Poisson Solver Code

Craig Lage

February 3, 2017

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

This 'hole17' branch is a major revision. It is still under development and may contain bugs. However, it was used to create all of the plots in the document docs/PACCD_Paper_2Feb17.pdf, so it is reasonably mature.

- **Mobile Charges:** The treatment of mobile charges, both holes and electrons, is the biggest change in the version. Mobile charges are now dealt with using the Quasi-Fermi level formalism. This self consistently solves for the potential and mobile charge densities in regions containing mobile charges. The hole-containing regions in these devices are a spatially continuous region which are in quasi-equilibrium, so a single hole Quasi-Fermi level serves to set the holes densities throughout the device. This is set in the .cfg ffile by the parameter qfh. Setting this parameter to a value of 0.4 Volts will set the potential in the hole containing region to be near 0 Volts, which is appropriate for the ITL devices. For the electrons, a different value of the electron Quasi-Fermi level is set in each pixel based on the number of electrons in this pixel. Because the relation between the electron Quasi-Fermi level and the number of electrons in the pixel is a compelx non-linear relationship, the code builds a look-up table for this. Building this look-up table is controlled by the BuildQFe, QFeMin, QFeMax, and NQFe parameters. This only need to be done each time the CCD parameters (voltages, dopings, etc.) are changed. If BuildQFeLookup = 0, the code will look for the *QFe.dat file to read in.
- **Geometry of gate and field oxide regions:** The code now has a more realistic gate and field oxide geometry, controlled by the GateOxide, FieldOxide, and FieldOxideTaper parameters.

4 Examples

There are a total of four 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 customize the Python plotting routines as well.

- Example 0: data/bfrun_0/bf.cfg
    1. Purpose: A simple 9x9 grid of pixels. The central pixel contains 80,000 electrons with an assumed charge density (adjustable in the .cfg file). No electron tracking or pixel boundary plotting is done. This is a lower resolution run that will run quickly.
    2. Syntax: src/Poisson data/bfrun_0/bf.cfg
    3. Expected run time: $\approx$ 1minutes.
    4. Plot Syntax: python Poisson_Plots.py data/bfrun_0/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.
    7. Plot Syntax: python ChargePlots_27Jan17.py data/bfrun_0/bf.cfg 0 3
    8. Expected plot run time: $\approx$ 1minute.
    9. Plot output: Detailed 3D Charge Distributions.
- Example 1: data/bfrun_1/bf.cfg
    1. Purpose: Same as example 0, but at higher resolution.
    2. Syntax: src/Poisson data/bfrun_1/bf.cfg
    3. Expected run time: $\approx$ 30minutes.
    4. Plot Syntax: python Poisson_Plots.py data/bfrun_1/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.
    7. Plot Syntax: python ChargePlots_27Jan17.py data/bfrun_1/bf.cfg 0 3
    8. Expected plot run time: $\approx$ 1minute.
    9. Plot output: Detailed 3D Charge Distributions.
- Example 2: data/bfrun_2/bf.cfg
    1. Purpose: Runs a B-F run, with 1,000,000 electrons in 101 10,000 electron steps.
    2. Syntax: src/Poisson data/bfrun_2/bf.cfg
    3. Expected run time: $\approx$ 3hours.
    4. Plot Syntax: python Poisson_Plots.py data/bfrun_2/bf.cfg 0
    5. Expected plot run time: $\approx$ 2minute.
    6. Plot Syntax: python ChargePlots_27Jan17.py data/bfrun_3/bf.cfg 0 9
    7. Expected plot run time: $<$ 1minute.
    8. Plot output: Detailed 3D Charge Distributions.
- Example 3: data/bfrun_3/bf.cfg
    1. Purpose: Same as example 1, but calculates pixel areas and shapes.
    2. Syntax: src/Poisson data/bfrun_3/bf.cfg
    3. Expected run time: $\approx$ 150minutes.
    4. Plot Syntax: python Poisson_Plots.py data/bfrun_3/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.
7. Plot Syntax: python ChargePlots_27Jan17.py data/bfrun_3/bf.cfg 0 3
8. Expected plot run time: $\approx$ 1minute.
9. Plot output: Detailed 3D Charge Distributions.
10. Plot Syntax: python VertexPlot.py data/bfrun_3/bf.cfg 0 2
11. Expected plot run time: $\approx$ 1minute.
12. Plot output: Detailed Pixel shapes