Poisson Solver Code

Craig Lage

November 11, 2016

1 Description

This code is a simple grid-based Poisson's equation solver intended to simulate pixel distortion effects in thick fully-depleted CCD's. The code builds a 3D rectilinear grid to represent a portion of the CCD, assigns the appropriate charge densities and applied potentials, then solves Poisson's equation using multi-grid methods. A 360^3 grid, which is adequate for most purposes, solves in less than one minute on a typical laptop. The code also includes prescriptions to propagate electrons from a given point of creation by an incoming photon down to the point of collection, including both drift and diffusion. Most data is saved as hdf files. The current code is configured to model the ITL STA3800 CCD, but other CCDs can be modeled by editting the configuration file. Plotting routines are available to plot the potentials, E-Fields, pixel shapes, and electron paths. A description of the code, the measurements which were used to validate the code, and some samples of the output are in the file docs/BF_White_Paper_28Sep16.pdf. Below is a basic description of how to install the code and a number of examples.

The code contains many options, and not all combinations have been tested together. If you find a set of options that does not work as you expect, please let me know. However, all of the example configuration files described in the Examples Section below have been tested.

This work is supported by DOE HEP Grant DE-SC0009999.

Installing: Read the Installation Section below.

Running: The basic syntax is:

src/Poisson < configurationfile >

More details are provided in the Examples Section.

Hopefully you find the code useful. Comments and questions are encouraged and should be addressed to: cslage@ucdavis.edu

2 Installation

Dependencies:

There are two dependencies that need to be installed before you can compile the Poisson code:

1. C++ Boost libraries. There are several options for installing these:
   (a) Ubuntu: Install the boost libraries using: sudo apt-get install libboost-all-dev
   (b) Mac OSX: Assuming you are using homebrew, install using: brew install boost

   (c) Build them from source. They can be downloaded from: www.boost.org

2. HDF5 libraries. There are several options for installing these:
   (a) Ubuntu: Install the hdf5 libraries using: sudo apt-get install hdf5-tools
   (b) Mac OSX: Assuming you are using homebrew, install using: brew install hdf5
   (c) Build them from source. They can be downloaded from: www.hdfgroup.org/HDF5/release/obtain5.html
3. After installing the above two dependencies, edit the src/Makefile lines BOOST_DIR and HDF5_DIR to point to their locations.
4. In the src directory, type "make". This should build the Poisson code, and create an executable called src/Poisson. Depending on where you have installed the above libraries, you may need to edit your LD_LIBRARY_PATH environment variable so the system can find the appropriate files for linking.
5. I have included in the src directory a file Makefile.nersc that works for me on NERSC Edison.

Running the python plotting routines also requires that you install h5py so that Python can read the HDF5 files.

If you run the forward modeling code in order to generate brighter-fatter plots as described in the bfrun1 example below, you will also need to build the forward.so Python extension. Instructions for this are in the forward_model_varying_i directory.

3 Changes in Most Recent Version

There are a number of changes and new control parameters in this more recent version. These are as follows:

- **File Formatting:** All output files are now have extensions of either ".hdf5" or ".dat" to make it clearer what type of file they are.
- **Changes to Formatting of Pts files:** The "*_Pts.dat", files, which have the information on electron tracking, have been re-formatted based on input and code from David Kirkby. The files now have a unique ID for each electron, and phase information of where the electron is in its track.
- **Non-Linear Z-axis:** This option "stretches" the z-axis to put more grid cells near the bottom of the silicon where things are changing rapidly, and fewer grid cells at the top where things change slowly. This allows higher resolution without increasing compute time significantly. It is controlled by the parameter "NZExp", which is the stretch factor at z=0. A value of NZExp = 1.0 reverts to a linear z-axis. A value of NZExp = 10.0 is recommended. This option was also in the "hole7" branch, but was hard coded at a value of NZExp = 10.0. More details are in Appendix D of the wihte paper BF_WP_11Nov16.pdf.
- **Ability to read in and continue a long simulation:** The control parameters "Continuation" and "LastContinuationStep" allow you to read in the mobile charge locations and contniue a simulation from where it left off. This is useful for simulations which have crashed or run out of time.
- **Option of free holes in the channel stop region:** One hypothesis still being explored is that the channel stop region is not fully depleted, and that there are free holes present in this region. This option was hard coded into the earlier "hole7" branch, but is now available as an option. It is controlled by the "UndepletedChannelStop" parameter (0 = no free holes; 1= free holes), and the "Vchannelstop" and "HoleConvergenceVoltage" parameters. The output file "*_Hole.hdf5" contains the locations of the free holes. It has been found that the inclusion of free holes in the channel stop region gives better agreement with measurements of the "brighter-fatter effect".

- **Better control of electron tracking:** Two parameters, "EquilibrateSteps" and "BottomSteps" have been added to give better control of the electron tracking. The first (which was previously hard coded at 100) counts the number of diffusion steps after the electron reaches the collecting well before we start storing the charge location, and the second (which was previously hard coded at 1000) counts the number of diffusion steps after the electron reaches the collecting well after we start storing the charge location.
- **Output of Grid files:** The code now outputs files called *_grid.dat (*=x,y,z) to document the grids used. This is based on a suggestion from David Kirkby, and makes the plotting much easier and more correct. This revision chnages the format from what David had suggested in order to include the cell boundaries as well as the cell center. This is needed to accomodate the non-linear Z-axis discussed above.

4 Examples

There are a total of nine examples included with the code. Each example is in a separate directory in the data directory, and has a configuration file of the form *.cfg. The parameters in the *.cfg files are commented to explain(hopefully) the purpose of each parameter. Python plotting routines are included with instructions below on how to run the plotting routines and the expected output. The plot outputs are placed in the data/*run*/plots files, so you can see the expected plots without having to run the code. If you edit the .cfg files, it is likely that you will need to cutomize the Python plotting routines as well.

- Example 0: data/run0/bf.cfg
  1. Purpose: A simple 9x9 grid of pixels. The central pixel contains 200,000 electrons with an assumed charge density (adjustable in the .cfg file). No electron tracking or pixel boundary plotting is done.
  2. Syntax: src/Poisson data/run0/bf.cfg
  3. Expected run time: $\approx$ 2minutes.
  4. Plot Syntax: python Poisson_Plots.py data/run0/bf.cfg 0
  5. Expected plot run time: $<$ 1minute.
  6. Plot output: Assumed boundary potentials and charge distribution as well as several views of the potential solution.
- Example1: data/run1/bf.cfg
  1. Purpose: The same as run0 above, but after solving Poisson's equation, a grid of electrons is traced to illustrate the pixel boundaries and electron paths. Diffusion is turned off for this electron tracing so one can see the impact of the brighter-fatter effect on distorting the electron paths around the central pixel. Plotting of the pixel boundaries and electron paths is relatively slow.
  2. Syntax: src/Poisson data/run1/bf.cfg
  3. Expected run time: $\approx$ 1hour.
  4. Plot Syntax: python Poisson_Plots.py data/run1/bf.cfg 0
  5. Expected plot run time: $\approx$ 2minute.
  6. Plot output: In addition to the plots from run0, there are plots of the pixels and electron paths.
  7. Plot Syntax: python ChargeDistribution_XYZ_N.py data/run1/bf.cfg 0 9
  8. Expected plot run time: a few seconds.
  9. Plot output: The assumed charge distribution of the collected electrons, as defined by the Collected-Charge*min(max) parameters in the .cfg file, as well as the distribution of free holes.
- Example 2: data/run2/bf.cfg

1. Purpose: The same as run1 above, but after solving Poisson's equation, the pixel boundaries and areas are found through a binary search which tracks the electrons down to a pixel location.
2. Syntax: src/Poisson data/run2/bf.cfg
3. Expected run time: $\approx$ 1hour.
4. Plot Syntax: python Poisson_Plots.py data/run2/bf.cfg 0
5. Expected plot run time: $\approx$ 2minute.
6. Plot Syntax: python ChargeDistribution_XYZ_N.py data/run2/bf.cfg 0 9
7. Expected plot run time: a few seconds.
8. Plot output: The assumed charge distribution of the collected electrons, as defined by the Collected-Charge*min(max) parameters in the .cfg file, as well as the distribution of free holes.
9. Plot Syntax: python VertexPlot.py data/run2/bf.cfg 0 3 (the last parameter determines how many pixels away from the central pixel are plotted).
10. Expected plot run time: a few seconds.
11. Plot output: The areas and shapes of the pixels surrounding the central pixel.

- Example3: data/run3/bf.cfg

  1. Purpose: A simple 9x9 grid of pixels. A Gaussian spot with a sigmax = sigmay of 10.0 microns (one pixel) is incident on the CCD and 1,000,000 electrons are tracked down to their final locations. The final location of the electrons is found in a self-consistent way and is not assumed as in runs 0-2. Poisson's equation is re-solved after each 10,000 electrons. The electron locations are saved after each step, the potential, charge, and E-field are saved after every 20 steps, and the pixel shapes are saved after 100 steps. This will generate about 8 GB of data.
  2. Syntax: src/Poisson data/run3/bf.cfg
  3. Expected run time: $\approx$ 5hours.
  4. Plot Syntax: python Poisson_Plots.py data/run3/bf.cfg x (x will determine which step is plotted)
  5. Expected plot run time: $\approx$ 2minute.
  6. Plot Syntax: python ChargeDistribution_XYZ_N.py data/run3/bf.cfg x 9(x will determine which step is plotted)
  7. Expected plot run time: a few seconds.
  8. Plot output: The self-consistent charge distribution of the collected electrons, and the location of free holes..
  9. Plot Syntax: python VertexPlot.py data/run3/bf.cfg x 3 (x will determine which step is plotted (0 or 100); the last parameter determines how many pixels away from the central pixel are plotted).
  10. Expected plot run time: a few seconds.
  11. Plot output: The areas and shapes of the pixels surrounding the central pixel.

- Example 4-7: These are identical to examples 0-3, except that these examples include free holes. This increases the run time by about 20%. This code is still not as robust as I would like, so some variations of the channel stop depth and doping may cause it not to converge. These examples should run fine, however.

- Example 8: data/bfrun1/bf.cfg

  1. Purpose: A simple 9x9 grid of pixels. A Gaussian spot with a sigmax = sigmay of 10.0 microns (one pixel) is incident on the CCD and 1,000,000 electrons are tracked down to their final locations. The final location of the electrons is found in a self-consistent way and is not assumed as in runs 1-3. Poisson's equation is re-solved after each 10,000 electrons. The electron locations are saved after each step, the potential, charge, and E-field are saved after every 10 steps, and the pixel shapes are saved

after 80 steps. A total of 63 different spots are run (each in a directory data/bfrun_x) with each spot having a random central location within the central pixel. After the spots are run, the 63 spots are forward modeled to produce a plot of the X and Y size of the spot as a function of flux (a "brighter-fatter" plot). A 64th spot is run (in directory data/bfrun_0) with 100,000 electrons self-consistently placed in the central pixel, and this is used to calculate the expected pixel-pixel correlations due to the brighter-fatter effect. This run will generate ≈ 250GB of data.

2. Syntax: This may vary depending on the system you are running on. The intent is to launch 64 copies(or however many you want) of the Run_BF_Multi.py code. This code will run the 64 spots and processor rank 0 will then make the BF plot. Two control files for launching the MPI jobs, python-mpi.sl(NERSC) and lage_script.moab(LLNL) are included. These will probably need modification, but should serve useful as starting points.
3. Expected run time: ≈ 7hours without free holes; ≈ 10hours with free holes.
4. Plot Syntax: The BF_Sim_101_64.png plot is generated automatically by the Run_BF_Multi.py code
5. Plot output: A plot of the brighter-fatter effect, showing the growth of the spots as the flux increases.
6. Plot Syntax: python AreaPlot_Corr.py data/bfrun_0/bf.cfg 80
7. Plot output: A plot of the expected pixel-pixel correlations (Area_UnBiased_Corr_0_80.pdf) due to the brighter-fatter effect.

5 Other Python Plotting Routines

I have included several other Python plotting routines, especially AreaTrend.py, VertexTrend.py, and PixelModel.py, which I have used to analyze the pixel area and vertex shifts. These have not been run with the current code revision, and will almost certainly need some customization to run on your output, but I include them since they are probably useful as a starting point.