`lattice_part` of the lattice elements. When we arrive at a cavity, we do the following:

1. Set momenta equal to 0.

2. Find length of cavity boundary by adding up all linear segments between the wall cross sections used to define the element geometry. Then we know the distance between each particle location for angle ϕ equal to 0 because they are evenly spaced along the wall.

3. Start with the interparticle distance divided by two, so that the last particle is also half the interparticle distance from the end boundary of the cavity. We then have the following logic, dependent upon a running distance and current cross section. The initial running distance is half of the interparticle distance, and the initial cross section is the first, at the entrance boundary of the cavity.

   a) If the running distance is greater than the linear segment length between the current cross section and the one following, we subtract the linear segment length from the distance and move on to the next cross section.

   b) If the running distance is the same length as the linear segment length between the current cross section and the one following, we create a particle location at the following cross section, set the running distance to be the interparticle distance, and move on to the next cross section.

   c) If the running distance is less than the linear segment length between the current cross section and the one following, we create a particle location at the running distance along the linear segment, then add the interparticle distance to the running distance while keeping the current cross section constant. This lets us find the next particle location while maintaining our use of only the cross section coordinates.

   d) We repeat these steps until we have reached the last cross section.

4. At each particle location, we generate a random angle ϕ (if desired), and create `numPhases` particles there with phases evenly spaced about the cavity period.

   - However, we only save particles if they have charge greater than 1/`max_charge_ratio` of the maximum charge of all particles created. To do this, we keep a record of the greatest effective charge we have encountered so far. We filter for particles twice: the first time, particles are saved into the array if their charge is sufficient compared to the

running maximum. The second, particles are saved to file if their charge is sufficient compared to the absolute maximum.

- To calculate effective charge, we use the Fowler-Nordheim equation. Before that, however, we check to see if the electric field at the cavity surface is pointing into or out of the cavity interior. If the field pushes electrons into the wall, then we set the particle charge equal to 0 because we know there will not be field emission at that phase.
- The charge from Fowler-Nordheim is then further weighted according to radius as previously described.

3 wall_generator.f90

This Fortran code parses a lattice file and outputs a particle track file where the points are not actually particle tracks, but instead points that trace out the intersection between the upper xs-plane and accelerator. This file can then be read in by Mathematica to draw a wall over any particle track or endpoint plot for qualitative analysis. Notice that we only generate points at the lattice cross sections, so this method relies upon the fact that the walls are created with only linear interpolation in the simulation. If cubic interpolation is used, this wall generator may not draw accurate walls.

3.1 Input

The wall generator program takes the lattice file name as a command line argument.

3.2 Output

The output is a particle track file as mentioned, with the name `wall.dat`.

3.3 Methodology

Because we only take points along a lateral cross section, we can immediately set all of time, y, momenta, hit angle, and charge equal to 0. That is, we only need to worry about the x- and s-coordinates. Then, for each element, we run the following:

1. If a wall structure exists for the element under consideration, we iterate along the element's cross sections and get its x-radius and s-coordinate. Both of these coordinates are always defined in the lattice in order to create a wall. We then write the coordinates to file in the same format as the `dark_current_tracker.f90` output file.
2. If a wall structure does not exist, then we take its aperture and plot two points. The first is at the very beginning of the element and has x-coordinate equal to the aperture radius. The second is idential, except at the end of the element.

4 wallCreator_datapts.nb

When we generate our EM field data file, we are also given a set of data points that describe the wall boundary. To create a `Bmad` 3D wall structure from these points, we can use the Mathematica notebook `wallCreator_datapts.nb`.

4.1 Input

Mathematica reads in the provided set of data points, which are formatted as two columns: s-coordinate and radius. The current filename for this data is `cavity7cell2d.wall`

4.2 Output

After running `wallCreator_datapts.nb`, we have a `Bmad` file that defines the wall for an element called `cavity7`. This file must be linked to from the main lattice file, and can be read directly by the `Bmad` parser. The current output file has name `wall.cavity7cell2d.bmad`.

In our lattice, we would also like our RF cavity to be centered. This may not be true from the EM field data points, so we calculate how far we need to shift our cavity to the left to make it as symmetric as possible. However, if we shift the cavity wall data, it will no longer line up with the EM field data. Therefore, we

need to instead shift the flanking pipe lengths. This Mathematica notebook calculates what that shift difference is and sets `xDist` equal to that value. The lattice files have a parameter that needs to be manually set to `xDist`, and all the pipes will automatically be changed accordingly.

4.3 Methodology

First, we describe our algorithm for creating a wall.

1. The current wall data has overlapping points, as well as points that are not necessarily in order of ascending s-coordinate. `Bmad`, however, expects no multiple cross sections and cross sections must be in order. So, we first sort the data points and elimate any duplicates.
    - We eliminate duplicates by iterating through the ordered list and keeping only points that do not have a duplicate before them.

2. If we use each data point to create a cross section for our cavity lattice element, then the `Bmad` parser will take too long processing the lattice file. Therefore, we must decimate the data set first. To do this, we set a value `decimFactor` by hand and run the commands that take one point out of every `decimFactor` points and plot the resulting curve. Once we are satisfied with this curve, we write the wall structure to file `wall.cavity7cell2d.bmad`.

Then, we calculate `xDist` through the steps below.

1. We take the data points after step 1 above and flip them across the vertical center of the cavity to get another data set.

2. By comparing the two curves, we can tell if they are symmetric from their level of overlap. If there is complete overlap, then the difference in their interpolating functions will be 0. Otherwise, it will be nonzero. Because the wall data is close to symmetric, we can conclude that the difference is correlated with the level of symmetry.

3. We cannot take the average of the difference of points along the curves because then largely positive and negative differences will still average to a small value. Additionally, the oscillatory characteristic of the cavity walls suggests that the average will be close to 0 regardless of shift size. To get a good value of `xDist`, we instead minimize the standard deviation.