A Python Implementation of a Drift-Diffusion Model to Capture Ion Migration in Perovskite Solar Cells

A thesis submitted to the faculty of the University of Minnesota by

Nathan J. Anderson

Department of Chemical Engineering

In partial fulfillment of the requirements for the degree of

Master of Science

Advised by Dr. Zhihua Xu

Swenson College June, 2021

Acknowledgements

I would like to express my sincere gratitude towards my advisor, Professor Zhihua Xu, for introducing me to this exciting field of research. Thank you for your patience and guidance while I found my direction with this thesis. It was a pleasure working with you.

Secondly, I would like to thank Professor Davis and Professor Rother for serving on my committee. Both of you were very influential with regards to my interest in numerical methods.

Lastly, I want to extend my thanks to Professor Lodge. I highly value the teaching experience and knowledge passed down from the particle technology lab.

It was a fun and memorable experience getting to know all of the graduate students, staff, and faculty in the Chemical Engineering Department. Thanks everyone.

Dedication

I would like to dedicate this work to my family and friends who supported me during this project.

A few honorable mentions:

My parents, who taught me the value of education at a young age. They worked very hard to provide me the opportunity to get an education and chase after my curiosity.

My older brother, Orie. He sparked my interest in chemistry and mathematics at the very start of my college education. His strong work ethic and influential character has always driven me to pursue my goals with purpose and the highest effort.

Josie, who made all the long nights and insurmountable work bearable. I couldn't have imagined doing this without you.

To the countless others that helped me along this path, thank you so much. I love you all.

Abstract

Charge carrier dynamics and ion migration are attributed to the current-voltage hysteresis in perovskite solar cells (PSCs). This study implements a drift-diffusion model in Python to simulate the characteristic current-voltage scans for realistic device architectures. The novel work in this research involve the integration of a transfer-matrix optical model to the drift-diffusion model with ion migration, the implementation of a Radau 5th order solver to the method of lines, and a demonstration that standard Python libraries can handle stiff systems of differential algebraic equations. A comparative analysis with published works was conducted to validate the algorithm. It was found that the simulation was able to capture fast carrier dynamics under a variety of experimental conditions. Lastly, it is shown that the model captures physically relevant trends in PSCs.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years perovskite solar cells (PSCs) have achieved significant advancements with device stability and power conversion efficiency, making them a promising material for use in the bulk-power generation market. Despite being in the infancy of commercialization there is still a demand for research and development regarding device stability, degradation, and optimization. Mathematical modelling of PSCs has been gaining traction in the perovskite research community in an effort to characterize various device phenomena in order to tackle these challenges. However, there are few numerical methods available that can achieve accurate simulations for realistic devices. The work presented in this thesis introduces a new numerical method based in Python that solves the drift-diffusion semiconductor equations with ion transport. A comparitive study between Courtier et al.'s IonMonger [3, 1] is used to verify the model and assess performance of the algorithm.

1.1 Objective of Research

Due to the complex nature of PSC charge carrier dynamics there have been limited resources of reliable device simulators. The existing open source code for drift-diffusion simulators are based in Matlab which require expensive software subscriptions [1, 2]. In an effort to increase accessibility to fast and accurate PSC simulation, a new algorithm based on the numerical methods presented in [1, 3, 4] was developed in Spyder using Python's Numpy and Scipy libraries. Lastly, this project aimed to build foundational code to expand the drift-diffusion model to two and three dimensions.

1.2 Challenges

The aim of this research intended to overcome challenges related to the application of mathematical models to device architecture, carrier dynamics, optics, and numerical

stability. This section gives a brief insight to some of the physical phenomena unique to PSCs. The forthcoming chapters of this thesis will explore types of models that are applied to some of the listed phenomena followed by the numerical methods used to execute the simulation.

1.2.1 Current-Voltage Hysteresis

The stability and performance of PSCs are typically measured by current-voltage scans. This experiment measures the current response of a solar cell to an applied voltage. It is standard practice to scan the cell at a forward and reverse bias to determine the stability of the cell. For many PSCs, a discrepancy is observed between the two scan directions that is commonly denoted as hysteresis [5]. A simulation of this phenomena is shown in Figure 1.1. Many publications have suggested several reasons for the existence of current-voltage hysteresis in PSCs. Some of the contributing factors include interfacial trapping, ferroelectric polarization, and slow ion migration [5, 6]. However, it is suggested that the relationship between ion migration and trap-assisted recombination is the primary driver for the transient behavior [7]. When subject to an applied voltage, ion and carrier concentrations accumulate at interfacial junctions on drastically different timescales. Due to the slow ionic transient behavior, electron and hole densities cannot reach a true steady state until mobile ions satisfy a steady state condition. As the net ionic charge varies throughout the bulk of the intrinsic layer, development of asymmetric electron-hole concentrations can affect bimolecular and trap-mediated recombination mechanisms.

Current-voltage scans are the primary method used to calculate the fill factor and efficiency of photovoltaic cells. Therefore, the device stability plays an important role in the characterization of PSCs. Unstable devices have presented issues when trying to obtain certified efficiency ratings. Additionally, it is important to consider the stability of power output for applications in large scale operations.

The detection of hysteresis relies on the applied voltage scan rate. Hysteresis tends to be more pronounced as the scan rate increases due to a combination of charge carrier and ion dynamics. Slow scan rates tend to exhibit little to no hysteresis. Measurement procedures typically involve preconditioning the cell at a positive bias, sweeping the applied voltage down to short circuit conditions, and lastly reversing the direction of the scan towards the original starting voltage.

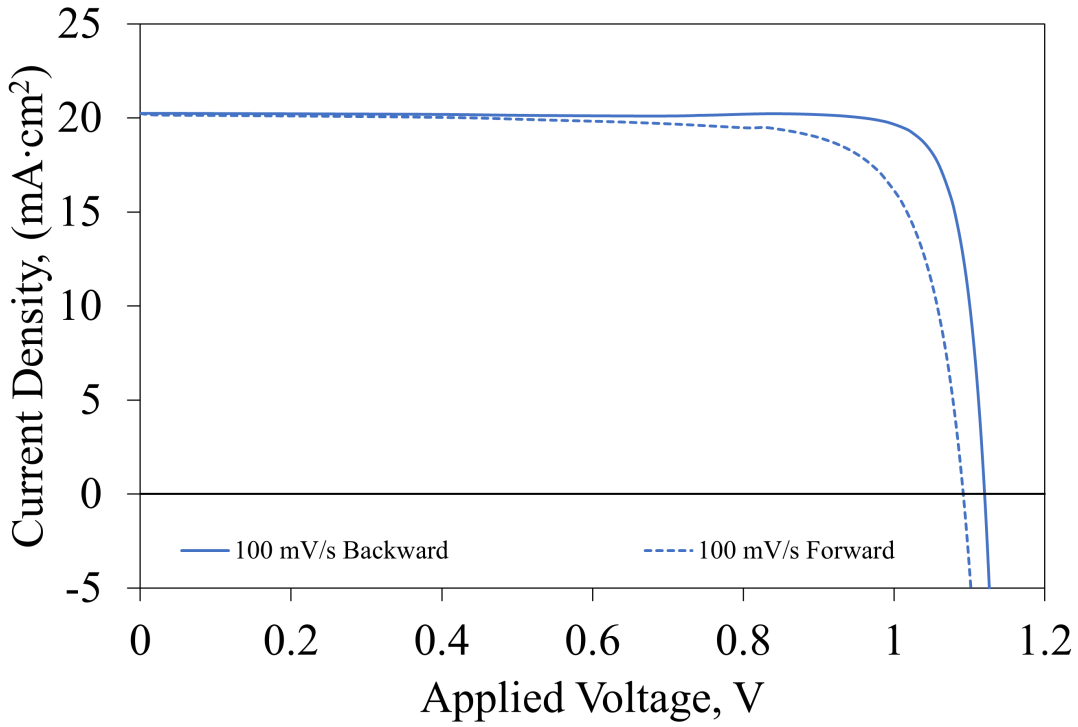

Figure 1.1: A simulated JV scan using a scan rate of 100 mV/s. The scan highlights the characteristic hysteresis curve for perovskite solar cells.

Proposed solutions to prevent ion migration involve the engineering of intrinsic layer composition, reduction of defects, reduction of grain boundary size, adjustments to device architecture, and enhancement of charge carrier extraction [7, 8, 9, 10, 11]. These examples further exemplify the importance of robust mathematical modelling in order to identify underlying mechanisms, plan experiments, and optimize devices.

1.2.2 Material Defects

Perovskites are a class of compounds composed of a metal halide crystal structure. Each type of perovskite is commonly given by an ABX_3 chemical formula, where A and B are cations and X is a halide anion. The chemical structure is a cubic crystalline material centered around a metal cation coordinated with an octohedron of anions. This thesis focuses on the methylammonium lead iodide ($MAPbI_3$) class of perovskites as they are the most widely used type of PSC due to their outstanding chemical and physical properties [12]. Lead based perovskites have achieved some of the most optimal bandgaps when compared to the Shockley-Quesser theoretical efficiency limit [13].

The crystal structure of perovskite is susceptible to imperfections due to the synthesis and manufacturing process [14]. These unique defect properties play a significant role regarding device stability and efficiency [5]. Point defects, surface boundaries, and grain boundaries are suspected to function as avenues for ion migration and potential wells for non-radiative recombination [15]. For $MAPbI_3$ perovskites, density functional theory calculations suggest that point defects with low formation energies create shallow traps and point defects with high formation energies create deep traps [15]. These unique characteristics indicate that the higher energies of formation for deep trap states down regulate non-radiative recombination and influence improved device efficiencies. However, the low activation energy for generating interstitial iodine defects can increase ion density at the grain boundaries producing deep potential wells. This phenomena up regulates non-radiative recombinations and lowers device efficiency.

Various types of methylammonium lead halide perovskites demonstrate self doping capabilities that can be tuned during synthesis. Intrinsic point defects, specifically in $MAPbI_3$, prevent the improvement of n-type conductivity through extrinsic doping. These types of perovskites tend to favor p-type doping profiles resulting from halide ion migration [15].

1.2.3 Device Degradation

For perovskites to become competitive in commerical markets they must retain optimal efficiencies for the lifetime of the device. Traditionally, these properties are determined by subjecting the solar cell to harsh accelerated aging tests. Over the years researchers have discovered various degradation mechanisms involving sensitivies to heat, mechanical stress, oxygen, humidity, and light soaking, many of which have been improved by incapsulating the device with protective layers or by modifying chemical structures and synthesis methods [16, 17, 18].

Currently, the available mathematical models provide a way to analyze short term transient experiments. However, they do not incorporate degradation reaction kinetics as a way to assess long term device performance. Expanding the drift-diffusion model to include reaction kinetics could provide a cheap and timely method to determine theoretical lifetimes of PSCs. The models provided in this work do not address this topic, however, the code provides a framework for future model expansion.

1.2.4 Environmental Impact

Most highly stable perovskites include lead in their crystal structure which has sparked concerns over the risk to human exposure and adverse environmental impact [19]. The toxicity and solubility of lead has the potential to leave permanent ecological damage. Debate over the ethical implications and environmental risks has suggested that lead-based

perovskites remain a viable option for the future of PSC development [20]. The effort to exchange lead with elements that have similar chemical and physical properties has remained a large area of study [21, 22, 23]. Theoretical studies using density functional theory calculations have proposed 10 potential elements that could satisfy the stability, lattice structure, and band gap requirements for PSCs [24]. However, the physical lead-free perovskites have not achieved efficiencies that are competitive with the standard lead-based structure. Further research has been suggested to engineer lead-free devices to achieve an optimistic goal of 15% efficiency [24].

1.3 Thesis Overview

This thesis is structured in a manner that addresses a range of topics involving semiconductor physics, discrete mathematics, numerical methods, programming, and simulation. Listed below is an overview of what each chapter covers, and a brief summary of what each chapter proposes.

Chapter 1. A detailed overview of the physics instigating the need for mathematical modelling of perovskite solar cells. The general introduction to current-voltage hysteresis, device defects, device degradation, and environmental impact. Lastly, a concise framework for the remainder of the thesis.

Chapter 2. The background regarding research history, device architecture and operation, and the recent progress made within the modelling community. The chapter concludes with an introduction to Python's capability as a scientific programming language.

Chapter 3. Defining the drift-diffusion model in terms of the parameters used in the simulation. This chapter seeks to identify the most critical equations used in the drift-diffusion model, their purpose, and physical relevance. The models used in this chapter references published single layer and three layer PSC models.

Chapter 4. A thorough explanation of the numerical methods used when executing the drift-diffusion model simulation. This chapter defines the functions and matrices that are used in the solution procedure and analysis of the Python code.

Chapter 5. Validation of the Python code by a comparitive analysis of published material. Simulations are used to replicate published test conditions and to assess the performance of the algorithm. The accuracy and reliability of the simulator is investigated.

Chapter 6. Simulation results are illustrated to identify characteristic trends of PSCs. Trends involving recombination mechanisms, device architecture, and current-voltage hysteresis are discussed.

Chapter 7. Concluding remarks regarding future research, other applications, and the outcomes of this thesis.

Chapter 2

Background

2.1 PSC Research History

Since the discovery their application to solar cells, perovskites have achieved expeditious advancements in device performance and stability. From their first publication in 2009, PSCs have attained an exceptional endorsed efficiency of 25.5% in single junction cells [26, 25]. Improvements to device architectures, chemical composition, and manufacturing techniques have largely driven their progress [7, 8, 9, 10, 11].

Perovskites became a material of interest due to a combination of desirable physical properties. Their high absorption coefficients coupled with the material having a direct band gap allow them to be manufactured as nanometer scale thin films [28, 29]. Secondly, the perovskite crystal structure can be easily synthesized using elementary chemical compounds [14]. This capability allows for a tunable band gap as well as a cost-effective manufacturing method for future scale-up. These attributes have given way to a wide variety of device architectures and different perovskite compositions in an effort to combat the challenges outlined in Section 1.2.

Due to the rapid increase in popularity and research potential, PSCs have gained the attention of the modelling community. Modelling has shown to be effective in the explanation of dominant recombination mechanisms, prediction of device efficiencies, and identification of mechanisms that influence device stability [1, 30, 31]. Drift-diffusion modelling in particular has provided a mathematical foundation to current-voltage hysteresis as a result of ion migration [1, 3]. Although much progress has been made by developing increasingly sophisticated models, there are still many questions driving this fluorishing research topic.

2.2 Device Architecture

Various solar cell architectures have been proposed to combat the challenges listed in Chapter 1. Single junction PSCs are typically constructed using conventional n-i-p or inverted p-i-n planar architectures. The efficiency of each type of junction is dependent on the ability to form ohmic contacts with electrodes as well as the optical properties of each layer. This study focuses on the simulation of a n-i-p junction which is illustrated in Figure 2.1. The figure is an important point of reference for later chapters because it includes the coordinate system used in the drift-diffusion model.

Tandem architectures are a second type of PSC that were introduced to maximize absorption potential [27]. They have the ability to surpass Shockley-Queisser limit by introducing an additional junction with a different absorber material. The highest certified efficiency of 29.15% was recorded by a multi-junction perovskite/silicon cell [25, 27].

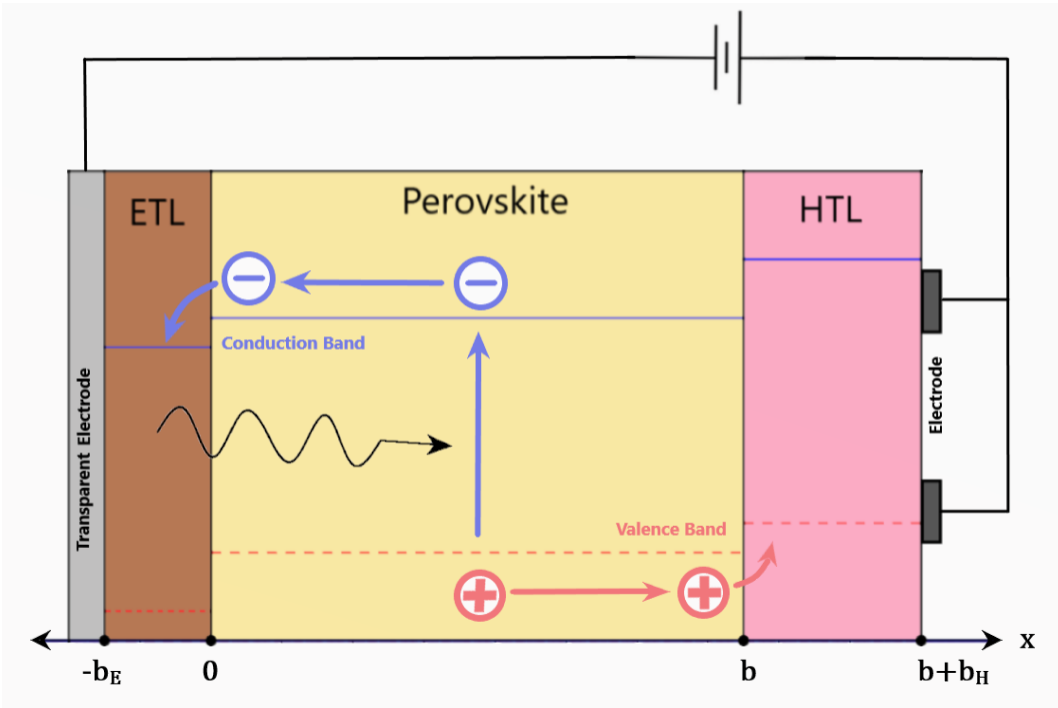

Figure 2.1: An image of a n-i-p junction sandwiched between two electrodes. Incident light passes through the transparent electrode where it excites electrons in the photosensitive material. The x-axis is defined where the origin meets the ETL/perovskite interface. The thicknesses of each layer are given by the parameters: b_E (ETL), b (perovskite), and b_H (HTL).

2.3 Operating Principles

The physics displayed in Figure 2.1 illustrate the photoelectric effect in a semiconductor. When illuminated, a photon enters the cell through the transparent electrode where it excites an electron to the conduction band. The vacancy produced by this excitation is characterized by an electron hole. Highly doped transport layers encapsulate the absorber layer to influence the flow of electrons in one direction. The electron transport layer (ETL) is an n-type semiconductor, and conversely, the hole transport layer (HTL) is a p-type semiconductor. These operate under the principle of high-level injection, where the intrinsic layer is administered excess charge from the n-type and p-type regions. Lastly, the cell is subject to an applied voltage which is used for the experimental conditions given in a current-voltage scan.

Charge carriers within semiconductors flow with respect to an electric field (drift) and a concentration gradient (diffusion). In metal halide perovskites, ions also follow these same principles. However, they move throughout interstitial spaces in the crystal lattice rather than between molecular orbitals. Migrating ions tend to accumulate near the boundaries of the perovskite layer and can produce a screening effect on an external electric field. Both ions and electrons require a certain activation energy to intitiate any current-density transients. In addition, electrons also possess the ability to be activated as a photocurrent in the absorber layer.

Lastly, the performance of a PSC is highly dependent on grain size, surface contacts, and defect density [5]. These properties create trap-states or potential wells that decrease charge carrier lifetimes. Poor layer contacts can result in the reduction of carrier extraction which can hinder the efficiency of the cell. Additionally, high defect densities can promote device degradation and higher reactivity.

2.4 Advances in Numerical Methods

Current methods for modelling charge carrier transport rely on the use of continuum mechanics to describe the evolution of carrier densities across layered materials. Variations of the drift-diffusion semiconductor equations are used to capture dynamics at mesoscopic, and more recently, nanoscopic scales [1, 3]. The equations are often paired with optical models to simulate generation rates, and recombination models to capture device defects and charge carrier annhilation. Ion migration dynamics are introduced to the models by incorporating additional transport equations.

Drift-diffusion models are most commonly solved using the Scharfetter-Gummel discretization scheme [32, 33, 34]. This scheme is aimed at decoupling the Poisson's and continuity equations by assuming the electric field is constant between iterations. Gobulev

et al. applied this method at steady state to PSC architectures with an ultrathin fullerene layer to provide a mechanism for the reduction of current-voltage hysteresis and increased device performance [32]. However, their model does not capture ion dynamics and can only be suitably applied to solar cells without current-voltage hysteresis.

Due to significant differences in the timescales that ion migration and electron transport occur, the system of equations cannot be solved at a steady-state. This renders the model a system of coupled partial differential equations (PDEs) that require extensive numerical algorithms. In addition to the challenging endeaver to solve coupled PDEs, the application of this model to PSCs impose an additional hurdle due to extreme numerical stiffness from discrepencies between charge carrier mobilities, large differences between the applied voltage and built-in voltage, and small Debye layers [1, 3, 4]. Various methods have been developed to provide the resolution needed to allow for solution convergence. To account for small Debye layers, staggered and non-uniform grids are used to increase the density of nodes near the layer interfaces. Secondly, solvers with adaptive time-steps are used to account for slow ion migration, scan rates, and differences in applied voltage. These constraints drastically limit the amount of applicable ready-made PDE solvers.

Methods have been proposed using software packages such as Quokka3 and COMSOL [35, 36]. However, these numerical methods were not thoroughly investigated in this research due to their lack of public availability. Additionally, a third-party review of these methods expressed limitations to single-layer models or the inability to replicate experimental results [1]. Two open source drift-diffusion solvers, namely, DriftFusion by Calado et al. and IonMonger by Courtier et al. were thoroughly investigated. Both solvers are based in Matlab and use different numerical methods in their approach. DriftFusion uses Matlab's PDEPE solver to consistently solve the coupled governing equations [2]. PDEPE uses a finite element approach to solve parabolic PDEs and notably requires an optimized spatial mesh for consistent convergence of solutions. The solver requires the user to format the governing equations into differential and source terms rather than the user manually discretizing the equations themselves. This in turn produces a more simplified algorithm. However, it was found that the solver does not perform well in realistic parameter spaces due to large condition numbers [3].

The results produced by Ionmonger can capture realistic charge carrier dynamics in PSCs. This method uses a finite element discretization in the spatial dimension and evolves the system forward in time using Matlab's ode15s integrator. A variety of numerical methods, including the use of a nonlinear spatial grid and an adaptive time-step allow for the solver to handle the stiff system of differential algebraic equations. Lastly, a collection of these numerical schemes are used as the basis for the Python implementation of the drift-diffusion model.

2.5 Python for Scientific Computing

Python is an object oriented programming language that has been used as a platform for small scale applications, machine learning, data science, and large scale networks like Wikipedia and Google [37]. Due to the readability of the coding language, expansive libraries, and rich development community, Python serves as an optimal starting point for novice programmers to gain coding experience. For applications related to science and engineering, the third-party Scipy and Numpy libraries offer toolkits that can approach solving systems of differential equations commonly encountered by researchers [44].

The development of the Numpy library served as a way to expand Python's computational power to handle advanced linear algebra operations and data manipulations. Recently, Python 3 added the '@' operator that simplifies matrix multiplication with Numpy arrays. This allows the user to easily distinguish between element wise matrix multiplication and standard matrix multiplication. Functions like `np.concatenate` and `np.linspace` offer simple methods to construct complex matrices. Additionally, Numpy's multidimensional arrays are the primary data structures used in the Scipy library.

Scipy is a Python library that includes a collection of optimization solvers, differential equation solvers, interpolation functions, and sparse matrices. However, it does not include packages that can handle systems of differential algebraic equations (DAEs). Installation of packages that wrap established solvers from different languages, for example the Assimulo package which wraps the popular Sundials solvers that are written in C, can be used as a workaround for this issue. Although, it was found that these solvers could not accurately calculate solutions for large systems of DAEs without an analytically derived Jacobian matrix. Additionally, these solvers cannot be easily modified without the user having a significant background in the source code.

Chapter 3

The Drift-Diffusion Model

The drift-diffusion equations are part of a classical macroscopic model used to describe the evolution of charge carrier densities in semiconductor devices. The equations are derived using the Boltzmann transport equations from statistical mechanics. Modifications to the model are made to include generation and recombination rates typically found in optoelectronic devices.

Drift-diffusion models are applicable when a semiconductor is assumed to be held at a quasi-steady state. This assumption implies the following conditions; there are no transient effects regarding ionic and carrier mobilities, isotropic temperature distribution throughout the device, and constant carrier energy. The model also requires the semiconductor device feature length to be much larger than the mean free path of the carriers.

The choice to simplify the model to one dimension was made to verify the single layer Python algorithm with the single layer model given by Courtier et al. The one dimensional model is valid under the assumption that two of the three system boundaries are well insulated in order for the flow of carriers to be dominant in one direction. However, it is recognized that the morphology of the active and transport layers likely plays a role in the charge carrier dynamics of PSCs.

This chapter introduces two models that simulate the n-i-p device architectures. First, a single layer is considered under the assumption that there are no carrier dynamics in the transport layers. Furthermore, the single layer model is expanded to a three layer model to incorporate the dominant carrier dynamics in the ETL and HTL. Lastly, the concluding sections contain a collection of generation rate and recombination models.

3.1 Single Layer Model

The single layer model implemented in the first rendition of the Python code was formulated in 2018 by Courtier et al. The model assumes that an equipotential exists in the

transport layers and the applied voltage boundary condition is satisfied at both p-type and n-type junctions.

The transient carrier continuity equations in one dimension are given by

$$\begin{aligned} \frac{\partial n}{\partial t} - \frac{1}{q}\frac{\partial j_n}{\partial x} &= G(x) - R(n,p), \\ \frac{\partial p}{\partial t} + \frac{1}{q}\frac{\partial j_p}{\partial x} &= G(x) - R(n,p), \end{aligned} \tag{3.1}$$

where n is the electron density; p is the hole density; j_n is the electron current density; j_p is the hole current density; $G(x)$ is the photo-generation rate; and $R(n,p)$ is the bulk recombination rate.

The current density equations take into account the two forces driving the flow of carrier concentrations, the Lorentz force (neglecting magnetic induction) and diffusion force. Both equations are written as

$$\begin{aligned} j_n &= qD_n\left(\frac{\partial n}{\partial x} - \frac{n}{V_T}\frac{\partial \phi}{\partial x}\right), \\ j_p &= -qD_p\left(\frac{\partial p}{\partial x} + \frac{p}{V_T}\frac{\partial \phi}{\partial x}\right), \end{aligned} \tag{3.2}$$

where V_T is the thermal voltage; D_n and D_p are the diffusivity coefficients; and ϕ is the electric potential. The electric field, E, is given by the gradient of the electric potential,

$$E = -\frac{\partial \phi}{\partial x} \tag{3.3}$$

To account for ion migration, an additional continuity equation is introduced to model the flow of positive vacancies. The terms are written in the following equation,

$$\frac{\partial P}{\partial t} + \frac{\partial F_P}{\partial x} = 0, \tag{3.4}$$

where P is the positive ion vacancy density; and F_P is the ion vacancy flux. Similar to (3.2), the flux term is given as

$$F_P = -D_+\left(\frac{\partial P}{\partial x} + \frac{P}{V_T}\frac{\partial \phi}{\partial x}\right), \tag{3.5}$$

where D_+ is the ion vacancy diffusivity coefficient.

To calculate the electic potential, E, the continuity equations are coupled with the

Poisson's equation by the following expression

$$\frac{\partial^2 \phi}{\partial x^2} = \frac{q}{\epsilon}\left(n - p - P\right), \tag{3.6}$$

where ϵ is the permittivity of free space.

3.1.1 Boundary Conditions

Due to the assumption that there is no transient behavior in the electron and hole transport layers, the interfacial carrier concentrations are given by the following Dirichlet boundary conditions

$$\begin{aligned} n_0|_{x=0} &= d_E \left(\frac{g_c}{g_{cE}}\right) e^{\frac{E_{cE}-E_c}{V_T}}, \\ p_0|_{x=b} &= d_H \left(\frac{g_v}{g_{vH}}\right) e^{\frac{E_v-E_{vH}}{V_T}}, \end{aligned} \tag{3.7}$$

where d_E and d_H are the effective doping densities for electrons and holes; (g_c/g_{cE}) and (g_v/g_{vH}) are the ratios between conduction and valence band density of states for the perovskite/transport layer interfaces; E_c and E_{cE} are the conduction band energy mininums for perovskite and the ETL; E_v and E_{vH} are the valence band energy minimums for perovskite and the HTL. Both relationships establish thermal equilibrium concentrations of their respective carrier with regard to the differences in band energy at the transport layer interfaces.

The second set of Dirichlet boundary conditions involve specifying the applied voltage. The potential drop across the cell is given by the difference between the built-in voltage and the applied voltage, $\phi_{bi} - \phi_{applied}$. The built-in voltage is calculated from the difference in band energies between the transport layers, $E_{fE} - E_{fH}$. The potential difference boundary conditions are given by

$$\begin{aligned} \phi|_{x=0} &= \frac{\phi_{applied}(t) - \phi_{bi}}{2}, \\ \phi|_{x=b} &= -\frac{\phi_{applied}(t) - \phi_{bi}}{2}. \end{aligned} \tag{3.8}$$

Perovskite layer ion migration does not leach into the neighboring transport layers which makes the system conserved regarding ion vacancies. The following Neumann boundary conditions assume zero ionic flux at both transport layer interfaces,

$$
\begin{aligned}
F_P|_{x=0} &= 0, \\
F_P|_{x=b} &= 0.
\end{aligned}
\tag{3.9}
$$

A second set of Neumann boundary conditions is needed to specify the carrier current densities at the transport layer interfaces. Due to the single layer model not accounting for ETL/HTL carrier dynamics and the constant conditions described by (3.7), the current density is only dependent on the interfacial recombination rates. The final boundary conditions are given by

$$
\begin{aligned}
j_n|_{x=0} &= -R_0(p), \\
j_p|_{x=b} &= -R_b(n).
\end{aligned}
\tag{3.10}
$$

3.1.2 Initial Conditions

Contrary to typical problems involving partial differential equations, the initial conditions for PSCs do not greatly affect the numerical solutions as long as the initial functions come close to satisfying the Poisson's equation. These solutions can be found using quasi-steady state approximations. To find initial distributions for electron and hole densities, it is assumed that there is no change in concentration with respect to time and that the cell is equipotential. This simplifies the continuity equations from (3.1) to

$$
\begin{aligned}
-D_n \frac{\partial^2 n}{\partial x^2} &= G(x) - R(n,p), \\
-D_p \frac{\partial^2 p}{\partial x^2} &= G(x) - R(n,p).
\end{aligned}
\tag{3.11}
$$

Solving the coupled system will yield distributions close enough to begin the solution procedure for the transient model. For the single layer model, simply using constant or linear charge carrier profiles with the concentrations from (3.7) at the perovskite layer boundaries will achieve convergence.

The final two initial conditions involve the initial ion distribution and potential function. The ion vacancy initial condition assumes a constant distribution of charge density. Lastly, due to the system being at an equipotential, the potential function is zero.

$$
\begin{aligned}
P_0(x) &= N_0, \\
\phi_0(x) &= 0.
\end{aligned}
\tag{3.12}
$$

3.1.3 Non-dimensionalization

The non-dimensionalization used for the single layer model follows the scaling given in [3]. Consequentially, this section will only cover the non-dimensionalization of the system parameters and the scaled governing equations relevant to the understanding of the numerical procedure.

Scaling the problem serves many purposes with regards to solving and analyzing systems of differential equations. Some examples include making variables and parameters of different orders comparable, the simplification of the system by reduction of parameters, increased numerical accuracy, and increased computational efficiency. These benefits are important to modelling PSCs due to the large disparities between time scales and narrow Debye layers. The Debye length is given by

$$L_d = \left(\frac{\epsilon_p kT}{q^2 N_0}\right)^{1/2}, \tag{3.13}$$

where ϵ_p is the permittivity of perovskite and N_0 is the density of ion vacancies. The non-dimensional parameters are expressed as the following,

$$\begin{aligned} \sigma &= \frac{D_+ b}{D_E L_d}, \quad \kappa_p = \frac{D_p}{D_E}, \quad \kappa_n = \frac{D_n}{D_E}, \quad \lambda = \frac{L_d}{b}, \\ \delta &= \frac{F_{ph} b}{D_E N_0}, \quad \bar{n} = \frac{n_0 D_E}{F_{ph} b}, \quad \bar{p} = \frac{p_0 D_E}{F_{ph} b}, \quad \Phi_{bi} = \frac{V_{bi}}{V_T}, \end{aligned} \tag{3.14}$$

where D_E is the ETL electron diffusion coefficient and F_{ph} is the incident photon flux.

After scaling the variables using the method from [3], the system of PDEs become

$$\frac{\partial P}{\partial t} = -\lambda \frac{\partial F_P}{\partial x}, \tag{3.15}$$

$$\frac{\partial E}{\partial x} = \frac{1}{\lambda^2}\left(P - 1 + \delta\left(p - n\right)\right), \tag{3.16}$$

$$\sigma \frac{\partial n}{\partial t} = \frac{\partial j_n}{\partial x} + G(x) - R(n,p), \tag{3.17}$$

$$\sigma \frac{\partial p}{\partial t} = -\frac{\partial j_p}{\partial x} + G(x) - R(n,p). \tag{3.18}$$

with the current density and ionic flux equations being

$$F_P = -\left(\frac{\partial P}{\partial x} - PE\right), \tag{3.19}$$

$$j_n = \kappa_n \left(\frac{\partial n}{\partial x} + nE\right), \tag{3.20}$$

$$j_p = -\kappa_p \left(\frac{\partial p}{\partial x} + pE\right). \tag{3.21}$$

3.2 Expanding to the Three Layer Model

Expansion to a three layer model incorporates dynamics from the electron and hole transport layers, negating the equipotential assumption given the single layer model. The three layer model used in the simulation can be found in [1]. This section will cover the new governing equations, boundary conditions, initial conditions, and rescaling of the model.

3.2.1 ETL and HTL Dynamics

Electron and hole transport layers are characterized by equations related to their respective majority carrier due to the materials being highly doped. The continuity equations resemble the left hand side of (3.1), lacking generation and recombination terms,

$$\begin{aligned} \frac{\partial n_E}{\partial t} - \frac{1}{q}\frac{\partial j_{n_E}}{\partial x} &= 0, \\ \frac{\partial p_H}{\partial t} + \frac{1}{q}\frac{\partial j_{p_H}}{\partial x} &= 0, \end{aligned} \tag{3.22}$$

with the corresponding current equations being

$$\begin{aligned} j_{n_E} &= qD_E \left(\frac{\partial n_E}{\partial x_E} - \frac{n_E}{V_T}\frac{\partial \phi_E}{\partial x_E}\right), \\ j_{p_H} &= -qD_H \left(\frac{\partial p_H}{\partial x_H} + \frac{p_H}{V_T}\frac{\partial \phi_H}{\partial x_H}\right). \end{aligned} \tag{3.23}$$

D_E and D_H are the electron and hole diffusivities for the ETL and HTL.

The electric fields for each layer are found using the Poisson's equation. Similar to (3.6), the Poisson's equations are

$$\begin{aligned}\frac{\partial^2 \phi_E}{\partial x^2} &= \frac{q}{\epsilon_E}\left(n_E - d_E\right), \\ \frac{\partial^2 \phi_H}{\partial x^2} &= \frac{q}{\epsilon_H}\left(d_H - p_H\right),\end{aligned} \tag{3.24}$$

where d_E and d_H are the effective doping densities given in (3.7); ϵ_E and ϵ_H are the permittivity of free space for the ETL and HTL materials.

3.2.2 Boundary Conditions

The three layer model includes a larger set of boundary conditions specifying the two transport/perovskite layer interfaces as well as the two interfaces with the electrodes. For the perovskite interfaces, equations (3.9) and (3.10) are applied for conservation of ions and interfacial recombination. The continuity conditions for carrier concentrations are given by proportionality constants,

$$\begin{aligned}k_E n|_{x=0^-} &= n|_{x=0^+}, \\ p|_{x=b^-} &= k_H p|_{x=b^+},\end{aligned} \tag{3.25}$$

where the constants, k_E and k_H, are produced from Boltzmann statistics using the following relationships,

$$\begin{aligned}k_E &= \left(\frac{g_c}{g_{cE}}\right) e^{\frac{E_{cE}-E_c}{V_T}}, \\ k_H &= \left(\frac{g_v}{g_{vH}}\right) e^{\frac{E_v-E_{vH}}{V_T}}.\end{aligned} \tag{3.26}$$

The electrostatic potential and potential gradient are continuous at the perovskite boundaries, given by

$$\begin{aligned}\phi|_{x=0^-} &= \phi|_{x=0^+}, \\ \phi|_{x=b^-} &= \phi|_{x=b^+}, \\ \epsilon_E \frac{\partial \phi}{\partial x}|_{x=0^-} &= \epsilon_p \frac{\partial \phi}{\partial x}|_{x=0^+}, \\ \epsilon_p \frac{\partial \phi}{\partial x}|_{x=b^-} &= \epsilon_H \frac{\partial \phi}{\partial x}|_{x=b^+}.\end{aligned} \tag{3.27}$$

It is assumed that the electrode interfaces are Ohmic contacts. Therefore, the potential

boundary conditions from (3.8) are applied at $x = -b_E$ and $x = b + b_H$. Lastly, the following Dirichlet boundary conditions are given for the carrier densities at the interfaces,

$$\begin{aligned} n_E|_{x=-b_E} &= d_E, \\ p_H|_{x=b+b_H} &= d_H. \end{aligned} \tag{3.28}$$

3.2.3 Initial Conditions

To provide valid initial conditions to the three layer system, steady state solutions are calculated for electrons and holes in the intrinsic layer using (3.1). The transport layer and ion vacancy distributions are at constant charge densities equal to their respective density of states. Due to the quasi-steady state assumption, the system is assumed to be at an equipotential following the same conditions from (3.12).

3.2.4 Rescaling the Problem

A selection of parameters are rescaled in order to proportionalize them to the concentrations in the ETL and HTL. The three layer model rescales the following parameters as

$$\begin{aligned} \kappa_p &= \frac{d_H k_H}{b^2 G_0} D_p, \quad \kappa_n = \frac{d_E k_E}{b^2 G_0} D_n, \\ \delta &= \frac{d_E k_E}{N_0}, \qquad\quad \sigma = \frac{d_E}{G_0 \tau_{ion}}, \end{aligned} \tag{3.29}$$

where τ_{ion} is the ionic timescale. This expression is given by

$$\tau_{ion} = \frac{b}{D_I} \sqrt{\frac{V_T \epsilon_p}{q N_0}}. \tag{3.30}$$

3.3 Generation and Recombination Rates

Capturing realistic generation and recombination mechanisms in the modelling of PSCs plays an important role when assessing device performance. These rates greatly impact the short-circuit current and open-circuit voltage, and ultimately the efficiency of the device. Various models can be applied to make design predictions and identify dominant recombination mechanisms. Recombination models are often combined to form a total recombination rate, $R(n, p)$.

The generation-recombination terms bring further numerical challenges to the drift-diffusion model through the addition of non-linear terms. The optical models used in this

simulation provide generation rates, $G(x)$, that are not dependent on carrier concentrations. However, the recombination models are dependent, requiring the solver to calculate the rates concurrently.

3.3.1 Beer-Lambert Law

The Beer-Lambert model offers a simplified approach to calculating the generation profile by assuming that the absorption coefficient, α, does not vary throughout the active layer. Additionally, this specific version of the law assumes that both the photon flux, F_{ph}, and absorption coefficeint are independent of the wavelength of light. The expression is given as an exponential function,

$$G(x) = F_{ph}\alpha e^{-\alpha x}. \tag{3.31}$$

Accurate results are found in systems with negligible light scattering and uniform absorber concentrations.

3.3.2 Transfer Matrix Optical Model

A second way to calculate the generation profile is to use a transfer matrix method. This method is derived from Maxwell's equations which illustrate the propagation of electromagnetic waves through thin films. Systems with more than two layers become increasingly complex due to internal reflections, transmission, magnetic fields, interference patterns, and absorbance. Transfer matrices offer an efficient solution to this problem by reducing the system to a single system matrix. Because the electric field is continuous at the layer interfaces, an energy balance can be applied to each boundary resulting in a combination of partial scattering matrices. A full derivation of the transfer matrices can be found in [38, 39].

The primary output of the transfer matrix algorithm is the total electric field for each layer, $E_j(x)$. These functions are then applied to an energy dissipation function which relates the absorptivity and refractive index of each material, given by

$$Q_j(x) = \frac{1}{2}c\epsilon_0\alpha_j\eta_j|E_j(x)|^2, \tag{3.32}$$

where c is the speed of light; η_j is the refractive index for material, j; and ϵ_0 is the permittivity of free space. The absorptivity is calculated using complex dielectric functions given from experimental data using the following relationship,

$$\alpha_j = \frac{4\pi\kappa_j}{\lambda}, \tag{3.33}$$

where κ_j is the complex refractive index term for each layer and λ is the wavelength. The generation rate can then be calculated by

$$G(x)_j = \frac{\lambda}{hc} Q_j(x). \tag{3.34}$$

To calculate the total generation rate, profiles for each absorbing wavelength of the photosensitive material are summed to produce a single function.

3.3.3 Shockley-Read-Hall Recombination

A general Shockley-Read-Hall (SRH) recombination model is introduced to capture the trap-assisted dynamics described in section 1.1.2,

$$R_{SRH}(n,p) = \frac{np - n_i^2}{\tau_p n + \tau_n p + k_3}. \tag{3.35}$$

The model takes into account both carrier lifetimes, τ_n and τ_p, which are dependent on the volumetric trap density of perovskite. The constant k_3 also relates to the carrier lifetimes and trap energies. Lastly, n_i is the intrinsic carrier density.

3.3.4 Monomolecular and Bimolecular Recombination

The SRH law can be simplified in cases where recombination is dominated by the hole concentration. Assuming that τ_p is much greater than τ_n, equation (3.35) can be reduced to

$$R_m(p) = \left(\frac{1}{\tau_p}\right) p. \tag{3.36}$$

This simplification reduces the nonlinearities presented by the full SRH equation and may be applicable to certain devices.

Bimolecular recombination is dependent on both carrier concentrations. This mechanism differs from monomolecular recombination because it captures both radiative and non-radiative recombination by means of direct electron/hole annihilation (radiative) or trap defects (non-radiative) which can accept both carriers. The model is given by

$$R_b(n,p) = \beta\left(np - n_i^2\right), \tag{3.37}$$

where β is the bimolecular recombination rate.

3.3.5 Surface Recombination

The surface recombination rates follow the same functions as the bimolecular and SRH recombination rates given in the bulk material. These rates remain consistent with the continuity boundary conditions for charge carriers and are dependent on carrier recombination velocities given at the interfaces. The expressions used in the code for the surface rates can be found in [1].

Chapter 4

Numerical Methods

The numerical scheme proposed in this chapter is the method of lines. This procedure functions by implementing a finite element discretization in the spatial dimension while leaving the temporal derivatives continuous. As a result, the system is converted into a large system of coupled ordinary differential algebraic equations. The accuracy and convergence of this method relies on the resolution, N, of the system.

Creating a solution procedure requires a multi-step approach. In most cases the initial conditions are not trivial and require steady state calculations and preconditioning steps. The program is written in a way that allows the discretized equations to be easily formatted for their appropriate application. Conditional statements are used to specify preconditioning modes used to prepare the simulated devices for current-voltage scans.

Python's object oriented programming is taken advantage of as a way to access parameters, equations, and solution structures through the construction of class objects. One benefit of this method of programming is the improved organization and readability of the code. This creates the possibility for making the code a collaborative effort for future expansions and improvements of the model. Another convenience involves the simple access from other collaberative scripts and the command window. Each script is written for a specific requisite function and requires class objects as input arguments. The hierarchy of classes are built starting from the parameters class, leading to matrices and spatial grid classes, and finally onto equation construction and solution procedures.

A number of built-in solvers from the Scipy and Numpy libraries are used for optimization problems. Currently, Python's libraries do not provide solvers that can tackle system of differential algebraic equations. This code uses a modification to Scipy's `Solve_IVP RADAU` solver to expand the solver to be compatible with a mass matrix [42].

4.1 Discretization Scheme

The finite element method used in this program follows the scheme from [1] as it has been demonstrated to effectively deal with extreme spatial stiffness. This section does not show the finite element derivation given in the published work, but rather defines some of most critical equations and non-trivial matrices used in the code

Following discretization, the system of DAEs take the form

$$\mathbf{M}\frac{\partial \mathbf{u}}{\partial t} = f(\mathbf{u}) \tag{4.1}$$

where $\mathbf{u}$ is a time dependent column vector, $[\boldsymbol{P}\ \boldsymbol{\phi}\ \boldsymbol{n}\ \boldsymbol{p}\ \boldsymbol{\phi_E}\ \boldsymbol{n_E}\ \boldsymbol{\phi_H}\ \boldsymbol{p_H}]^{\boldsymbol{T}}$, and $\mathbf{T}$ is the transpose of the vector. The size of each of the variable column vectors are dependent on the resolution of the three layers. Upon further extrapolation, the variable column vectors can be written in the form

$$\begin{aligned}
\boldsymbol{P} &= [P_0, P_1, \ldots, P_N], \\
\boldsymbol{\phi} &= [\phi_{N+1}, \ldots, \phi_{2N+1}], \\
\boldsymbol{n} &= [n_{2N+2}, \ldots, n_{3N+2}], \\
\boldsymbol{p} &= [p_{3N+3}, \ldots, p_{4N+3}], \\
\boldsymbol{\phi_E} &= [\phi_{E_{4N+4}}, \ldots, \phi_{E_{4N+NE+3}}], \\
\boldsymbol{n_E} &= [n_{E_{4N+NE+4}}, \ldots, n_{E_{4N+2NE+3}}], \\
\boldsymbol{\phi_H} &= [\phi_{H_{4N+2NE+4}}, \ldots, \phi_{H_{4N+2NE+NH+3}}], \\
\boldsymbol{p_H} &= [p_{H_{4N+2NE+NH+4}}, \ldots, p_{H_{4N+2NE+2NH+3}}],
\end{aligned} \tag{4.2}$$

where N, N_E, and N_H are the number of nodes for each respective layer. It should be noted that the indexing used here is accustomed to Python indexing where arrays begin at zero.

The mass matrix, $\mathbf{M}$, is a $(4N + 2N_E + 2N_H + 4) \times (4N + 2N_E + 2N_H + 4)$ singular tridiagonal matrix comprised of the coefficients of the time derivatives. Most notably, the zero entries along the portion of the diagonal corresponding to the electric potential equations, format the system into a set of DAEs.

4.1.1 Vectorization

In Python, it is generally recommended to avoid dynamically creating variables using loops. Similar to the Ionmonger code, this program vectorizes the variables by creating a set of operator matrices. The differencing, interpolation, and linear operators given in [1]

take the form of three tridiagonal matrices. The averaging operator is a $N \times (N+1)$ matrix given by

$$Av = \begin{pmatrix} 1 & 1 & 0 & \dots & 0 \\ 0 & 1 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & 1 & 1 \end{pmatrix}. \tag{4.3}$$

The interpolation operator is a $(N-1) \times N$ matrix given by

$$Dx = \begin{pmatrix} (\Delta x_0)^{-1} & 0 & \dots & \dots & 0 \\ -(\Delta x_1)^{-1} & (\Delta x_1)^{-1} & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & -(\Delta x_{N-1})^{-1} & (\Delta x_{N-1})^{-1} \end{pmatrix}, \tag{4.4}$$

where $\Delta x_i = x_{i+1} - x_i$
for $i = 0, 1, \dots, N-1$.

Lastly, the linear operator is a $(N-2) \times N$ matrix given by

$$Lo = \begin{pmatrix} a_0 & b_0 & c_0 & 0 & \dots & \dots & 0 \\ 0 & a_1 & b_1 & c_1 & 0 & \dots & 0 \\ \vdots & 0 & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & 0 & a_i & b_i & c_i \end{pmatrix}, \tag{4.5}$$

where the entries of the diagonals are

$$\begin{aligned} a_i &= \frac{1}{6}\Delta x_i, \\ b_i &= \frac{1}{3}\left(\Delta x_i + \Delta x_{i+1}\right), \\ c_i &= \frac{1}{6}\Delta x_{i+1}, \\ &\text{for } i = 0, 1, \dots, N-1. \end{aligned} \tag{4.6}$$

The product of the operator matrices with the variable vectors in (4.2) output a new linear system of equations used in the construction of the right-hand side of (4.1). All operator matrices used in both the single layer and three layer models can be found in `matrices.py`.

4.1.2 Spatial Mesh

A non-uniform grid is used to account for the rapid changes in solution in the narrow Debye layers. Both grids, for the single layer and three layer models, are based on hyperbolic tangent (tanh) functions from [1]. Each layer uses their own grid that is calculated from the input number of nodes for the perovskite layer. These functions concentrate 20% of the nodes in the Debye layers.

Figure 4.1: This example plot shows the concentration of grid points along the normalized x-axis for the three layer model. The grid spacing for each layer uses hyperbolic tangent distributions at $N = 50$.

4.2 Radau Solver Method

Evolving solutions forward in time requires the use of an adaptive time stepping solver to handle stiff numerical problems. A Radau 5th order subroutine was chosen due to its compatibility with systems of the form given in (4.1), dense output, adaptive step size control, and accuracy from previous applications to drift-diffusion models. However, the available Radau 5th order method in `scipy.integrate` was only compatible with explicit systems ($\mathbf{M} = \mathbf{I}$). An experimental adaptation was applied to the solver as a way to expand its capabilities to implicit problems.

Integration is accomplished by Newton iterations which require the evaluation of a Jacobian matrix for the right-hand side of (4.1). For small systems, the Jacobian is typically specified analytically in order to increase computational efficiency. However, the quantity of equations used in the method of lines is often greater than $O(10^3)$ which can result in a Jacobian matrix of $O(10^{16})$ or higher. Due to the impracticality of deriving an analytical expression, the matrix is approximated using finite differences. In a similar fashion to Matlab's JPattern option, a sparse matrix can be defined using Boolean values to identify which elements of the Jacobian remain zero and those that need to be approximated.

4.3 Solution Procedure

With the appropriate model equations, matrices classes, and solvers now defined, solution procedures can be constructed to simulate current-voltage scans and other experiments. A series of steps involving the calculation of generation rates, initial conditions, preconditioning regimes, and experiment simulations are given in Figure 4.2.

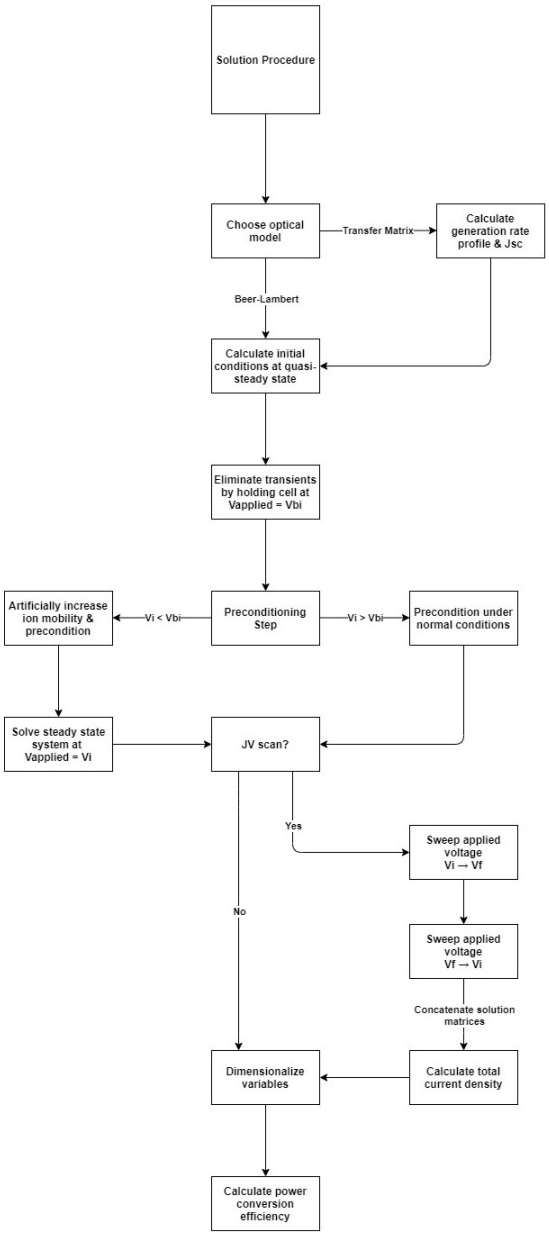

Figure 4.2: Flow chart depicting the logic used in the solution procedure.

The algorithm begins by identifying which optical model is selected. If the Beer-Lambert model is chosen, the program proceeds to calculate initial conditions at quasi-steady state using `scipy.integrate.solve_bvp`. The transfer matrix method requires additional steps because the model outputs discrete data points, rather than a continuous function which is required for the boundary value problem solver. The workaround used here applies a nonlinear regression of the transfer matrix data to a Beer-Lambert function given by the nondimensional form of (3.29). Note, the discrete transfer matrix data is still used when calculating the Radau solver solutions.

Once approximate initial conditions are found, a preconditioning step is conducted that holds the device at an applied voltage equal to the built-in voltage for an allotted amount of time. This step calculates a realistic initial condition by eliminating any unwanted transient behaviour resulting from the quasi-steady state solutions. For experiments at applied voltages other than V_{bi}, an additional preconditioning step linearly increases or decreases the applied voltage to an initial starting voltage, V_i. When the starting voltage is much less than the built-in voltage ($V_i < V_{bi}$), the ion mobility (σ) is artificially increased to an electron mobility to rapidly evolve the solution to a steady state. To address further convergence issues, the preconditioned solution is used as an initial condition for a steady-state calculation. The function `scipy.optimize.least_squares` solves the steady-state form of the drift-diffusion model.

After preconditioning has completed, the program can begin a current-voltage scan. Forward and reverse scans are automatically determined based off of the initial and final applied voltages specified in the parameters class. The program will automatically scan in both directions. Once solutions are reached, the `JV()` function will dimensionalize the variables and plot the curve. The user then has the option to use `efficiency_calculator.py` to find the theoretical efficiency and it's associated parameters. If the user specifies their own solution procedure, the `plot.py` script can be used to visualize the dimensional or non-dimensional solutions.

4.4 Methods of Analysis

A selection of scripts were written to assist in the analysis of data and determine theoretical device performance. The user has the option to choose between dimensional and non-dimensional formulations of the solutions when plotting results. Additionally, an animation function was written in order to visualize charge carrier dynamics during transient experiments. All visualizations utilize the comprehensive libraries provided by Matplotlib 3.4.1.

4.4.1 Efficiency Calculation

Efficiency calculations are made using the solution output of a current-voltage scan. The efficiency of a solar cell is characterized by

$$\eta = \frac{V_{OC} J_{SC} FF}{P_{in}}, \tag{4.7}$$

where η is the efficiency; V_{OC} is the open-circuit voltage; I_{SC} is the short-circuit current; FF is the fill factor; and P_{in} is the input power of incident light. The program calculates the efficiency based on solar cells measured under AM1.5 conditions. The open-circuit voltage and short-circuit current are simply found by seeking the curve intersections with the x- and y-axes. For J_{SC}, the J-V scan solutions are linearly interpolated and evaluated at $V = 0$. To find V_{OC} (i.e. the root of the function), an anonymous interpolation function using the output data and a dummy variable is fed to `scipy.optimize.least_squares`.

The fill factor is a measure of the squareness of the J-V curve. In other words, the largest rectangle bound between the curve and both axes. The full expression is given by the ratio

$$FF = \frac{V_{MP} J_{MP}}{V_{OC} J_{SC}}, \tag{4.8}$$

where the equation for a rectangle can be expressed as the product of the current-density at max power (J_{MP}) and the voltage at max power (V_{MP}). To calculate the numerator of (4.4), a minimization function is defined as

$$f_{min}(V_{MP}) = \frac{1}{V_{MP} \cdot f(V_{MP})}, \text{ where } 0 \leq V_{MP} \leq V_{OC}. \tag{4.9}$$

The function $f(V_{MP})$ is the linearly interpolated J-V solution curve as a function of the max power voltage. A least squares regression is subsequently applied to (4.5) to find the maximum value of V_{MP}.

4.4.2 Animation

An animation function was written to visualize the rapid changes in solution. It is particularly useful when analyzing the evolution of charge distributions and potential functions. The user can access the animation function through `animate.py`.

To capture the dynamics in real time, the frame rate needs be synchronized with the time dependent solution structure. However, the animation library provided by Matplotlib only

allows a constant frame rate while the adaptive time step from the Radau solver outputs a nonlinear time array. This is remedied by linearly interpolating the variables from (4.2) with respect to the discrete time array. A new array of evenly spaced time values is generated using the interpolated data. The amount of frames corresponds to the number of elements in the new time array. Lastly, the frame rate is determined by

$$FPS = \frac{frames}{t_f}. \tag{4.10}$$

Chapter 5

Model Validation

This chapter seeks to verify the model by comparing the results to the published works by Courtier et al. A selection of experiments were replicated in Python and Matlab using a set of test conditions found in [3]. Lastly, a discussion is presented to provide an interpretation to the findings of the study.

5.1 Comparison of Test Cases

To effectively compare test cases, the published Ionmonger code was simplified to a single layer model in Matlab [1, 43]. This code uses finite elements rather than finite differences, but closely resembles the current decay transients given in [3]. The method begins by preconditioning the device at the built-in voltage. After a steady state is reached, the applied voltage is rapidly decreased to zero using a hyperbolic tangent function given by

$$\Phi_{applied} = \Phi_{bi}\left(1 - \frac{\tanh(\beta t)}{\tanh(\beta t_{end})}\right) \text{ where } \beta = 10^6,\ t_{end} = 1. \tag{5.1}$$

The numerical solutions for the charge distributions and potential functions are directly compared in Figure 5.1.

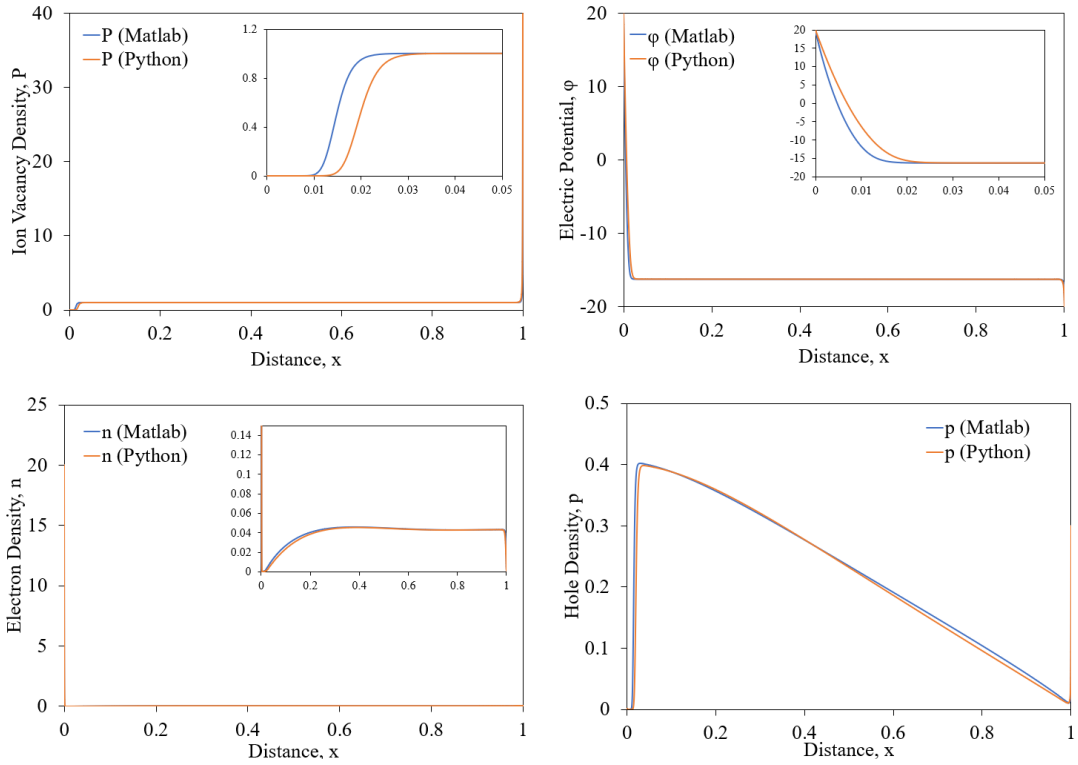

Figure 5.1: Non-dimensional potential functions and carrier distributions for the test conditions from Equation (5.1). Parameters were specified using the data from [3] and use the scaling given in Equation 3.14. The distance, x, is scaled by the perovskite layer thickness of 400 nm. All four plots compare the solutions from the single layer finite element method from the Matlab (blue) and Python (orange) codes. (Top left) ion vacancy density, (top right) electric potential, (bottom left) electron density, (bottom right) hole density.

The results displayed in Figure 5.1 show great agreement between the Python code and the finite element Matlab code, with both numerical solutions following similar trends compared to the figures presented in [3]. While these results demonstrate that the Radau solver method can handle the stiff system of DAEs, there are small differences between solutions in the Debye layers. Matlab's Ode15s solver produces solutions that concentrate charge carriers closer to the ETL/perovskite interface producing a steeper gradient in electric potential. This behaviour is likely attributed to differences in temporal accuracy between the two solvers. The error analysis presented in Figure 5.6 shows lower accuracy for solutions at the ETL/perovskite interface when compared to the error analysis in [1].

Ionmonger's solution procedure uses an iterative method to construct solution structures by calling Matlab's Ode15s solver for each iteration. Each element in a discrete time array dictates the interval at which Ode15s evolves the solution forward. This likely inceases the

temporal resolution needed to produce a highly accurate result.

5.2 Short-Time Solution Discrepancy

The current density as a function of time was investigated and compared to the results given in [3]. Both numerical solutions exhibit a fast electronic transient occuring at short times, followed by a gradual decrease in current density until the test condition reaches a steady state.

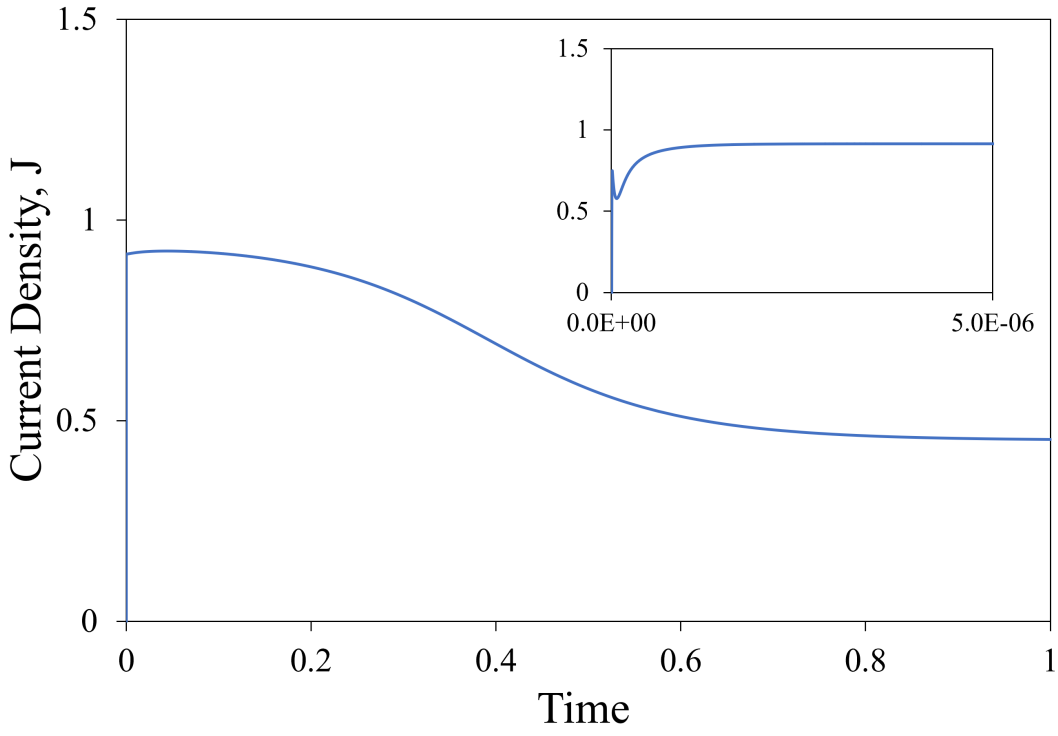

Figure 5.2: Current density as a function of time for the single layer model. The test conditions follow Equation (5.1). This experiment attempts to replicate the electronic transient given in [3]. The non-dimensional time corresponds to a current decay transient that occurs over 4.86 seconds. At the short-time scale in the top right corner, the time scale is within 24 microseconds.

The short time solution given by the Python code displays an electronic transient that occurs on a timescale an order of magnitude less (within 2 microseconds) than the Matlab code. Shortly after the solution reaches a maximum, it rapidly decays into a curve that follows the asymptotic expansion illustrated in Figure 5.3. The rapid transient that is observed is attributed to the fast electronic dynamics as the applied voltage decreases to short-circuit

conditions. These results suggest a faster and less intense current-density response when compared to the published results. The remainder of the decay curve is attributed to the slow migration of ion vacancies towards the perovskite/HTL interface. For this simulation, ions migrate on the order of $O(-17)$, while electrons and holes diffuse on an order of $O(-4)$. A comprehensive list of the system parameters and units can be viewed in Table 6.2.

A second consideration for this discrepancy is the accuracy of the time integration. It is possble that the Radau solver may have not been able to completely capture the rapid carrier and ion dynamics presented by Courtier et al. In the case that these results did not achieve the desired accuracy to capture the full range of dynamics, it does suggest that the Radau method has potential to improve by engineering the temporal resolution at short times. Currently, the Python algorithm uses a single call to the Radau solver which utilizes the dense output option to generate solutions at each time-step. In this case, the temporal tolerances are automatically determined by the `RadauDAE.py` script. However, future versions of the code could implement an iterative method that reduces the time-step for early iterations in the solution procedure as a way to improve the accuracy at short times.

When analyzing the complete solutions of the test case, it is apparent that the Radau method provides a reasonable approximation to the published solutions. This conclusion bestows confidence that the Python code outputs solutions that are physically relevant, and appropriate for simulations at nanoscopic scales.

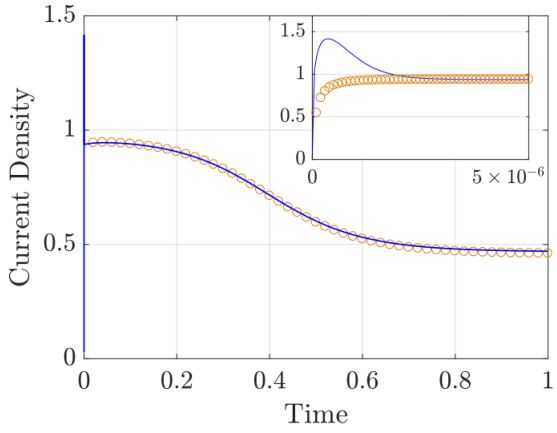

Figure 5.3: Current density as a function of time from [3]. Their simulation shows an electronic transient that differs from the asymptotic expansion. The blue line represents the numerical solution and the orange markers represent the asymptotic expansion.

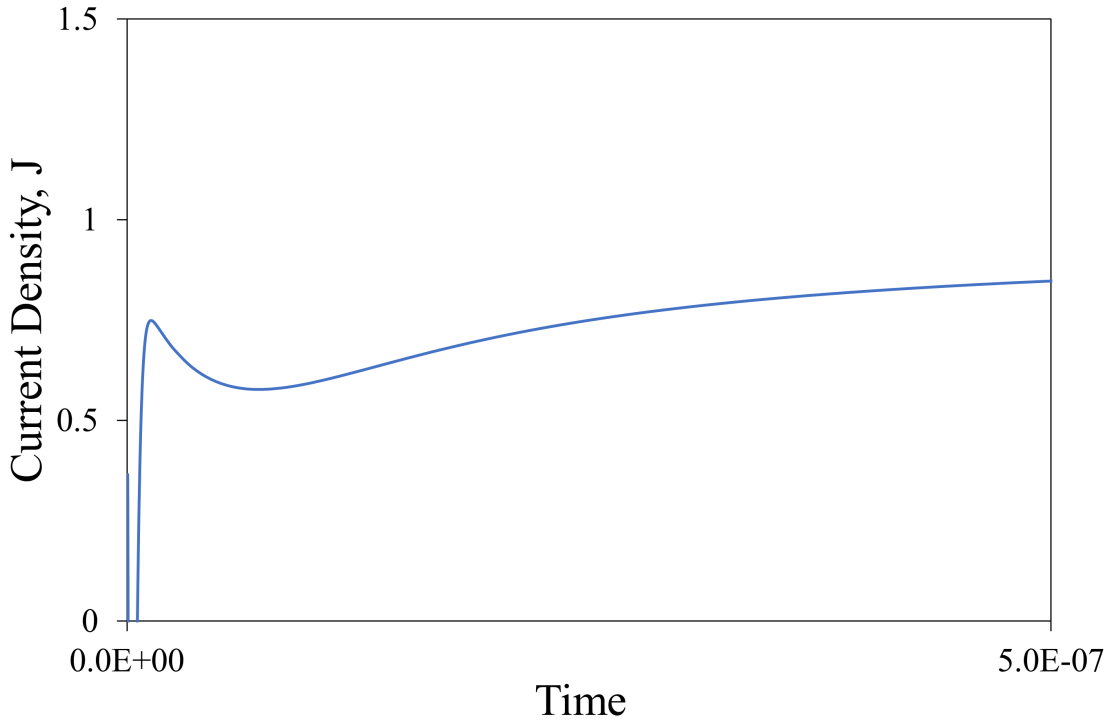

Figure 5.4: A rescaled image of Figure 5.2 where the time axis is reduced by one order of magnitude. The graph shows a rapid electronic transient that quickly follows the trend of the assymptotic expansion from Figure 5.3.

5.3 Comparison of Optical Models

Due to the reliability of the solutions presented in the previous sections of this chapter, an expansion of the drift-diffusion model was proposed to include a transfer-matrix optical model. The inclusion of the transfer-matrix method provides several advantages to modelling solar cells. Some examples involve the inclusion of refractive index data, an accounting for quantum efficiencies, and the engineering of layer thicknesses. To illustrate the differences between Beer-Lambert and transfer-matrix generation profiles, a nonlinear regression of the transfer-matrix model output was used to approximate the parameters in the Beer-Lambert model.

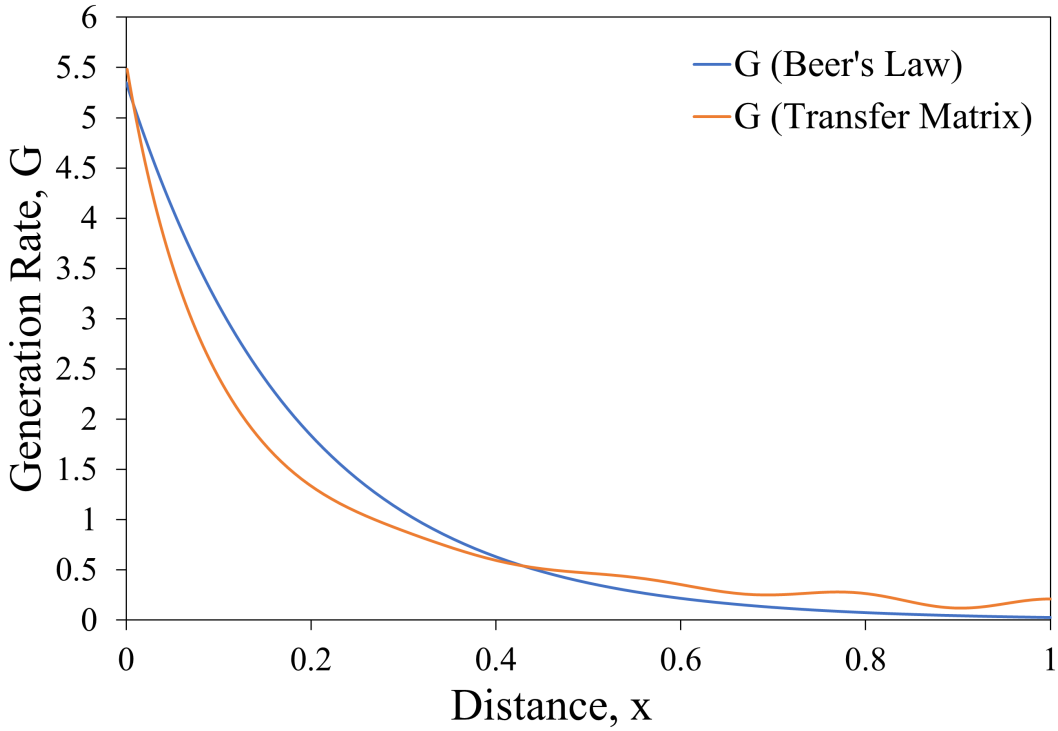

Figure 5.5: A comparison of the transfer matrix method to the Beer-Lambert model. The Beer-Lambert parameters are approximated from the transfer matrix data using the procedure described in Section 4.3. Optical data was obtained from [40].

The results displayed in Figure 5.5 show that the transfer-matrix method has a steeper generation profile and higher rates beyond the halfway point in the device. Additionally, the transfer-matrix model produces a wave-like function due to the inclusion of wave interference mechanics.

By incorporating the transfer-matrix method to the drift-diffusion model, it expected that equilibrium carrier concentrations will be more evenly distributed. This produces an effect that flattens the electric potential across the device. Due to these differences in the internal electric field and carrier concentrations, it is expected that recombination rates and device performance will also be influenced.

5.4 Assessing Simulator Performance

The performance of the Python code was determined by assessing the accuracy and speed at which a current-voltage solution procedure is carried out. To determine the accuracy of a result, a simulation was run using a highly refined mesh and compared to solutions at

lower grid spacings. The error is given by

$$\sigma = u_{Nmax} - u_N, \tag{5.2}$$

where u_{Nmax} is a dependent variable solution at a given point using the mesh at $N = 3200$, and u_N is a dependent variable solution at a mesh with lower grid spacing. The solutions of interest take place at the ETL/perovskite and perovskite/HTL interfaces using the three-layer model. Secondly, the errors are calculated at the end of the solution procedure. This is done instead of calculating time averaged errors because the Radau solver uses dense output, which generates solutions at non-replicable times due to the adaptive time-step. Lastly, the figures presented in this section are based on the total amount of nodes, N_{total}, where $N_{total} = N + NE + NH$.

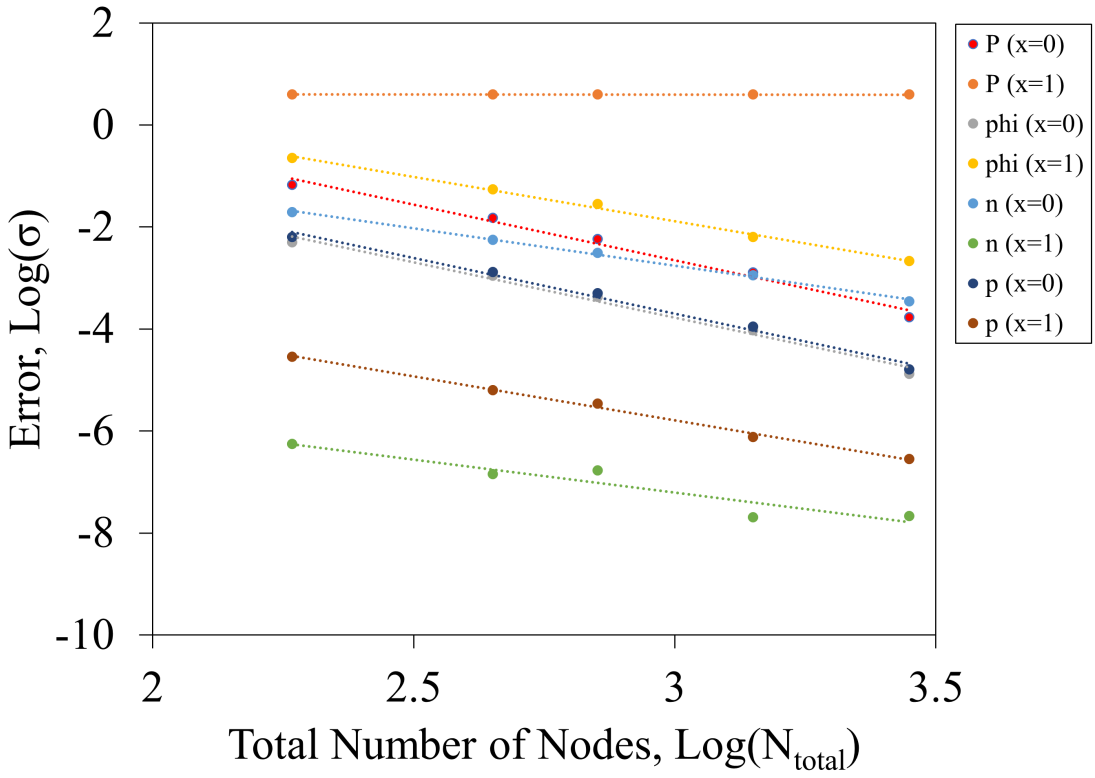

Figure 5.6: A Log_{10} plot of the errors at different mesh sizes. The errors are calculated at the PSC transport layer interfaces.

The errors produced in Figure 5.6 follow the same trends given in [1]. However, the errors are noticeably higher for all carrier concentrations and potentials. Additionally, as grid spacings increased, the accuracy of the ion vacancy density at the perovskite/HTL

interface did not change.

A speed test for this simulation was carried out by timing the solution procedure from the beginning of the preconditioning step to the end of the current-voltage scan. The simulations were carried out on a standard desktop computer using an Intel Core i5 Kaby Lake, 16 GB of RAM, and an NVIDIA GeForce GTX 1070.

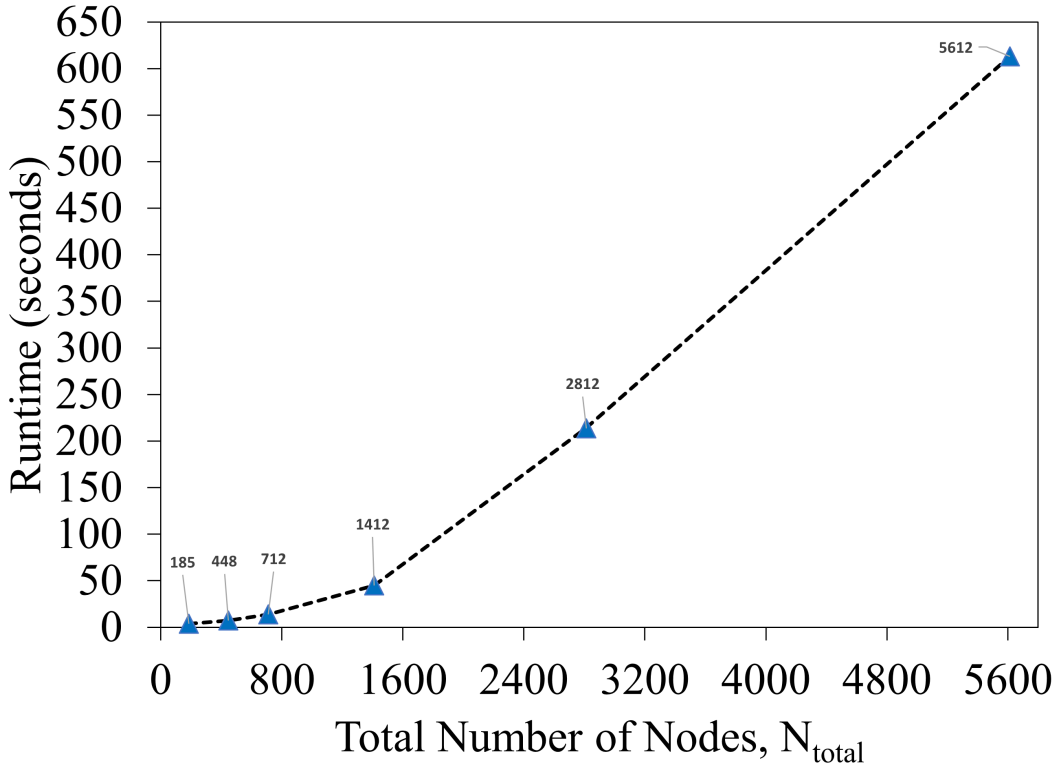

Figure 5.7: A plot of the performance time versus the total number of grid spacings.

As expected, the performance time increased as the computational mesh became more refined. When compared to the performance test in [1], the Python code operates at approximately a quarter of the speed as Ionmonger for similar grid spacings.

In conclusion, the Python code does not achieve the same performance standard as Courtier et al.'s Ionmonger. However, the accuracy of the solutions and the ability to achieve reasonable results within minutes demonstrate that this Python code can be used as a practical simulator for thin-layer devices with ion transport. Further work needs to be done to engineer the time integrator to capture fast carrier dynamics occurring during current-voltage sweeps.

Chapter 6

Simulation of PSCs

A variety of physical phenomena are characteristic of PSCs and vital to their performance. The introduction of this thesis gave insight to several engineering challenges related to the advancement of PSCs. This chapter focuses on capturing the distinctive trends of hysteresis, defect densities, surface recombination, and device architecture. The parameters used to simulate the devices are given in Tables 6.1 and 6.2. These values are referenced from [3, 1] and replicate some of their experimental conditions. All simulations were executed using the three layer model.

Table 6.1: List of parameters referenced from [1]. Table 1/2.

Symbol	Description	Value	Units
Tolerance & Resolution Parameters			
N	Intrinsic layer nodes	400	
N_E	ETL layer nodes	107	
N_H	HTL layer nodes	205	
r_{tol}	Temporal relative tolerance	1.0E-06	
a_{tol}	Temporal absolute tolerance	1.0E-10	
Scan Parameters			
V_i	Initial applied voltage	1.2	V
V_f	Final applied voltage	0	V
scan_rate	Scan rate of applied voltage	100	$mV \cdot s^{-1}$
Constants			
q	Elementary electric charge	1.60E-19	C
k_B	Boltzmann constant	8.62E-05	$eV \cdot K^{-1}$
T	Temperature	298	K
ε_0	Permittivity of free space	8.85E-12	$F \cdot m^{-1}$
F_{ph}	Incident photon flux	1.40E+21	$m^2 \cdot s^{-1}$
V_t	Thermal voltage	0.02568	eV

Table 6.2: List of parameters referenced from [1]. Table 2/2.

Symbol	Description	Value	Units
Perovskite Layer Parameters			
b	Intrinsic layer thickness	400	nm
ε_p	Perovskite permittivity	$24\varepsilon_0$	$F{\cdot}m^{-1}$
α	Perovskite absorption coefficient	1.3E+07	m^{-1}
E_c	Conduction band minimum	-3.7	eV
E_v	Valence band minimum	-5.4	eV
D_n	Perovskite electron diffusion coefficient	1.7E-04	$m^2{\cdot}s^{-1}$
D_p	Perovskite hole diffusion coefficient	1.7E-04	$m^2{\cdot}s^{-1}$
g_c	Conduction band density of states	8.1E+24	m^{-3}
g_v	Valence band density of states	5.8E+24	m^{-3}
Ion Migration Parameters			
N_0	Density of ion vacancies	1.60E+25	m^{-3}
D_I	Ion diffusion coefficient	1.01E-17	$m^2{\cdot}s^{-1}$
Electron Transport Layer Parameters			
d_E	Effective doping density	1.0E+24	m^{-3}
g_{cE}	Effective conduction band density of states	1.0E+25	m^{-3}
E_{cE}	Conduction band minimum	-4	eV
b_E	ETL layer thickness	100	nm
ε_E	ETL permittivity	$10\varepsilon_0$	$F{\cdot}m^{-1}$
D_E	Electron diffusion coefficient	1.0E-05	$m^2{\cdot}s^{-1}$
Hole Transport Layer Parameters			
d_H	Effective doping density	1.0E+24	m^{-3}
g_{vH}	Effective valence band density of states	1.0E+25	m^{-3}
E_{vH}	Valence band minimum	-5.1	eV
b_H	HTL layer thickness	2.0E+02	nm
ε_H	HTL permittivity	$3\varepsilon_E$	$F{\cdot}m^{-1}$
D_H	Hole diffusion coefficient	1.0E-06	$m^2{\cdot}s^{-1}$

6.1 Effect of Scan-Rate on Hysteresis

Current-voltage hysteresis is primarily attributed to the displacement of ions in the perovskite layer. Once subject to an applied voltage, the ion distributions shift towards the transport layer that matches the charged species, as seen in Figure 5.1. Because the ions move at a slow time-scale relative to electrons, the internal electric field is subject to a screening effect until the ions reach a steady state. Subjecting the cell to slow voltage scan rates can mitigate some hysteretic behaviour by allowing the ion migration to resolve. Con-

versely, fast scan-rates observe substantial hysteresis by not allowing the ions find stability. This trend is shown in the simulations presented in Figure 6.1.

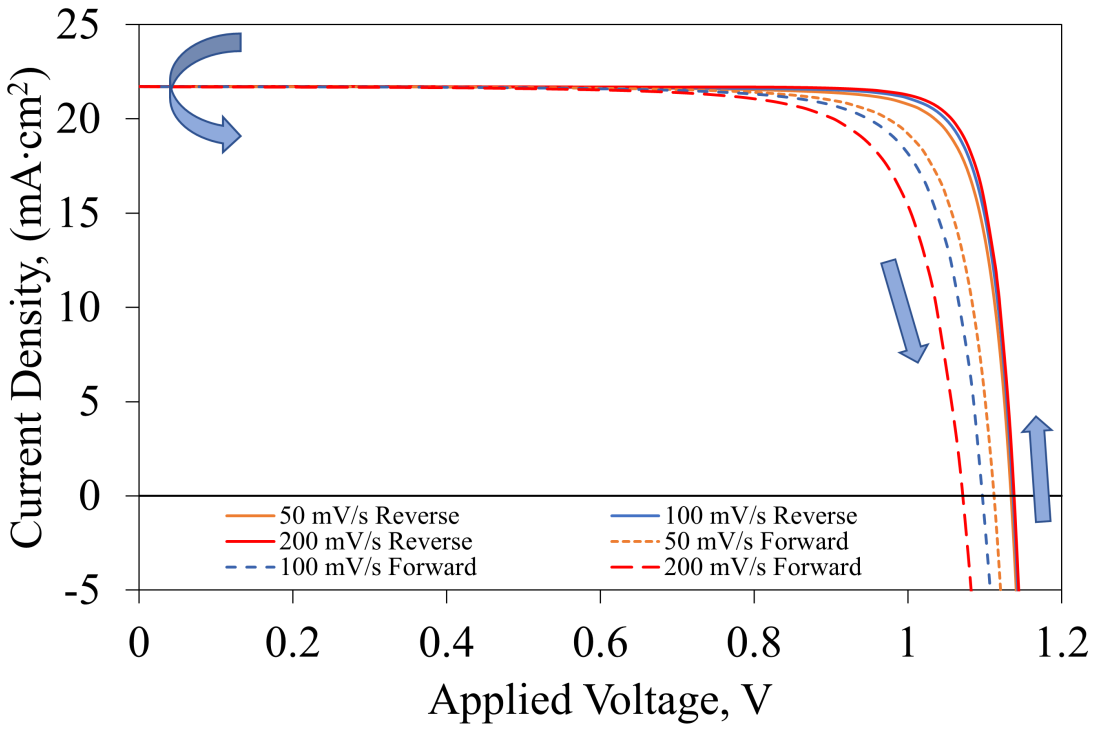

Figure 6.1: Three-layer model simulations at varying scan rates. The scan procedure begins from an applied voltage of 1.2 V and sweeps to the short-circuit current. Forward scans begin at an applied voltage of 0 V and return to the starting voltage. Reverse scans are shown as solid lines and the forward scans are dotted lines.

The discrepencies between the forward and reverse scans show a significant effect on the open-circuit voltage. As the scan-rate decreases the open-circuit voltage of the returning scan increases. These observations coincide with realistic devices, and demonstrate that the simulator can achieve a result that matches the natural world. Additionally, Figure 6.1 uses similar parameters to [1], which can be used as further evidence to validate the three layer model.

6.2 Impact of Layer Thicknesses on Device Performance

This section looks at the response of the transfer-matrix optical model with respect to layer thicknesses. Figures 6.2 and 6.3 illustrate current-voltage hysteresis curves along with generation rate profiles.

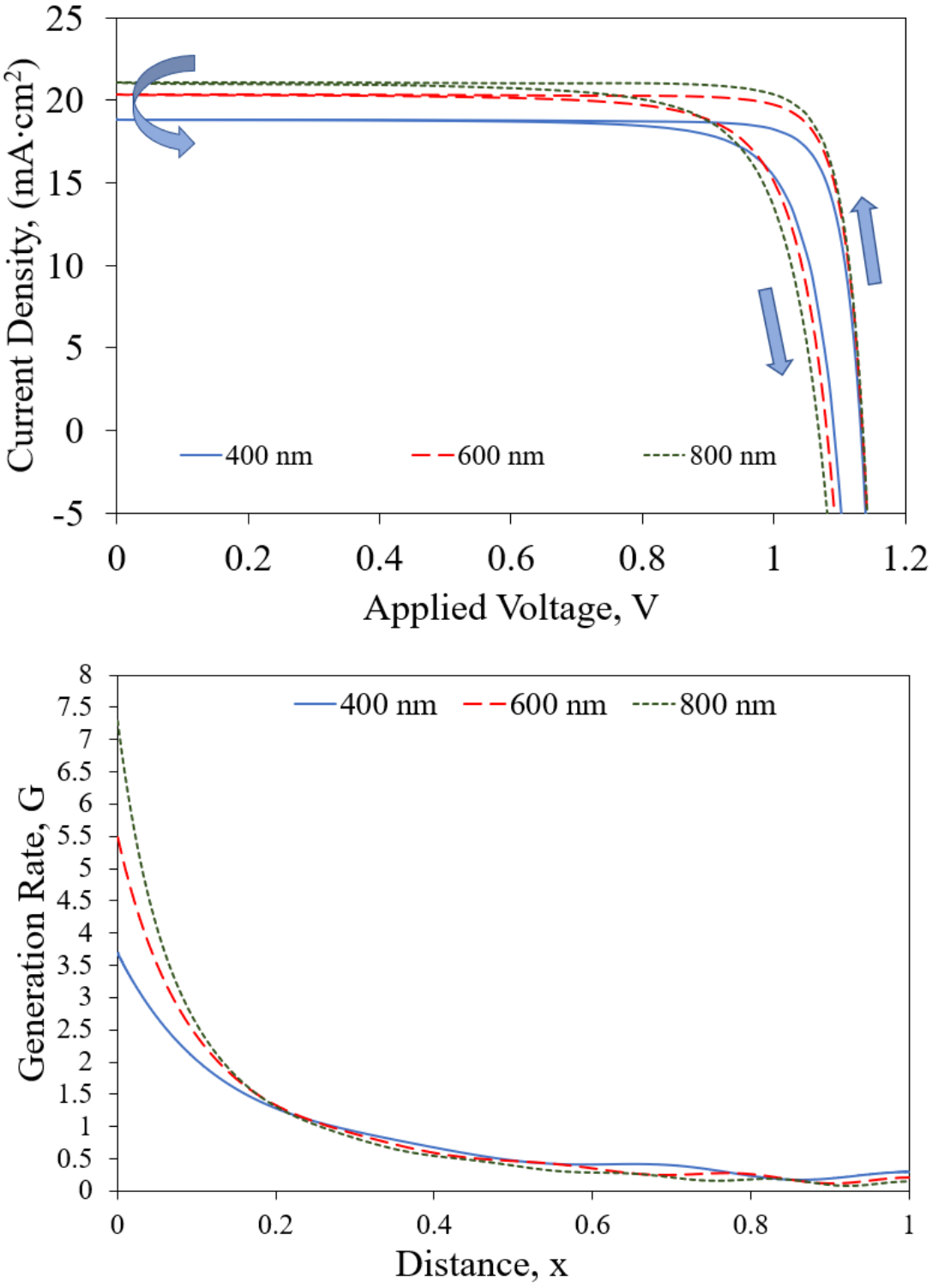

Figure 6.2: Current-voltage scans at different perovskite layer thicknesses. The remaining device parameters are given in Table 6.2. Two figures are presented showing the hysteresis curves (top) and the rescaled transfer-matrix generation rates (bottom). Optical data is from [40]

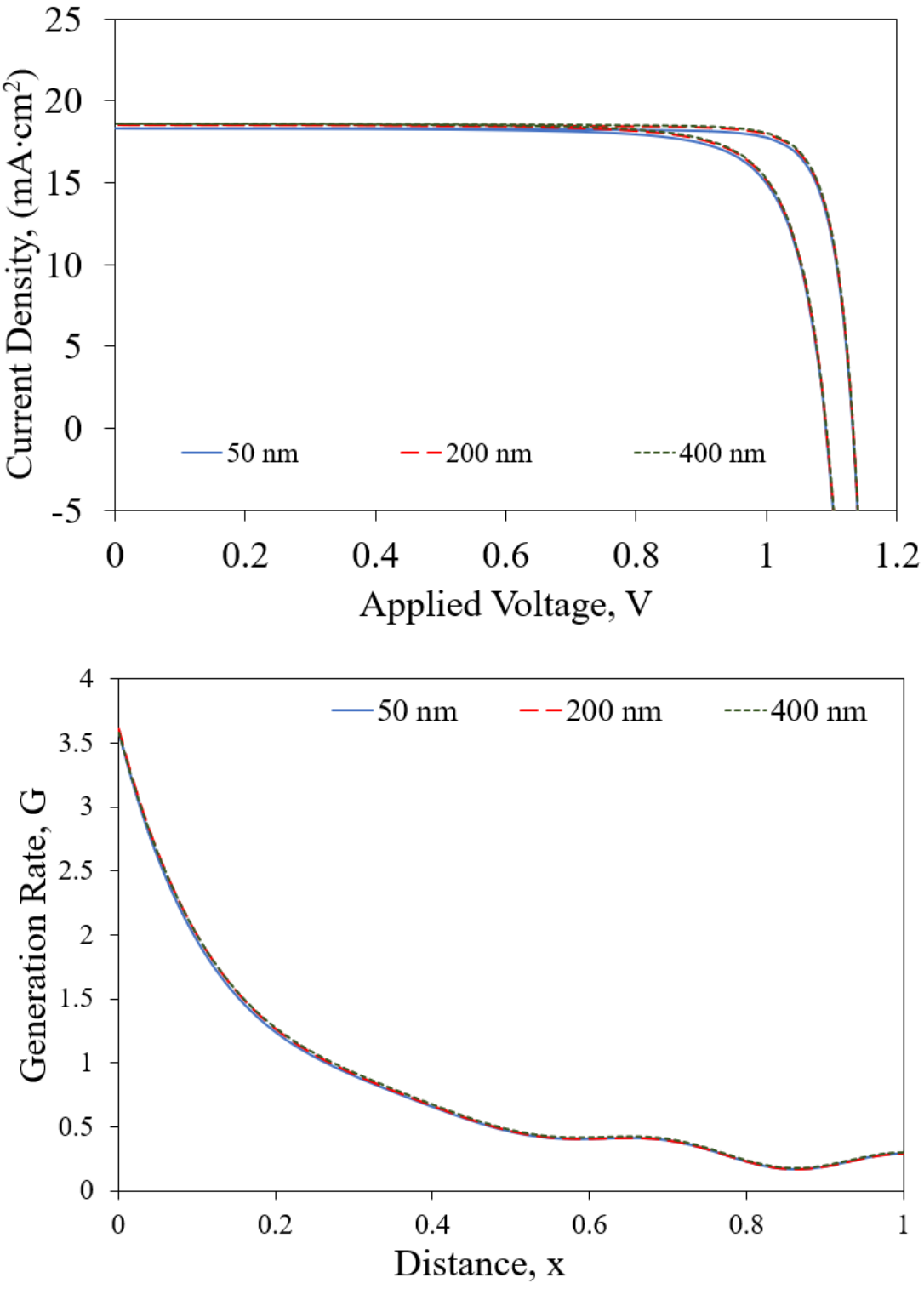

Figure 6.3: Current-voltage scans at different ETL layer thicknesses. The remaining device parameters are given in Table 6.2. Two figures are presented showing the hysteresis curves (top) and the rescaled transfer-matrix generation rates (bottom). Optical data is from [40]

As the perovskite layer increases in thickness, there is an increase in short-circuit current and a broadening of hysteresis. The transfer-matrix method predicts increased generation rates within the first 20% of the absorber layer as the thickness increases. This result provides an explanation for the improved short-circuit current. Hysteresis effects are likely attributed to an increase in ion migration distance. Lastly, as the thickness increases, the wave-like pattern produced by the transfer-matrix method loses amplitude. This suggests that the Beer-Lambert model retains higher accuracy at greater absorber layer thicknesses.

The results given by changing the ETL layer thickness did not produce significant changes in the generation rate profile. This is due to the low absorbance of the TiO_2 data used in the simulation. Additionally, at this scale this experiment produced no effects from the refractive properties of the ETL layer. However, there was a small reduction in short-circuit current in the 50 nm simulation indicating that charge carrier dynamics were effected.

In conclusion, these results demonstrate that the transfer-matrix method predicted physically relevant trends when applied to PSC drift-diffusion simulations. The methods used to implement the model also provide consistent and accurate solution convergence. Further sensitivity analysis should be applied with additional data to observe absorbance effects in the transport layers.

6.3 Recombination Parameter Trends

A set of numerical experiments were studied to investigate the defect physics described in Section 1.1.2. This was accomplished by varying the bulk and interfacial recombination parameters. The bulk recombination is described in terms of the average pseudo-lifetime of a charge carrier, τ_n or τ_p. Pseudo-lifetimes are characterized as a measure of the defects and non-radiative recombinations in the bulk of the perovskite layer. The interface parameters are given in terms of the recombination velocities, v_{nH} and v_{pE}. All the recombination parameters are implemented in the SRH model outlined in Chapter 3.

The ten experiments given in Table 6.3 were carried out using the current-voltage scan protocol. All efficiency calculations were made using `efficiency_calculator.py` and base their data off the initial reverse scan.

Table 6.3: Parameters and efficiency results for current-voltage simulations using various recombination parameters.

Experiment	τ_n (s)	τ_p (s)	v_{pE} (m/s)	v_{nH} (m/s)	V_{oc} (V)	J_{sc} (mA/cm^2)	FF	efficiency
Case 1	3.0E-09	3.0E-07	10	0.1	1.136	21.717	0.860	0.183
Case 2	3.0E-10	3.0E-08	10	0.1	1.123	21.392	0.838	0.169
Case 3	3.0E-11	3.0E-09	10	0.1	1.046	18.435	0.786	0.119
Case 4	3.0E-09	3.0E-07	10	10	1.056	21.704	0.875	0.176
Case 5	3.0E-09	3.0E-07	10	20	1.041	21.691	0.874	0.173
Case 6	3.0E-09	3.0E-07	10	30	1.029	21.678	0.876	0.171
Case 7	3.0E-07	3.0E-05	10	0.1	1.138	21.753	0.863	0.184
Case 8	3.0E-07	3.0E-05	10	10	1.056	21.740	0.877	0.176
Case 9	3.0E-11	3.0E-09	1	0.01	1.050	18.436	0.789	0.121
Case 10	3.0E-11	3.0E-09	0.1	0.001	1.049	18.436	0.790	0.121

The first three cases consider devices that vary in defect density by a factor of 10. As expected, when the carrier lifetimes decrease due to increase in trap-states, the efficiency of the device decreases. Additionally, a significant increase in hysteresis accompanies the loss of efficiency. Case 3 exhibits the largest loss of efficiency due to a reduction in short-circuit current.

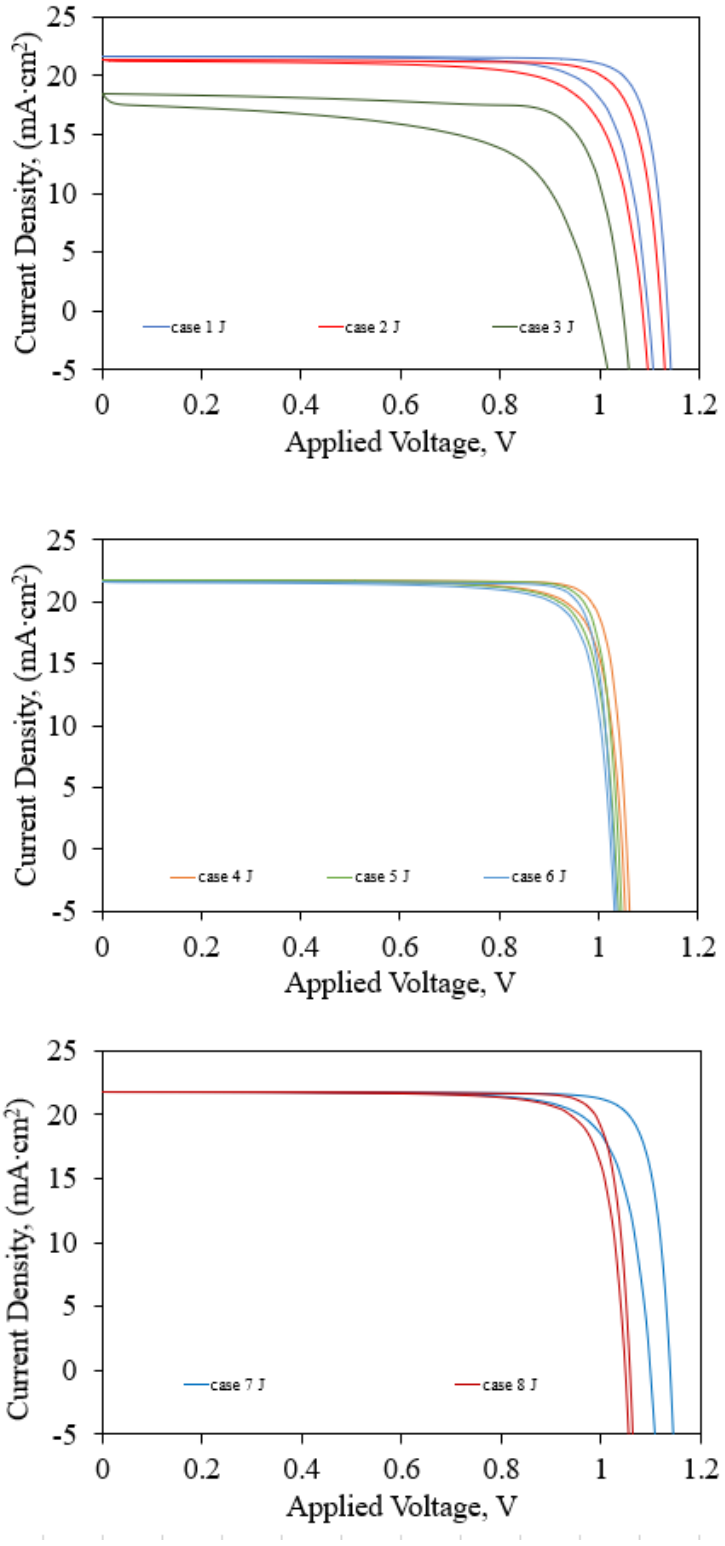

Figure 6.4: A collection of results from current-voltage simulations given by the experimental conditions in Table 6.3.

Cases 4-6 adjust the interface recombination parameters. These cases follow a similar trend to the bulk recombination parameters regarding their efficiency. However, they retained a relatively high efficiency compared to cases 1-3 and also exhibit improved device stability. These results suggest that an increase in carrier extraction at the HTL leads to a reduction in hysteresis.

The system described by case 7 has the lowest defect density and consequently the highest efficiency out of all of the test cases. However, this result produced a broad hysteresis curve. When increasing the electron recombination velocity at the HTL, given by case 8, a reduction of hysteresis is observed accompanied by an efficiency loss that makes the device perform like case 4. These results indicate that interfacial recombination impacts the efficiency and stability of the device greater than the effect of a perovskite with low defect density.

Lastly, cases 9 and 10 demonstrate poor performance with low carrier lifetimes and reduced interfacial recombination. The slightly improved efficiencies when compared to case 3 are due to the decreased recombination at the interface.

Chapter 7

Conclusions and Outlook

7.1 Other Applications

The method proposed in this thesis provides an implementation of the method of lines that could be applied to other dynamical systems. More specifically, this code provides an approach that can accurately solve large systems of stiff differential algebraic equations (DAEs). This also may be found useful for programmers who seek to improve simulation performance by making approximations to Jacobian matrices that don't have analytical solutions.

A second application can be found in machine learning. Python has been commonly used as a platform for this topic, which could be applied to fit experimental data or make predictions about device performance. Machine learning algorithms have been previously used to identify carrier recombination mechanisms found in PSCs.

7.2 Future Research

There are many directions that this research can approach for future iterations of the code. The research presented in this thesis suggest that improvements to the solver algorithm should be made to increase the accuracy of the simulation to capture fast carrier dynamics. Secondly, the model should be expanded to allow for alternate device architectures and additional layers.

Currently, the program has demonstrated the ability to simulate realistic devices. Once protocols are developed for more experimental conditions, future research should apply the model to experimental data to validate the effectiveness of the model. An implementation of a machine learning algorithm in Python could provide a method to fit real world experimental data.

This program has built a foundation for expansions to the drift-diffusion model. Future

directions for drift-diffusion modelling should include ion transport equations for minority migration ions, expansion to multiple spatial dimensions to account for morphological properties, and inclusion of degradation mechanics to predict the longevity of devices. Further studies will need to assess the accuracy this algorithm when expanding the system to larger systems of DAEs.

7.3 Concluding Statements

This study investigated topics involving charge carrier dynamics in perovskite solar cells; mechanisms that influence the device stablility and efficiency; and numerical methods for partial differential equations. A simulation was written that demonstrates that standard Python libraries can be used to solve stiff systems of differential algebraic equations while performing at reasonable computation times. After applying a comparative analysis between the Python code and the published works by Courtier et al., it was found that the Ionmonger algorithm outperformed this work in accuracy and performance time. However, the Radau solver method produced an electronic transient on short time scales that diverged from their asymptotic solutions, indicating the method has potential for future improvements. Additionally, the test cases provided reasonable accuracy for realistic device simulations on nanometer scales. Lastly, studies were performed to determine physically relevant trends when changing certain device parameters. It was found that this algorithm produced appropriate responses for realistic devices. In conclusion, this thesis demonstrates that Python can serve as a platform for future implementations of transient drift-diffusion models and that this Python code could serve as a tool for theoretical device simulations.

Bibliography

[1] Courtier, N. E.; Cave, J. M.; Walker, A. B.; Richardson, G.; Foster, J. M., IonMonger: a free and fast planar perovskite solar cell simulator with coupled ion vacancy and charge carrier dynamics. *J Comput Electron* **2019**, *18* (4), 1435-1449.

[2] Calado, P.; Gelmetti, I.; Hilton, B.; Azzouzi, M.; Nelson, J.; Barnes, P. R. F., Driftfusion: An open source code for simulating ordered semiconductor devices with mixed ionic-electronic conducting materials in one-dimension. **2020**.

[3] Courtier, N. E.; Richardson, G.; Foster, J. M., A fast and robust numerical scheme for solving models of charge carrier transport and ion vacancy motion in perovskite solar cells. **2018**.

[4] Courtier, N. E.; Foster, J. M.; O'Kane, S. E. J.; Walker, A. B.; Richardson, G., Systematic derivation of a surface polarization model for planar perovskite solar cells. **2017**.

[5] Snaith, Henry J; Abate, Antonio; Ball, James M; Eperon, Giles E; Leijtens, Tomas; Noel, Nakita K; Stranks, Samuel D; Wang, Jacob Tse-Wei; Wojciechowski, Konrad; Zhang, Wei. Anomalous Hysteresis in Perovskite Solar Cells. The journal of physical chemistry letters 2014, 5 (9), 1511–1515. https://doi.org/10.1021/jz500113x.

[6] Unger, E.L; Hoke, E.T; Bailie, C.D; Nguyen, W.H; Bowring, A.R; Heumüller, T; Christoforo, M.G; McGehee, M.D. Hysteresis and Transient Behavior in Current-Voltage Measurements of Hybrid-Perovskite Absorber Solar Cells. Energy & environmental science 2014, 7 (11), 3690–3698. https://doi.org/10.1039/c4ee02465f

[7] Kang, Dong-Ho; Park, Nam-Gyu. On the Current–Voltage Hysteresis in Perovskite Solar Cells: Dependence on Perovskite Composition and Methods to Remove Hysteresis. Advanced materials (Weinheim) 2019, 31 (34), e1805214–n/a. https://doi.org/10.1002/adma.201805214.

[8] Lin, Liangyou; Jones, Timothy W; Wang, Jacob Tse-Wei; Cook, Andre; Pham, Ngoc Duy; Duffy, Noel W; Mihaylov, Blago; Grigore, Mihaela; Anderson, Kenrick F; Duck,

Benjamin C; Wang, Hongxia; Pu, Jian; Li, Jian; Chi, Bo; Wilson, Gregory J. Perovskite Solar Cells: Strategically Constructed Bilayer Tin (IV) Oxide as Electron Transport Layer Boosts Performance and Reduces Hysteresis in Perovskite Solar Cells (Small 12/2020). Small (Weinheim an der Bergstrasse, Germany) 2020, 16 (12), 2070061–n/a. https://doi.org/10.1002/smll.202070061.

[9] Luo, Junsheng; Jia, Chunyang; Wan, Zhongquan; Han, Fei; Zhao, Bowen; Wang, Ruilin. The Novel Dopant for Hole-Transporting Material Opens a New Processing Route to Efficiently Reduce Hysteresis and Improve Stability of Planar Perovskite Solar Cells. Journal of power sources 2017, 342, 886–895. https://doi.org/10.1016/j.jpowsour.2017.01.010.

[10] Nagaoka, Hirokazu; Ma, Fei; deQuilettes, Dane W; Vorpahl, Sarah M; Glaz, Micah S; Colbert, Adam E; Ziffer, Mark E; Ginger, David S. Zr Incorporation into TiO2 Electrodes Reduces Hysteresis and Improves Performance in Hybrid Perovskite Solar Cells While Increasing Carrier Lifetimes. The journal of physical chemistry letters 2015, 6 (4), 669–675. https://doi.org/10.1021/jz502694g.

[11] Han, Fei; Luo, Junsheng; Malik, Haseeb Ashraf; Zhao, Bowen; Wan, Zhongquan; Jia, Chunyang. A Functional Sulfonic Additive for High Efficiency and Low Hysteresis Perovskite Solar Cells. Journal of power sources 2017, 359, 577–584. https://doi.org/10.1016/j.jpowsour.2017.05.084.

[12] Huang, Zhen; Wang, Duofa; Wang, Song; Zhang, Tianjin. Highly Efficient and Stable MAPbI3 Perovskite Solar Cell Induced by Regulated Nucleation and Ostwald Recrystallization. Materials 2018, 11 (5), 778. https://doi.org/10.3390/ma11050778.

[13] Zong, Yingxia; Wang, Ning; Zhang, Lin; Ju, Ming-Gang; Zeng, Xiao Cheng; Sun, Xiao Wei; Zhou, Yuanyuan; Padture, Nitin P. Back Cover: Homogenous Alloys of Formamidinium Lead Triiodide and Cesium Tin Triiodide for Efficient Ideal-Bandgap Perovskite Solar Cells (Angew. Chem. Int. Ed. 41/2017). Angewandte Chemie (International ed.) 2017, 56 (41), 12790–12790. https://doi.org/10.1002/anie.201708387.

[14] Jeon, Nam Joong; Noh, Jun Hong; Kim, Young Chan; Yang, Woon Seok; Ryu, Seungchan; Seok, Sang Il. Solvent Engineering for High-Performance Inorganic-Organic Hybrid Perovskite Solar Cells. Nature materials 2014, 13 (9), 897–903. https://doi.org/10.1038/nmat4014.

[15] Yan, Y.; Yin, W.-J.; Shi, T.; Meng, W.; Feng, C., Defect Physics of CH3NH3PbX3 (X = I, Br, Cl) Perovskites. In *Organic-Inorganic Halide Perovskite Photovoltaics: From Fundamentals to Device Architectures*, Park, N.-G.; Grätzel, M.; Miyasaka, T., Eds. Springer International Publishing: Cham, 2016; pp 79-105.

[16] Bryant, Daniel; Aristidou, Nicholas; Pont, Sebastian; Sanchez-Molina, Irene; Chotchunangatchaval, Thana; Wheeler, Scot; Durrant, James R; Haque, Saif A. Light and Oxygen Induced Degradation Limits the Operational Stability of Methylammonium Lead Triiodide Perovskite Solar Cells. Energy & environmental science 2016, 9 (5), 1655–1660. https://doi.org/10.1039/C6EE00409A.

[17] Chun-Ren Ke, Jack; Walton, Alex S; Lewis, David J; Tedstone, Aleksander; O'Brien, Paul; Thomas, Andrew G; Flavell, Wendy R. In Situ Investigation of Degradation at Organometal Halide Perovskite Surfaces by X-Ray Photoelectron Spectroscopy at Realistic Water Vapour Pressure. Chemical communications (Cambridge, England) 2017, 53 (37), 5231–5234. https://doi.org/10.1039/C7CC01538K.

[18] Zheng, Chao; Rubel, Oleg. Unraveling the Water Degradation Mechanism of CH3NH3PbI3. 2019. https://doi.org/10.1021/acs.jpcc.9b05516

[19] Babayigit, Aslihan; Ethirajan, Anitha; Muller, Marc; Conings, Bert. Toxicity of Organometal Halide Perovskite Solar Cells. Nature materials 2016, 15 (3), 247–251. https://doi.org/10.1038/nmat4572.

[20] Schileo, Giorgio; Grancini, Giulia. Lead or No Lead? Availability, Toxicity, Sustainability and Environmental Impact of Lead-Free Perovskite Solar Cells. Journal of materials chemistry. C, Materials for optical and electronic devices 2021, 9 (1), 67–76. https://doi.org/10.1039/d0tc04552g.

[21] Lyu, Miaoqiang; Yun, Jung-Ho; Chen, Peng; Hao, Mengmeng; Wang, Lianzhou. Addressing Toxicity of Lead: Progress and Applications of Low-Toxic Metal Halide Perovskites and Their Derivatives. Advanced energy materials 2017, 7 (15), 1602512–n/a. https://doi.org/10.1002/aenm.201602512.

[22] Leng, Meiying; Chen, Zhengwu; Yang, Ying; Li, Zha; Zeng, Kai; Li, Kanghua; Niu, Guangda; He, Yisu; Zhou, Qingchao; Tang, Jiang. Lead-Free, Blue Emitting Bismuth Halide Perovskite Quantum Dots. Angewandte Chemie (International ed.) 2016, 55 (48), 15012–15016. https://doi.org/10.1002/anie.201608160.

[23] Liu, Zhen; Dai, Shilei; Wang, Yan; Yang, Ben; Hao, Dandan; Liu, Dapeng; Zhao, Yiwei; Fang, Lu; Ou, Qingqing; Jin, Shu; Zhao, Jianwen; Huang, Jia. Photoresponsive Transistors Based on Lead-Free Perovskite and Carbon Nanotubes. Advanced functional materials 2020, 30 (3), 1906335–n/a. https://doi.org/10.1002/adfm.201906335.

[24] Ke, Weijun; Kanatzidis, Mercouri G. Prospects for Low-Toxicity Lead-Free Perovskite Solar Cells. Nature communications 2019, 10 (1), 965–965. https://doi.org/10.1038/s41467-019-08918-3.

[25] National Renewable Energy Laboratory (NREL) Best Research-Cell Efficiency Chart. https://www.nrel.gov/pv/cell-efficiency.html. **2021**, Accessed 15 May 2021.

[26] Kojima, Akihiro; Teshima, Kenjiro; Shirai, Yasuo; Miyasaka, Tsutomu. Organometal Halide Perovskites as Visible-Light Sensitizers for Photovoltaic Cells. Journal of the American Chemical Society 2009, 131 (17), 6050–6051. https://doi.org/10.1021/ja809598r.

[27] Duong, The; Pham, Huyen; Kho, Teng Choon; Phang, Pheng; Fong, Kean Chern; Yan, Di; Yin, Yanting; Peng, Jun; Mahmud, Md Arafat; Gharibzadeh, Saba; Nejand, Bahram Abdollahi; Hossain, Ihteaz M; Khan, Motiur Rahman; Mozaffari, Naeimeh; Wu, YiLiang; Shen, Heping; Zheng, Jianghui; Mai, Haoxin; Liang, Wensheng; Samundsett, Chris; Stocks, Matthew; McIntosh, Keith; Andersson, Gunther G; Lemmer, Uli; Richards, Bryce S; Paetzold, Ulrich W; Ho-Ballie, Anita; Liu, Yun; Macdonald, Daniel; Blakers, Andrew; Wong-Leung, Jennifer; White, Thomas; Weber, Klaus; Catchpole, Kylie. High Efficiency Perovskite-Silicon Tandem Solar Cells: Effect of Surface Coating Versus Bulk Incorporation of 2D Perovskite. Advanced energy materials 2020, 10 (9), 1903553–n/a. https://doi.org/10.1002/aenm.201903553.

[28] Ghosh, Biplab; Febriansyah, Benny; Harikesh, Padinhare Cholakkal; Koh, Teck Ming; Hadke, Shreyash; Wong, Lydia H; England, Jason; Mhaisalkar, Subodh G; Mathews, Nripan. Direct Band Gap Mixed-Valence Organic–inorganic Gold Perovskite as Visible Light Absorbers. Chemistry of materials 2020, 32 (15), 6318–6325. https://doi.org/10.1021/acs.chemmater.0c00345.

[29] Volonakis, George; Haghighirad, Amir Abbas; Milot, Rebecca L; Sio, Weng H; Filip, Marina R; Wenger, Bernard; Johnston, Michael B; Herz, Laura M; Snaith, Henry J; Giustino, Feliciano. Cs2InAgCl6: A New Lead-Free Halide Double Perovskite with Direct Band Gap. The journal of physical chemistry letters 2017, 8 (4), 772–778. https://doi.org/10.1021/acs.jpclett.6b02682.

[30] Le Corre, V. M.; Sherkar, T. S.; Koopmans, M.; Koster, L. J. A., Identification of the dominant recombination process for perovskite solar cells based on machine learning. *Cell Reports Physical Science* **2021**, *2* (2), 100346.

[31] Ren, Xingang; Wang, Zishuai; Sha, Wei E. I; Choy, Wallace C. H. Exploring the Way To Approach the Efficiency Limit of Perovskite Solar Cells by Drift-Diffusion Model. ACS photonics 2017, 4 (4), 934–942. https://doi.org/10.1021/acsphotonics.6b01043.

[32] Golubev, T.; Liu, D.; Lunt, R.; Duxbury, P., Understanding the impact of C 60 at the interface of perovskite solar cells via drift-diffusion modeling. *AIP advances* **2019**, *9* (3).

[33] Scharfetter, D. L.; Gummel, H. K., Large-signal analysis of a silicon Read diode oscillator. *IEEE transactions on electron devices* **1969**, *16* (1), 64-77.

[34] Stodtmann, S.; Lee, R. M.; Weiler, C. K. F.; Badinski, A., Numerical simulation of organic semiconductor devices with high carrier densities. *Journal of applied physics* **2012**, *112* (11), 114909.

[35] Walter, D.; Fell, A.; Wu, Y.; Duong, T.; Barugkin, C.; Wu, N.; White, T.; Weber, K., Transient Photovoltage in Perovskite Solar Cells: Interaction of Trap-Mediated Recombination and Migration of Multiple Ionic Species. *Journal of physical chemistry. C* **2018**, *122* (21), 11270-11281.

[36] Jacobs, D. A.; Shen, H.; Pfeffer, F.; Peng, J.; White, T. P.; Beck, F. J.; Catchpole, K. R., The Two Faces of Capacitance: New Interpretations for Electrical Impedance Measurements of Perovskite Solar Cells and Their Relation to Hysteresis. **2018**.

[37] "Quotes about Python". Python Software Foundation. Retrieved 8 January 2012.

[38] Pettersson, L. A. A.; Roman, L. S.; Inganäs, O., Modeling photocurrent action spectra of photovoltaic devices based on organic thin films. *Journal of applied physics* **1999**, *86* (1), 487-496.

[39] Peumans, P.; Yakimov, A.; Forrest, S. R., Small molecular weight organic thin-film photodetectors and solar cells. *Journal of applied physics* **2003**, *93* (7), 3693-3723.

[40] Löper, P.; Stuckelberger, M.; Niesen, B.; Werner, J.; Filipič, M.; Moon, S.-J.; Yum, J.-H.; Topič, M.; De Wolf, S.; Ballif, C., Complex Refractive Index Spectra of CH3NH3PbI3 Perovskite Thin Films Determined by Spectroscopic Ellipsometry and Spectrophotometry. *The journal of physical chemistry letters* **2015**, *6* (1), 66-71.

[41] Mielczarek, K. TransferMatrix_Python, Github: 2012. https://github.com/erichoke/Stanford/blob/master/TransferMatrix_Python

[42] laurent90git DAE-Scipy, Github: 2020. https://github.com/laurent90git/DAE-Scipy

[43] MATLAB version 9.4.0.813654 (R2018a), 2018.

[44] Python Software Foundation. Python Language Reference, version 3.8. Available at http://www.python.org

Appendix A

Core Python Code

The following appendices organize the Python source code in the order at which each class object and function is called. Each script is enclosed by its own section and border. This code is subject to change before being publicly released on Github.

A.1 parameters.py

```
# -*- coding: utf-8 -*-
"""
@author: Nathan

Parameters class. The user may choose between a single-layer or three-layer
    model along with various generation-recombination modes. Additionally,
    this class
non-dimensionalizes the parameter inputs.

The current parameters match the experimental conditions from Courtier et
   al.
Citations:

"""

import numpy as np
import math
from scipy.optimize import fsolve

class Params():

    def __init__(self):
```

```
"""
Experiment Modes:
-----------------
The modes listed below are test cases that were used in order to
    verify the single-layer model with the results from Courtier et
    al.

    please note,
    - if slm = 'off', tlm is assumed to be 'on'.
    - slm must be 'on' for test_case_slm to function.
    - to replicate the results from Courtier et al. single-layer
        model use Beer-Lambert optical model.
"""

slm = 'on'
test_case_slm = 'on'

""" Choose Optical Model:
- 'beer-lambert'
- 'transfer-matrix'
"""

self.model = 'beer-lambert'


"""
Parameter Inputs
----------------
- Each parameter is defined as an instance of the parameters class
    to allow for easy access through parameter objects.
----------------
"""

""" Tolerance and Resolution Parameters
    ----------------------------------
N: number of grid points - also used to calculate NE and NH for
    three-layer model
rtol: relative tolerance for Radau integrator timestep
atol: absolute tolerance for Radau integrator timestep
"""
self.N = 400
self.rtol = 1e-6
self.atol = 1e-10

""" J-V Scan Parameters
```

```
    ------------------
Vi: initial applied voltage/preconditioning voltage (V)
Vf: final applied voltage (V)
scan_rate: (V/s)
"""
self.Vi = 1.2
self.Vf = 0
self.scan_rate = 0.3

""" Constants
    ---------
q: proton charge (C)
kB: Boltzmann constant (eVK-1)
T: Temperature (K)
eps0: permittivity of free space (Fm-1)
Fph: incident photon flux (m-2s-1)
Vt: thermal voltage (V)
"""
self.q = 1.60217646e-19
self.kB = 8.61733035e-5
self.T = 298.0
self.eps0 = 8.85418782e-12
self.Fph = 1.4e21
self.Vt = self.kB*self.T

""" Perovskite Layer Parameter Inputs
    --------------------------------
b: perovskite layer thickness (m)
epsp: permittivity of perovskite (Fm-1)
alpha: perovskite absorption coefficient (m-1)
Ec: conduction band minimum (eV)
Ev: valence band maximum (eV)
Dn: perovskite electron diffusion coefficient (m2s-1)
Dp: perovskite hole diffusion coefficient (m2s-1)
gc: conduction band density of states (m-3)
gv: valence band density of states (m-3)
"""
self.b = 600e-9
self.epsp = 24.1*self.eps0
self.alpha = 1.3e7
self.Ec = -3.7
self.Ev = -5.4
self.Dn = 1.7e-4
self.Dp = 1.7e-4
self.gc = 8.1e24
self.gv = 5.8e24
```

```
""" Ion Parameter Inputs
    --------------------
N0: density of ion vacancies (m-3)
D: diffusivity relation
DIinf: high temperature vacancy diffusion coefficient (m2s-1)
EAI: iodide vacancy activation energy (eV)
DI: diffusion coefficient for iodide ions (m2s-1)
"""
self.N0 = 1.6e25
self.D = lambda Dinf, EA: Dinf*np.exp(-EA/(self.kB*self.T))
self.DIinf = 6.5e-8
self.EAI = 0.58
self.DI = self.D(self.DIinf, self.EAI)


"""
Three-Layer Model Parameter Inputs
----------------------------------
"""

""" ETL Parameter Inputs
    --------------------
dE: effective doping density of ETL (m-3)
gcE: effective conduction band density of states in ETL (m-3)
EcE: conduction band minimum in ETL (eV)
bE: ETL layer thickness (m)
epsE: permittivity of ETL (Fm-1)
DE: electron diffusion coefficient in ETL (m2s-1)
"""
self.dE = 1e24
self.gcE = 5e25
self.EcE = -4.0
self.bE = 100e-9
self.epsE = 10*self.eps0
self.DE = 1e-5

""" HTL Parameter Inputs
    --------------------
dH: effective doping density of HTL (m-3)
gvH: effective valence band density of states in HTL (m-3)
EvH: valence band minimum in HTL (eV)
bH: HTL layer thickness (m)
epsH: permittivity of HTL (Fm-1)
DH: hole diffusion coefficient in HTL (m3s-1)
"""
```

```
self.dH = 1e24
self.gvH = 5e25
self.EvH = -5.1
self.bH = 200e-9
self.epsH = 3*self.eps0
self.DH = 1e-6

""" Beer-Lambert Law Generation """
self.Y = 3.7
self.w = 2.4

""" Bulk Recombination Parameters
    ----------------------------
tn: SRH - electron pseudo-lifetime (s)
tp: SRH - hole pseudo-lifetime (s)
beta: bimolecular recombination rate (m3s-1)
"""
self.tn = 3e-9
self.tp = 3e-7
self.beta = 0.0

""" Interface Recombination Parameters
    ---------------------------------
betaE: ETL/perovskite bimolecular recombination rate (m3s-1)
betaH: perovskite/HTL bimolecular recombination rate (m3s-1)
vnE: SRH - electron recombination velocity - ETL (ms-1)
vpE: SRH - hole recombination velocity - ETL (ms-1)
vnH: SRH - electron recombination velocity - HTL (ms-1)
vpH: SRH - hole recombination velocity - HTL (ms-1)
"""
self.betaE = 0.0
self.betaH = 0.0
self.vnE = 1e5
self.vpE = 10
self.vnH = 0.1
self.vpH = 1e5

"""
PARAMETER INPUTS COMPLETE - Alteration to the following script may
    impact the accuracy of the simulation.
"""


""" Perovskite Calculated Parameters & Non-dimensionalization
    ------------------------------------------------------------
Eg: bandgap (eV)
```

```
LD: debye length (m)
lam: non-dimensional debye length parameter
lam2: non-dimensional debye length parameter squared
ni: intrinsic carrier density (m-3)
delta: ratio of electron and ion densities
chi: ratio of hole and electron densities
G0: typical rate of photogeneration (m-3s-1)
Tion: if DI !=0 then characteristic ionic timescale (s)
     if DI = 0 then characteristic electronic timescale (s)
sigma: ratio of carrier and electronic timescales
Kp: hole current parameter
Kn: electron current parameter
Upsilon: Beer-Lambert photogeneration parameter
"""
self.Eg = self.Ec - self.Ev
self.LD = np.sqrt(self.Vt*self.epsp/(self.q*self.N0))
self.lam = self.LD/self.b
self.lam2 = self.lam**2
self.ni = np.sqrt(self.gc*self.gv)*np.exp(-self.Eg/(2*self.Vt))
self.delta = self.dE/self.N0
self.chi = self.dH/self.dE
self.G0 = np.array((self.Fph/self.b))*(1-np.exp(-self.alpha*self.b))
if self.DI != 0:
    self.Tion = self.b/self.DI*np.sqrt(self.Vt*self.epsp/(self.q*
        self.N0))
else:
    self.Tion = self.dE/self.G0
self.sigma = self.dE/(self.G0*self.Tion)
self.Kp = self.Dp*self.dH/(self.G0*self.b**2)
self.Kn = self.Dn*self.dE/(self.G0*self.b**2)
self.Upsilon = self.alpha*self.b

""" Energy Level Parameters
    ----------------------
EfE: ETL workfunction (eV)
EfH: HTL workfunction (eV)
Vbi: built-in voltage (V)
pbi: non-dimensional built-in voltage
"""
self.EfE = self.EcE - self.Vt*np.log(self.gcE/self.dE)
self.EfH = self.EvH + self.Vt*np.log(self.gvH/self.dH)
self.Vbi = self.EfE - self.EfH
self.pbi = self.Vbi/self.Vt

""" Interface Parameters
    --------------------
```

```
kE: ratio between electron densities across ETL/perovskite interface
kH: ratio between hole densities across perovskite/HTL interface
n0: electron density in perovskite (m-3)
p0: hole density in perovskite (m-3)
"""
self.kE = self.gc/self.gcE*np.exp((self.EcE-self.Ec)/self.Vt)
self.kH = self.gv/self.gvH*np.exp((self.Ev-self.EvH)/self.Vt)
self.n0 = self.kE*self.dE
self.p0 = self.kH*self.dH



"""
Single-Layer Model Conditions
-----------------------------
- the single-layer model uses a different non-dimensionalization
    method than the three-layer model
- this condition modifies the few parameters that differ from the
    three-layer model
"""
if slm == 'on':
    """ Single-Layer Model Non-Dimensionalized Parameters """
    self.delta = self.Fph*self.b/(self.DE*self.N0)
    self.sigma = self.DI*self.b/(self.DE*self.LD)
    self.k_p = self.Dp/self.DE
    self.k_n = self.Dn/self.DE
elif test_case_slm == 'off':
    """ Do nothing """

""" Dirichlet Boundary Conditions
    ----------------------------
n_bar: ETL interface electron density
p_bar: HTL interface hole density
"""
self.n_bar = self.n0*self.DE/(self.Fph*self.b)
self.p_bar = self.p0*self.DE/(self.Fph*self.b)



""" Scan-rate scaling
    ----------------
ion_mobility_factor: typical ion mobility used to non-dimensionalize
     the scan rate
"""
self.ion_mobility_factor= 2.4025533333333337e-16
```

```
""" Non-Dimensional Scan Parameters """
self.psi_scan_rate = self.scan_rate*(self.LD*self.b/self.
   ion_mobility_factor)/self.Vt
self.phi_i = self.Vi/self.Vt
self.phi_f = self.Vf/self.Vt
self.phi_precondition = self.phi_i
self.tf_scan = np.abs((self.phi_i - self.phi_f)/self.psi_scan_rate)


""" Test Case Conditions
    --------------------
"""
if test_case_slm == 'on':
    self.pbi = 40
    self.DI = 2.4025533333333337e-16
    self.delta = 2.1*10e-7
    self.sigma = 5.8*10e-10
    self.n_bar = 20
    self.p_bar = 0.3
    self.lam = 2.4*10e-4
    self.lam2 = self.lam**2
    self.phi_precondition = 46.729618965853994
    self.phi_f = 0.0
    self.psi_scan_rate = 14.2
    self.tf_scan = np.abs((self.phi_precondition - self.phi_f)/self.
       psi_scan_rate)
    #self.tf_forward_scan = self.tf_reverse_scan
elif test_case_slm == 'off':
    """ Do nothing """

""" ETL & HTL Parameters
    --------------------
wE: relative width of ETL
wH: relative width of HTL
KE: ETL electron current parameter
KH: HTL hole current parameter
rE: relative ETL permittivity
rH: relative HTL permittivity
lamE2: relative ETL Debye length parameter squared
lamE: relative ETL Debye length parameter
lamH2: relative HTL Debye length parameter squared
lamH: relative HTL Debye length parameter
OmegaE: ETL charge density parameter
OmegaH: HTL charge density parameter
"""
self.wE = self.bE/self.b
```

```
self.wH = self.bH/self.b
self.KE = self.DE*self.Kn/self.Dn
self.KH = self.DH*self.Kp/self.Dp
self.rE = self.epsE/self.epsp
self.rH = self.epsH/self.epsp
self.lamE2 = self.rE*self.N0/self.dE*self.lam2
self.lamE = np.sqrt(self.lamE2)
self.lamH2 = self.rH*self.N0/self.dH*self.lam2
self.lamH = np.sqrt(self.lamH2)
self.OmegaE = np.sqrt(self.N0/(self.rE*self.dE))
self.OmegaH = np.sqrt(self.N0/(self.rH*self.dH))

""" Bulk Recombination Parameters
    ----------------------------
ni2: non-dimensional intrinsic carrier density squared
brate: bimolecular recombination rate constant
gamma: rate constant for SRH recombination
tor: ratio of SRH carrier lifetimes
tor3: constant from deep trap approximation
- gamma, tor, tor3 = 0 if no bulk SRH recombination
"""
self.ni2 = self.ni**2/(self.dE*self.dH)
self.brate = self.beta*self.dE*self.dH/self.G0
if self.tp>0 and self.tn>0:
    self.gamma = self.dH/(self.tp*self.G0)
    self.tor = self.tn*self.dH/(self.tp*self.dE)
    self.tor3 = (self.tn+self.tp)*self.ni/(self.tp*self.dE)
else:
    [self.gamma, self.tor, self.tor3] = [0,0,0]

""" Interface Recombination Parameters
    ----------------------------------
brateE: ETL bimolecular recombination rate constant
brateH: HTL bimolecular recombination rate constant
gammaE: ETL rate constant for SRH recombination
torE: ratio of SRH carrier lifetimes
torE3: constant from deep trap state approximation
- gammeE, torE, torE3 = 0 if no ETL/perovskite interface
    recombination
gammaH: HTL rate constnat for SRH recombination
torH: ratio of SRH carrier lifetimes
torH3: constant from deep trap state approximation
- gammaH, torH, torH3 = 0 if no perovskite/HTL interface
    recombination
"""
self.brateE = self.betaE*self.dE*self.dH/(self.b*self.G0)
```

```
        self.brateH = self.betaH*self.dE*self.dH/(self.b*self.G0)
        if self.vpE>0 and self.vnE>0:
            self.gammaE = self.dH*self.vpE/(self.b*self.G0)
            self.torE = self.dH*self.vpE/(self.dE*self.vnE)
            self.torE3 = (1/self.kE+self.vpE/self.vnE)*self.ni/self.dE
        else:
            [self.gammaE, self.torE, self.torE3] = [0,0,0]
        if self.vnH>0 and self.vpH>0:
            self.gammaH = self.dE*self.vnH/(self.b*self.G0)
            self.torH = self.dE*self.vnH/(self.dH*self.vpH)
            self.torH3 = (1/self.kH+self.vnH/self.vpH)*self.ni/self.dH
        else:
            [self.gammaH, self.torH, self.torH3] = [0,0,0]


        """ Spatial Discretization Parameters
            ---------------------------------
        X: percentage of grid points within ionic Debye length of the
            interface
        NE: number of grid points in ETL
        NH: number of grid points in HTL
        """
        self.X = 0.2
        self.tanhfun = lambda x, st: (math.tanh(st*(2*x-1))/math.tanh(st)+1)
            /2
        def func(st):
            return self.lam - self.tanhfun(self.X, st)
        self.st = float(fsolve(func, 2))
        self.A = lambda b: (math.tanh(self.st*(1-2/self.N))-(1-np.double(b))
            *math.tanh(self.st))/np.double(b)
        self.NE = np.int(np.round(2/(1-math.atanh(self.A(self.wE))/self.st))
            )
        self.NH = np.int(np.round(2/(1-math.atanh(self.A(self.wH))/self.st))
            )
```

A.2 grid.py

```
# -*- coding: utf-8 -*-
"""
@author: Nathan

Last modified 2/23/2021 - stable with accurate results
```

```
Spatial discretization for single-layer and three-layer models.
This class uses a parameters object to define the vectors.

Citations:
"""

import numpy as np

class Grid():

    def __init__(self, params):

        """ Define class variables from constructor class objects. """
        N = params.N
        NE = params.NE
        NH = params.NH
        st = params.st
        wE = params.wE
        wH = params.wH

        """ Calculate spatial mesh for single-layer and three-layer models.
            """
        # 'tanh' grid spacing.
        x = np.linspace(0, 1, N+1)
        self.x = (np.tanh(st*(2*x-1))/np.tanh(st)+1)/2
        xE = np.linspace(-wE, 0, NE+1)
        self.xE = wE*(np.tanh(st*(2*xE/wE+1))/np.tanh(st)+1)/2
        xH = np.linspace(1, 1+wH, NH+1)
        self.xH = 1 + wH*(np.tanh(st*(2*(xH-1)/wH-1))/np.tanh(st)+1)/2

        """ Differencing vectors. """
        self.dx = np.diff(self.x)
        self.dxE = np.diff(self.xE)
        self.dxH = np.diff(self.xH)

        """ The grid points at the interfaces are already included in the
            perovskite layer grid. """
        self.xE = self.xE[0:-1]
        self.xH = self.xH[1:]
```

A.3 matrices.py

```
# -*- coding: utf-8 -*-
```

```
"""
@author: Nathan

Last modified 2/23/2021 - stable with accurate results

Matrices class that defines single-layer and three-layer averaging matrices
     as well as constant charge densities used in the Poisson's Equation.
The matrices are tridiagonal matrices used for vectorization of the
    governing DD equations.
The class uses parameters and spatial grid class objects to define the
    matrices.

Citations:

Warning: The np.array used to create the diagonals matrices issue a
    deprecation warning from using ragged nested sequences. The dtype is
    specified to object to prevent this error
from being displayed in the command window. This script may lose
    functionality with future updates to the numpy library.

"""

from scipy.sparse import diags
import numpy as np

class Matrices():

    def __init__(self, params, grid):

        """ Define class variables from constructor class objects. """
        N = params.N
        NE = params.NE
        NH = params.NH
        dx = grid.dx
        dxE = grid.dxE
        dxH = grid.dxH

        """ Arrays used to define the sub-diagonals [-1], main-diagonals
            [0], and upper-diagonals [1] used in the instance variables. """
        diagonals_Av = np.array([np.zeros(N), np.ones(N+1), np.ones(N)],
            dtype=object)
        diagonals_AvE = np.array([np.zeros(int(round(NE))), np.ones(int(
            round(NE))+1), np.ones(int(round(NE)))], dtype=object)
        diagonals_AvH = np.array([np.zeros(int(round(NH))), np.ones(int(
            round(NH+1))), np.ones(int(round(NH)))], dtype=object)
```

```
diagonals_Lo = np.array([np.append(dx[0:-1]/6, [0.0]), np.append(np.
    append([0.0], [(dx[0:-1]+dx[1:])/3]), [0.0]), np.append([0.0],
    dx[1:]/6)], dtype=object)
diagonals_LoE = np.array([np.append(dxE[0:-1]/6, [0.0]), np.append(
    np.append([0.0], [(dxE[0:-1]+dxE[1:])/3]), [0.0]), np.append
    ([0.0], dxE[1:]/6)], dtype=object)
diagonals_LoH = np.array([np.append(dxH[0:-1]/6, [0.0]), np.append(
    np.append([0.0], [(dxH[0:-1]+dxH[1:])/3]), [0.0]), np.append
    ([0.0], dxH[1:]/6)], dtype=object)

diagonals_Dx = np.array([0./dx, np.append([-1./dx], [0.0]), 1./dx],
    dtype=object)
diagonals_DxE = np.array([0./dxE, np.append([-1./dxE], [0.0]), 1./
    dxE], dtype=object)
diagonals_DxH = np.array([0./dxH, np.append([-1./dxH], [0.0]), 1./
    dxH], dtype=object)

""" Averaging matrices. """
self.Av = diags(diagonals_Av, [-1,0,1]).toarray()/2; self.Av = self.
    Av[0:N, 0:N+1]
self.AvE = diags(diagonals_AvE, [-1,0,1]).toarray()/2; self.AvE =
    self.AvE[0:int(round(NE)), 0:int(round(NE))+1]
self.AvH = diags(diagonals_AvH, [-1,0,1]).toarray()/2; self.AvH =
    self.AvH[0:int(round(NH)), 0:int(round(NH))+1]

self.Lo = diags(diagonals_Lo, [-1,0,1]).toarray(); self.Lo = self.Lo
    [1:N,0:N+1]
self.LoE = diags(diagonals_LoE, [-1,0,1]).toarray(); self.LoE = self
    .LoE[1:int(round(NE)),0:int(round(NE))+1]
self.LoH = diags(diagonals_LoH, [-1,0,1]).toarray(); self.LoH = self
    .LoH[1:int(round(NH)),0:int(round(NH))+1]

""" Differencing matrices. """
self.Dx = diags(diagonals_Dx, [-1,0,1]).toarray(); self.Dx = self.Dx
    [0:N,0:N+1];
self.DxE = diags(diagonals_DxE, [-1,0,1]).toarray(); self.DxE = self
    .DxE[0:int(round(NE)),0:int(round(NE))+1]
self.DxH = diags(diagonals_DxH, [-1,0,1]).toarray(); self.DxH = self
    .DxH[0:int(round(NH)),0:int(round(NH))+1]

""" Vectors for constant cation vacancy and doping densities. """
self.NN = (dx[1:]+dx[0:-1])/2
self.ddE = (dxE[1:]+dxE[0:-1])/2
self.ddH = (dxH[1:]+dxH[0:-1])/2
```

A.4 mass_slm.py

```
# -*- coding: utf-8 -*-
"""
@author: Nathan

Last modified 2/23/2021 - stable with accurate results

Mass matrix class that creates a constant sparse matrix defining the
    coefficients to the temporal derivatives for the single-layer model.
The matrix is a singular tridiagonal matrix with zeroes in place for the
    Poisson's equation.
The class uses parameters and spatial grid class objects to define the
    matrices.

Citations:

Warning: The np.array used to create the diagonals matrices issue a
    deprecation warning from using ragged nested sequences. The dtype is
    specified to object to prevent this error
from being displayed in the command window. This script may lose
    functionality with future updates to the numpy library.

"""

import numpy as np
from scipy.sparse import diags
from scipy.sparse import csr_matrix

class Mass():

    def __init__(self, params, grid):

        """ Define class variables from constructor class objects. """
        N = params.N
        sigma = params.sigma
        dx = grid.dx

        """ Arrays used to define the sub-diagonals [-1], main-diagonals
            [0], and upper-diagonals [1] used in the instance variables. """
        diagonal_M11 = np.array([np.append(dx[0:-1]/6, dx[-1]/6), np.append(
            dx[0]/3, np.append((dx[0:-1]+dx[1:])/3, dx[-1]/3)), np.append(dx
            [0]/6, dx[1:]/6)], dtype=object)
```

```
        diagonal_M33 = sigma*np.array([np.append(dx[0:-1]/6, dx[-1]/6), np.
            append(dx[0]/3, np.append((dx[0:-1]+dx[1:])/3, dx[-1]/3)), np.
            append(dx[0]/6, dx[1:]/6)], dtype=object)
        diagonal_M44 = diagonal_M33

        """ Piece by piece construction and horizontal concatenation of zero
             and non-zero arrays. """
        # P equation
        M11 = diags(diagonal_M11, [-1,0,1]).toarray()
        M1 = np.concatenate((M11, np.zeros((N+1, 3*N+3))), 1)

        # Phi equation
        M2 = np.zeros((N+1,4*N+4))

        # n equation
        M33 = diags(diagonal_M33, [-1,0,1]).toarray(); M33[0,0] = 0; M33
            [0,1] = 0
        M312 = np.zeros((N+1,2*N+2))
        M34 = np.zeros((N+1,N+1))
        M3 = np.concatenate((M312, M33, M34), 1)

        # p equation
        M44 = diags(diagonal_M44, [-1,0,1]).toarray(); M44[-1,-1] = 0; M44
            [-1,-2] = 0
        M4123 = np.zeros((N+1,3*N+3))
        M4 = np.concatenate((M4123, M44), 1)

        """ Define instance variables. """
        # Vertically concatenate all rows.
        M = np.concatenate((M1, M2, M3, M4))
        self.M = csr_matrix(M)
```

A.5 mass_tlm.py

```
# -*- coding: utf-8 -*-
"""
@author: Nathan

Last modified 3/3/2021 - stable with accurate results

Mass matrix class that creates a constant sparse matrix defining the
    coefficients to the temporal derivatives for the three-layer model.
```

```
The matrix is a singular tridiagonal matrix with zeroes in place for the
    Poisson's equation.
The class uses parameters and spatial grid class objects to define the
    matrices.

Citations:

Warning: The np.array used to create the diagonals matrices issue a
    deprecation warning from using ragged nested sequences. The dtype is
    specified to object to prevent this error
from being displayed in the command window. This script may lose
    functionality with future updates to the numpy library.
"""

import numpy as np
from scipy.sparse import diags
from scipy.sparse import csr_matrix

class Mass():

    def __init__(self, params, grid, mode=None):

        """ Define class variables from constructor class objects. """
        chi = params.chi
        dx = grid.dx
        dxE = grid.dxE
        dxH = grid.dxH
        kE = params.kE
        kH = params.kH
        N = params.N
        NE = params.NE
        NH = params.NH
        sigma = params.sigma

        """ Arrays used to define the sub-diagonals [-1], main-diagonals
            [0], and upper-diagonals [1] used in the instance variables. """
        diagonal_M11 = np.array([np.append(dx[0:-1]/6, dx[-1]/6), np.append(
            dx[0]/3, np.append((dx[0:-1]+dx[1:])/3, dx[-1]/3)), np.append(dx
            [0]/6, dx[1:]/6)], dtype=object)
        diagonal_M33 = np.array([np.append(dx[0:-1]/6, dx[-1]/6), np.append(
            dx[0]/3, np.append((dx[0:-1]+dx[1:])/3, dx[-1]/3)), np.append(dx
            [0]/6, dx[1:]/6)], dtype=object)
        diagonal_M44 = np.array([np.append(dx[0:-1]/6, dx[-1]/6), np.append(
            dx[0]/3, np.append((dx[0:-1]+dx[1:])/3, dx[-1]/3)), np.append(dx
            [0]/6, dx[1:]/6)], dtype=object)
```

```
diagonal_M66 = np.array([dxE[0:-1]/6, np.append([0.0], (dxE[0:-1] +
    dxE[1:])/3), np.append([0.0], dxE[1:-1]/6)], dtype=object)
diagonal_M88 = np.array([np.append(dxH[1:-1]/6, [0.0]), np.append((
    dxH[0:-1] + dxH[1:])/3, [0.0]), dxH[1:]/6], dtype=object)

""" Piece by piece construction and horizontal concatenation of zero
     and non-zero arrays. """
M11 = diags(diagonal_M11, [-1,0,1]).toarray()
M12 = np.zeros((N+1, N+1))
M15 = np.zeros((N+1, NE))
M17 = np.zeros((N+1, NH))
M33 = sigma*diags(diagonal_M33, [-1,0,1]).toarray(); M33[0,0] =
    sigma*(dxE[-1]/kE + dx[0])/3
M36 = np.zeros((N+1, NE)); M36[0,-1] = sigma*dxE[-1]/6
M44 = sigma*chi*diags(diagonal_M44, [-1,0,1]).toarray(); M44[-1,-1]
    = sigma*chi*(dx[-1] + dxH[0]/kH)/3
M48 = np.zeros((N+1, NH)); M48[-1,0] = sigma*chi*dxH[0]/6
M51 = np.zeros((NE, N+1))
M55 = np.zeros((NE, NE))
M57 = np.zeros((NE, NH))
M63 = np.zeros((NE, N+1)); M63[-1,0] = sigma*(dxE[-1]/kE)/6
M66 = sigma*diags(diagonal_M66, [-1,0,1]).toarray()
M71 = np.zeros((NH, N+1))
M75 = np.zeros((NH, NE))
M77 = np.zeros((NH, NH))
M84 = np.zeros((NH, N+1)); M84[0,-1] = sigma*chi*(dxH[0]/kH)/6
M88 = sigma*chi*diags(diagonal_M88, [-1,0,1]).toarray()

""" Define the sparsity structure of each row. a-b-c-d-e-f-g-h are
    the blocks corresponding to each variable in the mass matrix.
    """
""" Row 1 """
a = M11; bcd = np.tile(M12, (3)); ef = np.tile(M15, (2)); gh = np.
    tile(M17, (2))
row1 = np.concatenate((a, bcd, ef, gh), 1)
""" Row 2 """
abcd = np.tile(M12, (4)); ef = np.tile(M15, (2)); gh = np.tile(M17,
    (2))
row2 = np.concatenate((abcd, ef, gh), 1)
""" Row 3 """
ab = np.tile(M12, (2)); c = M33; d = M12; e = M15; f = M36; gh = np.
    tile(M17, (2))
row3 = np.concatenate((ab, c, d, e, f, gh), 1)
""" Row 4 """
abc = np.tile(M12, (3)); d = M44; ef = np.tile(M15, (2)); g = M17; h
     = M48
```

```
        row4 = np.concatenate((abc, d, ef, g, h), 1)
        """ Row 5 """
        abcd = np.tile(M51, (4)); ef = np.tile(M55, (2)); gh = np.tile(M57,
            (2))
        row5 = np.concatenate((abcd, ef, gh), 1)
        """ Row 6 """
        ab = np.tile(M51, (2)); c = M63; d = M51; e = M55; f = M66; gh = np.
            tile(M57, (2))
        row6 = np.concatenate((ab, c, d, e, f, gh), 1)
        """ Row 7 """
        abcd = np.tile(M71, (4)); ef = np.tile(M75, (2)); gh = np.tile(M77,
            (2))
        row7 = np.concatenate((abcd, ef, gh), 1)
        """ Row 8 """
        abc = np.tile(M71, (3)); d = M84; ef = np.tile(M75, (2)); g = M77; h
             = M88
        row8 = np.concatenate((abc, d, ef, g, h), 1)
        """ Vertically concatenate all rows. """
        M = np.concatenate((row1, row2, row3, row4, row5, row6, row7, row8),
             0)

        """ Special Conditions """
        if mode == 'precondition':
            M[0:N+1,:] = sigma*M[0:N+1,:]

        """ Define instance variable. """
        self.M = csr_matrix(M)
```

A.6 jacobian_slm.py

```
# -*- coding: utf-8 -*-
"""
@author: Nathan

Last modified 2/23/2021 - stable with accurate results

Jacobian matrix class that defines the sparsity structure for the jacobian
    of the RHS of the single-layer model equations.
The arrays are comprised of boolean values that identify non-zero elements
    of the jacobian matrix. Because the jacobian cannot be determined
    analytically, the sparsity structure
is input to the Radau solver to significanly increase the computation time
    and accuracy of the algorithm.
```

```
This class uses a parameters object to define the matrix.

Citations:


"""

import numpy as np
from scipy.sparse import diags
from scipy.sparse import csr_matrix

class Jac():

    def __init__(self, params):

        """ Define class variable from constructor class objects. """
        N = params.N

        """ Piece by piece construction and horizontal concatenation of zero
             and non-zero arrays. """
        # J11 can be used to define all non-zero matrices in the sparsity
            structure.
        J11 = diags((np.ones(N), np.ones(N+1), np.ones(N)), [-1,0,1]).
            toarray()
        # First row, column 3 and 4.
        J13 = np.zeros((N+1, N+1))
        # First row, column 1 and 2.
        J_top_left = np.tile(J11, (2))
        # First row, column 3 and 4.
        J_top_right = np.tile(J13, (2))
        # Horizontally concatenate the top row.
        J_top = np.concatenate((J_top_left, J_top_right), 1)
        # Bottom three rows.
        J_bottom_3_rows = np.tile(J11, (3,4))

        # Vertically concatenate the top row with the bottom three rows.
        jac = np.concatenate((J_top, J_bottom_3_rows), 0)

        """ Define instance variables. """
        self.jac = csr_matrix(jac)
```

A.7 jacobian_tlm.py

```
# -*- coding: utf-8 -*-
"""
@author: Nathan

Last modified 3/3/2021 - stable with accurate results

Jacobian matrix class that defines the sparsity structure for the jacobian
    of the RHS of the three-layer model equations.
The arrays are comprised of boolean values that identify non-zero elements
    of the jacobian matrix. Because the jacobian cannot be determined
    analytically, the sparsity structure
is input to the Radau solver to significanly increase the computation time
    and accuracy of the algorithm.
This class uses a parameters object to define the matrix.

Citations:

"""

import numpy as np
from scipy.sparse import diags
from scipy.sparse import csr_matrix

class Jac():

    def __init__(self, params, mode=None):

        """ Define class variable from constructor class objects. """
        N = params.N
        NE = params.NE
        NH = params.NH

        """ Piece by piece construction and horizontal concatenation of zero
             and non-zero arrays. """
        J11 = diags((np.ones(N), np.ones(N+1), np.ones(N)), [-1,0,1]).
            toarray()
        J13 = np.zeros((N+1, N+1))
        J15 = np.zeros((N+1, NE))
        J17 = np.zeros((N+1, NH))
        J25 = np.zeros((N+1, NE)); J25[0, NE-1] = 1
        J27 = np.zeros((N+1, NH)); J27[N, 0] = 1
        J51 = np.zeros((NE, N+1))
        J52 = np.zeros((NE, N+1)); J52[NE-1, 0] = 1
        J55 = diags((np.ones(NE-1), np.ones(NE), np.ones(NE-1)), [-1,0,1]).
            toarray(); J55[0,1] = 0
```

```
J56 = diags((np.ones(NE-1), np.ones(NE), np.ones(NE-1)), [-1,0,1]).
    toarray(); J56[0,1] = 0; J56[0,0] = 0
J57 = np.zeros((NE, NH))
J71 = np.zeros((NH, N+1))
J72 = np.zeros((NH, N+1)); J72[0, N] = 1
J75 = np.zeros((NH, NE))
J77 = diags((np.ones(NH-1), np.ones(NH), np.ones(NH-1)), [-1,0,1]).
    toarray(); J77[NH-1, NH-2] = 0
J78 = diags((np.ones(NH-1), np.ones(NH), np.ones(NH-1)), [-1,0,1]).
    toarray(); J78[NH-1, NH-2] = 0; J78[NH-1, NH-1] = 0


""" Define the sparsity structure of each row. a-b-c-d-e-f-g-h are
    the blocks corresponding to each variable in the jacobian matrix
    . """
""" Row 1 """
ab = np.tile(J11, (2)); cd = np.tile(J13, (2)); ef = np.tile(J15,
    (2)); gh = np.tile(J17, (2))
row1 = np.concatenate((ab, cd, ef, gh), 1)
""" Row 2 """
abcd = np.tile(J11, (4)); ef = np.tile(J25, (2)); gh = np.tile(J27,
    (2))
row2 = np.concatenate((abcd, ef, gh), 1)
""" Row 3 """
abcd = np.tile(J11, (4)); ef = np.tile(J25, (2)); gh = np.tile(J17,
    (2))
row3 = np.concatenate((abcd, ef, gh), 1)
""" Row 4 """
abcd = np.tile(J11, (4)); ef = np.tile(J15, (2)); gh = np.tile(J27,
    (2))
row4 = np.concatenate((abcd, ef, gh), 1)
""" Row 5 """
a = J51; bc = np.tile(J52, (2)); d = J51; e = J55; f = J56; gh = np.
    tile(J57, (2))
row5 = np.concatenate((a, bc, d, e, f, gh), 1)
""" Row 6 """
a = J51; bc = np.tile(J52, (2)); d = J51; e = J56; f = J55; gh = np.
    tile(J57, (2))
row6 = np.concatenate((a, bc, d, e, f, gh), 1)
""" Row 7 """
a = J71; b = J72; c = J71; d = J72; ef = np.tile(J75, (2)); g = J77;
     h = J78
row7 = np.concatenate((a, b, c, d, ef, g, h), 1)
""" Row 8 """
a = J71; b = J72; c = J71; d = J72; ef = np.tile(J75, (2)); g = J78;
     h = J77
```

```
        row8 = np.concatenate((a, b, c, d, ef, g, h), 1)
        """ Vertically concatenate all rows. """
        jac = np.concatenate((row1, row2, row3, row4, row5, row6, row7, row8
            ), 0)

        if mode == 'precondition':
            jac[N,:] = 0.0; jac[N,0:N+1] = 1.0


        """ Define instance variables. """
        self.jac = csr_matrix(jac)
```

A.8 rhs_slm.py

```
# -*- coding: utf-8 -*-
"""
@author: Nathan

Last Modified 2/23/2021

This script defines the right-hand-side of the drift-diffusion
    semiconductor equations for the single-layer model.
The inputs of this function are:
    t: independent variable matrix
    u: dependent variable matrix
    psi: type of scan
    mode: solution condition

The function outputs the u matrix in the equation
    M*du/dt = u
"""


import numpy as np
import parameters, grid, matrices, generation_recombination


""" Create class objects. """
params = parameters.Params()
grid = grid.Grid(params)
mat = matrices.Matrices(params, grid)
genrec = generation_recombination.GR(params, grid)
```

```
""" Define parameters, grid, and matrices from class objects. """
chi = params.chi
delta = params.delta
lam = params.lam
lam2 = params.lam2
N = params.N
n_bar = params.n_bar
p_bar = params.p_bar
pbi = params.pbi
phi_precondition = params.phi_precondition
phi_f = params.phi_f
psi_scan_rate = params.psi_scan_rate


x = grid.x
dx = grid.dx

Av = mat.Av
Dx = mat.Dx
Lo = mat.Lo
NN = mat.NN

Rr = genrec.Rr
Rl = genrec.Rl
G = genrec.G
R = genrec.mm_recomb

def SLM(t, u, *psi):

    if ('pbi' in psi):
        psi = lambda t: pbi
    elif ('test1' in psi):
        psi = lambda t: pbi*(1 - (np.tanh(10**6*t)/np.tanh(10**6)))
    elif ('precondition' in psi):
        psi = lambda t: pbi*(1 - t/5) + phi_precondition*(t/5) if t<5 else
            phi_precondition
    elif ('reverse_scan' in psi):
        psi = lambda t: phi_precondition - psi_scan_rate*t
    elif ('forward_scan' in psi):
        psi = lambda t: phi_f + psi_scan_rate*t


    elif ('scan_1' in psi):
        if params.Vi < params.Vf:
            psi = lambda t: phi_precondition + psi_scan_rate*t
        elif params.Vi > params.Vf:
```

```
        psi = lambda t: phi_precondition - psi_scan_rate*t
    elif ('scan_2' in psi):
        if params.Vi < params.Vf:
            psi = lambda t: phi_f - psi_scan_rate*t
        elif params.Vi > params.Vf:
            psi = lambda t: phi_f + psi_scan_rate*t


    rhs = np.zeros(4*N+4)

    P = u[0:N+1]
    phi = u[N+1:2*N+2]
    n = u[2*N+2:3*N+3]
    p = u[3*N+3:4*N+4]

    GR = G(Av@x) - R(Av@p)

    mE = Dx@phi
    FP = lam*(Dx@P + mE*(Av@P))
    cd = NN - Lo@P + delta*(Lo@n - chi*Lo@p)
    fn = (Dx@n - mE*(Av@n))
    fp = -(Dx@p + mE*(Av@p))


    rhs[0] = FP[0]
    rhs[1:N] = FP[1:N] - FP[0:N-1]
    rhs[N] = -FP[N-1]

    rhs[N+1] = phi[0] + 0.5*(psi(t) - pbi)
    rhs[N+2:2*N+1] = mE[1:N] - mE[0:N-1] - cd/lam2
    rhs[2*N+1] = phi[-1] - 0.5*(psi(t) - pbi)

    rhs[2*N+2] = n[0] - n_bar
    rhs[2*N+3:3*N+2] = fn[1:N] - fn[0:N-1] + (dx[1:N]*GR[1:N]+dx[0:N-1]*GR
        [0:N-1])/2
    rhs[3*N+2] = -fn[-1] + dx[-1]*GR[-1]/2

    rhs[3*N+3] = -fp[0] + dx[0]*GR[0]/2
    rhs[3*N+4:4*N+3] = -(fp[1:N] - fp[0:N-1]) + (dx[1:N]*GR[1:N]+dx[0:N-1]*
        GR[0:N-1])/2
    rhs[4*N+3] = p[-1] - p_bar

    return rhs
```

A.9 rhs_tlm.py

```
# -*- coding: utf-8 -*-
"""
@author: Nathan

Last Modified 3/1/2021

This script defines the right-hand-side of the drift-diffusion
    semiconductor equations for the three-layer model.
The inputs of this function are:
    t: independent variable matrix
    u: dependent variable matrix
    psi: type of scan
    mode: solution condition

The function outputs the u matrix in the equation
    M*du/dt = u
"""

import numpy as np
import parameters, grid, matrices, generation_recombination

params = parameters.Params()
grid = grid.Grid(params)
mat = matrices.Matrices(params, grid)
genrec = generation_recombination.GR(params, grid)

chi = params.chi
delta = params.delta
kE = params.kE
KE = params.KE
kH = params.kH
KH = params.KH
Kn = params.Kn
Kp = params.Kp
lam = params.lam
lam2 = params.lam2
lamE = params.lamE
lamE2 = params.lamE2
lamH = params.lamH
lamH2 = params.lamH2
N = params.N
NE = params.NE
```

```
NH = params.NH
pbi = params.pbi
rE = params.rE
rH = params.rH

dx = grid.dx
dxE = grid.dxE
dxH = grid.dxH
x = grid.x

Av = mat.Av
AvE = mat.AvE
AvH = mat.AvH
Dx = mat.Dx
DxE = mat.DxE
DxH = mat.DxH
Lo = mat.Lo
LoE = mat.LoE
LoH = mat.LoH
ddE = mat.ddE
ddH = mat.ddH
NN = mat.NN

Rr = genrec.Rr
Rl = genrec.Rl
G = genrec.G
R = genrec.R

phi_precondition = params.phi_precondition
psi_scan_rate = params.psi_scan_rate
phi_f = params.phi_f

def TLM(t, u, *psi, mode=None):

    if ('pbi' in psi):
        psi = lambda t: pbi
    elif ('test1' in psi):
        psi = lambda t: pbi*(1 - (np.tanh(10**6*t)/np.tanh(10**6)))
    elif ('precondition' in psi):
        psi = lambda t: pbi*(1 - t/5) + phi_precondition*(t/5) if t<5 else
            phi_precondition
    elif ('reverse_scan' in psi):
        psi = lambda t: phi_precondition - psi_scan_rate*t
    elif ('forward_scan' in psi):
        psi = lambda t: phi_f + psi_scan_rate*t
```

```
elif ('scan_1' in psi):
    if params.Vi < params.Vf:
        psi = lambda t: phi_precondition + psi_scan_rate*t
    elif params.Vi > params.Vf:
        psi = lambda t: phi_precondition - psi_scan_rate*t
elif ('scan_2' in psi):
    if params.Vi < params.Vf:
        psi = lambda t: phi_f - psi_scan_rate*t
    elif params.Vi > params.Vf:
        psi = lambda t: phi_f + psi_scan_rate*t

# Allocate the residual vector.
rhs = np.zeros(4*N+2*NE+2*NH+4)

P = u[0:N+1]
phi = u[N+1:2*N+2]
n = u[2*N+2:3*N+3]
p = u[3*N+3:4*N+4]
phiE = u[4*N+4:4*N+NE+4]; phiE = np.append(phiE, phi[0])
nE = u[4*N+NE+4:4*N+2*NE+4]; nE = np.append(nE, n[0]/kE)
phiH = u[4*N+2*NE+4:4*N+2*NE+NH+4]; phiH = np.append(phi[-1], phiH)
pH = u[4*N+2*NE+NH+4:4*N+2*NE+2*NH+4]; pH = np.append(p[-1]/kH, pH)

mE = Dx@phi
mEE = DxE@phiE
mEH = DxH@phiH
FP = lam*(Dx@P + mE*(Av@P))
cd = NN - Lo@P + delta*(Lo@n - chi*Lo@p)
cdE = LoE@nE - ddE
cdH = ddH - LoH@pH
fn = Kn*(Dx@n - mE*(Av@n))
fnE = KE*(DxE@nE - mEE*(AvE@nE))
fp = Kp*(Dx@p + mE*(Av@p))
fpH = KH*(DxH@pH + mEH*(AvH@pH))

GR = G(Av@x) - R(Av@n, Av@p, Av@P)

# P equation
rhs[0] = FP[0]
rhs[1:N] = FP[1:N] - FP[0:N-1]
rhs[N] = -FP[N-1]

# phi equation
rhs[N+1] = mE[0] - rE*mEE[-1] - dx[0]*(1/2 - P[0]/3 - P[1]/6 + delta*(n
    [0]/3+n[1]/6 - chi*(p[0]/3 + p[1]/6)))/lam2 - rE*dxE[-1]*(nE[-2]/6+
```

```
        nE[-1]/3 - 1/2)/lamE2
    rhs[N+2:2*N+1] = mE[1:N] - mE[0:N-1] - cd/lam2
    rhs[2*N+1] = rH*mEH[0] - dx[-1]*(1/2 - P[-2]/6 - P[-1]/3 + delta*(n
        [-2]/6 + n[-1]/3 - chi*(p[-2]/6 + p[-1]/3)))/lam2 - rH*dxH[0]*(1/2 -
         pH[0]/3 - pH[1]/6)/lamH2

    # n equation
    rhs[2*N+2] = fn[0] - fnE[-1] - Rl(nE[-1],p[0]) + (dx[0]*GR[0])/2
    rhs[2*N+3:3*N+2] = fn[1:N] - fn[0:N-1] + (dx[1:N]*GR[1:N] + dx[0:N-1]*
        GR[0:N-1])/2
    rhs[3*N+2] = -fn[N-1] - Rr(n[N],pH[0]) + dx[N-1]*GR[-1]/2

    # p equation
    rhs[3*N+3] = fp[0] - Rl(nE[-1],p[0]) + dx[0]*GR[0]/2
    rhs[3*N+4:4*N+3] = fp[1:N] - fp[0:N-1] + (dx[1:N]*GR[1:N] + dx[0:N-1]*
        GR[0:N-1])/2
    rhs[4*N+3] = fpH[0] - fp[-1] - Rr(n[N],pH[0]) + (dx[-1]*GR[-1])/2

    # phiE equation
    rhs[4*N+4] = phiE[0] + 0.5*(psi(t) - pbi)
    rhs[4*N+5:4*N+NE+4] = mEE[1:NE] - mEE[0:NE-1] - cdE/lamE2

    # nE equation
    rhs[4*N+NE+4] = nE[0] - 1
    rhs[4*N+NE+5:4*N+2*NE+4] = fnE[1:NE] - fnE[0:NE-1]

    # phiH equation
    rhs[4*N+2*NE+4:4*N+2*NE+NH+3] = mEH[1:NH] - mEH[0:NH-1] - cdH/lamH2
    rhs[4*N+2*NE+NH+3] = phiH[-1] - 0.5*(psi(t) - pbi)

    # pH equation
    rhs[4*N+2*NE+NH+4:4*N+2*NE+2*NH+3] = fpH[1:NH] - fpH[0:NH-1]
    rhs[4*N+2*NE+2*NH+3] = pH[-1] - 1

    if mode == 'precondition':
        rhs[N] = np.trapz(P, x) - 1

    return rhs
```

A.10 generation_recombination.py

```
# -*- coding: utf-8 -*-
"""
```

```
@author: Nathan

Last modified 3/3/2021

Generation-recombination class. This script organizes all of the generation
    and recombination models into a single class so they can easily be
   accessed in the initial-conditions and
RHS scripts.

The class uses the parameter and grid class objects as the inputs.

Citations:
"""
import numpy as np

class GR():

   def __init__(self, params, grid):

       brate = params.brate
       brateE = params.brateE
       brateH = params.brateH
       gamma = params.gamma
       gammaE = params.gammaE
       gammaH = params.gammaH
       kE = params.kE
       kH = params.kH
       ni2 = params.ni2
       tor = params.tor
       torE = params.torE
       torH = params.torH
       tor3 = params.tor3
       torE3 = params.torE3
       torH3 = params.torH3
       Y = params.Y
       w = params.w
       x = grid.x

       model = params.model

       """ Generation Rates"""
       """ Beer-Lambert Law """
       self.beer = lambda x: Y*np.exp(-Y*x)
       if model == 'beer-lambert':
           self.G = self.beer
           self.G_init = self.G
```

```
    """ Transfer Matrix Optical Model """
    from transfermatrix import transfer_matrix
    from scipy.optimize import least_squares

    if model == 'transfer-matrix':
        self.G_dim, self.Jsc = transfer_matrix()
        self.G = lambda x: np.interp(x, grid.x, self.G_dim/params.G0)
        Y_func = lambda Y: Y*np.exp(-Y*grid.x) - self.G_dim/params.G0
        self.Y_init = least_squares(Y_func, 5).x
        self.G_init = lambda x: self.Y_init*np.exp(-self.Y_init*x)

    """ Recombination Rates"""
    """ Mono-molecular Recombination """
    self.mm_recomb = lambda p: w*p

    """ Bi-molecular Recombination """
    self.bm_recomb = lambda n, p: brate*(n*p - ni2)

    """ Schockley-Read-Hall (SRH) Recombination """
    self.SRH = lambda n, p, gamma, ni2, tor, tor3: \
        gamma*(p - ni2/n)/(1 + tor*p/n + tor3/n)*(n>=tor*p)*(n>=tor3) \
            + gamma*(p - ni2/p)/(n/p + tor + tor3/p)*(tor*p>n)*(tor*p>
                tor3) \
                + gamma*(p*n - ni2)/(n + tor*p + tor3)*(tor3>n)*(tor3>=
                    tor*p)

    """ Total Recombination """
    self.R = lambda n, p, P: self.bm_recomb(n, p) + self.SRH(n, p, gamma
        , ni2, tor, tor3)

    """ Total Interfacial Recombination """
    self.Rl = lambda nE, p: brateE*(nE*p - ni2/kE) + self.SRH(nE, p,
        gammaE, ni2/kE, torE, torE3)
    self.Rr = lambda n, pH: brateH*(n*pH - ni2/kH) + self.SRH(pH, n,
        gammaH, ni2/kH, torH, torH3)

    self.GR = lambda n, p, P: self.G(x) - self.R(n, p, P)
```

A.11 transfer_matrix.py

```
# -*- coding: utf-8 -*-
"""
```

```
@author: Nathan

Last modified 4/29/2020

This script defines the transfer matrix function. It also uses the Pandas
    library to import complex refractive index data used in the calculation
    .
The function outputs are:
    Jsc: short-circuit current
    gen: generation rate profile

This code was adapted from the Matlab code by George F. Burkhard, Eric T.
    Hoke, Stanford University. 2010

Note: This was my first attempt at trying to port code from Matlab, so it
    may seem disorganized. However, about 3/4 of the way through finishing
    it I found that it had already been done!
The reference is by Kamil Mielczarek, University of Texas at Dallas.

Citations:
"""

import numpy as np
import numpy.matlib
import pandas as pd
import matplotlib.pyplot as plt

import parameters, grid


params = parameters.Params()
grid = grid.Grid(params)

def transfer_matrix():
    thickness_ito = 80
    thickness_ETL = params.bE*1e9
    thickness_perovskite = params.b*1e9
    thickness_HTL = params.bH*1e9

    # Wavelength range inputs.
    lower_bound = 350
    upper_bound = 800
    wavelength = np.array(range(lower_bound,upper_bound + 1))

    # Electric field is calculated in discrete positions.
    step_size = 1
```

```
layers = ['ITO', 'TiO2', 'perovskite', 'Spiro-OMeTAD']
thicknesses = [thickness_ito, thickness_ETL, thickness_perovskite,
    thickness_HTL]
activelayer = 3

h = 6.62606957e-34
c = 2.99792458e8

# ----------------------Data Input

# Use pandas to obtain data from Excel file.
datafile = pd.read_excel('Complex_Refractive_Index_Data.xlsx')

# Add data designation here from the Excel file. The column names must
    be input to the bracketed portion of datafile.
# The program will fail if column names are mispelled.
data_wl_air = datafile['Air_wavelength'].values
data_n_air = datafile['Air_n'].values
data_k_air = datafile['Air_k'].values

data_wl_ITO = datafile['ITO_wavelength']
data_n_ITO = datafile['ITO_n']
data_k_ITO = datafile['ITO_k']

data_wl_TiO2 = datafile['TiO2_wavelength']
data_n_TiO2 = datafile['TiO2_n']
data_k_TiO2 = datafile['TiO2_k']

data_wl_MAPbI3 = datafile['MAPbI3_wavelength'].values
data_n_MAPbI3 = datafile['MAPbI3_n'].values
data_k_MAPbI3 = datafile['MAPbI3_k'].values

data_wl_spiro = datafile['Spiro_wavelength']
data_n_spiro = datafile['Spiro_n']
data_k_spiro = datafile['Spiro_k']

#------------------Data Processing
# Function to interpolate n and k data. Adds both values to produce
    total complex index of reflection.
# Uses raw data input from Excel file.
def interp_nk(wavelength_input, wavelength_data, n_data, k_data):
    wl_int = wavelength_data
    n_int = n_data
    k_int = k_data
```

```
    n = np.interp(wavelength_input, wl_int, n_int)
    k = np.interp(wavelength_input, wl_int, k_int)

    return n + 1j*k

# Complex refractive index for each material at the discrete values
    given in the wavelength range.
n_air = interp_nk(wavelength, data_wl_air, data_n_air, data_k_air)
n_ITO = interp_nk(wavelength, data_wl_ITO, data_n_ITO, data_k_ITO)
n_TiO2 = interp_nk(wavelength, data_wl_TiO2, data_n_TiO2, data_k_TiO2)
n_MAPbI3 = interp_nk(wavelength, data_wl_MAPbI3, data_n_MAPbI3,
    data_k_MAPbI3)
n_spiro = interp_nk(wavelength, data_wl_spiro, data_n_spiro,
    data_k_spiro)

n_matrix_dimensions = (len(layers), len(wavelength))
n = np.zeros(n_matrix_dimensions, dtype=complex)

# Input data into empty n matrix.
for index in range(0, len(n_ITO)):
    n[0,index] = n_ITO[index]
for index in range(0, len(n_TiO2)):
    n[1,index] = n_TiO2[index]
for index in range(0, len(n_MAPbI3)):
    n[2,index] = n_MAPbI3[index]
for index in range(0, len(n_spiro)):
    n[3,index] = n_spiro[index]

""" Transfer Matrix Optical Model """
""" Calculate incoherent power transmission through ITO substrate. """
T_ITO = np.abs(np.divide(4*n[0,:],(np.power((1 + n[0,:]),2))))
R_ITO = np.power(np.abs(np.divide(1 - n[0,:],1 + n[0,:])),2)

t = thicknesses
t[0] = 0
t_cumsum = np.cumsum(t)

x_pos = np.arange(step_size/2,sum(t),step_size)
#x_mat uses an inequality to produce boolean values. The remaining
    operations create a matrix that describes what layer number the
    corresponding point in x_pos is.
x_mat = sum(np.tile(x_pos,[len(t),1])>np.transpose(np.tile(np.transpose
    (t_cumsum),[len(x_pos),1])),1)
R = wavelength*0.0
T = np.ones(len(wavelength))*0.0
```

```
E_mat_dimensions = (len(x_pos),len(wavelength))
E = np.zeros(E_mat_dimensions, dtype=complex)

""" Define functions for I and L matrices. """
def I(n1,n2):
    r = (n1 - n2)/(n1 + n2)
    t = 2*n1/(n1 + n2)
    return [[1,r],[r,1]]/t

def L(n,d,wavelength):
    xi = 2*np.pi*n/wavelength
    return [[np.exp(-1j*xi*d),0],[0,np.exp(1j*xi*d)]]


#----------Construction of transfer matrices
print('Calculating␣Transfer␣Matrices')
for l in range(0,len(wavelength)):
    S = I(n[0,l],n[1,l])
    for matindex in range(2,len(t)):
        S = np.asarray(np.mat(S)*np.mat(L(n[matindex-1,l],t[matindex-1],
            wavelength[l]))*np.mat(I(n[matindex-1,l],n[matindex,l])))
    R[l] = np.abs(S[1,0]/S[0,0])**2
    T[l] = np.abs(2/(1+n[0,l]))/np.sqrt(1-R_ITO[l]*R[l])
    for material in range(1,len(t)):
        xi = 2*np.pi*n[material,l]/wavelength[l]
        dj = t[material]
        x_indices = np.argwhere(x_mat == material+1)
        x = x_pos[x_indices] - t_cumsum[material-1]
        S_prime = I(n[0,l],n[1,l])
        for matindex in range(2,material+1):
            S_prime = np.asarray(np.mat(S_prime)*np.mat(L(n[matindex-1,l
                ],t[matindex-1],wavelength[l]))*np.mat((I(n[matindex-1,l
                ],n[matindex,l]))))
        S_double_prime = np.eye(2)
        for matindex in range(material,len(t)-1):
            S_double_prime = np.asarray(np.mat(S_double_prime)*np.mat(I(n
                [matindex,l],n[matindex+1,l]))*np.mat(L(n[matindex+1,l],t
                [matindex+1],wavelength[l])))
        E[x_indices,l] = T[l]*(np.divide((S_double_prime[0,0]*np.exp(
            complex(0,-1.0)*xi*(dj - x)) + S_double_prime[1,0]*np.exp(
            complex(0,1.0)*xi*(dj - x))), S_prime[0,0]*S_double_prime
            [0,0]*np.exp(complex(0,-1.0)*xi*dj) + S_prime[0,1]*
            S_double_prime[1,0]*np.exp(complex(0,1.0)*xi*dj)))

#----------Generation Equations
def interp_1sun(wavelength_input, wavelength_data, sun_data):
```

```
    wl_int = wavelength_data
    sun_int = sun_data
    solar_spectrum = np.interp(wavelength_input, wl_int, sun_int)
    return solar_spectrum

sun_datafile = pd.read_excel('Sun_Spectrum.xlsx')
data_wl_sun = sun_datafile['Wavelength␣(nm)'].values
data_sun_spectrum = sun_datafile['Global␣tilt␣␣mW*cm-2*nm-1␣(1sun␣AM␣
    1.5)']

sun_mat = interp_1sun(wavelength, data_wl_sun, data_sun_spectrum)

a_mat_dimensions = (len(t),len(wavelength))
a = np.zeros(a_mat_dimensions)

for i in range(1,len(t)):
    a[i,:] = (4*np.pi*np.imag(n[i,:]))/(wavelength*1.0e-7)


activepos = np.argwhere(x_mat == activelayer)

Q = np.tile(a[activelayer-1,:]*np.real(n[activelayer-1,:])*sun_mat,(np.
    size(activepos),1))*(abs(E[activepos,:])**2)


G = (Q*1.0e-3)*np.tile(wavelength*1.0e-9,(np.size(activepos),1))/(h*c)

if len(wavelength) == 1:
    wl_step = 1

else:
    wl_step = (sorted(wavelength)[-1]-sorted(wavelength)[0])/(len(
        wavelength)-1)

Gx = np.sum(G,2)*wl_step

x_non_dim = np.append(np.append([0.0], (x_pos[activepos] -
    thickness_ETL)*1e-9/params.b), [1.0])
G_total = np.append(np.append(Gx[0,0], Gx[:,0]), Gx[-1,0])*1e6 # m3s-1
gen = np.interp(grid.x, x_non_dim, G_total)

Jsc = sum(Gx)*step_size*1e-7*params.q*1e3; Jsc = Jsc[0]
return gen, Jsc
```

A.12 initial_conditions.py

```
# -*- coding: utf-8 -*-
"""
@author: Nathan

Last modified 3/3/2021

Initial conditions class used for the single-layer model. A 4*N+4 column
    vector is calculated.
The class uses parameters and spatial grid class objects to define the
    vector.

Citations:


"""

import numpy as np
import parameters, grid, generation_recombination, jacobian_tlm
from rhs_tlm import TLM
from scipy.integrate import solve_bvp
from scipy.optimize import least_squares


""" Create class objects used in quasi-steady state functions. """
params = parameters.Params()
grid = grid.Grid(params)
jac = jacobian_tlm.Jac(params)
genrec = generation_recombination.GR(params, grid)

class Initial_Conditions():

    def __init__(self, params, grid):

        """ Define class variables from constructor class objects. """
        chi = params.chi
        kE = params.kE
        kH = params.kH
        N = params.N
        NE = params.NE
        NH = params.NH
        n_bar = params.n_bar
        p_bar = params.p_bar
```

```
        x = grid.x

        """ Define initial charge densities and electric potential. """
        P0 = np.ones(N+1)
        phi0 = np.zeros(N+1)
        n0 = p_bar*x + n_bar*(1 - x)
        p0 = n0

        """ Define instance variable for single layer model. """
        self.sol_init_slm = np.concatenate((P0, phi0, n0, p0), axis=0)


        """ Compute profiles for carrier concentrations from quasi-steady
            state boundary value problem. """
        y_guess_eqn = lambda x: [kH*x+kE*(1-x)/chi, 0.0*x, chi*kH*x+kE*(1-x)
            , 0.0*x]
        y_guess = y_guess_eqn(x)
        sol = solve_bvp(yode, ybcs, x, y_guess)
        p_init = sol.y[0]
        n_init = sol.y[2]

        """ Uniform carrier concentrations and electric potential for
            transport layers. """
        phiE_init = np.zeros(NE)
        nE_init = np.ones(NE)
        phiH_init = np.zeros(NH)
        pH_init = np.ones(NH)

        """ Concatenate initial concentrations. Create instance variables
            for three-layer model. """
        self.u0 = np.concatenate((P0, phi0, n_init, p_init, phiE_init,
            nE_init, phiH_init, pH_init), 0)

        """ Use fsolve to find steady-state solution at built in voltage.
            """
        print('Calculating␣consistent␣initial␣conditions.')
        jac_sparsity = jac.jac
        self.sol_init_tlm = least_squares(lambda u: TLM(0, u, 'pbi'), self.
            u0, jac_sparsity = jac_sparsity).x
        print('Complete')


""" Quasi-steady state functions used in the above class. """
def yode(x, y):
    G = genrec.G_init
```

```
    R = genrec.R
    Kp = params.Kp
    Kn = params.Kn


    dpdx = [-y[1]/Kp, G(x)-R(y[2],y[0],1), y[3]/Kn, -(G(x)-R(y[2],y[0],1))]
    return dpdx

def ybcs(ya, yb):
    kH = params.kH
    Rl = genrec.Rl
    kE = params.kE
    Rr = genrec.Rr

    res = [yb[0]-kH, ya[2]+Rl(1,ya[0]), ya[2]-kE, yb[3]+Rr(yb[2],1)]
    return res
```

Appendix B

Solver Python Code

B.1 initialize.py

```
# -*- coding: utf-8 -*-
"""
@author: Nathan
"""
def initialize():
    import parameters, grid, matrices, generation_recombination

    params = parameters.Params()
    grid = grid.Grid(params)
    mat = matrices.Matrices(params, grid)
    genrec = generation_recombination.GR(params, grid)

    return params, grid, mat, genrec

from JV import JV
from dimensionalize import dimensionalize_slm, dimensionalize_tlm

import parameters, grid, matrices, generation_recombination

params = parameters.Params()
grid = grid.Grid(params)
mat = matrices.Matrices(params, grid)
genrec = generation_recombination.GR(params, grid)

initialize()
```

B.2 JV.py

```
# -*- coding: utf-8 -*-
"""
@author: Nathan

Last modified 3/16/2021

This script defines the solution procedures for JV scans. The function
    takes the input 'single-layer' or 'three-layer' to define which model
    is used.
Both functions output the non-dimensional solution matrices: t_non_dim,
    u_matrix, psi
and dimensional solution matrices: x, t, P, phi, n, p , J_total, V_applied

The non-dimensional JV scan is automatically plotted. The remaining
    solutions can be dimensionalized and plotted using the dimensionalize
    and plot scripts in the command window.
The solutions can be accessed through the variable explorer.
"""

from rhs_slm import SLM
from rhs_tlm import TLM
from total_current import total_current_slm, total_current_tlm
from scipy.optimize import least_squares
from scipy.integrate import solve_ivp
from radauDAE import RadauDAE
from plot import plot_JV
from dimensionalize import dimensionalize_slm, dimensionalize_tlm

import numpy as np
import matplotlib.pyplot as plt

import parameters, grid, initial_conditions, mass_slm, jacobian_slm,
    mass_tlm, jacobian_tlm

import time

""" Common class objects used by both JV functions. """
params = parameters.Params()
grid = grid.Grid(params)
ic = initial_conditions.Initial_Conditions(params, grid)

""" Common solver settings used by both JV functions. """
tf = 1
```

```
method = RadauDAE
rtol = params.rtol
atol = params.atol

def JV(*model):

    if ('single-layer' in model):

        """ Create class objects, mass matrix, and jacobian under standard
            conditions. """
        mass = mass_slm.Mass(params, grid)
        jac = jacobian_slm.Jac(params)

        mass = mass.M
        jac = jac.jac

        """ Solution process for JV scan.
        - Dense output is only used for the scanning solution procedures.
        """

        sol_init = ic.sol_init_slm
        dae_fun = lambda t, u: SLM(t, u, 'pbi')

        print('Eliminating␣transient␣behavior␣at␣built-in␣voltage.')
        start_time = time.time()
        sol = solve_ivp(fun=dae_fun, t_span=(0.0, tf), y0=sol_init,
                        rtol=rtol, atol=atol, jac_sparsity=jac,
                        method=method, vectorized=False, dense_output=
                            False,
                        mass=mass)

        print('Preconditioning␣device␣to␣Vi.')
        dae_fun = lambda t, u: SLM(t, u, 'precondition')
        sol_init = sol.y[:,-1]

        sol = solve_ivp(fun=dae_fun, t_span=(0.0, 10*tf), y0=sol_init,
                        rtol=rtol, atol=atol, jac_sparsity=jac,
                        method=method, vectorized=False, dense_output=
                            False,
                        mass=mass)

        sol_init = sol.y[:,-1]
        tf_scan = params.tf_scan
        dae_fun = lambda t, u: SLM(t, u, 'scan_1')

        print('Beginning␣JV␣scan.')
```

```
sol = solve_ivp(fun=dae_fun, t_span=(0.0, tf_scan), y0=sol_init,
                  rtol=rtol, atol=atol, jac_sparsity=jac,
                  method=method, vectorized=False, dense_output=True
                      ,
                  mass=mass)

t_vector_1 = sol.t
u_matrix_1 = sol.y

""" These conditions allow for the correct calculations for psi
    depending if the scan begins reverse or forward. """
if params.Vi < params.Vf:
    psi_1 = params.phi_precondition*np.ones(t_vector_1.size) +
        params.psi_scan_rate*t_vector_1
elif params.Vi > params.Vf:
    psi_1 = params.phi_precondition*np.ones(t_vector_1.size) -
        params.psi_scan_rate*t_vector_1

J_reverse = total_current_slm(t_vector_1, u_matrix_1)[0]


sol_init = sol.y[:,-1]
dae_fun = lambda t, u: SLM(t, u, 'scan_2')
print('Scanning␣opposite␣direction.')
sol = solve_ivp(fun=dae_fun, t_span=(0.0, tf_scan), y0=sol_init,
                  rtol=rtol, atol=atol, jac_sparsity=jac,
                  method=method, vectorized=False, dense_output=True
                      ,
                  mass=mass)
print('Scan␣Complete')

t_vector_2 = sol.t
u_matrix_2 = sol.y

if params.Vi < params.Vf:
    psi_2 = params.phi_f*np.ones(t_vector_2.size) - params.
        psi_scan_rate*t_vector_2
elif params.Vi > params.Vf:
    psi_2 = params.phi_f*np.ones(t_vector_2.size) + params.
        psi_scan_rate*t_vector_2

J_forward = total_current_slm(t_vector_2, u_matrix_2)[0]

""" Concatenate t_vectors and u_matrices into complete data
    structures for both forward and backward scans. """
```

```
    t_non_dim = np.concatenate((t_vector_1, t_vector_1[-1] + t_vector_2)
        , 0)
    u_matrix = np.concatenate((u_matrix_1, u_matrix_2), 1)
    psi = np.concatenate((psi_1, psi_2), 0)

    """ Dimensionalize results. """
    dimensional_sol = dimensionalize_slm(grid.x, t_non_dim, u_matrix,
        psi)

    x = dimensional_sol[0]
    t = dimensional_sol[1]
    P = dimensional_sol[2]
    phi = dimensional_sol[3]
    n = dimensional_sol[4]
    p = dimensional_sol[5]
    J_total = dimensional_sol[6]
    V_applied = dimensional_sol[7]

    """ Plot results. """
    plot_JV(V_applied, J_total, 'dimensional')

    scan1_size = psi_1.size
    scan2_size = psi_2.size

    """""
    plt.plot(psi_1, J_reverse)
    plt.plot(psi_2, J_forward)
    plt.xlim([0,40])
    plt.ylim([0,0.8])
    """

    print("---␣%s␣seconds␣---" % (time.time() - start_time))




elif ('three-layer' in model):

    """ Create class objects, mass matrix, and jacobian under standard
        conditions. """
    mass = mass_tlm.Mass(params, grid, mode=None)
    jac = jacobian_tlm.Jac(params, mode=None)

    mass = mass.M
    jac = jac.jac
```

```
""" Solution process for JV scan.
- Dense output is only used for the scanning solution procedures.
"""
sol_init = ic.sol_init_tlm
dae_fun = lambda t, u: TLM(t, u, 'pbi', mode=None)

start_time = time.time()
print('Eliminating␣transient␣behavior␣at␣built-in␣voltage.')
sol = solve_ivp(fun=dae_fun, t_span=(0.0, tf), y0=sol_init,
                rtol=rtol, atol=atol, jac_sparsity=jac,
                method=method, vectorized=False, dense_output=
                    False,
                mass=mass)

dae_fun = lambda t, u: TLM(t, u, 'precondition', mode=None)
sol_init = sol.y[:,-1]

""" Adjust class objects to increase ion mobility to precondition at
    a voltage other than Vbi. """

mass = mass_tlm.Mass(params, grid, mode='precondition')
jac = jacobian_tlm.Jac(params, mode=None)

mass = mass.M
jac = jac.jac

print('Preconditioning␣device␣to␣Vi.')
sol = solve_ivp(fun=dae_fun, t_span=(0.0, 10*tf), y0=sol_init,
                rtol=rtol, atol=atol, jac_sparsity=jac,
                method=method, vectorized=False, dense_output=
                    False,
                mass=mass)

sol_precondition = sol.y[:,-1]
tf_precondition = sol.t[-1]

""" Redefine jacobian to ensure conservation of ions for steady
   state solution. """
jac = jacobian_tlm.Jac(params, mode='precondition')
jac = jac.jac

if params.Vi < params.Vbi:
   sol_init = least_squares(lambda u: TLM(tf_precondition, u, '
       precondition', mode='precondition'), sol_precondition,
       jac_sparsity=jac).x
```

```
else:
    sol_init = sol.y[:,-1]

""" Initial conditions for the JV scan are now calculated. Calculate
     non-dimensional scan time and reset class objects to normal
    conditions. """
tf_scan = params.tf_scan
dae_fun = lambda t, u: TLM(t, u, 'scan_1', mode=None)

mass = mass_tlm.Mass(params, grid, mode=None)
jac = jacobian_tlm.Jac(params, mode=None)

mass = mass.M
jac = jac.jac

print('Beginning␣JV␣scan.')
sol = solve_ivp(fun=dae_fun, t_span=(0.0, tf_scan), y0=sol_init,
                 rtol=rtol, atol=atol, jac_sparsity=jac,
                 method=method, vectorized=False, dense_output=True
                     ,
                 mass=mass)

t_vector_1 = sol.t
u_matrix_1 = sol.y

""" These conditions allow for the correct calculations for psi
    depending if the scan begins reverse or forward. """
if params.Vi < params.Vf:
    psi_1 = params.phi_precondition*np.ones(t_vector_1.size) +
        params.psi_scan_rate*t_vector_1
elif params.Vi > params.Vf:
    psi_1 = params.phi_precondition*np.ones(t_vector_1.size) -
        params.psi_scan_rate*t_vector_1

sol_init = sol.y[:,-1]
dae_fun = lambda t, u: TLM(t, u, 'scan_2', mode=None)
print('Scanning␣opposite␣direction.')
sol = solve_ivp(fun=dae_fun, t_span=(0.0, tf_scan), y0=sol_init,
                 rtol=rtol, atol=atol, jac_sparsity=jac,
                 method=method, vectorized=False, dense_output=True
                     ,
                 mass=mass)
print('Scan␣Complete')
print("---␣%s␣seconds␣---" % (time.time() - start_time))
```

```
        t_vector_2 = sol.t
        u_matrix_2 = sol.y

        if params.Vi < params.Vf:
            psi_2 = params.phi_f*np.ones(t_vector_2.size) - params.
                psi_scan_rate*t_vector_2
        elif params.Vi > params.Vf:
            psi_2 = params.phi_f*np.ones(t_vector_2.size) + params.
                psi_scan_rate*t_vector_2

        """ Concatenate t_vectors and u_matrices into complete data
            structures for both forward and backward scans. """
        t_non_dim = np.concatenate((t_vector_1, t_vector_1[-1] + t_vector_2)
            , 0)
        u_matrix = np.concatenate((u_matrix_1, u_matrix_2), 1)
        psi = np.concatenate((psi_1, psi_2), 0)

        """ Dimensionalize results. """
        dimensional_sol = dimensionalize_tlm(grid.x, grid.xE, grid.xH,
            t_non_dim, u_matrix, psi)

        x = dimensional_sol[0]
        t = dimensional_sol[1]
        P = dimensional_sol[2]
        phi = dimensional_sol[3]
        n = dimensional_sol[4]
        p = dimensional_sol[5]
        J_total = dimensional_sol[6]
        V_applied = dimensional_sol[7]

        """ Plot results. """
        plot_JV(V_applied, J_total, 'dimensional')

        scan1_size = psi_1.size
        scan2_size = psi_2.size

    return t_non_dim, u_matrix, psi, x, t, P, phi, n, p , J_total,
        V_applied, scan1_size, scan2_size
```

B.3 slm_solver.py

```
# -*- coding: utf-8 -*-
"""
```

```
@author: Nathan

Last Modified 2/23/2021


"""
from rhs_slm import SLM
from scipy.integrate import solve_ivp
from radauDAE import RadauDAE
from plot import plot_distributions_slm
from total_current import total_current_slm
import matplotlib.pyplot as plt
import parameters, grid, initial_conditions, mass_slm, jacobian_slm

import time
start_time = time.time()

params = parameters.Params()
grid = grid.Grid(params)
ic = initial_conditions.Initial_Conditions(params, grid)
mass = mass_slm.Mass(params, grid)
jac = jacobian_slm.Jac(params)

sol_init = ic.sol_init_slm
mass = mass.M
jac = jac.jac

tf = 1
method = RadauDAE
rtol = params.rtol
atol = params.atol

dae_fun = lambda t, u: SLM(t, u, 'pbi')

sol = solve_ivp(fun=dae_fun, t_span=(0.0, tf), y0=sol_init,
                  rtol=rtol, atol=atol, jac_sparsity=jac,
                  method=method, vectorized=False, dense_output=False,
                  mass=mass)

"""
dae_fun = lambda t, u: SLM(t, u, 'precondition')
sol_init = sol.y[:,-1]

sol = solve_ivp(fun=dae_fun, t_span=(0.0, tf), y0=sol_init,
                  rtol=rtol, atol=atol, jac_sparsity=jac,
                  method=method, vectorized=False, dense_output=True,
```

```
                  mass=mass)
"""
sol_init = sol.y[:,-1]
dae_fun = lambda t, u: SLM(t, u, 'test1')

sol = solve_ivp(fun=dae_fun, t_span=(0.0, 1), y0=sol_init,
                  rtol=rtol, atol=atol, jac_sparsity=jac,
                  method=method, vectorized=False, dense_output=True,
                  mass=mass)

t = sol.t
u = sol.y

plot_distributions_slm(grid.x, u)
J = total_current_slm(t, u)

plt.figure(5)
plt.plot(t, J[0])
plt.xlim([0,5*10e-7])
plt.ylim([0,1.5])
plt.xlabel('Time')
plt.ylabel('Current Density')
plt.title("Ref Figure 5")


n = u[2*params.N+2:3*params.N+3,-1]

plt.figure(6)
plt.plot(grid.x, n)
plt.xlim([0,1])
plt.ylim([0,0.06])
plt.xlabel('Distance, x')
plt.ylabel('Electron Concentration, n')
plt.title("Ref Figure 6")

print("--- %s seconds ---" % (time.time() - start_time))
```

B.4 tlm_solver.py

```
# -*- coding: utf-8 -*-
"""
@author: Nathan
"""
```

```
from rhs_tlm import TLM
from plot import plot_distributions_tlm
from total_current import total_current_tlm
from scipy.integrate import solve_ivp
from radauDAE import RadauDAE
from matplotlib import pyplot as plt

import numpy as np
import parameters, grid, initial_conditions, mass_tlm, jacobian_tlm

import time


params = parameters.Params()
grid = grid.Grid(params)
ic = initial_conditions.Initial_Conditions(params, grid)
mass = mass_tlm.Mass(params, grid)
jac = jacobian_tlm.Jac(params)

sol_init = ic.sol_init_tlm
mass = mass.M
jac = jac.jac

tf = 1
method = RadauDAE
rtol = params.rtol
atol = params.atol

dae_fun = lambda t, u: TLM(t, u, 'pbi')
start_time = time.time()

sol = solve_ivp(fun=dae_fun, t_span=(0.0, tf), y0=sol_init,
                  rtol=rtol, atol=atol, jac_sparsity=jac,
                  method=method, vectorized=False, dense_output=False,
                  mass=mass)

t = sol.t
dae_fun = lambda t, u: TLM(t, u, 'test1')
sol_init = sol.y[:,-1]


sol = solve_ivp(fun=dae_fun, t_span=(0.0, 10*tf), y0=sol_init,
                  rtol=rtol, atol=atol, jac_sparsity=jac,
                  method=method, vectorized=False, dense_output=True,
                  mass=mass)
```

```
sol_init = sol.y[:,-1]
tf_reverse_scan = params.tf_scan
dae_fun = lambda t, u: TLM(t, u, 'reverse_scan')

sol = solve_ivp(fun=dae_fun, t_span=(0.0, tf_reverse_scan), y0=sol_init,
                  rtol=rtol, atol=atol, jac_sparsity=jac,
                  method=method, vectorized=False, dense_output=True,
                  mass=mass)

t_reverse = sol.t
u_reverse = sol.y

psi_applied_r = params.phi_precondition*np.ones(t_reverse.size) - params.
    psi_scan_rate*t_reverse
J_reverse = total_current_tlm(t_reverse, u_reverse)[0]


sol_init = sol.y[:,-1]
tf_forward_scan = params.tf_scan
dae_fun = lambda t, u: TLM(t, u, 'forward_scan')

sol = solve_ivp(fun=dae_fun, t_span=(0.0, tf_forward_scan), y0=sol_init,
                  rtol=rtol, atol=atol, jac_sparsity=jac,
                  method=method, vectorized=False, dense_output=True,
                  mass=mass)

t_forward = sol.t
u_forward = sol.y

psi_applied_f = params.phi_f*np.ones(t_forward.size) + params.psi_scan_rate
    *t_forward
J_forward = total_current_tlm(t_forward, u_forward)[0]

J_nondim = np.concatenate((J_reverse, J_forward), 0)
psi_scan = np.concatenate((psi_applied_r, psi_applied_f), 0)

u_matrix = np.concatenate((u_reverse, u_forward), 1)
t_vector = np.concatenate((t_reverse, t_forward), 0)

u = sol.y
t = sol.t
x = grid.x
xE = grid.xE
xH = grid.xH
```

```
plot_distributions_tlm(x, xE, xH, u[:,-1])

N = params.N
x = grid.x
"""
P = u[0:N+1,-1]
phi = u[N+1:2*N+2,-1]
n = u[2*N+2:3*N+3,-1]
p = u[3*N+3:4*N+4,-1]




plt.figure(0)
plt.plot(x,P)
plt.figure(1)
plt.plot(x,phi)
plt.figure(2)
plt.plot(x,n)
plt.figure(3)
plt.plot(x,p)

plt.figure(4)
plt.plot(psi_applied_r, J_reverse)
#plt.plot(psi_applied_f, J_forward)
plt.xlim([0,40])
plt.ylim([-0.2,0.8])
"""
print("---␣%s␣seconds␣---" % (time.time() - start_time))
```

B.5 total_current.py

```
# -*- coding: utf-8 -*-
"""
@author: Nathan

Last modified 2/24/2021

This script defines two functions for the single-layer and three-layer
    models that calculate the total current-density from the non-
    dimensional solutions.
```

```
Citations:
"""
import numpy as np
import parameters, grid, matrices

params = parameters.Params()
grid = grid.Grid(params)
mat = matrices.Matrices(params, grid)


b = params.b
delta = params.delta
DI = params.DI
epsp = params.epsp
G0 = params.G0
Kn = params.Kn
Kp = params.Kp
lam = params.lam
lam2 = params.lam2
N = params.N
NE = params.NE
NH = params.NH
N0 = params.N0
q = params.q
sigma = params.sigma
Tion = params.Tion
Vt = params.Vt

dx = grid.dx

Av = mat.Av
Dx = mat.Dx

def total_current_slm(t_vector, u_matrix, *model):

    P = u_matrix[0:N+1,:]
    phi = u_matrix[N+1:2*N+2,:]
    n = u_matrix[2*N+2:3*N+3,:]
    p = u_matrix[3*N+3:4*N+4,:]

    mE_data = np.zeros((N, t_vector.size))
    dd_data = np.zeros((N, t_vector.size))
    FP_data = np.zeros((N, t_vector.size))
    fn_data = np.zeros((N, t_vector.size))
    fp_data = np.zeros((N, t_vector.size))
```

```
    dt = np.append([0.0], np.diff(t_vector))

    for i in range(0, t_vector.size):

        mE = Dx@phi[:,i]
        dd = mE*dt[i]
        FP = lam*(Dx@P[:,i] + mE*(Av@P[:,i]))
        fn = (Dx@n[:,i] - mE*(Av@n[:,i]))
        fp = -(Dx@p[:,i] + mE*(Av@p[:,i]))

        mE_data[:,i] = mE
        dd_data[:,i] = dd
        FP_data[:,i] = FP
        fn_data[:,i] = fn
        fp_data[:,i] = fp

    J_data = fn_data + fp_data - (sigma*lam2/delta)*dd_data + (sigma*lam/
        delta)*FP_data

    midpoint = int(np.ceil((N+1)/2))
    J_total = J_data[midpoint,:]

    return J_total, J_data, FP_data, dd_data, fn_data, fp_data

def total_current_tlm(t_vector, u_matrix, *model):

    P = u_matrix[0:N+1,:]
    phi = u_matrix[N+1:2*N+2,:]
    n = u_matrix[2*N+2:3*N+3,:]
    p = u_matrix[3*N+3:4*N+4,:]

    mE_data = np.zeros((N, t_vector.size))
    dd_data = np.zeros((N, t_vector.size))
    FP_data = np.zeros((N, t_vector.size))
    fn_data = np.zeros((N, t_vector.size))
    fp_data = np.zeros((N, t_vector.size))

    dt = np.append([0.0], np.diff(t_vector))

    for i in range(0, t_vector.size):
        mE = Dx@phi[:,i]
        dd = mE*dt[i]
        FP = lam*(Dx@P[:,i] + mE*(Av@P[:,i]))
        fn = Kn*(Dx@n[:,i] - mE*(Av@n[:,i]))
        fp = -Kp*(Dx@p[:,i] + mE*(Av@p[:,i]))
```

```
        mE_data[:,i] = mE
        dd_data[:,i] = dd
        FP_data[:,i] = FP
        fn_data[:,i] = fn
        fp_data[:,i] = fp

    J_data = fn_data + fp_data - (epsp*Vt/(q*G0*b**2*Tion))*dd_data + (DI*
        N0/(G0*b**2))*FP_data

    midpoint = int(np.ceil((N+1)/2))
    J_total = J_data[midpoint,:]

    return J_total, J_data, FP_data, dd_data, fn_data, fp_data
```

Appendix C

Analysis Python Code

C.1 dimensionalize.py

```
# -*- coding: utf-8 -*-
"""
@author: Nathan
Last modified 3/13/2021 - stable with accurate results

A script that defines two functions (for single-layer and three-layer
    models) that dimensionalize the solution outputs from the DAE solver.
    Additionally, both functions dimensionalize
the current-densities used to plot the JV scans.

The function outputs are: x, t, P, phi, n, p, J_total, V_scan

Citations:

"""

import parameters
import numpy as np

from total_current import total_current_slm, total_current_tlm

""" Define parameters class object and declare parameters. """
params = parameters.Params()

b = params.b
dE = params.dE
dH = params.dH
DE = params.DE
DI = params.DI
```

```
epsp = params.epsp
Fph = params.Fph
G0 = params.G0
LD = params.LD
N = params.N
NE = params.NE
NH = params.NH
N0 = params.N0
q = params.q
Tion = params.Tion
Vt = params.Vt

""" Single-layer model dimensionalization. """
def dimensionalize_slm(x_vector, t_vector, u_matrix, psi_vector):
    """ The variable arrays are 2-dimensional to account for dense output.
        """
    P = u_matrix[0:N+1,:]
    phi = u_matrix[N+1:2*N+2,:]
    n = u_matrix[2*N+2:3*N+3,:]
    p = u_matrix[3*N+3:4*N+4,:]

    x = x_vector*b
    t = t_vector*(LD*b/DI)

    P = N0*P
    phi = Vt*phi
    n = (Fph*b/DE)*n
    p = (Fph*b/DE)*p


    """ Non-Dimensional Current Density """
    FP_data = total_current_slm(t_vector, u_matrix)[2]
    dd_data = total_current_slm(t_vector, u_matrix)[3]
    fn_data = total_current_slm(t_vector, u_matrix)[4]
    fp_data = total_current_slm(t_vector, u_matrix)[5]

    J_dimensional = q*G0*b*(fn_data + fp_data) - (epsp*Vt/(b*Tion))*dd_data
        + (q*DI*N0/b)*FP_data

    midpoint = int(np.ceil((N+1)/2))
    J_total = J_dimensional[midpoint,:]

    V_scan = psi_vector*Vt

    return x, t, P, phi, n, p, J_total, V_scan
```

```
""" Three-layer model dimensionalization. """
def dimensionalize_tlm(x_vector, xE_vector, xH_vector, t_vector, u_matrix,
    psi_vector):
   """ The variable arrays are 2-dimensional to account for dense output.
       """
   P = u_matrix[0:N+1,:]
   phi = u_matrix[N+1:2*N+2,:]
   n = u_matrix[2*N+2:3*N+3,:]
   p = u_matrix[3*N+3:4*N+4,:]
   phiE = u_matrix[4*N+4:4*N+NE+4,:]
   nE = u_matrix[4*N+NE+4:4*N+2*NE+4,:]
   phiH = u_matrix[4*N+2*NE+4:4*N+2*NE+NH+4,:]
   pH = u_matrix[4*N+2*NE+NH+4:4*N+2*NE+2*NH+4,:]

   phi = np.concatenate((phiE, phi, phiH), 0)
   n = np.concatenate((nE, n, n[-1]*np.ones((NH,t_vector.size))), 0)
   p = np.concatenate((p[0]*np.ones((NE,t_vector.size)), p, pH), 0)

   x = np.concatenate((-xE_vector[::-1], x_vector, xH_vector), 0)*b
   t = t_vector*params.LD*params.b/params.ion_mobility_factor
   P = N0*P
   phi = Vt*phi
   n = dE*n
   p = dH*p

   """ Non-Dimensional Current Density """
   J_total = q*G0*b*total_current_tlm(t_vector, u_matrix)[0]/10
   V_scan = psi_vector*Vt

   return x, t, P, phi, n, p, J_total, V_scan
```

C.2 animate.py

```
# -*- coding: utf-8 -*-
"""
Created on Thu Mar 18 02:22:47 2021

@author: Nathan
"""

import numpy as np
```

```
def animate_JV(t_vector, V_scan, J_total):

    import matplotlib.pyplot as plt
    import matplotlib.animation as animation

    fig = plt.figure()
    ax = plt.axes(xlim=(np.min(V_scan), np.max(V_scan)), ylim=(-5, 25))
    line, = ax.plot([], [], lw=2)

    # initialization function
    def init():
        # creating an empty plot/frame
        line.set_data([], [])
        return line,

    # lists to store x and y axis points
    xdata, ydata = [], []


    dt = 0.1
    values = np.arange(0, t_vector[-1], dt)
    V_scan_interp = np.interp(values, t_vector, V_scan)
    J_total_interp = np.interp(values, t_vector, J_total)


    def animate(i):

        x = V_scan_interp[i]
        y = J_total_interp[i]


        xdata.append(x)
        ydata.append(y)
        line.set_data(xdata, ydata)

        return line,

    # setting a title for the plot
    plt.title('J-V␣Scan␣Animation')
    # hiding the axis details
    plt.axis('on')

    # call the animator
    anim = animation.FuncAnimation(fig, animate, init_func=init, frames=len
        (J_total_interp), interval=55, blit=True)
```

```
    # save the animation as mp4 video file
    anim.save('JV.gif',writer='pillow')

    return anim



def animate_distribution(t_vector, x_mesh, particle, title):

    import matplotlib.pyplot as plt
    import matplotlib.animation as animation

    fig = plt.figure()
    ax = plt.axes(xlim=(np.min(x_mesh), np.max(x_mesh)), ylim=(np.min(
        particle), np.max(particle)))
    line, = ax.plot([], [], lw=1)

    # initialization function
    def init():
        # creating an empty plot/frame
        line.set_data([], [])
        return line,

    # lists to store x and y axis points
    xdata, ydata = [], []


    dt = 0.1
    values = np.arange(0, t_vector[-1], dt)

    particle_interp = np.zeros((x_mesh.size, values.size))

    for i in range(0,x_mesh.size):
        particle_interp[i,:] = np.interp(values, t_vector, particle[i,:])


    def animate(i):
        x = x_mesh; x[0] = None
        y = particle_interp[:,i]
        xdata.append(x); xdata[i-1] = np.zeros(x_mesh.size)
        ydata.append(y); ydata[i-1] = np.zeros(particle_interp[:,i].size)

        line.set_data(xdata, ydata); plt.title(title); time = int(values[i
            -1])+1; plt.title('%i␣s' %time, loc='right')
        return line,
```

```
    # setting a title for the plot
    plt.title('J-V␣Scan␣Animation')
    # hiding the axis details
    plt.axis('on')

    frames=len(values)
    # call the animator
    anim = animation.FuncAnimation(fig, animate, init_func=init, frames=
        frames, interval=55, blit=True)

    # save the animation as mp4 video file
    anim.save('distribution.gif', fps=frames/t_vector[-1] , writer='pillow'
        )

    return anim
```

C.3 plot.py

```
# -*- coding: utf-8 -*-
"""
Created on Wed Mar 3 23:08:34 2021

@author: Nathan
"""

import matplotlib.pyplot as plt
import numpy as np
import parameters

params = parameters.Params()

N = params.N
NE = params.NE
NH = params.NH

def plot_distributions_slm(x, u):
   P = u[0:N+1,-1]
   phi = u[N+1:2*N+2,-1]
   n = u[2*N+2:3*N+3,-1]
   p = u[3*N+3:4*N+4,-1]

   fig1, ax = plt.subplots()
```

```
    ax.plot(x, P, label='ions')
    ax.set_xlabel('non-dimensional space')
    ax.set_ylabel('non-dimensional P')
    ax.set_title("Ion Distribution")
    ax.legend()

    fig2, ax = plt.subplots()
    ax.plot(x, phi, label='phi')
    ax.set_xlabel('non-dimensional space')
    ax.set_ylabel('non-dimensional phi')
    ax.set_title("Electric Potential")
    ax.legend()

    fig3, ax = plt.subplots()
    ax.plot(x, n, label='n')
    ax.set_xlabel('non-dimensional space')
    ax.set_ylabel('non-dimensional n')
    ax.set_title("Electron Distribution")
    ax.legend()

    fig4, ax = plt.subplots()
    ax.plot(x, p, label='p')
    ax.set_xlabel('non-dimensional space')
    ax.set_ylabel('non-dimensional p')
    ax.set_title("Hole Distribution")
    ax.legend()

    return fig1, fig2, fig3, fig4

def plot_distributions_tlm(x, xE, xH, u):
    P = u[0:N+1,-1]
    phi = u[N+1:2*N+2,-1]
    n = u[2*N+2:3*N+3,-1]
    p = u[3*N+3:4*N+4,-1]
    phiE = u[4*N+4:4*N+NE+4,-1]
    nE = u[4*N+NE+4:4*N+2*NE+4,-1]
    phiH = u[4*N+2*NE+4:4*N+2*NE+NH+4,-1]
    pH = u[4*N+2*NE+NH+4:4*N+2*NE+2*NH+4,-1]

    xtotal = np.concatenate((-xE[::-1], x, xH), 0)
    phitotal = np.concatenate((phiE, phi, phiH), 0)
    ntotal = np.concatenate((nE, n, n[-1]*np.ones(NH)))
    ptotal = np.concatenate((p[0]*np.ones(NE), p, pH))

    fig1, ax = plt.subplots()
    ax.plot(x, P, label='ions')
```

```
    ax.set_xlabel('non-dimensional space')
    ax.set_ylabel('non-dimensional P')
    ax.set_title("Ion Distribution")
    ax.legend()

    fig2, ax = plt.subplots()
    ax.plot(xtotal, phitotal, label='phi')
    ax.set_xlabel('non-dimensional space')
    ax.set_ylabel('non-dimensional phi')
    ax.set_title("Electric Potential")
    ax.legend()

    fig3, ax = plt.subplots()
    ax.plot(xtotal, ntotal, label='n')
    ax.set_xlabel('non-dimensional space')
    ax.set_ylabel('non-dimensional n')
    ax.set_title("Electron Distribution")
    ax.legend()

    fig4, ax = plt.subplots()
    ax.plot(xtotal, ptotal, label='p')
    ax.set_xlabel('non-dimensional space')
    ax.set_ylabel('non-dimensional p')
    ax.set_title("Hole Distribution")
    ax.legend()

    return fig1, fig2, fig3, fig4

def plot_JV(psi, J, *dim):
    fig1, ax = plt.subplots()
    ax.plot(psi, J, label='J')

    voltage_range = [params.Vi, params.Vf]
    upper_lim = np.max(voltage_range)
    lower_lim = np.min(voltage_range)

    if ('non-dimensional' in dim):
        ax.set_xlabel('psi - non-dimensional applied voltage')
        ax.set_ylabel('J - non-dimensional current-density')
        plt.xlim([lower_lim/params.Vt, upper_lim/params.Vt])
        plt.ylim([0,1])
    elif ('dimensional' in dim):
        ax.set_xlabel('Applied Voltage (V)')
        ax.set_ylabel('Current Density ($mA \cdot cm^{-2}$)')
        plt.xlim([lower_lim, upper_lim])
        plt.ylim([-5,25])
```

```
    ax.set_title("J-V␣Scan")
    ax.legend()

    plt.plot([-100,100], [0,0], color='black', linestyle='dashed',
        linewidth=0.5)

    return fig1
```