B8_03: Simulation of Charge Packet Shapes in a One-dimensional Charged-Coupled Device

Candidate Number: 1101608

University of Oxford

Supervisor: Dr. Daniel Weatherill

Trinity 2020

Simulation of Charge Packet Shapes in a CCD

Candidate Number: 1101608
Supervisor: Dr. Daniel Weatherill
Department of Physics, University of Oxford
(Dated: May 10, 2020)

Being able to numerically simulate charge-couple devices used in telescopes is important for building more durable and precise measuring instruments. In this project, we attempt to simulate drift-diffusion by solving non-linear Poisson equations and continuity equations using the Gummel's iterative method with the ultimate goal of simulating charge-coupled device detectors. Due to difficulties in writing the numerical solver, this was limited to being able to analyse the no-bias steady state solutions of one-dimensional systems.

I. INTRODUCTION

The Legacy Surve of Space and Time (LSST) is a ground-based telescope that is currently under construction, and, once finished, would allow mapping the night sky at an unprecedent pace. The aim of the telescope is to further the search in understanding dark matter and energy, investigate the structure of the Milky Way, and track hazardous asteroids in the Solar System, among other things. The main features of the telescope are its wide field of vision, very high read-out rate and the high resolution camera. The camera consists of 189 16-megapixel silicon detectors arranged to provide a total of 3.2 gigapixels of detection[1]. In this project, we attempt to simulate the photodetection capabilities of the charge-coupled devices (CCD) in the camera.

Being able to simulate the operating efficiencies of CCDs is of upmost importance, partly because there does not exist accurate analytical nor computational solutions to the system, and partly because of the potential to severely lessen the effects hindering the efficiency of a CCD. One of such limitations is the Brighter-Fatter Effect (BFE), in which charge already accumulated in a pixel alters the electric field geometry and causes new charge to be deflected away from brighter pixels[2]. The effect is specially dominant in higher luminosity objects and the errors it introduces in pixel values can be up to several percentages in some sensors. The other limiting effect is the Charge Transfer Inefficiency (CTI) which occurs during the readout phase. When shifting electrons from one site to the next, some of the electrons are lost, and most of the time they get re-emitted, as later pixels are clocked out. This introduces "smearing" in one direction[3].

There are various attempts to computationally simulate the effects of BFE and CTI, but most of the sources either make vast simplifications (e.g. assuming static charge distributions[4]) or rely on outdated libraries[5]. The aim of this project is to offer an open source package able to produce solutions to three-dimensional CCD systems. A significant part of the package was developed together with my colleague. Later on, I worked on verifying the validity of the solutions found in simpler one-dimensional systems such as a PN junction or a linear dielectric while my colleague investigated potential avenues for using parallel computation tools and different data types alongside fitting the work into a package.

II. THEORETICAL FRAMEWORK

A. non-equilibrium statistical mechanics of semiconductors

The governing equations of a semiconductor can be derived by treating the semiconductor as having the charge carriers be a non-degenerate classical gas of "quasi-electrons" (different effective mass than that of an electron) and holes. Although holes are just a lack of electrons, we can treat them as having separate populations with some recombination reactions happening between the two. In this case, we assume that the recombination is very small compared to the evolution of each of the populations. This results in two systems that are perpetually in equilibrium (and can be assigned a chemical potential) that have some recombination happening between them. We can then apply Fermi-Dirac statistics on the particles to get the following distributions:

$$f_n(E) = \frac{1}{1+e^{\beta\left(E-E_F^n\right)}},$$
$$f_p(E) = 1 - f_n(E) = \frac{1}{1+e^{\beta\left(E_F^p-E\right)}}.$$

Here, n and p denote the electron and hole densities respectively. For an intrinsic semiconductor (no doping), $n = p = n_i$. The carrier concentrations are given by

$$n = N_C e^{\beta(E_F^n - E_C)}, \quad p = N_V e^{\beta\left(E_V - E_F^p\right)},$$

where N_C and N_V are the effective densities of states at the conduction and valence bands respectively[6]. The equations were derived by assuming a parabolic shape for the bands, which does hold for small perturbations from the band gap but gets less accurate at bigger deviations. Notably, the system has three independent unknown variables: electron and hole concentrations, n and p, and voltage V. In order for the system to be solvable, three independent binding equations are needed. Two come from the continuity equations (derived from the Boltzmann Transport Equation) for n and p:

$$\frac{\partial n}{\partial t} = \frac{1}{e}\boldsymbol{\nabla}\cdot\boldsymbol{J}_n + R_n, \tag{1}$$
$$\frac{\partial p}{\partial t} = -\frac{1}{e}\boldsymbol{\nabla}\cdot\boldsymbol{J}_p + R_p, \tag{2}$$

where

$$\boldsymbol{J}_n = en\mu_n\boldsymbol{E} + eD_n\boldsymbol{\nabla}n,$$
$$\boldsymbol{J}_p = ep\mu_p\boldsymbol{E} - eD_p\boldsymbol{\nabla}p.$$

Here, μ is the mobility, and D the diffusivity, and they are related by the Einstein relation $D = \mu k_{\mathrm{B}}T$.

The third and final binding equation in non-equilibrium is simply the Poisson equation arising from the first Maxwell equation:

$$\boldsymbol{\nabla}\cdot(\varepsilon\boldsymbol{\nabla}V) = e(n - p + N_A - N_D), \tag{3}$$

where N_A and N_D are the acceptor and donor concentrations respectively[5]. Although the Poisson equation on its own is linear, it is referred as being non-linear due to the exact forms of n and p being highly non-linear in nature.

B. Quasi-Fermi level formulation

Instead of solving for the previously defined V, n, p (natural formulation), we choose, for convenience, to solve for V, ϕ_n, ϕ_p (quasi-fermi level formulation) which satisfy

$$n = n_i \exp\left(\frac{e(V-\phi_n)}{k_{\mathrm{B}}T}\right),$$
$$p = n_i \exp\left(\frac{e(\phi_p - V)}{k_{\mathrm{B}}T}\right).$$

ϕ_n and ϕ_p can be interpreted as quasi-fermi levels that are with respect to the applied voltage V. This is done to get rid of the valence and conduction band energies. To be exact, $\phi_n = V - E_F^n$, $\phi_p = V - E_F^p$, and $n_i = N_C e^{-\beta E_C} = N_V e^{\beta E_V}$.

C. Recombination

So far, we have not mentioned recombination. Recombination refers to the R_n and R_p terms in the continuity equations (1) and (2) and corresponds to the creation or elimination of mobile charge carriers. Recombination can arise from many different sources. Here we will outline two of the main sources.

Band-to-band recombination occurs when an electron falls directly from an occupied state in the conduction band into the empty state in the valence band associated with a hole. This band-to-band transition is typically a radiative transition in direct band gap semiconductors, or it can be phonon-assisted in indirect gap materials.

Trap-assisted recombination (SRH) occurs when an electron falls into a "trap," an energy level within the band gap caused by the presence of a foreign atom or a structural defect[5].

In practice, the band-to-band recombination is much smaller than trap-assisted in almost engineering semiconductors, and in particular in Silicon and other indirect band gap materials.

III. 1D NUMERICAL ANALYSIS

A. Derivatives and numerical instabilities

In the course of the numerical analysis, all the values are evaluated on a discrete grid. Derivatives are found by multiplying vectors through with the corresponding derivative matrix. Special care needs to be taken to express the special operators such as the divergence in a way to avoid numerical instabilities. The root cause for this is that the divergence is evaluated at the mid-points of the mesh, and simple linear extrapolation is not enough to guarantee stability.

The Scharfetter-Gummel interpolation was used for the current:

$$J^n_{i-1/2} = \\ = \frac{eD^n_{i-1/2}}{\Delta}\left[n_i B\left(\frac{V_i - V_{i-1}}{V_T}\right) - n_{i-1}B\left(\frac{V_{i-1}-V_i}{V_T}\right)\right],$$

where

$$B(x) = \frac{x}{e^x - 1}$$

is the Bernoulli function[5] and $\Delta = L_D$ the grid step size.

The following matrices were used for derivatives:

$$\frac{\mathrm{d}}{\mathrm{d}x} \to D_1 = \frac{1}{2\Delta}\begin{pmatrix} -1 & 1 & & & & & \\ -1 & 0 & 1 & & & & \\ & -1 & 0 & & & & \\ & & & \ddots & & & \\ & & & & 0 & 1 & \\ & & & & -1 & 0 & 1 \\ & & & & & -1 & 1 \end{pmatrix},$$

$$\frac{\mathrm{d}^2}{\mathrm{d}x^2} \to D_2 = \frac{1}{\Delta^2}\begin{pmatrix} -1 & 1 & & & & & \\ 1 & -2 & 1 & & & & \\ & 1 & -2 & & & & \\ & & & \ddots & & & \\ & & & & -2 & 1 & \\ & & & & 1 & -2 & 1 \\ & & & & & 1 & -1 \end{pmatrix}.$$

Notably, the choice of interpolation and derivative matrices worked well for the continuity equations but there were problems with stability with the Poisson equation (3) in non-constant dielectrics.

B. Scaling

For the purpose of making the applied numerical methods be more optimised, the equations need to be scaled to avoid too large values on one side of the equation. To this end, we will introduce thermal voltage $V_T = \frac{k_\mathrm{B}T}{e}$ and the Debye length

$$L_D = \sqrt{\frac{\tilde{\varepsilon}\varepsilon k_\mathrm{B}T}{e^2\tilde{N}}},$$

where $\tilde{\varepsilon}$ is a chosen scaling permittivity, and $\tilde{N}$ maximum doping density. For our use, both were chosen as the maximum absolute value of the corresponding quantity in the system. The Debye length is used as the grid size in the simulation and is a measure of the charge carrier's effective length of influence in the system, below which we expect the unknown quantities to be smooth. Notably, for low temperatures or high dopant concentrations, very fine grid simulations are needed. In a typical charge-coupled device (doping $\sim 1\times10^{13}\,\mathrm{cm}^{-3}$, $T\sim 300\,\mathrm{K}$), the Debye length is $\sim 0.024\,\mathrm{\mu m}$

The following quantities are then scaled (scaled values are denoted with a bar on top of it):

$$\begin{aligned}
\bar{V} &= \frac{V}{V_T},\\
\bar{\varepsilon} &= \frac{\varepsilon}{\tilde{\varepsilon}},\\
\bar{\nabla} &= L_D\nabla,\\
\bar{\phi}_n &= \frac{\phi_n}{V_T},\\
\bar{\phi}_p &= \frac{\phi_p}{V_T},\\
\bar{C} &= \frac{N_A-N_D}{n_i},\\
\bar{n}_i &= \frac{n_i}{\tilde{N}}.
\end{aligned}$$

The non-linear Poisson takes the convenient form

$$\begin{aligned}
&\bar{\boldsymbol{\nabla}}\cdot(\bar{\varepsilon}\bar{\boldsymbol{\nabla}}\bar{V}) = \\
&\quad = \bar{n}_i\left(\exp(\bar{V})\exp(-\bar{\phi}_n) - \exp(-\bar{V})\exp(\bar{\phi}_p) + \bar{C}\right).
\end{aligned} \tag{4}$$

Transforming (1) and (2) is not particularly useful in our use case.

C. Gummel's iterative method

In order to solve the three equations in a self-consistent manner, we use the Gummel's iterative method. The method can be described as having a lower convergence rate while being more robust when it comes to the initial guess.

The method can generally be separated into an inner and an outer loop, where, during the loops, $\bar{\phi}_n$, $\bar{\phi}_p$, and $\bar{V}$ are continuously updated. Let us look at one iteration of the inner loop:

first we consider the equation (4). $\bar{\phi}_n$ and $\bar{\phi}_p$ are kept constant, and $\bar{V}$ is perturbed to $\bar{V} = \bar{V}_\mathrm{old} + \delta\bar{V}$. The equation is linearised in $\delta\bar{V}$ and (4) is brought to the matrix form $A\delta V = b$, where

$$\begin{aligned}
A &= (D_1\bar{\varepsilon})D_1 - \\
&\quad - \bar{n}_i\left[\exp(\bar{V}_\mathrm{old})\exp(-\bar{\phi}_n) + \exp(-\bar{V}_\mathrm{old})\exp(\bar{\phi}_p)\right],\\
b &= \bar{C} - D_1(\bar{\varepsilon}D_1\bar{V}_\mathrm{old}) + \\
&\quad + \bar{n}_i\left[\exp(\bar{V}_\mathrm{old})\exp(-\bar{\phi}_n) - \exp(-\bar{V}_\mathrm{old})\exp(\bar{\phi}_p)\right].
\end{aligned}$$

The equation is solved using the Biconjugate gradient stabilized method, freely available from external libraries, SciPy was used in our considerations[7]. The value for $\bar{V}$ is then updated. This step is done repeatedly until the change in $\bar{V}$ drops below a threshold (usually looking at the square norm is enough)

The same process is also done for (1), and (2). For (1), $\bar{\phi}_n$ is changed while other variables are kept constant. Similarly, for (2), $\bar{\phi}_p$ is changed.

The inner loop thus consists of three separate updates for $\bar{V}$, $\bar{\phi}_n$, and $\bar{\phi}_p$. The outer loop simply applies the inner loop repeatedly until the three variables stop changing significantly.

D. Boundary conditions

The only form of the boundary condition that was investigated was the fixed voltage one. To this end, the first and last elements of the vector b defined in the last section were modified. The modification would guarantee that V is fixed at the boundary.

IV. COMPUTATIONAL ANALYSIS

The computational analysis was limited to three simple one-dimensional systems due to the difficulties arising from getting the more complicated systems working.

A. One dimensional PN junction

First, a simple PN junction was investigated. The total length was $L = 50\,\mathrm{\mu m}$, half made up of a donor (p), and the other half of an acceptor (n) with doping densities $N_D = 1\times10^{22}\,\mathrm{m}^{-3}$, and $N_A = 2\times10^{22}\,\mathrm{m}^{-3}$ respectively. The relative permittivity was taken to be a constant of $\varepsilon_r = 11.68$.

B. Simple dielectric

Secondly, a dielectric of length $L = 50\,\mathrm{\mu m}$ was investigated. The dielectric was made up of a segment of length $x_0 = 35\,\mu\text{m}$ with a relative permittivity $\varepsilon_{r1} = 4$ and secondly a segment of length $L-x = 15\,\mu\text{m}$ with permittivity $\varepsilon_{r2} = 1$. Boundary conditions of $V_l = 1\,\text{V}$ and $V_r = 0\,\text{V}$ were applied, and the ensuing distribution of V was calculated using the same method as in the previous part.

C. one dimensional CCD

Lastly, an attempt was made to investigate the one-dimensional CCD. The CCD was made to be of length $L = 50\,\mu\text{m}$ with a segment of length $2\,\mu\text{m}$ made of Poly-Silicon, an insulating layer of Silicon dioxide of length $0.25\,\mu\text{m}$ (no doping to simulate an insulator) and the rest made up of silicon with doping density $1\times10^{13}\,\text{cm}^{-3}$. Poly-Silicon in this case can be considered as a conductor, a very high doping $1\times10^{16}\,\text{cm}^{-3}$ was used for the simulation as a pure conductor environment had not been implemented. Because there were difficulties with getting non-constant dielectric coefficient working with doping, a constant $\varepsilon_r = 11.68$ was used. A sketch of the CCD is given below.

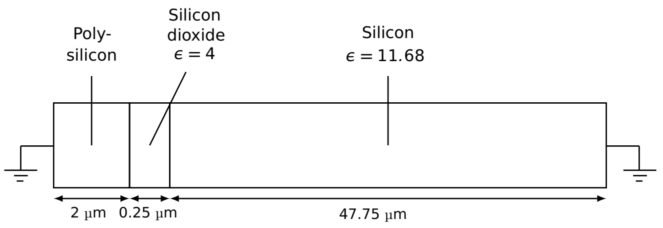

FIG. 1: An illustrative sketch of a one-dimensional CCD

V. RESULTS

A. PN junction

The steady state with no applied voltage and no recombination was simulated successfully. The region of interest at the PN boundary is shown on figure 2. Notably, the solver would break down any time recombination or bias were introduced.

Theoretical calculations of the no bias steady state give the following expressions for the voltage difference, characteristic width of the central area, and finally the maximum electric field at the center:

$$\Delta V = V_T \ln\left(\frac{N_A N_D}{n_i^2}\right) = 0.738\,\text{V},$$

$$W = \sqrt{\frac{2\varepsilon_r\varepsilon_0\,(N_A+N_D)\,\Delta V}{eN_AN_D}} = 0.378\,\mu\text{m},$$

$$E_{\max} = -\frac{eN_AN_DW}{\varepsilon_r\varepsilon_0\,(N_A+N_D)} = 3.9\times10^6\,\text{V}\,\text{m}^{-1}.$$

Comparing these to the simulated values

$$\begin{aligned} \Delta V &= 0.738\,\text{V}, \\ W &\approx 0.35\,\mu\text{m}, && \text{—FWHM} \\ E_{\max} &= 3.6\times10^6\,\text{V}\,\text{m}^{-1}, \end{aligned}$$

we see a very good overlap between the theoretical and simulated values. The discrepancies in $E_{\max}$ can be attributed to the finite size of the grid.

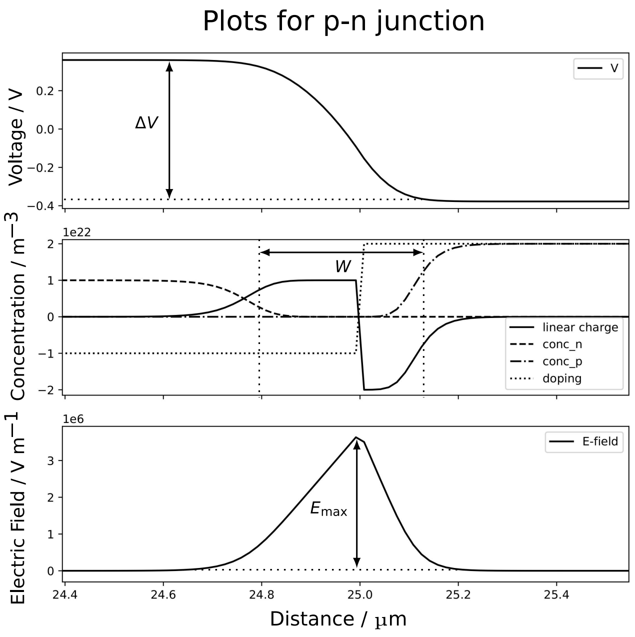

FIG. 2: central region of the $L = 50\,\mu\text{m}$ PN junction.
a) voltage as a function of x.
b) Total linear charge density alongside doping, electron, and hole charge densities as a function of x.
c) Electric field as a function of x.

B. Dielectric

The steady state with applied voltage between the two ends in a simple dielectric system was simulated successfully as can be seen on figure 3.

Given the formulation of the problem, it is straightforward to find the electric field as a function of coordinate:

$$E = \begin{cases} \frac{\Delta V}{x + \frac{\varepsilon_1}{\varepsilon_2}(L-x)} & \text{if } x \leq x_0, \\ \frac{\Delta V}{\frac{\varepsilon_2}{\varepsilon_1}x + (L-x)} & \text{if } x > x_0. \end{cases}$$

Notably, the voltage of the "turning point" is given by

$$V(x = x_0) = \frac{\Delta V}{1 + \frac{\varepsilon_2}{\varepsilon_1}\frac{x}{L-x}} = 0.63\,\mathrm{V},$$

and this agrees exactly with the simulated result.

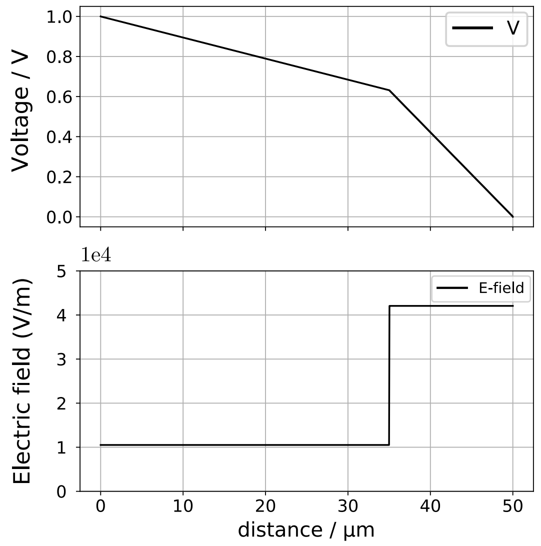

FIG. 3: One-dimensional CCD
a) voltage as a function of x.
b) Total linear charge density alongside doping, electron, and hole charge densities as a function of x.
c) Electric field as a function of x.

C. CCD

The simulation of the CCD was unsuccessful due to various reasons:

1. Recombination was not implemented successfully. Fortunately, recombination does not significantly affect electron storage aspect of a CCD.

2. Due to the limitations of the code, non-constant dielectric coefficient and doping could not be simulated at the same time.

3. Boundary conditions were not enforced successfully due to recombination not working. Hence, only the no-current case was considered

4. The properties of the conductive Poly-Silicon are not fully reflected by setting the doping density to be as high as possible.

As expected, the produced distributions do not follow what they should look like.

Notably, the right hand side of the graph has a sharp increase in voltage that should not appear at all. The doping has a misleading peak at the beginning (which corresponds to the conductive region). The voltage graph looks reasonable on the left hand side with the conducting region having virtually constant voltage.

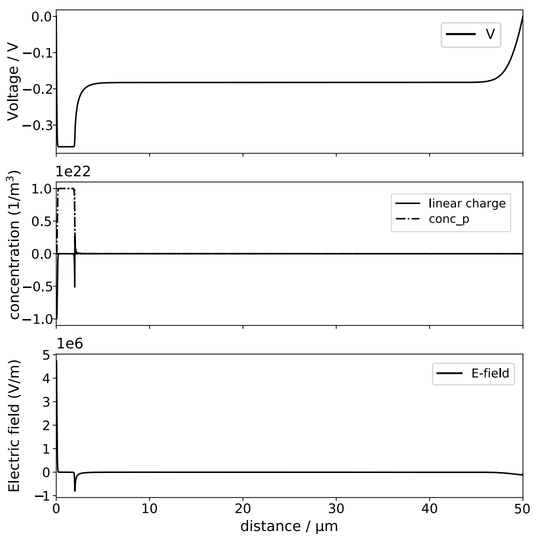

FIG. 4: Voltage and electric field as a function of coordinate in a $L = 50\,\mu\mathrm{m}$ dielectric with two connected homogeneous regions.

VI. DISCUSSION

The initial aim of the project was to analyse the full three-dimensional model of a CCD and observe how the electrons that get trapped inside a potential well in the CCD leak while idling or while being shifted in the read-out phase.

This however needed a working numerical solver for an arbitrary three-dimensional system with given boundary conditions. To that end, we started with a one-dimensional solver, and the initial plan was to move on to higher dimensions. Unfortunately we did not get the one-dimensional case working for recombination between the holes and the electrons, and for any non-zero bias applied to the system. Further, non-constant dielectric coefficient and non-zero doping could not be made to work together, this was caused by the unsuccess in figuring out how to make the first and second order differential operators agree with each-other ($D_1^2 \neq D_2$). Several weeks were spent on debugging, but to no avail. However, two simple one-dimensional systems did produce correct results, as illustrated in the previous section. The one-dimensional CCD did not produce correct results as it is a biased system with non-constant dielectric coefficient and non-zero doping with more complicated components that cannot be simply described by a doping density.

This project can definitely be further worked on, the mistakes in the one-dimensional case still need to be found and the code needs to be extended into further dimensions. In higher dimensions, more complicated data structures need to be used for optimisation purposes and in order to take into account the more complicated forms for the differential equations (1), (2), and (3).

[1] Ž. Ivezić, S. M. Kahn, J. A. Tyson, B. Abel, E. Acosta, R. Allsman, D. Alonso, Y. AlSayyad, S. F. Anderson, J. Andrew, and et al., Astrophys. J. **873**, 111 (2019), arXiv:0805.2366.

[2] C. M. Hirata and A. Choi, (2019), 10.1088/1538-3873/ab44f7, arXiv:1906.01846.

[3] J. Rhodes, A. Leauthaud, C. Stoughton, R. Massey, K. Dawson, W. Kolbe, and N. Roe, (2010), 10.1086/651675, arXiv:1002.1479.

[4] D. Weatherill, *Charge Collection in Silicon Imaging Sensors*, Ph.D. thesis (2016).

[5] D. Vasileska, *Computational Electronics: Semiclassical and Quantum Device Modeling and Simulation.*

[6] S. H. Simon, *The Oxford Solid State Basics.*

[7] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. Jarrod Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. Carey, İ. Polat, Y. Feng, E. W. Moore, J. Vand erPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and S. . . Contributors, Nature Methods **17**, 261 (2020).

APPENDIX – SIMULATION CODE IN PYTHON

The complete code used in the simulations.

setup.py

setup.py is located in the parent directory compared to other files which are in the CCD1D package directory.

```
from setuptools import setup

setup(name='CCD1D',
     version='0.0.1',
     description='One-dimensional CCD simulation.',
     classifiers=[
         'Development Status :: 3 - Alpha',
         'Intended Audience :: Science/Research',
         'Topic :: Scientific/Engineering :: Semiconductor Devices',
         'License :: OSI Approved :: MIT License',
         'Operating System :: OS Independent',
         'Programming Language :: Python :: 3.8',
     ],
     keywords='CCD 1D gummel',
     author='...', # Redacted for examination anonymity purposes
     license='MIT',
     python_requires='>=3.8',
     zip_safe=False)
```

__init__.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from . import config
from . import constants
from . import gummel_solver1D
from . import recombination
from . import utils

__author__ = '...' # Redacted for examination anonymity purposes
__credits__ = 'Daniel Weatherill'
__version__ = '0.0.1'
__status__ = 'Under construction'
```

utils.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
Provides utility functions which are of use throughout the entire package.
"""

import numpy as np
from scipy.sparse import dia_matrix, lil_matrix
```

```
def first_derivative_operator(length: int, grid_step_size: float) -> np.ndarray:
    """Returns the matrix corresponding to the first derivative."""
    N = length
    filler = np.array((np.full(N, 1), np.full(N, 0), np.full(N, -1)))
    filler[1][0] = -1
    filler[1][N - 1] = 1
    offset = np.array([1, 0, -1])
    operator = dia_matrix((filler, offset), shape=(N, N)) / 2 / grid_step_size
    return operator


def first_derivative(vec: np.ndarray, grid_step_size: float) -> np.ndarray:
    """Calculates the first derivative using a central difference method."""
    return first_derivative_operator(len(vec), grid_step_size) * vec


def second_derivative_operator(length: int, grid_step_size: float) -> np.ndarray:
    """Returns the matrix corresponding to the second derivative."""
    N = length
    filler = np.array((np.full(N, 1), np.full(N, -2), np.full(N, 1)))
    filler[1][0] = -1
    filler[1][N - 1] = -1
    offset = np.array([-1, 0, 1])
    operator = dia_matrix((filler, offset), shape=(N, N)) / grid_step_size**2
    return operator


def second_derivative(vec: np.ndarray, grid_step_size: float) -> np.ndarray:
    """Calculates the second derivative using a central difference method."""
    return second_derivative_operator(len(vec), grid_step_size) * vec


def rescale(vec, new_min=0, new_max=1):
    """Rescales the values of an vector from new_min to new_max via the min-max method."""
    old_min = vec.min()
    old_max = vec.max()
    return np.array([lambda x: (((x - old_min) * (new_max - new_min)) / (old_max - old_min)) + new_min for
        x in vec])
```

config.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
Configures the CCD1D class.
"""

import math
from typing import Tuple

import numpy as np
from scipy.sparse import csr_matrix, lil_matrix, diags
from scipy.sparse.linalg import bicgstab

from . import constants, utils, recombination


def setup_1D_PN_junction():
    N = 3000
    length = 50e-6
    epsilon_r = np.full(N, 11.68)
    doping = np.append(np.full(N//2, -1e16 * 1e6), np.full(N//2, 2e16 * 1e6))
    n_i = 5.29E19 * (constants.T / 300)**2.54 * np.exp(-6726 / constants.T) * 1e6
    mu_n, mu_p = (1400e-4, 450e-4)
    V_l, V_r = (0, 0)
    V = np.arcsinh(-doping / 2 / n_i) * constants.V_t

    return CCD1D(length, doping, (V_l, V_r), epsilon_r, (mu_n, mu_p), n_i, V)


def setup_1D_basic_CCD():
    length = 50e-6
    N = 3000
    epsilon_r = np.full(N, 4) # Right now, non-constant dielectric and non-zero doping don't work
        simultaneously
    doping = np.append(np.append(np.full(N//25, 1e22), np.full(N//200, 0)), np.full(N - N//25 - N//200,
        1e13 * 1e6))
    #doping = np.append(np.full(N//50, 0), np.full(N - N//50, 1e13 * 1e6))
    n_i = 5.29E19 * (constants.T / 300)**2.54 * np.exp(-6726 / constants.T) * 1e6
    mu_n, mu_p = (1400e-4, 450e-4)
    V_l, V_r = (0, 0)
    #V = np.append(np.append(np.full(N//25, 1e30), np.full(N//200, 0)), np.full(N - N//25 - N//200, 1e13 *
        1e6))
    V = np.arcsinh(-doping / 2 / n_i) * constants.V_t

    return CCD1D(length, doping, (V_l, V_r), epsilon_r, (mu_n, mu_p), n_i, V)


class CCD1D:
    """
    Class containing all the physical parameters describing a CCD.
    The CCD is divided into N segments, each of size grid_step_size such that
    N = floor(length / grid_step_size) + 1.
    each segment has its own doping factor.

    Base attributes:
        length - Length of the CCD in micrometers
        epsilon_r - Relative permittivity
        doping - Doping density of the dielectric. plus is for acceptor, negative for donor
        n_i - No chemical potential n,p density. Depends on the material
        mu_n, mu_p - Mobility of the electrons and holes in m^2/( Vs)
        V_l, V_r - Boundary conditions for the voltage on the lhs and rhs of the CCD
        recombination - Object that deals with calculating recombination rates across the CCD

    Derived attributes:
        D_n, D_p - Diffusion coefficients of the electrons and holes in SI units. Related to mobility via
            the Einstein relation
        number_of_steps - Number of steps on the grid
        debye_length - Debye length of the CCD. Defined through the maximal values of doping and epsilon_r
        grid_step_size - Grid step size
        V - Voltage along the CCD in Volts
        phi_n, phi_p - Chemical potential of electrons and holes in units of V_t
        conc_n, conc_p - Concentrations of electrons and holes respectively (1/m^3)
        conc_charged_particle - Total charge density along the CCD (C/m)
        electric_field - Electric field at each point in the CCD (V/m)
        current_n, current_p - n-current and p-current along the CCD (A/m^2)

    Normalised attributes:
        reduced_V_l, reduced_V_r - V_l, V_r in units of V_t
        reduced_epsilon_r - Relative permittivity in units of the scaling epsilon (max value of epsilon)
        reduced_V - V in units of V_t
        reduced_phi_n, reduced_phi_p - phi_n and phi_p in units V_t
    reduced_doping - Doping density in units of the scaling doping density (max value of doping)
    reduced_n_i - n_i in units of the scaling doping density (max value of doping)
    reduced_grid_step_size - grid_step_size in units of Debye length
"""

def __init__(self, length: float = None, doping: np.ndarray = None, V_lr: Tuple[float, float] = None,
     epsilon_r: np.ndarray = None, mu_np: Tuple[float, float] = None, n_i: float = None, V: np.ndarray =
     None):
    # If no corresponding arguments are passed, initialise to predefined values
    self.length = length if length is not None else 50e-6
    self.epsilon_r = epsilon_r if epsilon_r is not None else np.full(3000, 4)
    self.doping = doping if doping is not None else np.append(np.full(1500, -1e16 * 1e6), np.full(1500,
        1e16 * 1e6))
    self.n_i = n_i if n_i is not None else 5.29E19 * (constants.T / 300)**2.54 * np.exp(-6726 /
        constants.T) * 1e6
    self.mu_n, self.mu_p = mu_np if mu_np is not None else (1400e-4, 450e-4)
    self.V_l, self.V_r = V_lr if V_lr is not None else (0, 0)
    self.recombination = recombination.Recombination()

    # Derive other quantities and set initial guesses for the variables
    self.D_n = self.mu_n * constants.k_b * constants.T / constants.e
    self.D_p = self.mu_p * constants.k_b * constants.T / constants.e
    self.number_of_steps = self.doping.size
    self.grid_points = np.linspace(0, self.length, self.number_of_steps)
    self.grid_step_size = self.length / (self.number_of_steps - 1)
    self.debye_length = math.sqrt(max(self.epsilon_r) * constants.epsilon_0 * constants.k_b *
        constants.T / constants.e**2 / max(abs(self.doping)))
    if V is not None:
        self.V = V
    else:
        self.V = np.arcsinh(-self.doping / 2 / self.n_i) * constants.V_t
    self.phi_n = np.zeros(self.number_of_steps)
    self.phi_p = np.zeros(self.number_of_steps)
    self.conc_n = self.n_i * np.exp(self.V / constants.V_t)
    self.conc_p = self.n_i * np.exp(-self.V / constants.V_t)

    sum_of_donors = abs((self.doping * (self.doping > 0)).sum())
    sum_of_acceptors = abs((self.doping * (self.doping < 0)).sum())
    self.conc_n = self.conc_n * sum_of_donors / self.conc_n.sum()
    self.conc_p = self.conc_p * sum_of_acceptors / self.conc_p.sum()
    self.conc_charged_particle = self.conc_p - self.conc_n - self.doping

    self.electric_field = -utils.first_derivative(self.V, self.grid_step_size)
    self.current_n = constants.e * self.conc_n * self.mu_n * self.electric_field
    self.current_n += constants.e * self.D_n * utils.first_derivative(self.conc_n, self.grid_step_size)
    self.current_p = constants.e * self.conc_p * self.mu_p * self.electric_field
    self.current_p -= constants.e * self.D_p * utils.first_derivative(self.conc_p, self.grid_step_size)

    self.reduced_V_l = self.V_l / constants.V_t
    self.reduced_V_r = self.V_r / constants.V_t
    self.reduced_epsilon_r = self.epsilon_r / max(self.epsilon_r)
    self.reduced_V = self.V / constants.V_t
    self.reduced_phi_n = self.phi_n / constants.V_t
    self.reduced_phi_p = self.phi_p / constants.V_t
    self.reduced_doping = self.doping / max(abs(self.doping))
    self.reduced_n_i = self.n_i / max(abs(self.doping))
    self.reduced_grid_step_size = self.grid_step_size / self.debye_length

def update_unreduced(self):
    """
    Updates the three initially unknown quantities based on the reduced quantities.
    Needed since the Gummel method we use updates reduced values.
    """
    self.V = self.reduced_V * constants.V_t
```

```
        self.phi_n = self.reduced_phi_n * constants.V_t
        self.phi_p = self.reduced_phi_p * constants.V_t
        self.conc_n = self.n_i * np.exp(self.reduced_V - self.reduced_phi_n)
        self.conc_p = self.n_i * np.exp(-self.reduced_V + self.reduced_phi_p)
        self.conc_charged_particle = self.conc_p - self.conc_n - self.doping

        self.electric_field = -utils.first_derivative(self.V, self.grid_step_size)
        self.current_n = constants.e * self.conc_n * self.mu_n * self.electric_field
        self.current_n += constants.e * self.D_n * utils.first_derivative(self.conc_n, self.grid_step_size)
        self.current_p = constants.e * self.conc_p * self.mu_p * self.electric_field
        self.current_p -= constants.e * self.D_p * utils.first_derivative(self.conc_p, self.grid_step_size)
        self.recombination.update_recombination(self.conc_n, self.conc_p, self.n_i)

    def print_info(self):
        print("CCD parameters:")
        print(f"\t{'total length: ':>28}{1e6 * self.length:.3f }m,")
        print(f"\t{'step size: ':>28}{1e6 * self.grid_step_size:.3f }m,")
        print(f"\t{'number of steps: ':>28}{self.number_of_steps}")
        print(f"\t{'maximum epsilon_r: ':>28}{max(self.epsilon_r):.3f},")
        print(f"\t{'Debye length: ':>28}{1e6 * self.debye_length:.3f }m,")
        print(f"\t{'step size / Debye length: ':>28}{self.grid_step_size / self.debye_length:.3f},")
        print(f"\t{'n_i: ':>28}{self.n_i/1e6:.3E} 1/cm^3,")
        print(f"\t{'max doping: ':>28}{max(abs(self.doping))/1e6:.3E} 1/cm^3,")
        print(f"\t{'electron mobility: ':>28}{self.mu_n * 1e4:.3f} cm^2/(Vs),")
        print(f"\t{'hole mobility: ':>28}{self.mu_p * 1e4:.3f} cm^2/(Vs),")
        print(f"\t{'electron diffusivity: ':>28}{self.D_n * 1e4:.3f} cm^2/s,")
        print(f"\t{'hole diffusivity: ':>28}{self.D_p * 1e4:.3f} cm^2/s,")
```

gummel_solver1D.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
Uses Gummel iterations to solve the van Roosbroeck system of equations.
"""

from math import exp

import numpy as np
from scipy.sparse import dia_matrix, lil_matrix
from scipy.sparse.linalg import bicgstab

from . import config, constants, utils


def lhs_helper(V, phi_n, phi_p):
    # Helper function that is used to construct the LHS of the inner loop.
    return np.exp(V) * np.exp(-phi_n) + np.exp(-V) * np.exp(phi_p)


def rhs_helper(V, phi_n, phi_p):
    # Helper function that is used to construct the RHS of the inner loop.
    return np.exp(V) * np.exp(-phi_n) - np.exp(-V) * np.exp(phi_p)


def bern(x):
    # Generating function for the Bernoulli numbers, used as a helper function for Scharfetter-Gummel
        discretisation.
    if x > 0.01:
        return x * exp(-x) / (1 - exp(-x))
    elif x < 0 and abs(x) > 0.01:
        return x / (exp(x) - 1)
    elif x == 0:
        return 1.
    else:
        term = 1.
        sm = term
        i = 0
        while sm != sm + term:
            i += 1
            term = term * x / (i+1)
            sm += term
        return 1./sm


def bernnp(t1,t2,n):
    prefac = 1 if n else -1
    return bern(prefac*(t1-t2))


def assemble_lhs_cont_matrix(grid_step_size, reduced_V, diffusivity, is_n):
    mat = lil_matrix((len(reduced_V), len(reduced_V)))
    for i in range(1, len(reduced_V) - 1):
        mat[i,i-1] = diffusivity / grid_step_size**2 * bernnp(reduced_V[i - 1], reduced_V[i], is_n)
        mat[i,i] = - diffusivity / grid_step_size**2 * bernnp(reduced_V[i], reduced_V[i - 1], is_n)\
                   - diffusivity / grid_step_size**2 * bernnp(reduced_V[i], reduced_V[i + 1], is_n)
        mat[i,i+1] = diffusivity / grid_step_size**2 * bernnp(reduced_V[i + 1], reduced_V[i], is_n)
    return mat.tocsr()


class GummelSolver1D:
    """
    A class that takes a 1D CCD and solves for its voltage, electron and hole concentration using the
        Gummel's Method.
    For each CCD, a separate instance of the class needs to be called.

    Attributes:
        ccd - the 1D CCD to be solved for.

    NOTE: Issues arise when attempted to enforce boundary conditions, e.g. bias voltages.
    """
    def __init__(self, ccd: config.CCD1D):
        self.ccd = ccd

    def enforce_partial_boundary(self, fraction, A, b):
        left_goal = self.ccd.V_l * fraction / constants.V_t
        right_goal = self.ccd.V_r * fraction / constants.V_t
        A[0, 1] = 0
        b[0] = A[0, 0] * (left_goal - self.ccd.reduced_V[0])
        N = self.ccd.number_of_steps
        A[N - 1, N - 2] = 0
        b[N - 1] = A[N - 1, N - 1] * (right_goal - self.ccd.reduced[N - 1])
        A.eliminate_zeros()

    def inner_loop(self, dV_norm_tol, n_max_iter, debug = False):
        """
        Inner loop of the Gummel's Method.
        If dV is the change in voltage for an iteration, then the inner loop
        repeatedly linearises the non-linear Poisson equation and brings it to the form
        A dV = b, where A is a matrix and b is a vector. This is solved using Bi-CGSTAB until
        the l2-norm (sum of its individual elements squared) is smaller than dV_norm_tol or
        until the number of iterations reaches n_max_iter.
        """
        dV_norm_previous = np.inf
        for i in range(n_max_iter):
            if (debug): print(f"\tIteration {i}, previous norm {dV_norm_previous}")

            A_diagonal = -self.ccd.reduced_n_i * lhs_helper(self.ccd.reduced_V, self.ccd.reduced_phi_n,
                 self.ccd.reduced_phi_p)
            A = utils.first_derivative_operator(self.ccd.number_of_steps,
                 self.ccd.reduced_grid_step_size).multiply(
                utils.first_derivative(self.ccd.reduced_epsilon_r, self.ccd.reduced_grid_step_size))
            A += utils.second_derivative_operator(self.ccd.number_of_steps,
                 self.ccd.reduced_grid_step_size).multiply(self.ccd.reduced_epsilon_r)
            A += dia_matrix((A_diagonal, 0), shape=(self.ccd.number_of_steps, self.ccd.number_of_steps))

            b = self.ccd.reduced_n_i * rhs_helper(self.ccd.reduced_V, self.ccd.reduced_phi_n,
                 self.ccd.reduced_phi_p)
            b += self.ccd.reduced_doping - self.ccd.reduced_epsilon_r *
                 utils.second_derivative(self.ccd.reduced_V, self.ccd.reduced_grid_step_size)

            dV, status = bicgstab(A, b, tol=1E-3)
            if status:
                print("\tInner loop: Error in BiCGSTAB iteration.")
                raise RuntimeError("Unsuccessful BiCGSTAB run")

            dV_norm = np.sum(dV**2)
            if (debug): print(f"\tnorm: {dV_norm:.3E}")
            if dV_norm > dV_norm_previous:
                print(f"\tInner loop {i}: previous norm: {dV_norm_previous:.4E}, current norm: {dV_norm:.4E}")
                print("\tInner loop: Norm increased, iteration doesn't converge. Improve the initial guess.")
                raise RuntimeError("Norm increased")

            self.ccd.reduced_V += dV
            if dV_norm < dV_norm_tol:
                print(f"\tInner loop: Norm withing tolerance after {i} iterations, breaking..")
                break

            dV_norm_previous = dV_norm

            if i == n_max_iter - 1: print("\tInner loop: Maximum number of iterations reached, did not
                 converge.")

        self.ccd.update_unreduced()
        if dV_norm_previous == np.inf:
            dV_norm_previous = 0
        print(f"\tInner loop: residual norm of reduced V: {dV_norm_previous:.3E}")

    def outer_loop(self):
        """
        Outer loop of the Gummel's Method.
        If we ignore recombination, we do not need to iterate because the equation can be solved exactly.
        """
        A_n = assemble_lhs_cont_matrix(self.ccd.grid_step_size, self.ccd.reduced_V, self.ccd.D_n, True)
        A_p = assemble_lhs_cont_matrix(self.ccd.grid_step_size, self.ccd.reduced_V, self.ccd.D_p, False)

        # NOTE: No recombination.
        b_n = -self.ccd.recombination.get_recombination()
        b_p = -b_n.copy()

        # Normalise the lhs and rhs!
        A_n *= self.ccd.debye_length**2
        A_p *= self.ccd.debye_length**2
        b_n *= self.ccd.debye_length**2
        b_p *= self.ccd.debye_length**2
        # NOTE: The outer loop does not work properly with recombination, so we initialise these vectors to
            zero for now.
        b_n = np.zeros(self.ccd.number_of_steps)
        b_p = np.zeros(self.ccd.number_of_steps)

        # If no recombination, the new solutions have to conserve the norm for n and p.
        # This has to be done separately because Bi-CGSTAB finds the kernel and that can be scaled freely.
        if np.count_nonzero(b_n) == 0:
            n_old_sum = abs(self.ccd.conc_n.sum())
            p_old_sum = abs(self.ccd.conc_p.sum())


        conc_n, status = bicgstab(A_n, b_n, x0=self.ccd.conc_n)
        if status:
            print("\tOuter loop: Error in electron BiCGSTAB iteration.")
            raise RuntimeError("Unsuccessful BiCGSTAB run")
        relative_change_n = np.average(np.abs(1 - conc_n / self.ccd.conc_n))
        self.ccd.conc_n = conc_n

        conc_p, status = bicgstab(A_p, b_p, x0=self.ccd.conc_p)
        if status:
            print("\tOuter loop: Error in hole BiCGSTAB iteration.")
            raise RuntimeError("Unsuccessful BiCGSTAB run.")
        relative_change_p = np.average(np.abs(1 - conc_p / self.ccd.conc_p))
        self.ccd.conc_p = conc_p

        # Deal with conserving the number of n-s and p-s if b = 0
        if np.count_nonzero(b_n) == 0:
            n_new_sum = abs(self.ccd.conc_n.sum())
            p_new_sum = abs(self.ccd.conc_p.sum())
            self.ccd.conc_n *= n_old_sum / n_new_sum
            self.ccd.conc_p *= p_old_sum / p_new_sum

        # Update normalised quantities
        self.ccd.phi_n = -np.log(self.ccd.conc_n / self.ccd.n_i / np.exp(self.ccd.reduced_V)) * constants.V_t
        self.ccd.phi_p = np.log(self.ccd.conc_p / self.ccd.n_i * np.exp(self.ccd.reduced_V)) * constants.V_t
        self.ccd.reduced_phi_n = self.ccd.phi_n / constants.V_t
        self.ccd.reduced_phi_p = self.ccd.phi_p / constants.V_t
        print(f"\tOuter loop: relative change of the concentration of n: {relative_change_n:.3E}, p:
            {relative_change_p:.3E}")

    def solve_initial_conditions(self):
        print(f"--------------------------------------")
        for i in range(5):
            self.inner_loop(1E-2, 100)
            print(f"Computing steady state... iteration {i}:")
            self.outer_loop()
            print(f"--------------------------------------")
```

recombination.py

Recombination was not implement into the solver successfully. `recombination.py` is included for completeness.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

class Recombination:
    """
    Calculates the recombination rate.
    """
    def __init__(self):
        self.rate = None
        self.coef = 4e-18 # m^3/s

    def update_recombination(self, n, p, ni):
        self.rate = self.coef * (n * p - ni * ni);

    def get_recombination(self):
        return self.rate
```

main.py (for p-n junction and CCD)

main.py is located in the parent directory compared to other files which are in the CCD1D package directory.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
Script for producing plots. This script is meant for the PN junction and the CCD.
"""

import matplotlib.pyplot as plt

from CCD1D.config import setup_1D_PN_junction, setup_1D_basic_CCD
from CCD1D.gummel_solver1D import GummelSolver1D


def generate_plot(ccd, left_x, right_x):
    monochrome = ["k-","k--","k-.","k:"]

    fig, axs = plt.subplots(3, sharex=True)
    fig.suptitle("1D CCD Data")
    plt.xlabel(r"distance / $\mathrm{\mu m}$")
    axs[0].plot(x, ccd.V, monochrome[0], label="V")
    axs[0].set_ylabel("Voltage / V")
    axs[0].legend()
    axs[1].plot(x, ccd.conc_charged_particle, monochrome[0], label="linear charge")
    axs[1].plot(x, ccd.conc_n, monochrome[1], label="conc_n")
    axs[1].plot(x, ccd.conc_p, monochrome[2], label="conc_p")
    axs[1].plot(x, ccd.doping, monochrome[3], label="doping")
    axs[1].legend()
    axs[1].set_ylabel("concentration ($1/\\mathrm{m}^3$)")
    plt.ticklabel_format(axis='y', style='sci', scilimits=(0, 0))
    axs[2].plot(x, ccd.electric_field, monochrome[0], label="E-field")
    axs[2].set_ylabel("Electric field (V/m)")
    axs[2].legend()
    plt.xlim(left_x, right_x)


if __name__ == '__main__':
    ccd = setup_1D_PN_junction()
    #ccd = setup_1D_basic_CCD()
    ccd.print_info()
    solver = GummelSolver1D(ccd)
    solver.solve_initial_conditions()

    x = ccd.grid_points * 1e6

    generate_plot(ccd, 0, ccd.length * 1e6)

    plt.show()
```

simple_dielectric.py

simple_dielectric.py is located in the parent directory compared to other files which are in the CCD1D package directory.

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
Script for producing plots. This script is meant for the simple dielectric.
"""

import matplotlib.pyplot as plt

from CCD1D.config import setup_1D_dielectric
from CCD1D.gummel_solver1D import GummelSolver1D


def generate_dielectric_plot(ccd):
     monochrome = ["k-", "k--", "k-.", "k:"]

     fig, axs = plt.subplots(2, sharex=True)
     fig.suptitle("1D CCD Data")
     plt.xlabel(r"distance / $\mathrm{\mu m}$")
     axs[0].plot(x, ccd.V, monochrome[0], label="V")
     axs[0].set_ylabel("Voltage / V")
     axs[0].legend()
     axs[1].plot(x, ccd.electric_field, monochrome[0], label="E-field")
     axs[1].set_ylabel("Electric field (V/m)")
     axs[1].legend()
     axs[0].grid('both')
     axs[1].grid('both')
     axs[1].set_ylim(0, 50000)


if __name__ == "__main__":
     ccd = setup_1D_dielectric()
     ccd.print_info()
     solver = GummelSolver1D(ccd)
     solver.solve_initial_conditions()

     x = ccd.grid_points * 1e6
     # The expected plot shape is given here: https://www.desmos.com/calculator/ykncppbjpc
     generate_dielectric_plot(ccd)
     plt.show()
```