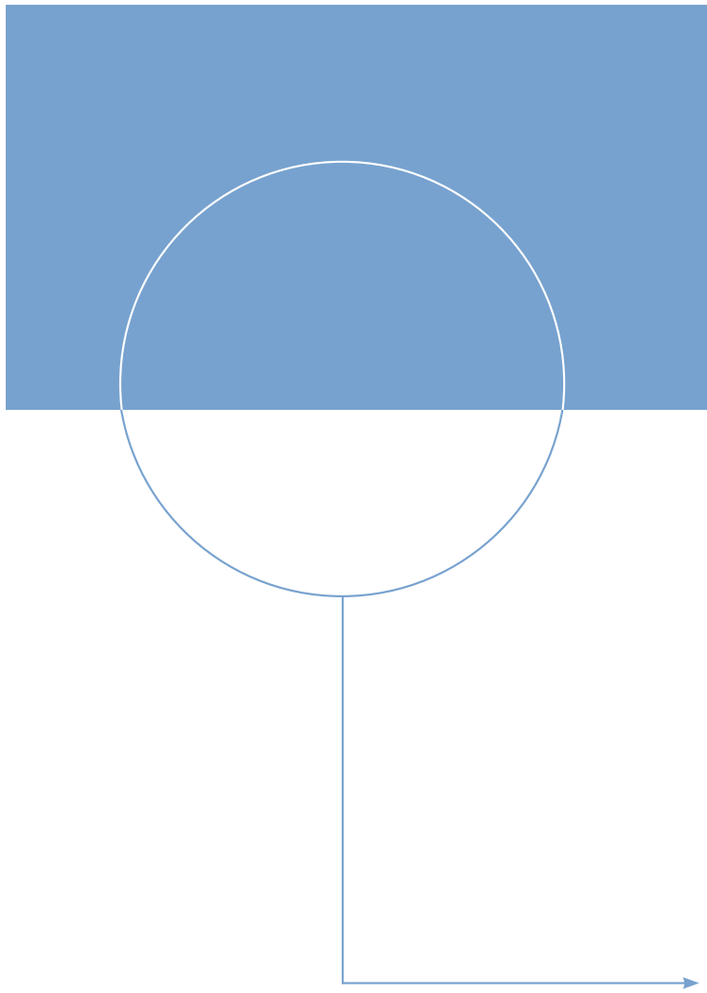


Master's thesis

2020

Andreas Bolstad

NTNU
Norwegian University of
Science and Technology
Faculty of Natural Sciences
Department of Physics

Andreas Bolstad

Development of a 3D Particle-Based Device Simulator

A Self-Consistent Monte Carlo Approach Using Tetrahedral Grids

June 2020

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Development of a 3D Particle-Based Device Simulator

A Self-Consistent Monte Carlo Approach Using Tetrahedral Grids

Andreas Bolstad

Applied Physics and Mathematics
Submission date: June 2020
Supervisor: Jon Andreas Støvneng (IFY)
Co-supervisor: Trond Brudevoll (FFI)
Asta Katrine Storebø (FFI)

Norwegian University of Science and Technology
Department of Physics

I dedicate this work to my family.

Abstract

The ability to simulate semiconductor devices with realistic geometrical and material properties is essential to their optimization. A new simulation program is presented which combines a particle-based simulation approach using tetrahedral grids to simulate high-energy charge-carrier transport in devices with complex 3D geometry. The report details how the different program components are coupled and interact. Key aspects are 1) generation of device discretizations with Gmsh; 2) field updates computed with a finite element solver; 3) carrier transport simulated with a full-band Monte Carlo approach; and 4) particle boundary conditions coupled with an unstructured grid. The program's capabilities are then demonstrated by simulating the response of applying a $-7\,\mathrm{V}$ bias on a $30\,\mu\mathrm{m}$ x $10\,\mu\mathrm{m}$ x $1\,\mu\mathrm{m}$ HgCdTe-alloy avalanche photodiode.

It is shown that the program is capable of simulating the geometrically complex features of a large range of micro-scale semiconductor devices. Furthermore, methods to increase execution speed such as parallelization are discussed for obtaining high accuracy results in a reasonable amount of time.

Sammendrag

Essensielt for optimering av halvlederkomponenter er å kunne simulere dem med realistiske geometri- og material-egenskaper. Et nytt simuleringsprogram presenteres som kombinerer en partikkelbasert simuleringsmetode med bruk av tetraedriske rutenett for å simulere høyenergi-ladningstransport i komponenter med kompleks 3D-geometri. Denne rapporten beskriver hvordan de ulike komponentene til programmet er koblet og virker sammen. Sentrale punkt i oppgaven omhandler å 1) diskretisere komponenter med Gmsh, 2) beregne felt med en endelig element løser, 3) simulere ladningstransport med bruk av fullbånds Monte Carlo metode, og 4) forbinde partiklers grensebetingelser til et ustrukturert rutenett. Deretter demonstreres programmets egenskaper ved å simulere responsen fra å påsette en $-7\,\mathrm{V}$ spenning på en $30\,\mu\mathrm{m}$ x $10\,\mu\mathrm{m}$ x $1\,\mu\mathrm{m}$ avalanche-fotodiode laget av en HgCdTe-legering.

Det vises at programmet er i stand til å simulere de geometrisk komplekse egenskapene til et bredt spekter av mikroskala halvlederkomponenter. Videre diskuteres metoder for å øke programmets kjørehastighet – for eksempel parallellisering – som kan gi nøyaktige resultater til rimelig tidsforbruk.

Preface

This Master's thesis is submitted to the Norwegian University of Science and Technology (NTNU), fulfilling the final requirements of the Master of Science degree in Applied Physics and Mathematics at the Department of Physics (IFY). This work was carried out during the spring semester of 2020, and the official subject title was *TFY4900 Physics, Master's Thesis*.

My supervisors at FFI, Trond Brudevoll and Asta Katrine Storebø, have my sincere appreciation for their support and encouragement throughout this project, always replying to my many questions with even longer answers. I hope my contribution will be of great use to them, which they certainly deserve. Professor Jon Andreas Støvneng has been my supervisor at NTNU, and I thank him for helping this project get underway. I would also like to thank my family for their strong and consistent moral support.

Andreas Bolstad,
June 22nd, 2020

Contents

List of Tables

List of Figures

Abbreviations

2D	two-dimensional
3D	three-dimensional
APD	avalanche photodiode
API	application programming interface
BTE	Boltzmann transport equation
BZ	Brillouin zone
CMT	cadmium mercury telluride
CPU	central processing unit
DD	drift-diffusion
EMC	ensemble MC
FB	full-band
FD	finite-difference
FE	finite-element
FFI	Norwegian Defence Research Establishment
GUI	graphical user interface
HD	hydrodynamic
IR	infrared
MC	Monte Carlo
MCS	MC simulation
NTNU	Norwegian University of Science and Technology
QM	quantum mechanical
RAM	random access memory
ROIC	read-out integrated circuit
SCMC	self-consistent MC

1 Introduction

Progress in semiconductor research has paved the way for a new technological era by making digital computers and electronic devices commonplace. Despite these great technological advances, our understanding of semiconductor devices is still limited. It is not uncommon that a device is developed in the lab long before the governing physics is understood. Fortunately, computer simulation is closing this knowledge gap.

Charge transport phenomena are fundamental to the study of semiconductor devices, but are notoriously difficult to analyze analytically. This is why numerical methods and particularly device physics simulators are essential for device optimization. For the analysis of devices near equilibrium, classical methods such as drift-diffusion have been greatly successful, and today there are multiple high-quality software packages available. However, classical models are insufficient for describing highly non-equilibrium effects, in which case semiclassical[1] models are required, and the topic at hand.

A popular particle-based approach for simulation of highly non-equilibrium charge transport is the semiclassical Monte Carlo (MC) method. Studying transient device behavior requires the electric field to be updated during simulation, in which case the method is referred to as self-consistent Monte Carlo (SCMC).

Device manufacturers require accurate simulation in 3D for studying increasingly complex device models. New SCMC software and yearly increases in computational power show that simulation of nanoscale devices in 3D is becoming feasible, but at high computational costs associated with frequent electric field updates [1]. For the simulation of devices with arbitrary 3D geometry and field complexity, a robust but computationally demanding approach is to use unstructured grids in combination with the finite element (FE) method. Most research on this topic has been conducted by a research group at the University of Swansea, who have created a parallel 3D FE SCMC simulator to study non-planar transistors [2]. In the last few years, another research group at the University of Granada have made a similar 3D FE SCMC simulator with multi-subband capabilities, also mainly used for nanoscale transistor research [3, 4]. Today, this approach might be applied for simulation of microscale devices as increasing computational power greatly increases the applicability of this advanced model. This is further considered in this thesis.

[1] A semiclassical model treats one part quantum-mechanically and the other classically.

1.1 Previous work

A set of general-purpose Monte Carlo simulation tools referred to as FFI-MCS have been developed since 2007 by researchers at the Norwegian Defence Research Establishment (FFI) in cooperation with Master's students and summer interns. The primary application of FFI-MCS has been to model charge transport in cadmium mercury telluride (CMT) alloys and study electro-optical processes. It is currently adapted for studying avalanche photodiodes (APDs) for the Epitek group at FFI.

Full-band (FB) particle simulation is the most consequential and heavily developed functionality of FFI-MCS, which is essential for accurate high-energy simulations, and therefore central to modeling the operation of APDs made at FFI. The first openly available conference presentation using FB FFI-MCS was in 2017, where it was used to demonstrate how alloy scattering could be exploited to amplify signals in HgCdTe-alloy APDs by an order of magnitude [5]. To continue this research, FFI intends to continue the development of FFI-MCS for many years to come. Hopefully, parts of FFI-MCS can eventually be shared with the larger research community, considering that the availability of open source software is limited and development requires years of hard work.

Until 2013, FFI-MCS only supported analytical band representations. In recent years, the analytical bands have only been used as a starting point for faster simulation during development of new functionality. However, the code structure and capabilities of the analytical-band and full-band program have diverged over time. This has made it increasingly difficult to successfully integrate new functionality made for an analytical program version into the main FB program.

When this thesis work began, the FB program only supported one electric field solver, the two-dimensional (2D) finite-difference (FD) Poisson solver for uniform grids created by Kirkemo back in 2012 [6]. In recent years, three new Poisson solvers have been made for analytical-band versions of FFI-MCS, which are listed in table 1.1. However, none of them has yet been fully coupled with the full-band program.

Table 1.1: List of Poisson solvers developed for FFI-MCS in recent years.

Year	Author	Poisson solution method
2015	J. Harang [7]	2D finite difference method with tensor grids
2016	D. Åsen [8]	2D finite element method with triangular grids
2018	S. Fatnes [9]	3D finite element method with tetrahedral grids

1.2 Aim and Approach

Researchers at FFI warrant new and faster Poisson solvers for studying new APD designs with larger dimensions and more advanced geometry. To this end, I have continued the development of Fatnes's program MCFEM [9], to expand FFI-MCS capabilities to complex device domains in three dimensions (3D) and provide a program framework for future development.

Most developers of 3D SCMC simulators have used them to study nanoscale devices. One exception is a research group at Boston University, who have studied microscale IR detectors with full-band MC simulations [10], similar to FFI's use cases for FFI-MCS. However, the Boston group have yet to demonstrate the efficiency or any applications of their 3D-FE component [11, ch. 7]. Thus, there is a research question in need of an answer:

How well suited is 3D-FE-SCMC simulation for studying microscale devices?

Before answering this question, it is necessary to build a functioning simulator, which is a significant challenge in itself. Considering the low number of 3D SCMC simulators created, despite the apparent success of its developers, shows the difficulty in building a simulator that works correctly and is sufficiently accurate and efficient. Usability is another equally important consideration. A program nobody can use is worthless. Developing simulators tailored to specific test cases is generally a time-consuming task, and existing simulators are often too specialized for efficient adaptation to new what-if scenarios. This is the main reason simulations are not seeing the widespread use one would expect considering most researchers are well-aware of the benefits and insights simulations can provide [12].

Thus, this work aims to provide new solutions and insights into the following aspects of 3D SCMC simulation:

Applicability What functionality is suitable for the widest range of applications?

Efficiency What can help users carry out a modelling task as fast as possible?

Maintainability How can we structure the program to ensure good extensibility and generally make continued development as easy as possible?

To achieve the set goals, this work proceeded in a series of stages, each leading to the next:

1. Identify and rectify errors in the 3D SCMC program called MCFEM [9], which was unable to run longer MC simulations prior to this work.

2. Design a new MC program capable of accommodating the full-band components of FFI-MCS, the 3D Poisson solver from MCFEM, and other features likely to be added in the future.

3. Implement a prototype of the program to verify the main features work as expected.

4. Streamline the development process by making new systems for utilizing configuration files, automatic building, defining device geometry and running simulations on computing clusters.
5. Implement new modules for particle initialization and boundary conditions coupled with unstructured grids.
6. Verify correct program operation and analyze numerical stability.
7. Benchmark computational performance.

1.3 Thesis Content

This thesis was written with a physics audience in mind, who have a good understanding of fundamental semiconductor physics. An introductory level of numerical and computational understanding is assumed as well. The report is structured as follows:

Chapter 1 provides a brief description of context, motivation and objectives for this thesis.

Chapter 2 introduces the underlying numerical methodology used throughout this work.

Chapter 3 provides an overview of how the new simulation program's structure and operation.

Chapter 4 presents the software tools used throughout this work and explains how the new MC simulator was implemented.

Chapter 5 shows test results and evaluates the program's correctness, stability, efficiency and applicability.

Chapter 6 lists recommended paths for further program development.

Chapter 7 concludes this thesis.

2 Background

2.1 Scope of the Monte Carlo Method

In general, the Monte Carlo (MC) method is a stochastic solution method that relies on random number generation. This is mainly used to solve optimization and integration problems, which is useful for a wide range of applications, from applied statistics and finance to computational physics and chemistry [13].

Herein we consider the semiclassical MC method for the simulation of charge-carrier transport in semiconductors. MC methods have been successful applied to the investigation of a great variety of semiclassical transport phenomena due its ability to include stochastic processes in general, and scattering processes in particular [14]. During the 80s, the MC method became popular for device simulation, made possible by the steady increase in computational power [15]. Todays MC simulators are capable of modeling a vast range of time scales and device dimensions. This is because the MC method is more robust and extensible to a wider range of applications than most other methods [16]. For example, for the simulation of small devices it is possible to add quantum corrections, which is essential for nanoscale transistor modeling.

The topic of MC simulation (MCS) for semiconductors and semiconductor devices is vast, and this chapter can only cover the main aspects relevant for this thesis. For a thorough introduction to modern semiconductor device simulation, with emphasis on MCS, the book by Vasileska et al. [17] is recommended. It provides a good beginner's introduction to MCS, its range of application and comparison to other simulation methods. Jungemann and Meinerzhagen's book [18] provides in-depth descriptions of important numerical aspects of MCS, including full-band simulation. The classic textbooks by Moglestue [19] and Lugli and Jacobini [20] are also recommended for their thorough treatment of the fundamentals of MCS.

2.2 Semiclassical Transport Theory

Semiclassical transport of charge carrier ensembles, electrons and holes, can be treated with the classical Boltzmann transport equation (BTE) of a kinetic gas extended with two quantum mechanical components:

1. The semiconductor's band structure

2. Scattering rates determined by Fermi's golden rule [17, p. 100]

When the particle motion is collision-less the particle is in a so-called free flight, which can be represented by Newton's laws of motion,

$$\mathbf{v} = \frac{1}{\hbar}\nabla_k \mathbf{E_n}(\mathbf{k}), \tag{2.1}$$

and

$$\dot{\mathbf{k}} = \frac{q\mathbf{F}(\mathbf{r})}{\hbar}, \tag{2.2}$$

where we have the energy dispersion relation $E(\mathbf{k})$, electric field $\mathbf{F}$, wave vector $\mathbf{k}$ and band index n.

Free flight and scattering can be coupled through the Boltzmann transport equation (BTE), which can be expressed as [17, p. 241]:

$$\frac{\partial f}{\partial t} + \frac{1}{\hbar}\nabla_k E_n(\mathbf{k})\nabla_r f + \frac{q\mathbf{F}(\mathbf{r})}{\hbar}\nabla_k f = \left[\frac{\partial f}{\partial t}\right]_{\text{collision}} \tag{2.3}$$

where f is the one-particle distribution function of occupying a state with position $\mathbf{x}$, pseudo-momentum $\mathbf{k}$, and time t. BTE accurately describes charge transport in semiconductors when the semiclassical approximations apply: The effective mass approximation, which incorporates quantum effects arising from the crystal's periodicity; the Born approximation for collisions, which is small perturbations in electron-phonon interactions and instantaneous collisions; the series of flight and scatterings is a Markov process, which means the probability of a future event only depends on the system's current state. Note that the BTE can be solved analytically only in special circumstances, which is why numerical treatment is required.

2.3 Bulk Simulation

This section outlines the MC components required for basic bulk semiconductor simulation.

2.3.1 Monte Carlo Simulation of Charge Carriers

The BTE can be formulated in integral form, which can be solved with the semiclassical MC method [18]. The MC solution treats the collision term of BTE quantum mechanically by stochastic evaluation of each distinct scattering mechanism, unlike the drift-diffusion (DD) and hydrodynamic (HD) methods where the collision term is treated with a classical approximation. This is why MC

can accurately emulate a large range of highly non-equilibrium semiconductor systems where other models are insufficient [17, p. 242].

The MC method evaluates the BTE integral through a series of particle free flights and scattering events. The approach is mainly comprised of three steps:

1. Generating Free Flight Duration

The charge carrier's phase-space trajectory, which is comprised of position $\mathbf{r}$ and wavevector $\mathbf{k}$, changes continuously during a free flight. If we define $\lambda\left[\mathbf{k}(t)\right]$ as the probability of scattering during a tiny time interval dt, it can be shown that the probability $P(t)$ for a scattering to occur during the time interval dt at a time t, with the previous scattering having occurred at $t = 0$, is

$$P(t)dt = \lambda\left[\mathbf{k}(t)\right] \exp\left[-\int_0^t dt' \lambda\left[\mathbf{k}(t')\right]\right] dt. \tag{2.4}$$

A free flight duration δt can be sampled by generating a random number $r \in U(0,1)$ and evaluating the equation

$$r = \int_0^{\delta t} dt P(t), \tag{2.5}$$

which by inserting equation (2.4) may be rewritten to the more practical form

$$-\ln r = \int_0^{\delta t} \lambda(\mathbf{k}(\mathbf{t}))dt. \tag{2.6}$$

A proper derivation of the above equations is found in [19, ch. 5].

2. Selecting Scattering Mechanism

At the end of a free flight, a scattering mechanism is selected. The probability for selecting the i-th mechanism, out of n possible mechanisms, is

$$P_i(\mathbf{k}) = \frac{\lambda_i(\mathbf{k})}{\lambda(\mathbf{k})}, \quad \lambda(\mathbf{k}) = \sum_{i=1}^{n} \lambda_i(\mathbf{k}). \tag{2.7}$$

Then, again by generating a random number $r \in U(0,1)$, we accept the j-th mechanism that satisfies the inequality

$$\sum_{i=1}^{j-1} \frac{\lambda_i(\mathbf{k})}{\lambda(\mathbf{k})} < r < \sum_{i=1}^{j} \frac{\lambda_i(\mathbf{k})}{\lambda(\mathbf{k})}. \tag{2.8}$$

Note that the numerical evaluation of equation (2.6) is usually simplified by introducing a self-scattering term λ_0 [21].

3. Selecting State after Scattering

After determining the scattering mechanism to terminate the flight, a new state for the wavevector $\mathbf{k}_f$ after scattering must be chosen. This is done stochastically by generating $\mathbf{k}_f$ according to the differential cross section of the mechanism in question.

Executing the above steps is a computationally demanding task, which has created a high demand for efficient numerical methods. The method used in this work is presented in section 4.3.

2.3.2 Ensemble Monte Carlo

The above-described Monte Carlo approach allows for simulating the motion of individual particles. Macroscopic quantities can be obtained by sampling statistics for the simulation of many particles. In order to improve statistics over short and/or transient simulations, the most common approach is to simulate a synchronous ensemble of particles [16, p. 10], which is known as ensemble Monte Carlo (EMC) method.

Superparticles

The number of actual particles in a system is usually too large to simulate of all of them individually in a reasonable amount of time. Therefore, each simulated particle is a so-called *superparticle*, representing several real particles with identical properties. A superparticle's charge is proportional to the number of real particles it represents, while it behaves as a single real particle in scattering events.

Ensemble Averages

Macroscopic quantities are usually computed by taking the average over the ensemble at several time steps throughout simulation. For example, the drift velocity at time t can be calculated as an ensemble average of the velocity

$$\bar{v}(t) \approx \frac{1}{N} \sum_{i=1}^{N} v_i(t). \tag{2.9}$$

where N is the number of superparticles in the ensemble. $\bar{v}$ is an estimator of the actual velocity, with a standard error given by [16, eq. 34]

$$s = \frac{\sigma}{\sqrt{N}}, \tag{2.10}$$

where σ^2 is the variance. We see that the estimation error decreases slowly with a rate proportional to $1/\sqrt{N}$. Thus, a very large N is required to achieve accurate results, making EMC simulation in general computationally demanding.

2.4 Device Simulation

Moving from a bulk simulator to a device simulator, we require additional components:

1. A real-space discretization method for representing device models.
2. A framework for imposing particle boundary conditions at device boundaries.
3. A Poisson solver to resolve the electric field

Each part is covered briefly in this section.

Charge carriers are accelerated by the electric field, which in turn is influenced by the carrier distribution. Consequently, the MC solver needs to be coupled with a field solver to update the electric field during simulation [22]. This simulation method is called self-consistent (ensemble) Monte Carlo (SCMC).

Some device properties can be simulated non-consistently using a fixed electric potential, which is typically produced with a preliminary SCMC simulation or using another tool such as a DD or HD simulator. This approach is usually much less computationally costly compared to self-consistent simulation, but it is not well suited for transient simulation [18, ch. 6].

2.4.1 Discrete Device Representation

The device can be discretized by dividing it into cells, which collectively are called a grid, also known as a mesh. Using the grid discretization, we can represent different device parameters in two main ways [18, ch. 6]:

1. Specifying device regions with constant values, where each region is comprised of multiple cells. Most device properties (except for global ones) are imposed this way.
2. Specifying a discrete scalar field with values at each grid node and using interpolation to calculate values within each cell during simulation. This approach is mainly used where the first approach provides insufficient detail.

Mesh Types

There are many possible types of mesh to choose from, but some are better suited for MC simulation. Generally, structured meshes are preferred because it is simple to transform particle positions to grid cell indices. Widely used mesh types for device discretization are illustrated in figure 2.1. The rectangular uniform grid is the most basic one, dividing the domain into equally sized cells. Tensor and octree grids provide simple and efficient techniques to achieve higher refinement in specific regions. However, these grids are not suited for representing complex geometries such as curved

surfaces. Unstructured grids, typically comprised of triangles in 2D and tetrahedra in 3D, are usually preferred when complex shapes are needed because of few limitations on grid arrangement.

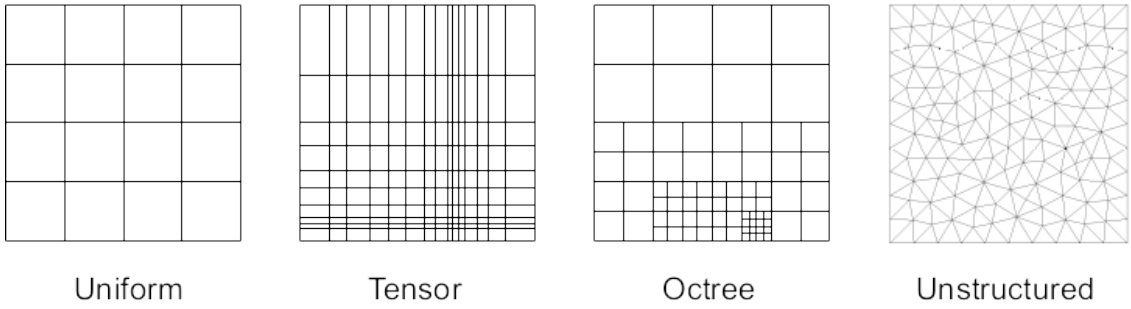


Figure 2.1: Illustration of three different types of structured grid (left) and a triangular unstructured grid (right).

Particle Boundary Conditions

There are several possible interactions that can be applied when a particle meets an interface between two grid cells, as illustrated by figure 2.2. The mechanisms implemented in this work are *enter*, *reflect* and *absorb*, which are required for device simulation. *Enter* is the default condition, letting particles transfer to another cell without any change in momentum. *Reflect* is typically used at surfaces where we want zero particle flux. The reflection is specular with quasi momentum parallel to the surface and particle energy being conserved. *Absorb* can be used to simulate contacts, where particles are allowed to leave the device. Typically, the particle is simply removed from the simulation, and particle data is saved for statistics such as terminal current.

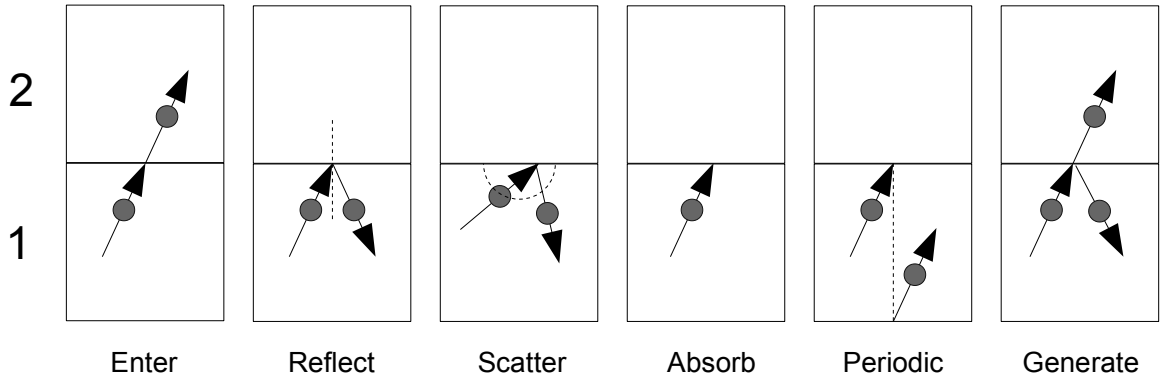


Figure 2.2: Possible particle boundary conditions at the interface between two device regions 1 and 2. This figure is based on a similar figure by Jungemann and Meinerzhagen [18].

Device Regions

The device can be divided into multiple regions using the discretization, mainly used to specify material properties throughout the device and boundary conditions along device boundaries. For example, charge density may be specified by marking one region negatively doped (N), and another positively doped (P). Isolation (I) regions are used to represent any type of isolating material charge carriers are cannot enter. At (I) interfaces, particles are usually reflected. Another common region is contact (C), used to apply special properties for modeling metal-semiconductor interfaces. A particle leaving into a metal contact is usually deleted, marked by a boundary between (I) and (C) regions.

The complexity of implementing regions depends on how complex the discretization is. A method for digitally representing regions for unstructured grids was implemented in this work as presented in 4.4.

2.4.2 Finite Element Poisson Solver

Herein we assume a negligible magnetic field and a quasi-static electric field, which is a fair approximation for the behavior of a large range of semiconductor devices. The field can be obtained by calculating the gradient of the electric potential, which is approximated by a numerical solution of the Poisson equation.

Poisson Equation

The linear Poisson equation relates the electric potential u to the charge distribution ρ as

$$-\nabla^2 u(\mathbf{x}) = \frac{1}{\epsilon_r \epsilon_0}\rho(\mathbf{x}), \tag{2.11}$$

where ϵ_r is the material's relative static dielectric constant. For a point particle system with N_p particles, the Poisson equation can be expressed as

$$-\nabla^2 u = \frac{1}{\epsilon_r \epsilon_0}\sum_{p=1}^{N_p} q_p \delta(\mathbf{x} - \mathbf{x}_p), \tag{2.12}$$

where q_p is the charge and $\mathbf{x}_p$ is the position of particle p, and δ is the Dirac delta function. Here, the particles p represent both doping atoms and free charge carriers. The electric field can generally be computed using the electric potential with the expression

$$\mathbf{E} = -\nabla u. \tag{2.13}$$

Boundary Conditions

The potential u is restricted to the device domain Ω with constraints along the boundaries. To model reflective surfaces we impose a zero Neumann condition, whereas Dirichlet conditions are applied along contact boundaries. This corresponds well with reality when contacts are kept at constant potentials and there is no electric field parallel to isolating surfaces [20, ch. 5.2]. These conditions are typically formulated as

$$u = g(\mathbf{x}) \text{ on } \partial\Omega_D, \qquad \frac{\partial u}{\partial \mathbf{n}} = 0 \text{ on } \partial\Omega_N, \tag{2.14}$$

where n is the surface normal of the boundary.

Finite Element Method

The finite element (FE) method can be used to solve the Poisson equation in this work. This is because the method is generally stable, efficient and allows the use of unstructured grids. The main steps of any finite element method can be summarized as [23, ch. 2.3]:

1. Partitioning the domain Ω into a triangulation T composed of non-overlapping (finite) elements K_i

2. Selecting sets of interpolation functions $\{\varphi_j\}_i$ for each element K_i

3. Formulating the system of equations as a solvable boundary condition problem

4. Implementing a computer algorithm to solve the problem numerically

While the theory of FE solution is well established, making more accurate and efficient numerical implementations is an ongoing research topic. Which methods to apply depend on the problem at hand, but the main challenges are usually the same:

- Efficiently and accurately solving a large system of equations

- Minimizing the number of elements required to achieve the level of accuracy desired

For our purposes, the first point is thoroughly covered by Fatnes [9], while the latter point is further considered in later chapters.

A proper introduction to FE methods is not provided here. The reader should instead consult the many introductory textbooks on the subject: Quarteroni [24] for a mathematically rigorous treatment, Jin [23] for applications and implementation strategies for electromagnetics in general, and Mobbs [25] for applying FE methods to semiconductor device modelling in particular.

2.4.3 Numerical Stability

Since Monte Carlo device simulation comes at a high computational cost, it is important to optimize for computational efficiency. However, the numerical stability of SCMC simulation depends on resolution in both time and space, putting hard limits on how much accuracy can be sacrificed for increased efficiency.

An intuitive requirement for ensuring stability is to restrict the charge carrier's movement between electric field updates. Otherwise, carriers will cross multiple grid cells in one time step and the real electric field applied to the particle no longer corresponds to reality. This restriction can be expressed with the following criterion [26, ch. 6.7]:

$$\Delta t < \frac{\Delta x}{v_{\max}}, \tag{2.15}$$

where h is the grid spacing (length between mesh nodes), and $v_{\max}$ is the highest velocity of a carrier, which typically is $\approx 10^8\,\mathrm{cm\,s^{-1}}$. There are however other more stringent stability criterions for both the time and space discretization which are covered next.

Time Resolution

Charge carrier's in a carrier plasma have a characteristic oscillation frequency, the so-called plasma frequency, which can be expressed as

$$\omega_p = \sqrt{\frac{e^2 n}{\epsilon_r \epsilon_0 m^*}}, \tag{2.16}$$

where n is the carrier density. According to Hockney and Eastwood [22] (1988), a stable simulation requires Δt to smaller than the inverse plasma frequency ω_p^{-1},

$$\Delta t < \frac{2}{\omega_p}. \tag{2.17}$$

A more rigorous stability analysis is provided by Rambo and Denavit [27] from 1993, showing that equation (2.17) is not applicable other than in special circumstances. They presented new stability criteria for simulations including particle collisions, and showed that MC simulations are always unstable without collisions. Using the same notation as in [27], the collision rate ν_c can be related to the low field mobility μ by

$$\nu_c = \frac{e}{m^* \mu}, \tag{2.18}$$

which is used in Rambo and Denavit's stability criterion:

$$\Delta t \leq \frac{2\nu_c}{\omega_p^2}. \tag{2.19}$$

The reader should be warned that most textbooks on MCS only present the old criterion from equation (2.17), including relatively modern books such as [26] from 2000 and [17] from 2010. One exception is Jungemann and Meinerzhagen, who provide detailed time stability analysis in their book [18, ch. 6.4] from 2003.

Spatial Resolution

A natural criterion for the spatial resolution is the smallest wavelength of charge variations, which for a non-degenerate semiconductor is the Debye length [17]. At an interface between two differently doped regions, the carriers diffuse into the less doped region with an excess charge density that decays exponentially towards the bulk's equilibrium density. The characteristic length of this decay is one interpretation of the Debye length λ_D, such that

$$\lambda_D = \sqrt{\frac{\epsilon_r \epsilon_0 k_B T}{e^2 n}}. \tag{2.20}$$

There is little research on how low spatial resolution can lead to instability compared to the time stability analysis discussed earlier. However, one 2006 research paper by Palestri et al. [28] provides a detailed analysis and a guideline on the choice of grid spacing for SCMC with a linear Poisson solution scheme. In the zero scattering limit $\nu_c \to 0$ it was shown that simulations are stable when

$$\frac{\Delta x}{\lambda_D} \leq \pi. \tag{2.21}$$

The stability criterion can be relaxed if scattering rates are high. If the system only has elastic scattering mechanisms, there is a cut-off scattering rate $\nu_c \sim \omega_p$. Above cut-off, the grid spacing Δx can be tens of times larger than the Debye length λ_D. The stability criteria changes depending on which types of scattering mechanisms are included. Palestri et al. find simulations are stable for much larger grids than the Debye length when $\nu_c \gtrapprox 0.4\omega_p$ in one scenario, and when $\nu_c \gtrapprox 0.25\omega_p$ in another.

With non-elastic scattering, especially emission rates where energy is lost, the stability criteria can be further relaxed. In a device simulation, other mechanisms also make the system more and less stable, which means a stability criteria $x < \frac{\nu_c}{\omega_p}$ needs evaluated on a case-by-case basis.

Knowledge of the relaxed stability criteria is crucial for our purposes, as it can help to drastically reduce computational costs. If the grid spacing is increased 10 times, it is equivalent to reducing the number of cells 1000 times in 3D simulation. These criteria are further analyzed in section 5.4.

Particle Self-Forces

Another stability issue called *self-forces* arises when using an unstructured grid. The electric field for each particle is determined by approximating the potential gradient at the particle's position. The field contribution from a particle will act with a force on the same particle, which is a force not present in a real system. Self-forces are almost negligible for high-energy simulations with many particles according to Aldegunde and Kalna [29], which is why self-force corrections have not been prioritized in this work.

3 | Program Overview

The goal of this chapter is to provide an overview of the operation and structure of the new MC program that I have named MonteFFI. A brief description of the program workflow is given in section 3.1 and program components in section 3.2. Development history of the larger FFI-MCS project is presented in section 3.3.

3.1 Workflow

The use of MonteFFI is only one of multiple steps in a computational workflow to obtain physical device insights. To begin with, we require the physical parameters of interest to be defined. This includes material properties and various physical quantities, such as the electric field $\mathbf{E}$ and lattice temperature T_0 [30]. For device simulation, we also require the device geometry and mesh resolution with specified particle densities and boundary conditions. The next step is to specify MC simulation settings that determine the accuracy and speed of the simulation, for example which scattering effects to include and the time resolution of the MC loop and field updates.

With all parameters defined, the computer calculations can be performed. MonteFFI requires input data from pre-processing programs and its output treated by post-processing. The typical order in which the different programs are executed is presented in table 3.1.

Table 3.1: Computational workflow and software for 3D-FB-SCMC simulation.

Step	Computational work	Software
1.	Calculate numerical (full) band structure	KPBAND[1], ABINIT [31], WIEN2k [32]
2.	Calculate scattering rates	SCRATES [33]
3.	Generate mesh	GMSH [34]
4.	MC simulation	MonteFFI
5.	Post-processing	Python and Matlab scripts

[1] In-house band structure program using the $\vec{k} \cdot \vec{p}$-method originally created by Halvorsen [33] which has been updated several times later, most recently by Karlsen [35].

3.2 Program Structure

3.2.1 Program Components

MonteFFI is a combination of multiple program components, largely built on top of previous versions of FFI-MCS. The main program components of MonteFFI are briefly presented below.

Full-Band and Scattering MonteFFI uses a discrete representation of the full energy band structure, which is included with a module using pre-tabulated band structures to calculate particle energies and energy gradients through interpolation. In turn, these values are used to determine free-flight particle motion in another module. Before simulation, the energy band is also used to tabulate scattering rates. A scattering module is included to retrieve scattering rates through interpolation based on the state of each particle at each time step during simulation.

Finite-Element Solver To resolve the electric field required for self-consistent solution MonteFFI uses a 3D finite-element Poisson solver which I have fixed as described in section 4.2. A 3D point location algorithm is used to determine the element in the unstructured grid each particle is in, before charges are distributed between mesh nodes. The Poisson equation in the FE approximation is represented as a linear system of equations, which is solved using a sparse preconditioned conjugate gradient method. Lastly, the electric field in each particle position is interpolated from the potential gradient at the vertices of the element the particle is residing in. The program modules performing these operations were all created by Fatnes and are thoroughly described her thesis [9].

Components largely used to connect the other program components were created during this work:

- A tetrahedral grid based device discretization representation.
- Particle initialization and boundary conditions coupled to the mesh.
- An MC procedure connecting the particle-band, particle-field and particle-mesh interactions, which is called *flight and scattering* herein.

Each of these features are thoroughly covered later in chapter 4.

3.2.2 Module Hierarchy

To ensure MonteFFI remains maintainable, its program structure was made markedly modular. Figure 3.1 illustrates the functional hierarchy of MonteFFI, and gives an overview of which parts of implementation are the result of this thesis work. The goal was to keep lower-level components independent of higher-level components, keep interfaces between modules minimal and flexible, and generally minimize the amount of work required to add, remove or replace program features.

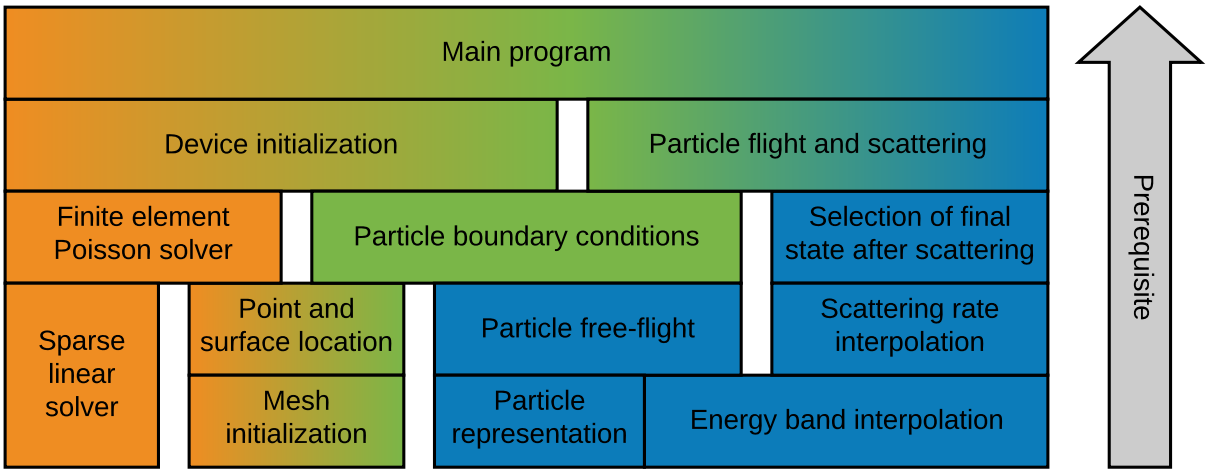


Figure 3.1: Simplified functional hierarchy of MonteFFI's main components. Green color indicates which components are new or have been changed during this thesis, yellow components are from MCFEM [9], and blue components are from the latest full-band version of FFI-MCS.

3.2.3 Control Flow

There are two main steps in the execution of MonteFFI: initialization as shown in figure 3.2, and a main simulation loop as shown in figure 3.3. The main simulation happens inside a time loop, where each iteration has a time duration Δt. The electric field applied to each particle is updated after a fixed time step, which is a fixed multiple of Δt, making the program self-consistent.

3.3 Development History

To understand FFI-MCS possibilities and limitations it is important to have a good grasp on previous development. Therefore, I have compiled a chronological list of student contributions as presented in table 3.2. This can be of great help to anyone looking to further develop MonteFFI or making a similar simulator.

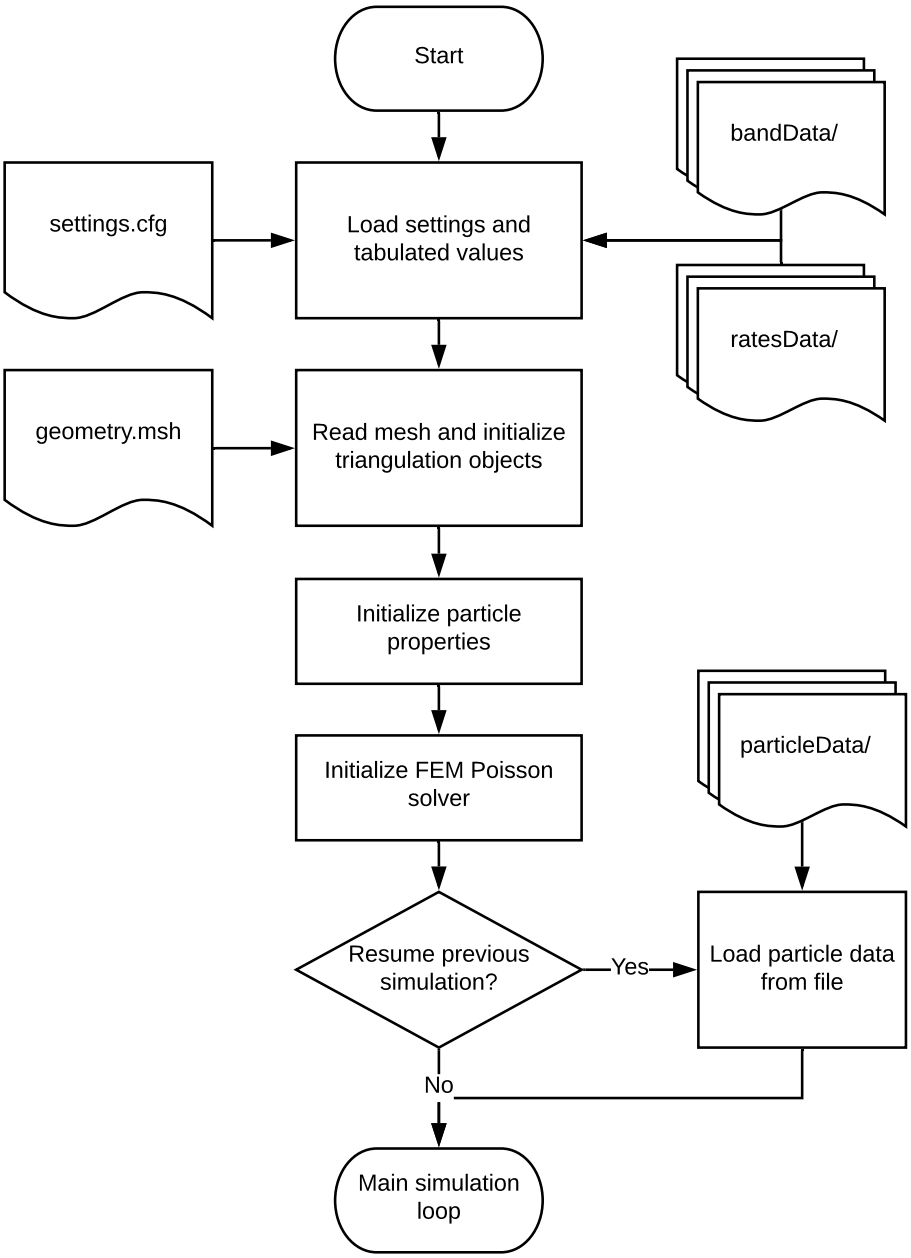


Figure 3.2: Control flow of initialization procedures in MonteFFI.

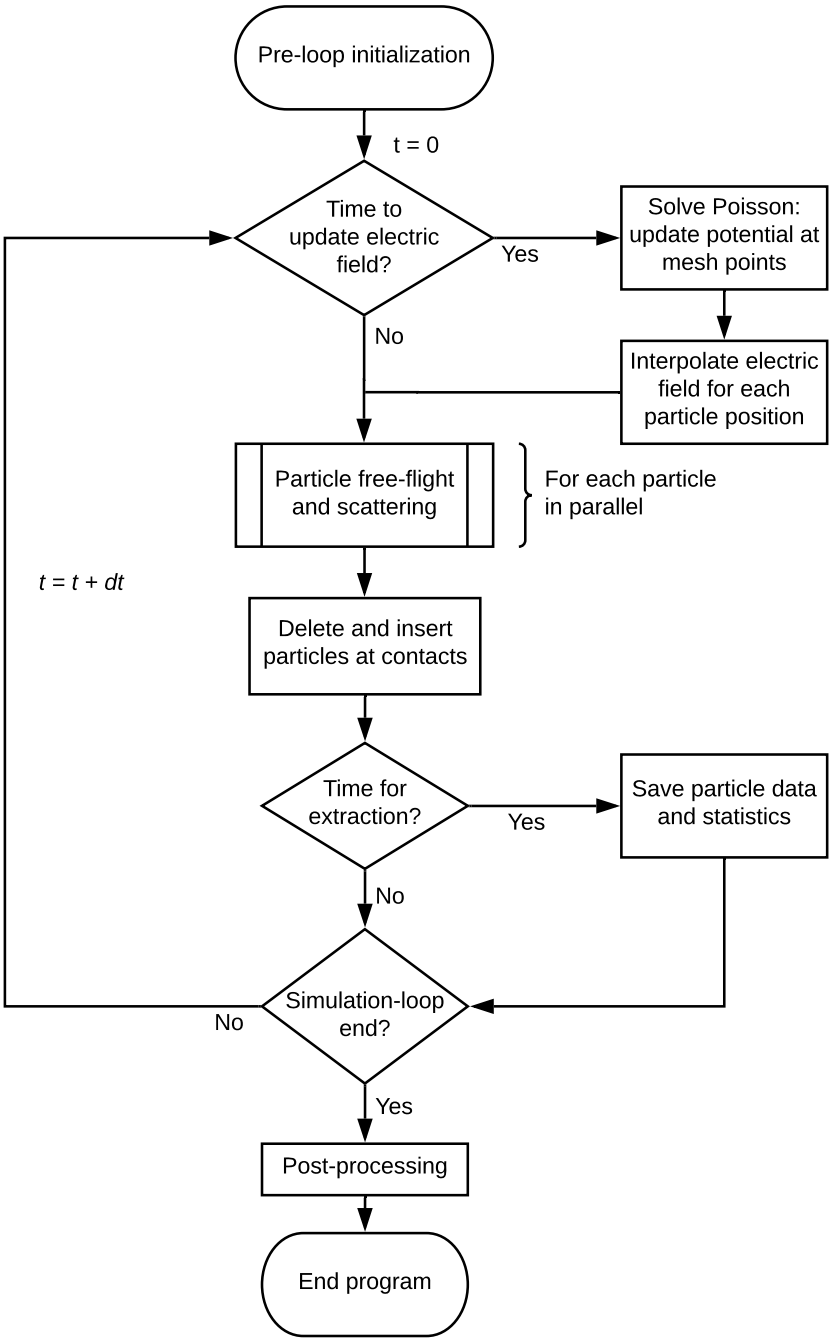


Figure 3.3: Control flow of the main MC simulation loop in MonteFFI.

Table 3.2: Development history of FFI-MCS. Note that this list describes what each developer worked on, which does not necessarily represent features fully integrated with maintained versions of FFI-MCS.

Year	Developer	Work description	Ref.
2007	H. Brox	Start of FFI-MCS: A bare-bones EMC simulator for bulk semiconductors.	N/A
2009	Ø. Olsen	Incorporated SCRATES [33] for pre-calculating scattering rates. Prototyped carrier-carrier scattering and Pauli exclusion modules.	[36]
	O. C. Norum	Added multiple scattering mechanisms and a basic 2D FD Poisson solver.	[37]
2010	Ø. Skåring	Improved accuracy of Pauli exclusion, hot phonon and screening mechanisms.	[38]
2011	C. N. Kirkemo	SCMC simulation: Introduced an improved 2D FD Poisson solver and simulated PN-junctions and CMT APD devices.	[6]
2012	A. J. V. Vestby	Used Shockley-Ramo analysis to calculate terminal currents in single photon excited APDs.	[39]
2013	K. V. Falck	Studied Auger-recombination models.	[40]
	B. Karlsen	Provided full-band tables created with $\vec{k}\cdot\vec{p}$ and *ab initio* methods.	[35]
	T. S. Bergslid	Enabled use of full-band structures to calculate scattering rates and selection of final states.	[41]
2014	J. Selvåg	Improved precision for selection of final states.	[42]
2015	J. J. Harang	Created a 2D FD Poisson solver for tensor grids.	[7]
2016	T. Chirac	Simulated photoconductive terahertz switches.	[43]
	D. K. Åsen	Created a self-force-free 2D FE Poisson solver.	[8]
2017	D. Goldar	Studied wavefunction overlaps with Wien2k.	[44]
2018	M. Haug	Studied NEGF and Schrödinger-Poisson techniques.	[45]
2018	S. N. Fatnes	Created a 3D FE Poisson solver.	[9]
2019	M. Estensen	Studied optimal mesh generation.	[46]
2020	A. Bolstad	Parallelized single-carrier flight and scattering.	[47]

4 Implementation

This chapter presents the implementation of MonteFFI, starting with section 4.1 where the main development tools used throughout this work are presented. The first part of development was fixing the existing program MCFEM as described in section 4.2. The program was restructured to incorporate full-band features, requiring flight and scattering to be re-implemented as presented in section 4.3. With a functioning program in place, the next step was to make a workflow to generate device discretizations for unstructured grids, presented in section 4.4. Finally, sections 4.5 and 4.6 present the components coupling particles with the mesh, adding the final missing components needed to simulate devices with complex 3D geometry.

4.1 Development Tools

It is important that running MonteFFI will not require several days of having to deal with computer science concepts and programming tools, considering the primary user is an expert in physics and not Linux or high performance computing. The below described tools have been selected and applied to MonteFFI to facilitate easy development and fast program execution.

Gmsh To generate device discretizations I have used Gmsh[1], an open-source mesh generator for 3D finite element solvers specifically designed to be *fast, light and user-friendly* [34]. Previous developers of FFI-MCS [9, 8] have also used Gmsh because it is free and widely used. Developers at Swansea have for more than a decade been using their in-house meshing software [48], but in a recent (2019) paper [49] it is specifically stated that they are now using Gmsh for 3D SCMC simulation, which is a good indication that Gmsh well suited for our purposes. A list of other potential meshing suited for our purposes are listed in [50, ch. 1.2]. More details of how Gmsh was applied in this work is covered in section 4.4.

Modern Fortran MonteFFI is written in Fortran 2008, a general-purpose and compiled programming language well suited to numeric and scientific calculations. Object-oriented features, added with Fortran 2003, have been used to make the program more flexible and compact, greatly reducing

[1]The Gmsh project's official website: `http://gmsh.info/`. Last accessed: 03.05.2020

time spent on writing code and improving readability. However, class abstraction was generally avoided in computationally demanding routines, as it can make performance hard to analyze and improve.

Modern Fortran language standards[2] were strictly followed to compile the MonteFFI on any standard compiler. This required substantial modifications to FFI-MCS's code originally written in Fortran 90. Hundreds of bugs were discovered and removed in the process. The debugging effort presented in section 4.2 would likely not have succeeded without this code cleanup.

Automated Code Building GNU Make, or simply Make, is a standard tool for compiling programs on Unix systems. Other versions of FFI-MCS have used simple bash scripts for compilation, but I made a transition to Make for two main reasons: 1) With more than 50 source code files in MonteFFI, manually maintaining the ordered list of module dependencies required for compilation is cumbersome and error-prone. 2) Building (compilation and linking) can take a long time, slowing development. The current build time is close to a minute when compiling from scratch, and will further increase as more features are added. Using Make in combination with a script called *fortdepend*[3] for automatic dependency file generation, the end user can build and install MonteFFI without having to know any details of the program.

The Makefile (build recipe) created for this project provides support for both Gfortran and Ifort compilers. It picks up all files in the source code folder, so that the code can be altered without restrictions on repository structure. However, the Makefile must be altered sometimes, for example to change settings for working on computing clusters. Make's scripting language is powerful but often confusing, and simple yet feature rich Makefiles for Fortran programs are hard to come by online. Therefore, I made a new thoroughly commented Makefile from scratch, which is presented in appendix B.3.

Shared-Memory Parallelization Parts of MonteFFI were parallelized using the Open Multi-Processing library (OpenMP). OpenMP is the de facto industry standard application programming interface (API) used to create multi-threaded (parallel) programs for shared-memory systems [51]. It is platform independent and included in most Fortran and C compilers by default. Code regions to be parallelized are indicated with compiler directives written directly in the code, which made it straightforward to parallelize the Fortran code of MonteFFI.

Profiling Tools In order to analyze computational performance, the following profiling tools were used:

Gprof is a fast and easy-to-use tool that provides sampling and call-graph profiling. It was the preferred tool when quick analysis was required.

[2]Fortran standards website: `http://fortranwiki.org/fortran/show/Standards`
[3]Fortdepend on GitHub: `https://github.com/ZedThree/fort_depend.py`

Valgrind is a package with several tools which also are easy to use. In this work, the Callgrind tool of Valgrind was used to get detailed line-by-line profiling. However, using Callgrind results in a performance penalty, typically increasing execution time more than 50 times. Thus, Valgrind was used mainly when Gprof could not provide sufficient detail.

HPCToolkit can do what Valgrind does while being much less computationally costly and supporting performance analysis of parallelized code. However, HPCToolkit was more difficult to use and install compared with Gprof and Valgrind. HPCToolkit was therefore only used when computationally efficient profiling was required or parallel performance was a concern.

Configuration File Parser A configuration file parser was added to MonteFFI as its own Fortran program module, which is a modified version of "config_fortran"[4]. The parser is used to read and write settings to and from files, which allows using the same source code for various simulations without recompiling. Any primitive data type in Fortran and one-dimensional arrays of these types are supported. The program can also read in multiple configuration files, and options may be overwritten in the command line. In total, this makes it much faster to modify and save settings, which reduces the time spent on fine-tuning simulations and preparing multiple similar simulations.

Version Control A private repository on GitHub was used to host the source code of MonteFFI. This provided easy version control through Git. Larger files, such as tabulated bands, were stored in compressed format to stay below GitHub's free storage quota. In sum, this made it easy to share the code with others, debug and maintain multiple program versions, and upload and run prepared simulations on computing clusters.

4.2 Fixing MCFEM

This section serves as an erratum to Fatnes's thesis [9], and motivates the emphasis in this work on program maintainability. The 3D FE solver created by Fatnes in 2018 was shown to solve the Poisson equation accurately and efficiently, but MC simulation results were unphysical. This was mainly caused by two errors resolved in this work.

4.2.1 Scattering Error

An unexpected phenomenon of "freezing holes" was first noticed by Estensen in 2019 [46], which has only been a problem with the analytical band version (not the full-band version of FFI-MCS). After a few picoseconds of simulated time, a majority of holes would completely stop moving.

[4] config_fortran on GitHub: `https://github.com/jannisteunissen/config_fortran`

Estensen found that this is because whenever a particle obtained a k-value outside the first Brillouin zone it was set to zero. He wrote that an Umklapp process should be implemented but did not address the issue further.

The existing function for computing particle velocity assumed input wavevectors are in the first Brillouin zone (BZ), and instead of aborting with an error message the function would return zero velocity if the wavevector was outside the BZ, letting erroneous input pass through unnoticed.

The program should not allow k-values outside the BZ, which can be enforced with periodic boundary conditions on reciprocal space. A procedure was implemented according to algorithm 4.1 to move any wavevector back to the BZ. The procedure is called whenever a particle's k-value is updated in flight and scattering routines, so that any process adding a momentum Δk causing $k + \Delta k > k_{\max}$ is treated as an Umklapp process. The full-band program already has such an algorithm, but with the zincblende BZ for CMT instead of the spherical BZ used in the analytical program.

Algorithm 4.1 Periodic Boundary Conditions of Reciprocal Space with Spherical Brillouin zone.

1: $k \leftarrow ||\vec{k_i}||$
2: **if** $k < k_{\max}$ **then**
3: $\quad \kappa \leftarrow [(k + k_{\max}) \mod (2k_{\max})] - k_{\max}$
4: $\quad$ **return** $\frac{\kappa}{k}\vec{k_i}$
5: **else**
6: $\quad$ **return** $\vec{k_i}$

With the new BZ function, the holes stopped freezing, but new erroneous behavior appeared. Holes moved increasingly erratic with time and plotting the average energy per particle over time showed a near linear growth. Several possible causes were examined one by one, including unreasonably large electric fields and instability due to time or mesh refinement. Ruling out possibilities was a difficult and slow process because the other error covered below was also contributing to the unexpected behavior.

A call graph generated with Gprof provided a surprising insight, showing the procedure for selection of particle state after scattering was *not* a significant consumer of CPU resources. As discussed in my project thesis [47, ch. 3], state selection is one of the most computationally demanding parts of MC simulation. This was indicative of no scattering events being simulated, contradictory to the statistics sampled during simulation that showed reasonably high scattering frequencies.

The error was finally identified after a thorough code inspection. After choosing a scattering mechanism and saving statistics, the mechanism selector would be reinitialized to zero (instead of another similarly named variable), causing self-scattering to occur instead of the selected mechanism. Note that this component is not used in MonteFFI, which uses the newly implemented full-band version described in section 4.3.

In summary, the "freezing holes" error was caused by the following combination of program deficiencies:

1. Holes gained unreasonably high energies because of no scattering,
2. causing k-values to leave the BZ without being returned to the BZ,
3. causing the hole velocity procedure to incorrectly return zero velocity.

Removing the erroneous code and patching the other program issues resulted in more stable simulations. However, it was only after removing the second error described later that MCFEM simulations started producing physically reasonable results.

4.2.2 Scaling Error

Problem Description

A second error was identified in MCFEM during this work, which I have called "speeding electrons". In the APD bias simulations, electrons would rapidly accelerate towards contacts and exit the device. This behavior can be seen in figures 7.6-9 in Fatnes's thesis [9], where almost all electrons have left the device after a few picoseconds.

The potential is expected to increase in areas where electron density is decreasing, but this effect was not observed. On the contrary, the change in potential after most electrons had left the device was negligible. This indicated that an incorrect representation of particle charges was used in the Poisson solution.

Testing with doubling the charge per carrier had no noticeable impact on the resulting potential. However, through trial-and-error, it was found that a charge per carrier 10^6 times higher resulted in the electric potential expected for a functioning simulation. This pointed to an error with scaling, where values equal 10^6 were present.

Scaling System

A short explanation of the scaling factors and their appropriate values is required to understand the erroneous behavior. Scaling is used to create a dimensionless set of equations, which in this is applied to equations (2.12) and (2.13) from electrostatics.

We define four scaling equations, each with a variable in SI units on the left-hand side and a scaling factor times a dimensionless variable on the right-hand side:

$$\begin{aligned} \mathbf{x}^* &= L\mathbf{x} & u^* &= Uu \\ q^* &= Qq & \mathbf{E}^* &= E\mathbf{E} \end{aligned} \tag{4.1}$$

Using that $[\nabla^2] = \mathrm{m}^{-2}$, and in 3D $[\delta(\mathbf{x}^*)] = \mathrm{m}^{-3}$, we find the dimensionless Poisson equation

$$-\nabla^2 u = \alpha \sum_{p=1}^{N_p} q_p \delta(\mathbf{x} - \mathbf{x}_p), \quad \alpha = \frac{Q}{UL\epsilon_r\epsilon_0}, \tag{4.2}$$

and the dimensionless electric field

$$\mathbf{E} = -\beta\nabla u, \quad \beta = \frac{U}{LE}. \tag{4.3}$$

The values of each scaling factor currently implemented in the program are listed in table 4.1.

Table 4.1: Values of scaling factors used in MCFEM and MonteFFI for electrostatic equations (4.2) and (4.3). q_{sup} is the (positive) charge of a superparticle, which is evaluated during runtime.

Symbol	Unit	Value
L	m	10^{-6}
U	V	1
Q	C	q_{sup}
E	$\mathrm{V\,m^{-1}}$	10^6

Solution

The error's cause was identified by examining Fatnes's thesis. Two scaling factors $(UL)^{-1} = 10^6$ were found missing from the variable α defined in equation (7.4) [9, p. 55], and the same incorrect equation was found implemented in MCFEM. Adding the scaling factors, we get

$$\alpha = \frac{q_{\mathrm{sup}}}{UL\epsilon_0\epsilon_r}, \tag{4.4}$$

where q_{sup} is the (absolute) charge of the superparticle. Replacing the old α in MCFEM with α in equation (4.4) resulted in the correct potential contribution from charge carriers. It should also be noted that a factor α is missing from several equations in [9, ch. 2]. Replacing q_p with αq_p wherever α is not already present corrects the equations.

4.3 Renewed Flight and Scattering

Adding new functionality to the MC program requires a good understanding of how the flight and scattering algorithm is implemented. However, maintaining a clear structure for the program procedure can be difficult because of the large number of mechanisms incorporated in it. The procedure's structure in previous versions of FFI-MCS were suitable for a small program, but with

increased program size it has become cumbersome and error-prone to add and remove functionality. Thus, a new flight and scatter procedure was built for MonteFFI, which will help facilitate many years of future development.

To numerically evaluate equation (2.6) I have implemented the constant time method by Yorston [21]. The method is designed to make the number of self-scattering events low by producing a function $\Gamma(t)$ which closely fits the local, real, scattering rate.

The constant time method is one of the more robust and efficient algorithms and is especially well suited for self-consistent simulation. The particle ensemble in EMC simulation is synchronized at every time step, allowing for field updates at regular intervals. Nonlinear effects, for example carrier-carrier interactions and Pauli exclusion mechanisms, can be separately updated each time step after the main flight and scatter routine [16, p. 10].

Parallelization is applicable to the particle flight and scattering loop since each particle is simulated independently. The shared-memory parallelization approach presented in my project thesis [47] was used to parallelize MonteFFI, and increased execution speed of flight and scattering by about 18.6 times for 20 CPU cores.

4.3.1 Constant Time Method

The constant time method works as follows. Let $\Gamma(t) = \lambda(t) + \lambda_0(t)$, where $\lambda(t)$ is the total scattering rate and λ_0 is the self-scattering rate. In the constant time method, $\Gamma(t)$ is piecewise constant, and every discontinuity occur at multiples of a fixed time increment Δt. The constant value of each interval is determined by

$$\Gamma(t) = \max(\lambda(t_1), \lambda(t_2)), \quad t \in [t_1, t_2]. \tag{4.5}$$

Figure 4.1 illustrates how we evaluate the integral of equation (2.6). Partial integrals ΔI_j for each time interval j are summed up until $\sum_j \Delta I_j \geq -\ln r$. The difference between $-\ln r$ and the accumulated integral value is used to determine the time t_s of the scattering event.

After simulating the scattering event (determining a new particle state), the integral from the scattering time t_s to the end of the time step t_2 is evaluated and a new value $-\ln r$ is generated. This is done repeatedly if necessary, to resolve any number of scattering events during a single time step.

Note that previous versions of FFI-MCS could only resolve scattering events at the end of flight intervals [42, p.48]. Thus, MonteFFI is the first version of FFI-MCS with a complete implementation of the constant time method.

4.3.2 Flight and Scattering Control Flow

An overview of the flight and scattering procedure is given by the control flow in figure 4.2. Aside from the integral evaluation, the rest of the procedure is captured by two major components:

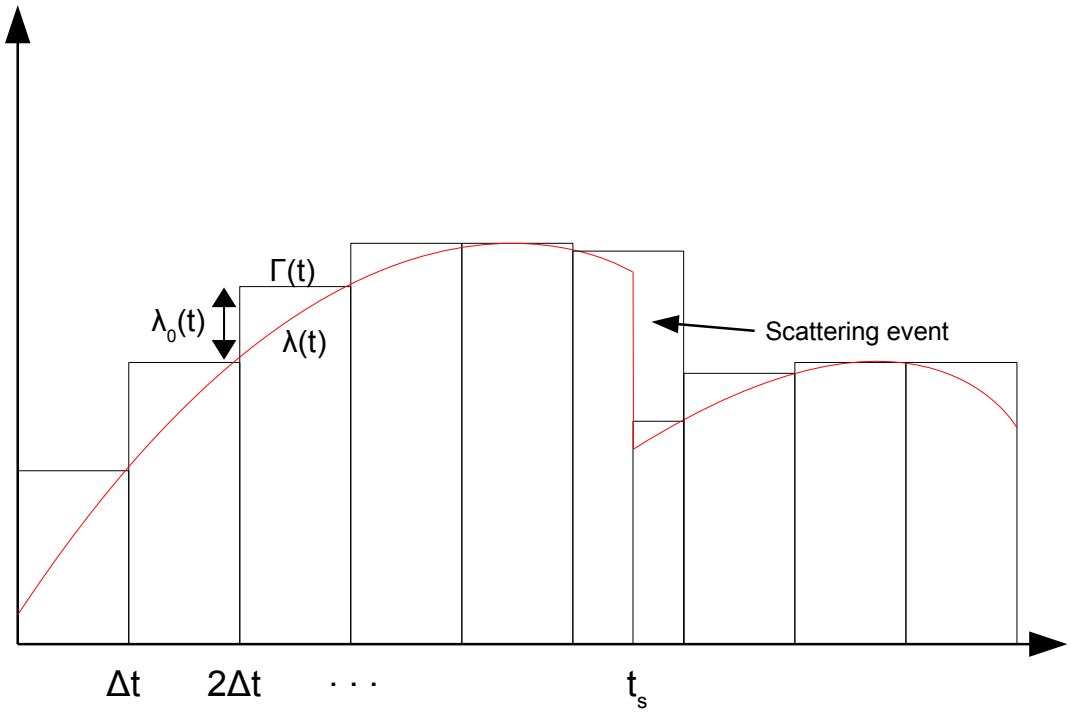


Figure 4.1: Illustration of the scattering rate integration over time using the constant time method.

1. Particle flight and boundary conditions, called FLIGHTANDBC in MonteFFI. When changing type of simulation from 3D to bulk or 2D device simulation, the developer mainly needs to change FLIGHTANDBC. This component is further discussed later in this thesis.

2. Selection of scattering mechanism and final state, simply called SCATTER in MonteFFI. This component was taken from the latest full-band version of FFI-MCS, only receiving minor interface changes.

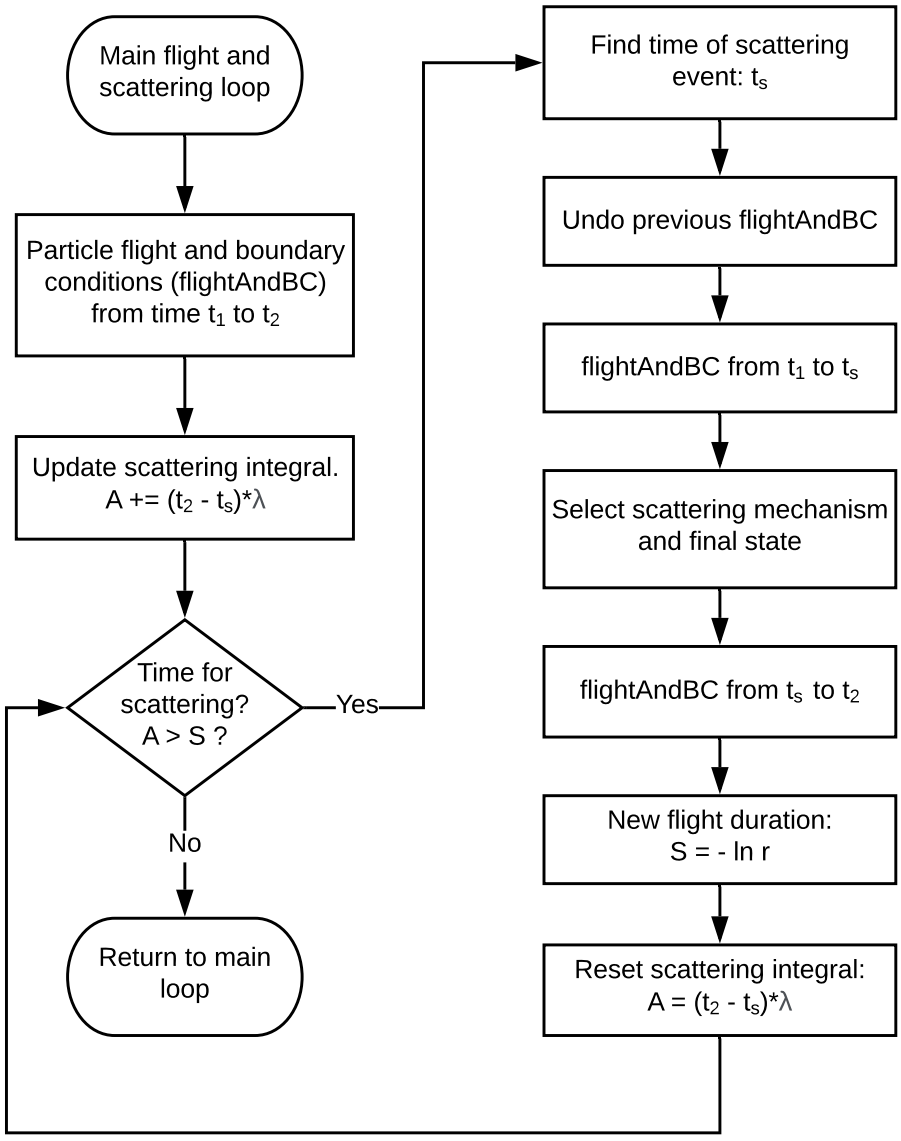


Figure 4.2: Simplified control flow of the particle flight and scattering procedure using the constant time method after Yorston [21].

4.4 Device Discretization with Gmsh

Before running the MC simulation, we require the unstructured grid that represents a device discretization as input (as explained in section 2.4). This section presents how device discretizations were made for MonteFFI using three capabilities of Gmsh:

Geometry Definition Complex 3D device geometry can be made both using built-in features of Gmsh or importing geometry from other programs.

Mesh Generation With a great range of geometric utilities and local grid refinement[5] techniques, Gmsh allows for rapid development of optimized unstructured meshes.

Physical Tags Groups of geometrical entities can be given an ID, called a "physical tag" in Gmsh, which is exported as an integer in the mesh file.

Workflow The pre-simulation device discretization with Gmsh has an overall workflow that can be summarized as a series of steps, usually in the following order:

1. Specify device geometry
2. Split the device into regions by assigning physical tags to geometrical entities.
3. Specify mesh resolution with a background field.
4. Save Gmsh (.geo) script, generate mesh, and export mesh (.msh) file.
5. Specify settings for each region in configuration files.
6. Run simulation with mesh and configuration files as input.
7. (Optional) Make an adaptable .geo script (replace constants with variables)

More details regarding the individual steps are provided throughout the rest of this section.

Script and GUI Gmsh comes with its own graphical user interface (GUI) and scripting language, which can be combined to make an efficient workflow. In addition, Gmsh features automatic script generation based on GUI input. In other words, the actions done in the GUI are saved as lines of code in a script (.geo) file. During and after model creation, the script can be edited to replace parameters, from hard-coded numbers to adjustable variables. This provides a way to effectively reuse models. For our case, it is particularly useful for the following reasons:

[5]To refine a mesh usually means to increase mesh density in regions requiring higher accuracy and reducing mesh density where less accuracy is needed in order to increase efficiency.

- For efficient optimization of device dimensions, we want the ability to easily adapt geometry parameters. Reconstructing the whole model for each simulation is cumbersome.
- Accuracy and computational efficiency requirements can frequently vary. Adapting an existing mesh by tuning variables in a script is usually much faster than generating a new mesh from scratch.

Drawing Geometry There are mainly two ways of generating geometry in Gmsh:

1. From the ground up: Define a set points, connect two points or extrude a point to form a line segment, and similarly one can make surfaces from lines and volumes from surfaces. This approach was used in Fatnes's work with MCFEM [9, App. A].
2. Using predefined geometry: Generate geometry objects using Gmsh's built-in classes or import existing CAD models. More complex objects can be generated using Boolean operations, for example the difference of two concentric cylinders can generate a hollow cylinder. The lower-dimensional entities of the 3D object (points, lines, and surfaces) are generated automatically.

Defining each point manually is much work when creating more than basic geometry, and possibly an unusable approach when advanced geometry is required. Therefore, mainly the second approach was used in this work.

Overlapping Surfaces A difficulty when designing a device comprised of disjoint regions is avoiding overlapping entities. Consider a scenario where we add two adjacent boxes such that one side from each box is overlapping the other. Unfortunately, the overlapping surfaces do not automatically combine to a shared surface. If we use this geometry the resulting mesh would be unconnected and thus unusable for simulation. To solve this type of problem, a particularly useful Boolean operation in Gmsh is `BooleanFragments`. It takes two sets of geometrical entities as input and returns the fragments, meaning the intersection and the differences of the two geometrical sets as illustrated in figure 4.3. Thus, we may split the device geometry and replace overlapping entities with shared entities in a straightforward and intuitive manner.

Mesh Refinement One approach to mesh refinement which has been used by previous developers of FFI-MCS is to specify the desired grid spacing at each point, which Gmsh uses to automatically generate a mesh by interpolating grid spacing values between points. However, this approach is not well suited on its own unless we are constructing the geometry point by point. Fortunately, grid spacing can be controlled in other ways. Presented here is an easy-to-use and generally applicable mesh refinement approach for our purposes.

Mesh refinement can be controlled in Gmsh by specifying a background mesh size, comprised of so-called mesh size fields. There are many field types to choose from, with varying degrees of complexity. The following mesh size fields are well suited for this purpose:

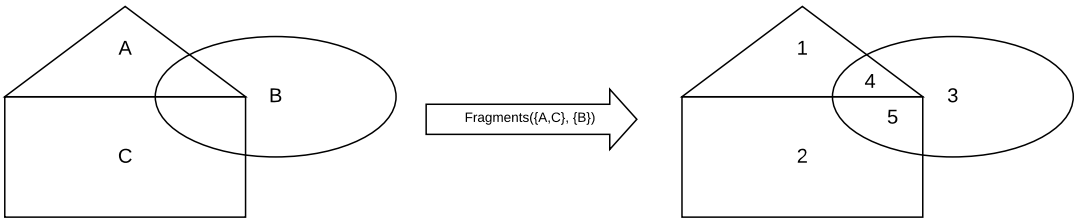


Figure 4.3: Illustration of Gmsh's `BooleanFragments` function, taking two sets of geometrical entities as input and returning the "fragments" of the two sets. On the left-hand side, regions are overlapping. On the right-hand side, there is no overlap between regions except shared borders.

- The `Box` field specifies element size inside and outside a parallelepipedal region.
- The `Min` field specifies element sizes using the minimum of other fields.
- The `Distance` field in conjuction with a `Threshold` field specifies element sizes as a function of the distance to specified entities.

This approach provides intuitive and fine control of local refinement and puts few restrictions on using Gmsh's many other features to create more advanced meshes.

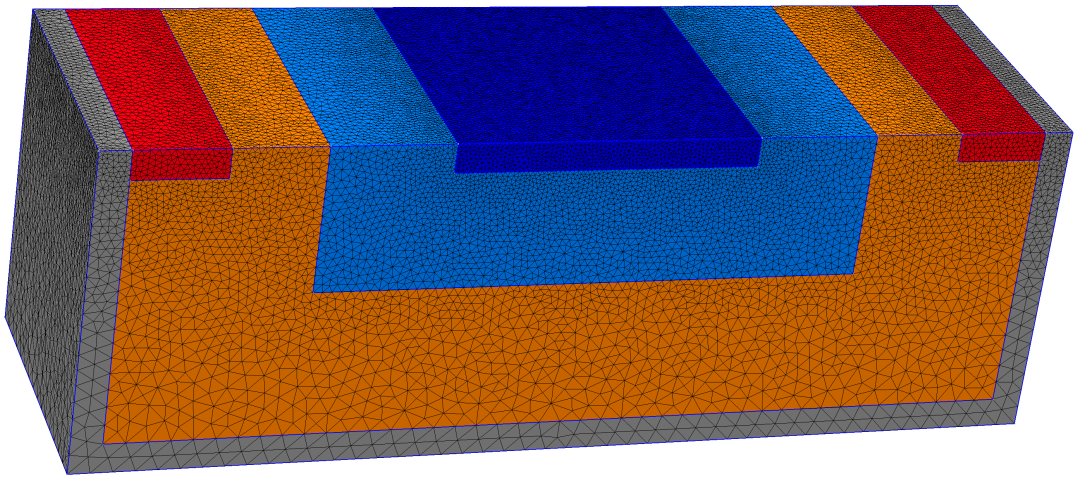
A basic mesh can be created by setting the background field to a `Min` field of geometrical fields such as `Box` field. Figure 5.2 shows a simple mesh using these features next to a mesh generated with element sizes specified at points. The `Distance` and `Threshold` fields are particularly useful to increase refinement along surfaces of interest, such as PN-junctions. In sum, these features are well suited for specifying mesh refinement for a large range of device geometries.

Identifying Regions In the device simulation we have different regions as described in section 2.4.1. With advanced geometry, it is impractical to hardcode the geometry into the simulator. Instead we can use a device discretization represented by the mesh as described here.

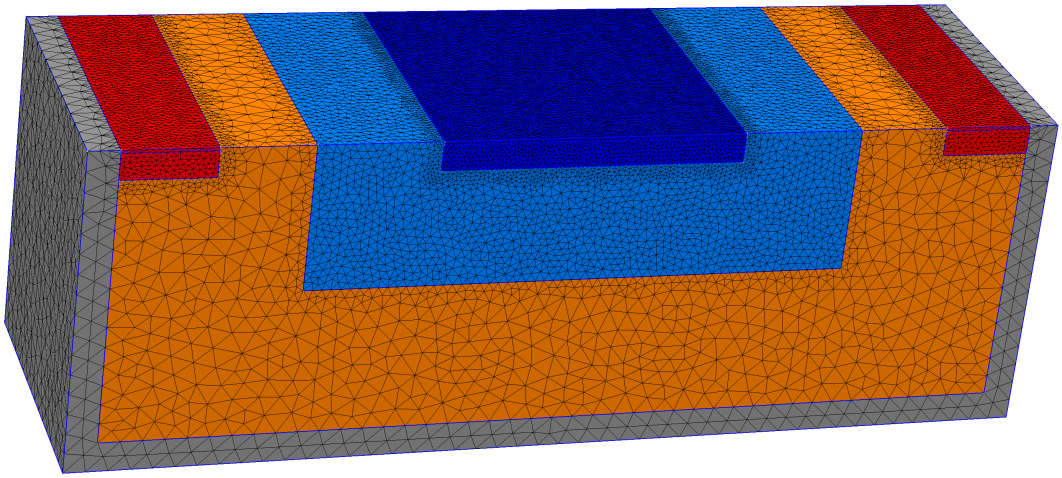
Gmsh possesses a feature for tagging geometrical entities. The intended purpose for the tags by Gmsh's developer's is to easily specify boundary nodes for a finite element solver. This is also used in MonteFFI, a feature previously developed by Fatnes, to identify which nodes to impose Dirichlet boundary conditions on. I have used this feature to also identify which elements belong to which device region.

When the final mesh is exported, each element is stored in the exported .msh file with the physical tag of the region it belongs to. This causes any element generated in the tagged region to store the region's tag with it, which is a positive integer the user specifies when creating the region.

A group of parameters in a configuration file represent the settings for each region, and each group is identified by its physical tag. Thus, the mesh and configuration file serve as a representation of the device discretization applicable for MC simulation.



(a)



(b)

Figure 4.4: Example meshes generated with Gmsh for an APD model, with grid spacing specified at points in (a) and specified with a `Min`-field of several `Box`-fields in (b). The colors show the physical tag specified for each element, indicating which device region each element belongs to. Each different region is given different grid spacing to reduce the number of elements in regions where we require less elements to get accurate results. Both presented meshes have the same grid spacing along region borders, but mesh (b) has $19.4\,\%$ fewer elements compared to (a). This is not a major difference, but for more complex meshes the difference can be much greater, particularly when the highest and lowest mesh resolutions are far apart.

4.5 Particle Initialization

Before simulation can start, we need to determine the number of particles and select a location for each particle. This is a straightforward task with parallelepipedal geometry, which in previous versions of FFI-MCS was solved by hardcoding these features for each model. The matter is more complicated with complex geometry. In this section I show how particles can be initialized for any geometry by using unstructured grids, which has the additional benefit that the user does not have to change any source code when switching between device models.

4.5.1 Superparticles

Before we can initialize any superparticles, we need to determine the charge and number of particles to initialize. Here we will use that the device grid features physical tags for each element as detailed in section 4.4. The total number of superelectrons N_s^- and the impurity density n_i of each device region (each physical tag) i are specified by the user in config files. The remaining parameters for superparticle initialization can then be determined computationally as presented below.

The volume of each individual element is calculated during initialization and stored in a table for repeated use later on. The equation used to calculate the volumes is

$$V = \frac{|(\mathbf{p}_2 - \mathbf{p}_1) \cdot (\mathbf{p}_3 - \mathbf{p}_1) \times (\mathbf{p}_4 - \mathbf{p}_1)|}{6}, \tag{4.6}$$

which applies for any tetrahedron with vertices $\mathbf{p}_i$. The volume of each region i is computed using

$$V_i^{Region} = \sum_{\text{Each element with physical tag } i} V_j^{Element} \tag{4.7}$$

Next, we find the number of charge carriers in each region,

$$N_i = \sum_i V_i n_i, \tag{4.8}$$

and calculate the total number of electrons

$$N_{tot}^- = \sum_{\text{All negatively doped regions}} N_i. \tag{4.9}$$

The number of particles per superparticle is

$$R_{\text{sup}} = \frac{N_{tot}^-}{N_s^-}, \tag{4.10}$$

and thus we can calculate the number of superholes

$$N_s^+ = \left(\sum_{\text{All positively doped regions}} N_i \right) / R_{\text{sup}} \tag{4.11}$$

and determine the (unsigned) charge per superparticle

$$q_{\text{sup}} = eR_{\text{sup}}. \tag{4.12}$$

4.5.2 Particle Positions

With the number of superparticles initialized, we are ready to initialize positions for the charge carriers. Now we need a method for placing particles according to the device model's carrier distribution $n(\mathbf{r})$. This is simple in principle, but the implementation can be complicated as it depends on how $n(\mathbf{r})$ is represented.

I have created an approach suitable when the carrier density is defined in each geometrical region, allowing for arbitrary 3D geometries discretized with tetrahedra. Particle positions are selected by repeatedly executing two steps:

1. Weighted selection of an element based on the carrier density and element volume.

2. Selection of a random position inside a tetrahedron.

Note that electrons and holes are treated separately, but with the same procedures presented below.

Weighted Element Selection Assuming each element has a predetermined constant carrier density, which in our case is determined by which device region the element belongs to, we can use the following procedure. We compute the number of particles N_i inside each element i with

$$N_i = n_i V_i, \tag{4.13}$$

where n_i is the carrier density and V_i the volume of the element. With this we can create a carrier density weighted probability distribution for placing a particle in each element i,

$$P_i = \frac{N_i}{\sum_j N_j}, \tag{4.14}$$

with the discrete cumulative probability distribution

$$S_i = \sum_{j=1}^{i} P_j. \tag{4.15}$$

Now we can make a carrier-per-element-weighted element selection using the cumulative distribution. This is done by generating a random number $r \in U(0,1)$ and finding the element i where

$$S_{i-1} \leq r \leq S_i. \tag{4.16}$$

Thus, the problem of selecting an element has been reformulated to a pseudo-random number sampling from a discrete probability distribution, for which several efficient algorithms exist.

I have implemented "bisect left" into MonteFFI, which takes S and r as input and outputs i of equation (4.16). Based on binary search, it is a simple algorithm having a time complexity $O(lg(n))$ [52], where n is the number of discretization points in the cumulative distribution.

A popular alternative to bisect left is the Alias method, doing the same operation with a time complexity $O(1)$ [53, ch. 3]. While asymptotically faster than bisect left, the alias method is somewhat complicated to implement and use effectively. For the present purpose, bisect left is sufficiently efficient and more straightforward to use.

Random Point inside a Tetrahedron After selecting an element we need to randomly select a position inside. With tetrahedral elements, we need a function to make random samples from a uniform distribution on a tetrahedral domain. Points in a tetrahedron can be represented with barycentric coordinates, which have the useful property that $\sum_i \lambda_i = 1$. Thus, the problem is reduced to sampling random points on a unit simplex.

A simple method for selecting a random point in an arbitrary n-dimensional simplex is presented in [54, pp. 73-74]. A more thorough treatment of the method with a correctness proof is found in [55]. The implementation is presented in pseudo-code in algorithm 4.2, which can be executed with $n = 4$ to get the four barycentric coordinates required for a tetrahedron. Finally, the new particle position is computed by mapping the barycentric coordinates to a cartesian point

$$\mathbf{r} = \lambda_1 \mathbf{p}_1 + \lambda_2 \mathbf{p}_2 + \lambda_3 \mathbf{p}_3 + \lambda_4 \mathbf{p}_4, \tag{4.17}$$

Figure 4.5 shows that the implementation for generating positions uniformly on a tetrahedron work as expected.

Algorithm 4.2 Sample random vector over a unit simplex

1: **procedure** SAMPLEUNITSIMPLEX(n)
2: Generate n random values $X_i \in U(0,1)$
3: $\{X_{(i)}\} \leftarrow \{X_i\}$ sorted in increasing order
4: $Y_1 \leftarrow X_{(1)}$
5: **for** $i = 2$ **to** n **do** $Y_i = X_{(i)} - X_{(i-1)}$
6: **return** $\{Y_i\}$

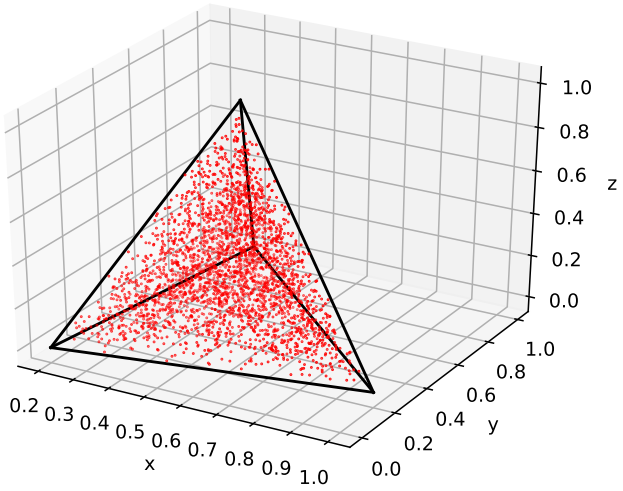


Figure 4.5: 3000 points sampled from a uniform tetrahedral distribution function. The vertices of the tetrahedral domain are (0.2, 0, 0), (1, 0.2, 0), (0.3, 1, 0) and (0.5, 0.5, 1).

Alternate Approach: Rejection Sampling An alternative approach was considered but was not implemented in MonteFFI. However, it could prove useful in the future, for example to quickly implement charge distributions independent of the mesh.

There are two components required for the rejection sampling:

1. A minimum bounding box B enclosing the device domain Ω.
2. A representation of the carrier density distribution $n(\mathbf{r})$ over the bounding box B.

Using positions uniformly distributed on B, the rejection method can pick out positions using the distribution

$$f(\mathbf{r}) = \frac{n(\mathbf{r})}{\max_{\mathbf{r}\in B} n(\mathbf{r})}. \tag{4.18}$$

This makes the resulting distribution of particles a stochastic approximation of $n(\mathbf{r})$.

A problem with the rejection method is its poor efficiency in certain corner cases. Consider a region with relatively high charge density enclosing the rest of the domain like a thin shell, such that its volume is a very small fraction v/V of the total domain volume V. In this case a very small fraction of iterations $\leq v/V$ are accepted, leading to a large number of total iterations. This can be remedied at the price of a more complicated algorithm. For example, the domain may be split into multiple regions, each being initialized independently.

The rejection approach is simple, but not always practical for complex domains. The implemented approach was preferred because it is generally more reliable and efficient. Furthermore,

program components for particle placement can be reused for particle injection as covered in the next section.

4.6 Particle Boundary Conditions

Treating particle boundary conditions accurately is of great importance to the simulation's accuracy in mimicking real device behavior [56]. A basic requirement is having an elastic reflection to keep particles confined inside device boundaries, and we require deletion and injection of particles for modelling contacts. Widely used methods for particle boundary conditions are described by Hockney and Eastwood [22], but are only applicable to structured grids without extending the methodology. Previous versions of FFI-MCS only works for rectangular cuboid devices, which if applied to MonteFFI defeats the purpose of having an unstructured discretization.

To support complex geometry, I have implemented a new module for modeling particle boundary conditions with unstructured grids in 3D. As there were no available open source code or algorithmic details from other 3D SCMC developers, I chose to implement this part of the program based on an approach outlined in the documentation [57] of OpenMC, "a high-fidelity MC simulator for neutron and photon transport" [58]. The program framework is built to facilitate the incorporation of a large range of surface interactions, particularly the mechanisms depicted in figure 2.2.

4.6.1 Reflection and Deletion

Particle Free-Flight At the beginning of a particle flight with boundary conditions we first compute the result of a free flight (flight without boundary conditions). From this we obtain the change in position $\Delta\mathbf{x}$ and an updated value of the wavevector $\mathbf{k}$. We define the distance traveled $d_0 = ||\Delta\mathbf{x}||$ for later use.

Detecting Surface Crossings Next, we determine whether a particle has crossed a surface or not. To this end I have used Fatnes's 3D point location method for unstructured grids [9, ch. 4], which is based on an algorithm by Chorda et al. [59]. The algorithm can be summarized as follows:

1. The input specifies the end point in cartesian coordinates $\mathbf{x}_p$.
2. The search begins from a point $\mathbf{x}_0$ at the center of some element K_0.
3. The search direction is $S = \mathbf{x}_p - \mathbf{x}_0$.
4. Having stored the neighbours of each element, the algorithm traverses elements from $\mathbf{x}_0$ along the search trajectory until it finds the element containing $\mathbf{x}_p$.

Note that the algorithm requires the mesh to be convex. Otherwise the program can crash because the search can leave the domain when the trajectory passes through void regions (outside the mesh).

To detect particles crossing over to isolation or void regions, the point location algorithm was modified as follows:

1. $\mathbf{x}_0$ is set to the particle location at the beginning of flight (not the center of the element). This is necessary to get the correct intersection line.

2. As the algorithm traverses the mesh, if it encounters an element of isolation type, or no element at all (void), it returns the intersection surface X_s and the element K encountered before intersection. Otherwise, it returns the element enclosing the end point as before.

If a surface crossing is detected, a boundary condition model is selected based on which region K belongs to. If K is in a contact region the particle is marked for deletion, which means it will be removed from the simulation after exiting the flight and scatter routine. Else, if K is in a "normal" region, the particle is reflected off the surface.

Surface Reflection To reflect the particle specularly off the located surface, we first determine the intersection point $\mathbf{x}_s$ of a line along the particle's trajectory with the intersection surface using basic vector algebra. Then, the distance from the original particle position to the intersection point, $d_1 = ||\mathbf{x}_s - \mathbf{x}_0||$, and the distance traveled after reflection, $d_2 = d_1 - d_0$, are computed.

Next, the new flight direction is determined using the Householder transformation [60, ch. 11.3.2]:

$$\mathbf{v}' = \mathbf{v} - 2(\mathbf{v} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} \tag{4.19}$$

where $\hat{\mathbf{n}}$ is the surface's unit vector normal. This can be understood intuitively by noticing that $(\mathbf{v} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$ is the projection of $\mathbf{v}$ onto $\hat{\mathbf{n}}$. At last, we find the new point after reflection by moving a distance d_2 from $\mathbf{x}_s$ along direction of $\mathbf{v}'$.

In some cases, for example a corner in a cubic device, multiple reflections may occur during a single flight. In this case we update the parameters $\mathbf{x}_0 \leftarrow \mathbf{x}_s$ and $d_0 \leftarrow d_1$, and repeat the reflection process.

4.6.2 Ohmic Contact Model

An ohmic contact (OC) is a device region characterized by local charge neutrality and thermal equilibrium. Several approaches for reproducing this behavior is described by González and Pardo [56]. They concluded the best applicable approach is to delete particles crossing the OC boundary and inject new particles as needed to maintain a neutral contact region. For structured grids, the OC region is comprised of the cells adjacent to the OC boundary. For unstructured grids, the OC model needs to be generalized to facilitate complex geometries.

Fatnes [9, ch. 5] has already worked on this problem, and while her approach has its merits it is limited to planar contacts. This is because the approach relies on defining a fixed contact depth in one direction. Moreover, her program implementation only allowed rectangular contacts lying in the xz-plane. Therefore, I have implemented a new OC scheme, largely based on the approach proposed by Aldegunde et al. [1].

Contact Region Construction Aldegunde et al. constructed contact regions by locating all volume elements with one or more nodes on contact surfaces defined before simulation. My implementation does the opposite by constructing contact regions directly in Gmsh. During initialization of a simulation, each contact surface is created from triangles located along the border between contact and isolation/void regions. This makes it easy to compute the surface normal of each triangle into the device needed to specify particle injection velocity. My model gives increased control over OC region size, which can be crucial for several applications where accurate OC modeling is required. This framework can also be useful for a future implementation of more advanced, non-Ohmic, contact models. The construction of contact regions requires an extra step compared to Aldegunde et al.'s approach. Fortunately, the user only needs to extrude a (contact) surface in Gmsh to make a contact region, which is an elementary operation.

Determining Charge Deficit Computing the charge deficit of an OC region is straightforward. First, we fetch the number of background impurities determined during initialization as described in section 4.5.1. Next, we locate the number of particles currently in the region. For a negatively doped contact region we calculate

$$N_{\text{insert}} = N_{\text{impurity}} - N_{\text{electrons}} + N_{\text{holes}} \tag{4.20}$$

and vice versa for positively doped contacts.

Selecting Injection Points Each particle is assigned to a surface triangle after weighted selection using the area of each triangle, analogous to the random tetrahedron selection presented in section 4.5.2. Afterwards, we need a random position inside the selected triangle. Using algorithm 4.2 with $n = 3$ generates three random barycentric coordinates λ_i needed to represent a point on a triangle. Then the cartesian coordinate point is found using the transformation

$$\mathbf{r} = \lambda_1 \mathbf{p}_1 + \lambda_2 \mathbf{p}_2 + \lambda_3 \mathbf{p}_3, \tag{4.21}$$

where $\mathbf{p}_i$ are the triangle's vertices.

Injection Velocity Injected particles require a velocity *into* the device. A preexisting routine from FFI-MCS is used to sample a wavevector $\mathbf{k}$ from a Maxwellian distribution, which is also used

when initializing particles at the beginning of a simulation. Already having tabulated the surface normal $\hat{\mathbf{n}}$ of each triangle into the device, we can evaluate $\hat{\mathbf{n}} \cdot \mathbf{k} < 0$. If the relation is true, $\mathbf{k}$ is reflected according to equation (4.19). This effectively inserts a particle into the device with a velocity sampled from a half-Maxwellian distribution.

According to Pardo and González [56], a more physically correct implementation is to use a velocity-weighted half-Maxwellian distribution. This is intuitive since a particle with a higher velocity have a higher chance to enter the device. A thorough description of sampling wavevectors from a velocity-weighted half-Maxwellian distribution is provided by Lundstrom [26, pp. 274-272]. Furthermore, a random flight should be added after injection to emulate that a particle can be injected at any time during a time step, not just at the end. With limited time, this work is left for future developers to finish.

5 | Testing

This chapter presents verification tests of MonteFFI's program behavior and analyzes simulation stability and efficiency. First, the computational resources used are described in section 5.1. Second, a simulation test model is presented in section 5.2 and simulation results are presented and discussed in section 5.3. Third, the simulation is further analyzed in terms of stability in section 5.4 and terms of execution time in section 5.5. Finally, a more advanced simulation model is presented in section 5.6 to showcase MonteFFI's support for complex device geometry.

5.1 Computational Resources

To carry out this project, most development was conducted on a standard laptop and heavier simulations were executed on a computing cluster. This is because the computing cluster has far superior computational power but is more cumbersome to work with compared to the laptop.

The computing cluster used throughout this work, called IDUN, is managed by the HPC group at NTNU in Trondheim.[1] IDUN currently has 68 work nodes which are all available to registered users. A batch system controls when the simulation starts, which means each simulation is submitted as a job placed in a queue where it has to wait until enough computational resources become available. Other hurdles include a lack of administrator privileges for the ordinary user, and the process of uploading and downloading data from the server can be slow depending on internet connection quality.

Access to IDUN made it possible to benchmark the program by running multiple simulations simultaneously, each with multiple cores in parallel to speed up the flight and scattering loop. Performance testing was restricted to nodes with the same specifications to produce consistent and reproducible results. IDUN has 27 such nodes with specifications listed in table 5.1. Furthermore, the cluster provided more than sufficient RAM and storage resources for this project. The only significant hardware restriction was CPU resources.

To help future developers use the same or a similar cluster, I have created a guide for running MonteFFI on IDUN which is included in appendix A.3.

[1] HPC group and IDUN homepage: `https://www.hpc.ntnu.no/`. Accessed 21.06.2020

Table 5.1: System specifications for computing nodes used during this work.

Item	Specification
CPU model	Intel Xeon E5-2630 v4
CPU cores	20
RAM	128 GB
ISA	x86-64
OS	CentOS Linux release 7.4 (Core)
Gfortran	version 4.8.5
OpenMP	version 3.1

5.2 APD Model

Central to validating the new program modules and the complete simulation system was to run one known test scenario to check that MonteFFI can produce results with sufficient accuracy. The test case was a bias simulation of a wide-area avalanche photodiode (APD) for infrared (IR) radiation detection, which is depicted in figure 5.1. The purpose of such simulations are made clear in [5, 61] by Brudevoll and Storebø, who have tested and verified the APD model against experiments with previous versions of FFI-MCS.

APD Operation The APD has the same fundamental purpose as any photodiode, which is to transform a light signal to a current pulse. Photons are captured in the absorption region through photo-excitation, causing electrons to enter the conduction band. By applying a reverse bias, an electric field is created which accelerate electrons in the N- region towards the contact at the top of the N+ region. The accelerated electron excites other electrons through impact ionization processes, causing an "electron avalanche" [62]. By the time the accelerated electrons reach the contact they have created a much larger cluster of electrons which amplifies the read-out signal.

Device Parameters The APD is made from a $\mathrm{Hg_{0.72}Cd_{0.28}Te}$-alloy with the doping densities listed in table 5.2 and device dimensions listed in table 5.3. Typical device dimensions for real APDs studied at FFI are typically larger than $10\,\mu\mathrm{m}$ in all directions. However, simulation of such a large device is not practical for testing because even low-accuracy simulations can require weeks to finish. In this work, a short device depth $L_z = 1\,\mu\mathrm{m}$ was applied to decrease the computational burden. The behavior of a shorter model is almost identical to a longer model due to the model's symmetry in the z-direction. This is also why researchers at FFI with 2D FD simulation have been able to successfully generate experimentally accurate results for similar APD models.

Not shown in the model, but present in the applied device discretization are $1\,\mu\mathrm{m}$ isolation layers along the left, right and bottom sides of the APD. These layers have no impact on the simulations

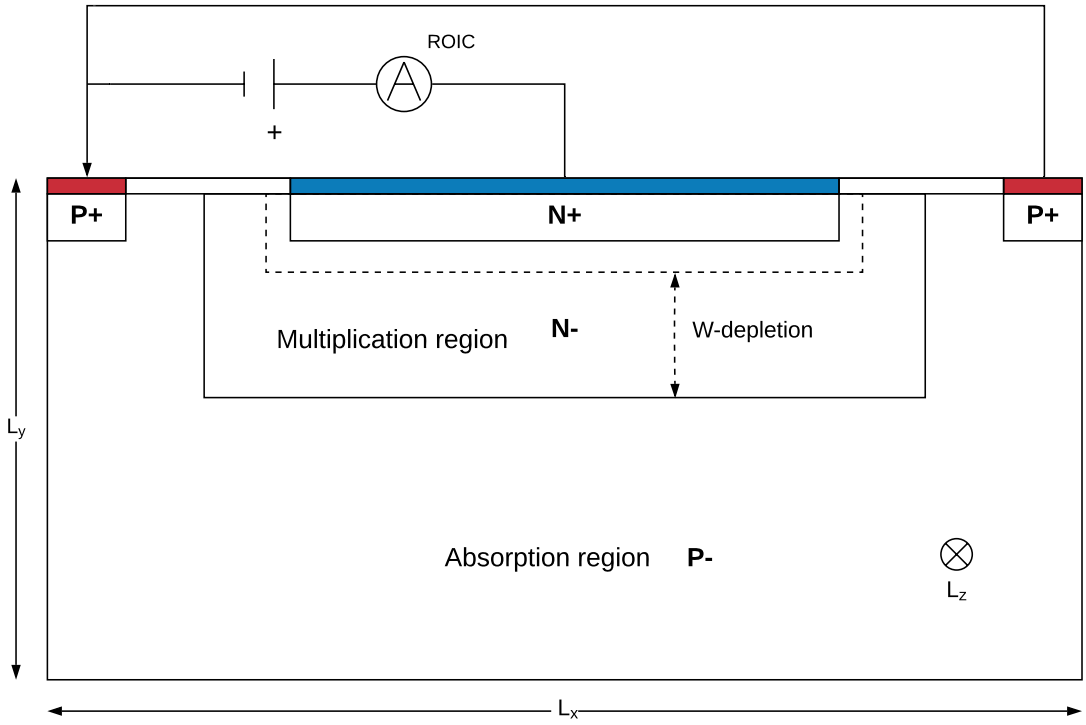


Figure 5.1: Planar layout of the CMT electron-initiated APD, with a voltage supply and a read-out integrated circuit (ROIC). The third dimension extends into the paper plane with a length L_z. The device is separated into positively doped (P) regions and negatively doped (N) regions, where minus (-) indicates low doping density and plus (+) indicates high doping density. The dotted line marks the inner border where we expect no electrons (depletion) in a steady-state system with applied reverse bias.

presented here, but were included in the discretization because they will be necessary for future simulation with alloy gradient fields used in FFI's research [61].

The simulated system was initialized in a neutral state and the applied contact voltages were kept constant throughout simulation. $-7\,\mathrm{V}$ were applied to the two outer P+ contacts and $0\,\mathrm{V}$ to the central N+ contact. The entire device has a constant temperature of $77\,\mathrm{K}$ throughout simulation. The scattering rates for a system with the presented material, doping and temperature parameters have been tabulated by Selvåg [42], and were used to include the following mechanisms in my simulations:

- Acoustic phonon scattering
- Polar optical phonon scattering
- Non-polar optical phonon scattering
- Ionized impurity scattering
- Alloy scattering

Other scattering mechanisms in CMT APDs are only relevant in special circumstances [63]. For example, impact ionization mechanisms can be ignored since we assume the amount of photogenerated charge is small.

Table 5.2: APD doping densities

Parameter	Value [cm^{-3}]
n_{N+}	$2.5 \cdot 10^{17}$
n_{N-}	$5.0 \cdot 10^{15}$
n_{P+}	$2.0 \cdot 10^{16}$
n_{P-}	$1.0 \cdot 10^{16}$

Table 5.3: APD device dimensions

Parameter	Value [µm]	Description
L_x	30	Device length in x-direction without isolation layer
L_y	10	Device length in y-direction without isolation layer
L_z	1	Device length in z-direction without isolation layer
L_x^{P+}	3	Length in x-direction for P+ regions
L_y^{P+}	1	Length in y-direction for P+ regions
L_x^{N-}	18	Length in x-direction for N- region
L_y^{N-}	5	Length in y-direction for N- region
L_x^{N+}	10	Length in x-direction for N+ region
L_y^{N+}	1	Length in y-direction for N+ region
d_y^{P+}	0.1	Contact depth (y-direction) for P+ region
d_y^{N+}	0.1	Contact depth (y-direction) for N+ region

5.3 Steady-State Simulation

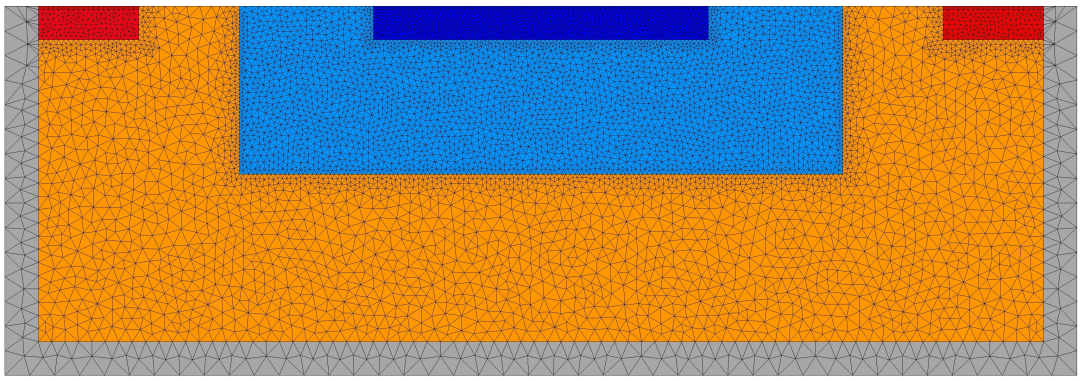
To verify program correctness, it is necessary to have all components interact in a full simulation. For this purpose, I have analyzed simulation results of the APD bias model, particularly to see if a reasonable steady-state can be achieved. Studying the behavior of the APD model in equilibrium is a good starting point because it well researched by previous students [6, 43, 8, 9] and researchers at FFI [61, 5].

Simulation Input

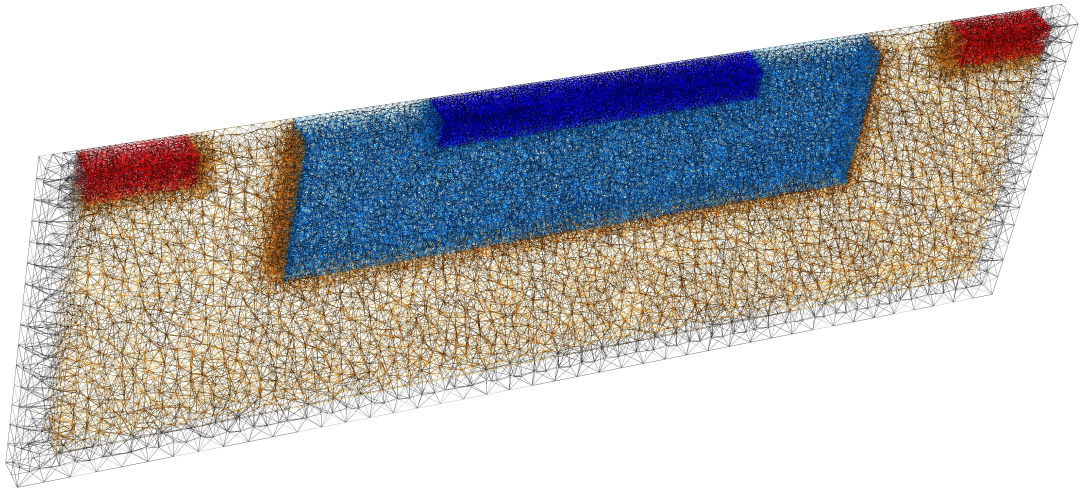
Table 5.4 lists the main parameters which control the simulation accuracy. These parameters were specified independent of the source code by using the .cfg file, presented in appendix B.2. The applied mesh presented in figure 5.2 was generated with the .geo script presented in appendix B.1.

Table 5.4: Resolution parameters for steady-state APD simulation.

Parameter	Value	Description
τ	2 fs	Time step between MC iterations
t_{tot}	160 ps	Total simulated time
N_e^{init}	200 000	Number of initial superelectrons
N_h^{init}	211 765	Number of initial superholes
$\frac{N_{\text{real}}}{N_{\text{sup}}}$	10.2	Real particles per superparticle
Δt_P	2 fs	Time step between field updates
N_n	31 884	Number of mesh nodes



(a)

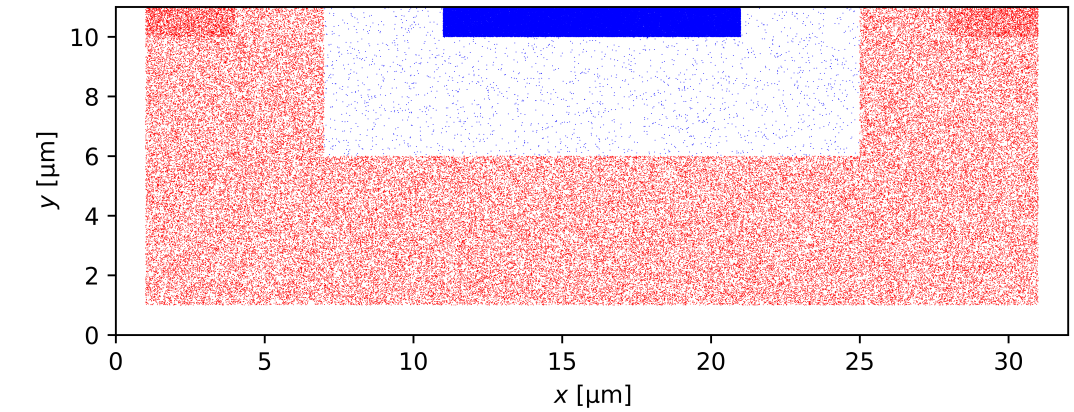


(b)

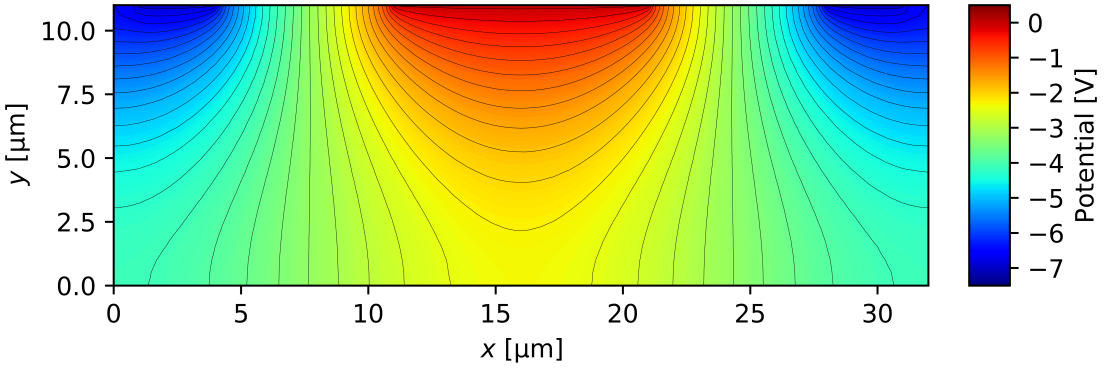
Figure 5.2: Mesh of APD model with $\Delta z = 1\,\mu\mathrm{m}$, shown from a 2D (a) and 3D (b) perspective. Note that the refinement is much higher close to region borders than in bulk semiconductor material.

Initial State

The component is initially charge neutral, where the moving charges and frozen doping atoms exactly match each other's position. Figure 5.3a shows the initial particle distribution, where we see that particles are uniformly distributed by a piecewise-constant carrier density with discontinuities along material region boundaries as expected. Thus, the initial Poisson solution is only influenced by surface charges represented as constant voltages along the Ohmic contacts. Figure 5.3b shows the initial potential where we see the boundary conditions are correctly applied: A potential difference of $7\,\mathrm{V}$; Dirichlet conditions forming constant potential along contact surfaces; and equipotential lines perpendicular to all Neumann boundaries (here: all non-Dirichlet boundaries).



(a)

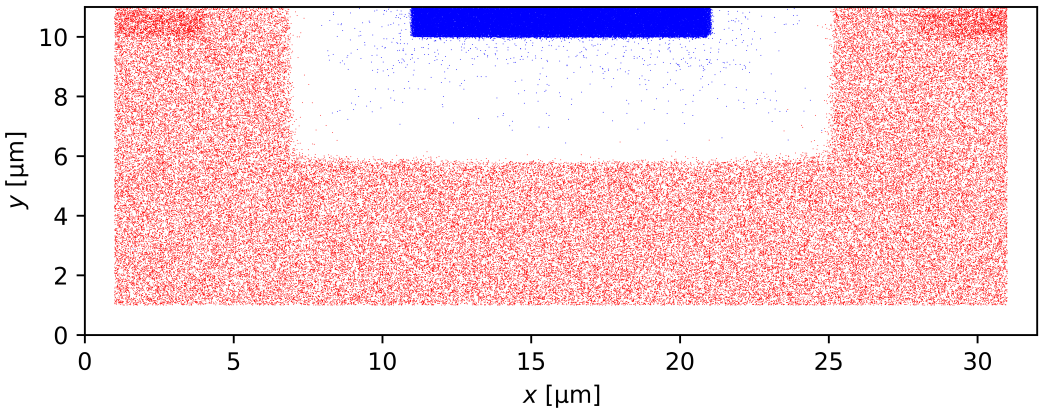


(b)

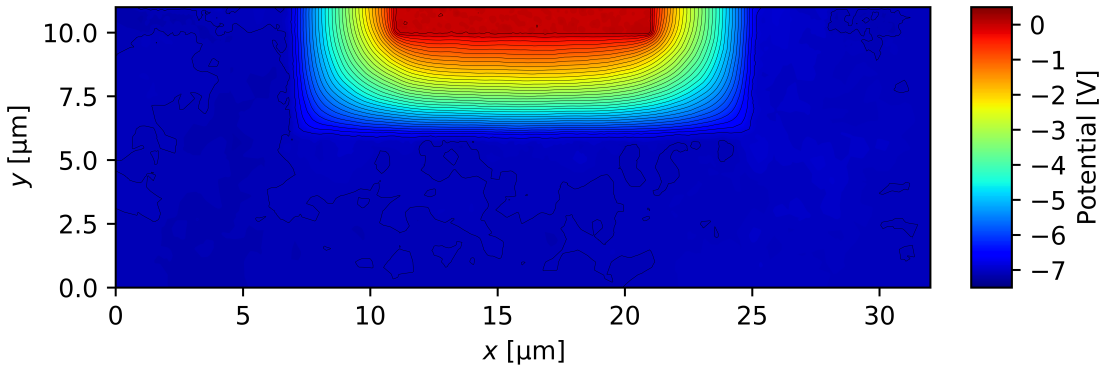
Figure 5.3: Initial state ($t = 0$) of the bias APD simulation. Figure (a) shows positions for $10\,\%$ of the simulated particles. Each blue dot represents a superelectron and each red dot represents a superhole. Figure (b) shows the electric potential through the xy-plane $z = 0.5\,\mu\mathrm{m}$.

Steady-State

Figures 5.4a and 5.4b respectively show particle positions and electric potential when the system has reached equilibrium. Most electrons have left the N- region creating a depletion region. This is due to large potential differences, creating an electron accelerating potential gradient pointing towards the N-contact characteristic for the APD model.



(a)

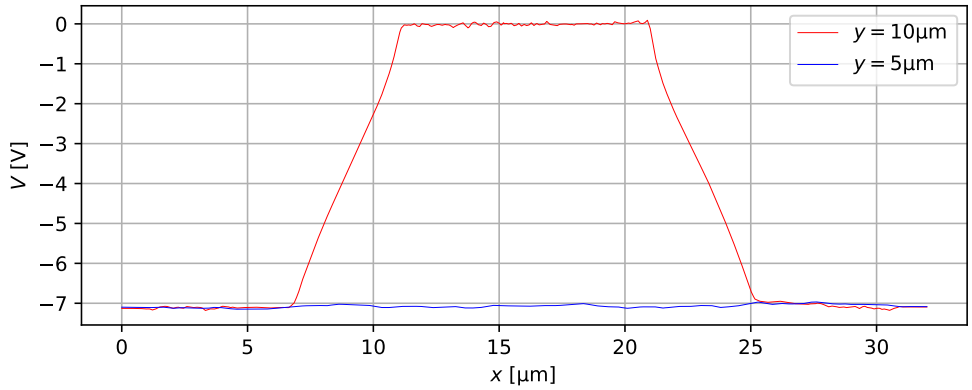


(b)

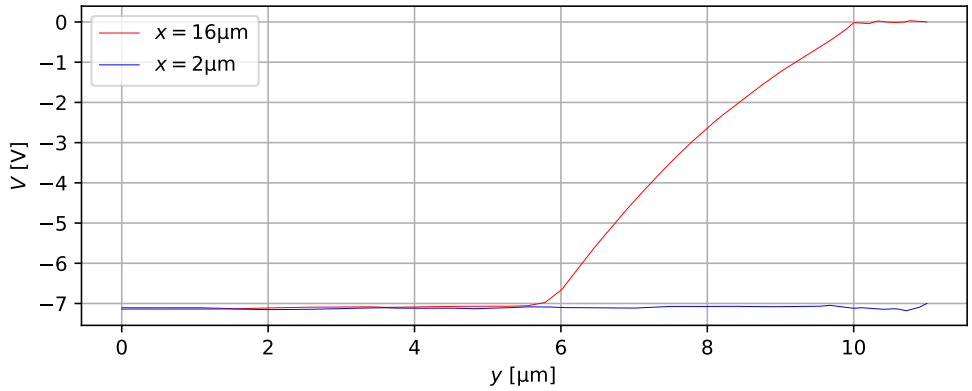
Figure 5.4: Same as figure 5.3 with $t = 160\,\mathrm{ps}$.

The numerical noise is clearly visible in the potential lines shown in figure 5.5. There are ripples causing deviations of up to $100\,\mathrm{mV}$ from the mean potential for every region except N-. This is to be expected without a finer mesh and a larger number of superparticles. Noise in the N- and N+ regions is smaller than what was achieved with the 2D FD version of FFI-MCS using about two times shorter grid spacing and many times more superparticles [47, p. 26]. On the other hand, there is more noise in the P regions in the presented results. This is because of an intentional trade-off, where mesh refinement is kept low in P-regions to increase efficiency in order to apply higher mesh refinement in N-regions. Another reason is that the overall system stability is more sensitive to the

motion of electrons than holes (further discussed in section 5.4).



(a)



(b)

Figure 5.5: Electric potential lines from the APD simulation at $t = 160\,\mathrm{ps}$. Figure (a) shows lines parallel to to the x-axis, and figure (b) shows lines parallel to the y-axis. All lines are in the plane $z = 0.5\,\mu\mathrm{m}$.

Oscillations in the Electron-Hole Plasma

To verify that the final bias state is in fact a steady-state we examine the simulation's transient behavior in terms of particle populations.

Figure 5.6 shows the particle populations at each time step, which clearly oscillate before stabilizing to a new steady state. Transient oscillations in the electron-hole plasma can be attributed to natural phenomena. The hole population oscillation has the same period as the electron oscillation, but consistently lags behind by a few picoseconds due to inertia. After about $80\,\mathrm{ps}$ the oscillations have subsided due to the damping caused by momentum randomizing scattering.

As expected, many electrons leave the depletion zone and eventually the device through the contacts, defining a new capacitance along the PN-junction as a result of the applied reverse bias.

The rapid expulsion of electrons initially causes a similar number of holes to leave through the P-contacts. Both carrier populations then start increasing again before oscillating around new mean values. However, the new mean values show a slight deficit of 1365 (11765-10400) positive charge carriers, since the number of holes in the settled biased state has decreased more than the number of evacuated electrons, as seen in figure 5.6b. The deficit appears to decrease slowly with time until the end of the simulation at 160 ps and further on, but it is doubtful whether a longer simulation would result in substantial changes.

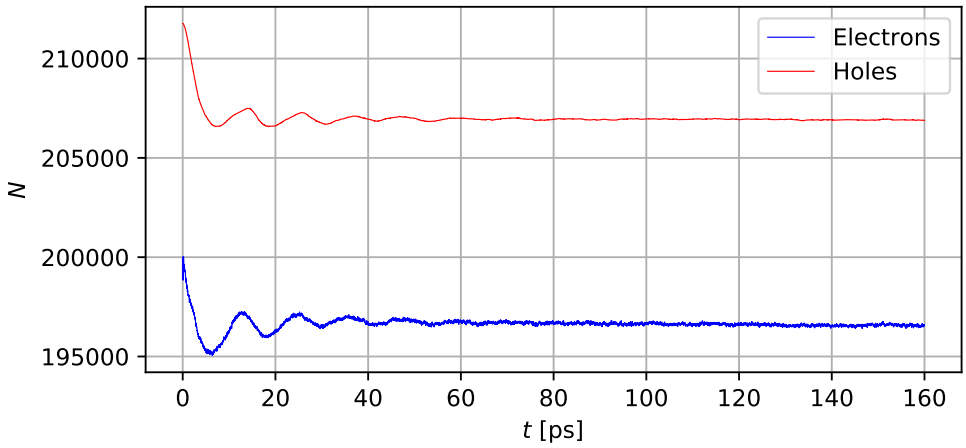
Note that the particle populations only include free internal charge carriers in the MC simulation, and not surface charges which the Poisson solver might include with the Dirichlet conditions applied along Ohmic contacts. The observations above are indicative of additional capacitive effects beside the PN-junction.

While further analysis is outside the scope of this thesis, a newly submitted paper [64] from my research group at FFI provides a more thorough analysis of a similar phenomenon of *signal current* oscillations. This was done simulating a larger APD component using the 2D FD version of FFI-MCS, which MonteFFI will supplement. A summary of our findings is presented in the following paragraphs.

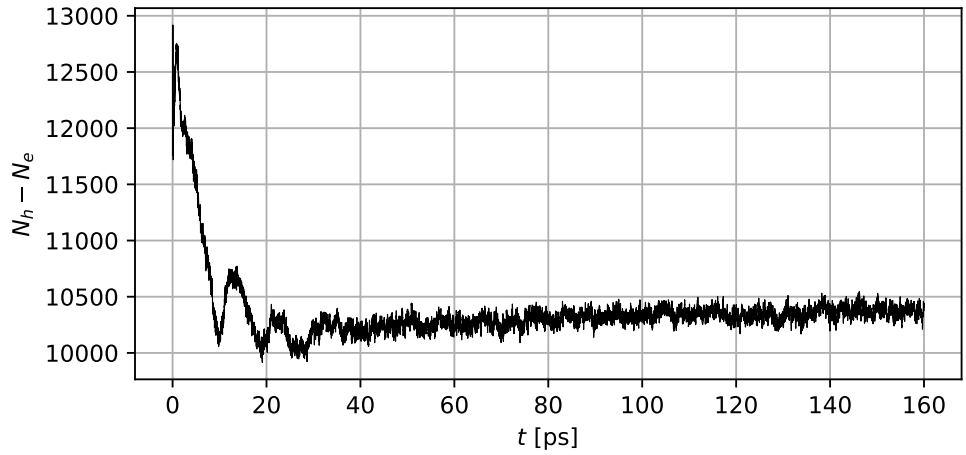
Non-Linear Signal Current Response Analysis The paper [64] investigates the effect on the small-signal current due to the junction capacitance and varying internal or external series resistances. It was shown that high-intensity irradiation delivered in the form of a $1\,\mathrm{ns}$ laser pulse could readily induce current and particle number oscillations for series resistances lower than $3\,\mathrm{k}\Omega$. This effect is due to a feedback mechanism where the avalanche gain creates charge clouds which screen away a small portion of the junction bias, causing part of the junction voltage to shift onto the series resistance, thereby temporarily reducing the avalanche gain and signal current, causing the voltage shift to return onto the junction again, increasing the avalanche gain and current again and so on.

A point made in the study is the realization that this effect could not be reproduced by "static" bias voltage/avalanche gain curves which show the gain as a function of a static junction voltage bias. In those curves, the junction voltage is due to a 'global' junction field which is imposed by an external voltage source. Rather, the modulation effect on the gain demonstrated in [64] is a local and dynamic effect: A severe perturbation of the local electric field between the generated charge clouds "pinches off" avalanche gain in that critical region and thus strongly reduces the total gain, but influences the 'global' junction voltage significantly less than the values of bias voltage we would need for creating the same gain variations statically.

Having a high series resistance stabilized the oscillations by suppressing transit-time effects, and the current response became more RC-like, dominated by the junction capacitance and the series resistance. Even though a high series resistance can protect the APD from excessive signal currents, the response of the APD becomes slower, and a tendency to afterpulsing was noted when Geiger-like series resistances of $32\,\mathrm{k}\Omega$ were simulated.



(a)



(b)

Figure 5.6: Particle populations throughout at each time step throughout the simulation. The number of superelectrons N_e and superholes N_h are shown in (a), and the difference $N_h - N_e$ is shown in (b).

5.4 Numerical Stability and Accuracy

This section analyses how well settings for the long APD simulation of section 5.3 correspond with the stability criteria previously introduced in section 2.4.3. Understanding different stability criteria is important to achieve stable simulation effectively optimize parameters determining simulation accuracy and efficiency. Stability criteria from literature are based on simple, typically one-dimensional, models. Complex device simulation can however deviate significantly from these models, and generally require careful evaluation against experiments. Still, simple stability criteria may serve guidelines for initial selection of simulation parameters, which can save significant development time.

Time Stability

The plasma frequency and relaxation rates for each region are listed in table 5.5. Values for the two time-stability criteria are presented in table 5.6, using only the parameters for the N+ region. This is because the same (shortest) time step applies to all regions, and the strictest time-stability criteria for the APD model can be shown to be for the N+ region.

Table 5.5: Plasma frequency ω_p and transfer rate ν_c for each of the APD model's differently doped regions. The low-field electron and hole mobilities needed to calculate ν_c were determined with the FFI-MCS bulk simulator [42].

Region	ω_p $[\mathrm{s}^{-1}]$	ν_c $[\mathrm{s}^{-1}]$
N+	$4.67 \cdot 10^{14}$	$1.0 \cdot 10^{12}$
N-	$2.33 \cdot 10^{13}$	$1.0 \cdot 10^{12}$
P+	$1.48 \cdot 10^{14}$	$1.8 \cdot 10^{14}$
P-	$1.04 \cdot 10^{14}$	$1.8 \cdot 10^{14}$

Table 5.6: Time-stability criterion values specific to the material and doping of the APD model's N+ region.

ω_p^{-1}	$2\nu_c/\omega_p^2$
21.4 fs	0.92 fs

A value $\Delta t_P < 5\,\mathrm{fs}$ was required to avoid destabilizing oscillations in the electric field. This is because of the criteria that Δt_P needs to be much smaller than ω_p^{-1}. On the other hand, the criterion $\Delta t_P < 2\nu_c/\omega_p^2$ does not necessarily have to be strictly satisfied because of stabilizing effects present in a complex device model not present in the idealized models the stability criteria were derived for. For example, the high electric fields increase scattering rates which reduces destabilizing oscillations

in the potential. This was seen in the APD simulations, where decreasing Δt_P below $2\,\mathrm{fs}$, about twice the value of $2\nu_c/\omega_p^2$, did not noticeably reduce noise in the electric potential. However, to achieve sufficiently high *accuracy*, smaller Δt_P could be necessary to model effects sensitive to fluctuations in the potential.

Spatial Stability

The spatial stability criteria are considered here for each differently doped region. This is because we can use fewer computational resources with a non-uniform grid by specifying the number of cells in each region independently.

The stability criteria are evaluated with values for both λ_D and ν_c/ω_p listed in table 5.7. With Palestri et al.'s relaxed stability criterion, presented in section 2.4.3, we know that generally requiring grid spacing Δx to be smaller than the Debye length λ_D is an unnecessarily strict criterion for MC simulation. This theory is corroborated by the APD simulation results presented herein, where stable solutions were obtained with grid spacing for P-doped regions $\Delta x \gg \lambda_d$. The reason Δx can be an order of magnitude larger than λ_D in P-regions is most likely because $\nu_c/\omega_p \gg 0.1$.

Table 5.7: Debye length, λ_D and spatial stability relaxation criterion, ν_c/ω_p, for differently doped regions of the APD model presented in section 5.2.

Region	λ_D [m]	ν_c/ω_p [1]
N+	$5.51 \cdot 10^{-9}$	$2.1 \cdot 10^{-2}$
N-	$1.10 \cdot 10^{-7}$	$4.3 \cdot 10^{-1}$
P+	$1.74 \cdot 10^{-8}$	$1.2 \cdot 10^{1}$
P-	$2.46 \cdot 10^{-8}$	$1.7 \cdot 10^{1}$

The N- region is an outlier requiring $\Delta x \lessapprox 0.2 \ll \lambda_D$. The potential is rapidly changing in the N- region, which demands high resolution for accurate simulation. Similarly, high second derivatives of the potential require extra refinement along junctions.

The N+ region has a very small value for ν_c/ω_p, which requires $\Delta x < \pi\lambda_D = 0.017\,\mu\mathrm{m}$ for stable simulation. For the tetrahedral grid we do not have a fixed grid spacing, but we do have a maximum grid spacing determined by the longest element edge length. The edge length is about $0.12\,\mu\mathrm{m}$ of elements in the N+ region, which is far greater than $\pi\lambda_D$. Still the system is stable, possibly because instabilities are damped by the larger device system and frequent momentum randomizing scattering because of the high electron energies in this system.

Velocities can be as high as $10^8\,\mathrm{cm\,s^{-1}}$ with highly mobile electrons in the N+ region. During a single time step $\Delta t_P = 1\,\mathrm{fs}$ a fast electron moves $\delta x = 0.1\,\mu\mathrm{m}$, which is about the same length as the grid spacing Δx of the current mesh. Therefore, a finer grid resolution in the N+ region could not increase simulation accuracy further without a proportional decrease in Δt_P, which would

greatly increase the computational costs of simulation. Therefore, 3D SCMC simulation require large computational resources with high doping densities, even for nanoscale device simulation [65].

Despite the large grid spacing in N+, the overall simulations are stable. Researchers at FFI have run 2D FD simulations of a similar APD model with $\Delta x = 0.05\,\mu\mathrm{m}$, resulting in similar overall results, and the read-out current they have measured corresponded well with experiments. Thus, the stability criteria are not absolute requirements for obtaining useful results, but still serve as the best guidelines we have for specifying initial resolutions in time and space.

5.5 Execution Time

To effectively reduce execution time, it is beneficial to identify code sections most limiting performance. Table 5.8 gives an overview of MonteFFI's time distribution when executing the previously presented bias APD simulation on IDUN. The total execution time was $132\,822\,\mathrm{s} \approx 37\,\mathrm{h}$, and the time per simulation step was $1.33\,\mathrm{s}$. Most resources were spent on the Poisson solver, electric field interpolation, and flight and scattering, while the CPU costs of the new contact module are comparatively small.

Table 5.8: Percentage of execution time of various code regions with the simulation settings presented in section 5.3.
(*) Run in parallel with 20 CPU cores $\implies t_{\mathrm{CPU}} \approx 20 t_{\mathrm{Execution}}$.

Program section	$t_{\mathrm{Section}}/t_{\mathrm{Total}}\ [\%]$
Poisson solution	36.7
E-field interpolation	12.2
Flight and scattering (*)	46.5
Contact dynamics	1.7
Save statistics	2.6
Other	0.3

Control Parameters The two main parameters the user can alter to affect execution time are the number of particles N_{sup} and the number of mesh nodes N_n. Graphs in figure 5.8 show the execution times for varying N_{sup} and N_n. The Poisson solver's execution time increases superlinearly with N_n, and slowly increases with N_{sup}. For flight and scattering and electric field interpolation it is the other way around, with execution time increasing near linearly with increasing N_{sup}, and remaining unchanged with increasing N_n.

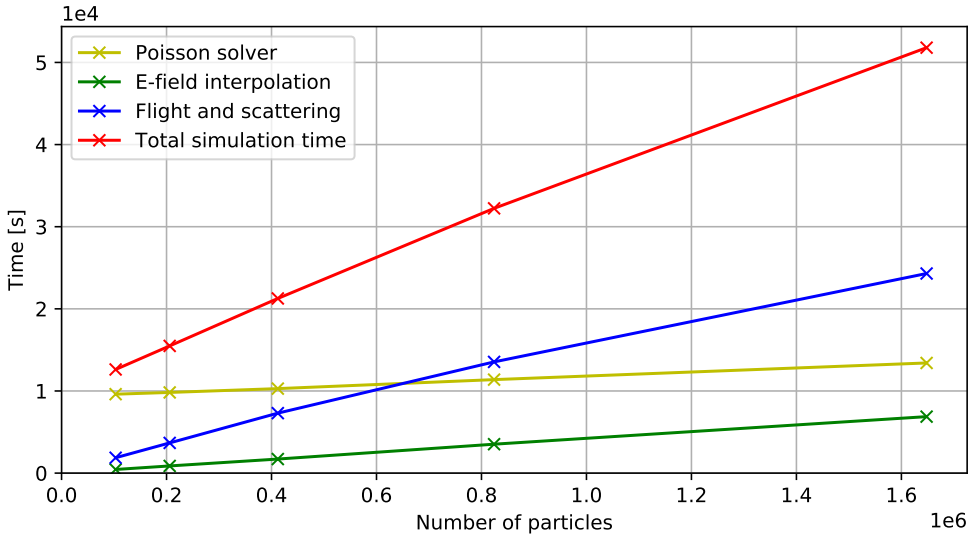


Figure 5.7: Execution times of simulations with a variable number of particles, a constant number of mesh nodes $N_n = 49983$ and 10^4 time iterations. The same initial ratio of holes and electrons, $N_h^{\text{Init}}/N_e^{\text{Init}}$, applied for all simulations.

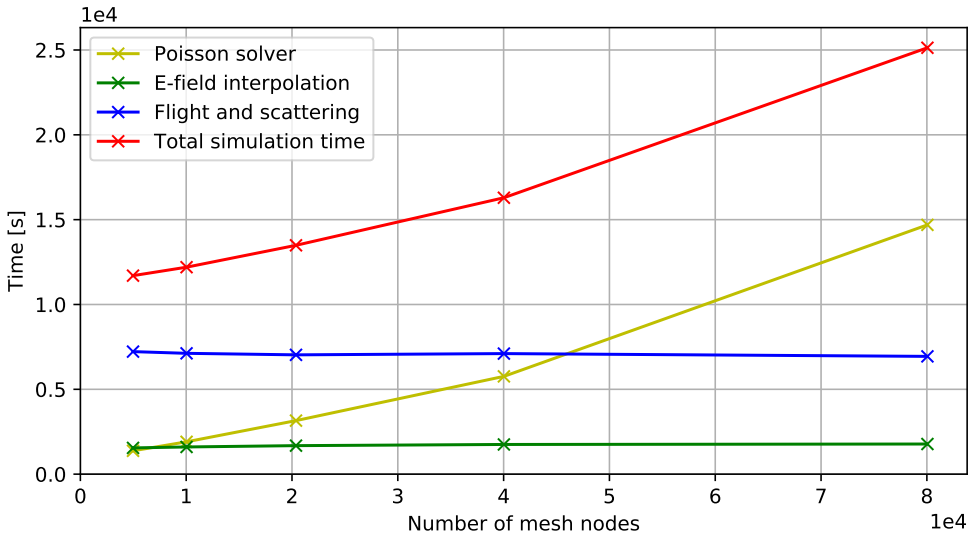


Figure 5.8: Execution times with a variable number of mesh nodes, a constant number of particles $N_{\text{sup}} = 417665$ and 10^4 time iterations.

If device complexity or doping density is increased, we generally expect larger increases in N_n than N_{sup} to maintain high accuracy. Thus, to reduce execution time, improving the Poisson solver's performance should be prioritized first. Major reductions in execution time will however require improvements to all three major program components, especially for the simulation of low doping

density devices where the E-field can be updated less frequently.

Poisson Solver Comparison The FE Poisson solver incorporated into MonteFFI seems to outperform Aldegunde et al.'s solver [65] run in serial (not parallel). Table 5.9 shows that our solver has a superior execution speed which cannot be explained by differences in simulation settings and hardware. It is worth reiterating that the full-band particle flight and scattering will be the main bottleneck in most cases if we increase the performance of the Poisson solver by an order of magnitude. Furthermore, the full-band particle flight and scattering already require multi-core hardware for efficient execution. With these considerations in mind, applying parallelization is currently the easiest and most beneficial approach for increasing execution speed.

Table 5.9: Comparison of execution time for the Poisson solution t_{Poisson} for two different 3D FE SCMC programs. Case A is the APD simulation presented in this chapter, and case B is the result from [65] using 1 CPU core.

Case	CPU clock frequency	Number of mesh nodes	$t_{\mathrm{Poisson}}/N_{\mathrm{Timesteps}}$
A	$2.2\,\mathrm{GHz}$	$31\,184$	$0.5\,\mathrm{s}$
B	$1.5\,\mathrm{GHz}$	$17\,629$	$24.9\,\mathrm{s}$

5.6 Complex Geometry

A smaller 3D APD model is presented here to demonstrate MonteFFI's ability to handle complex geometry. The same region types and doping levels of the previously considered APD model are used, but the geometry of this model is more complex demanding an unstructured device discretization.

The device geometry and mesh, seen in figure 5.9, was entirely created with Gmsh using basic geometrical and Boolean features as described in section 4.4. Drawing geometry takes some time and practice, but applying the model to MonteFFI was simple. Switching from the previous APD model to the new one only required altering device region specific settings before running a simulation. No changes to the source code were required.

Both contact regions and an isolation layer between N+ and P+ regions have curved surfaces, which MonteFFI can handle due to the support of an unstructured grid in the new Poisson solution and particle dynamics components of MonteFFI. The isolating "groove" is added to prevent premature breakdown between the electrodes/contacts at the top. The APD model was tested by running a bias simulation with the main simulation settings listed in table 5.10 and with a constant applied bias of $-1\,\mathrm{V}$ on the outer P-contact and $0\,\mathrm{V}$ on the inner N-contact.

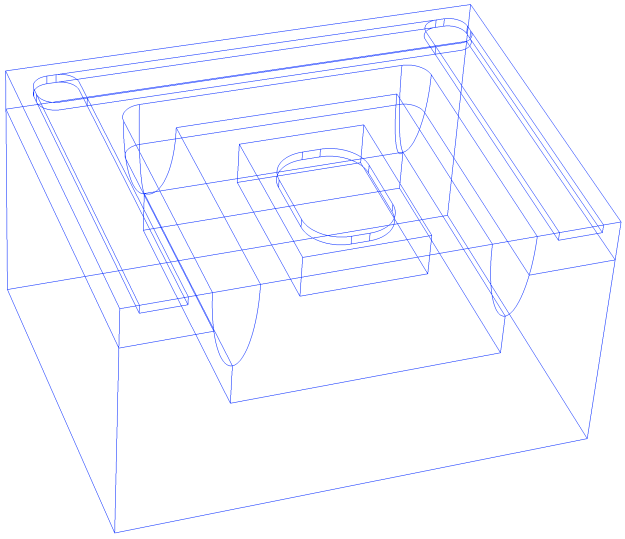
Due to lack of time the mesh for this model was not optimized. Still, the potentials presented in figure 5.10 show that MonteFFI has no apparent difficulty handling a model with complex geometry, except for significant numerical noise mainly caused by using a suboptimal mesh. Potential profiles

Table 5.10: Resolution parameters for steady-state APD simulation.

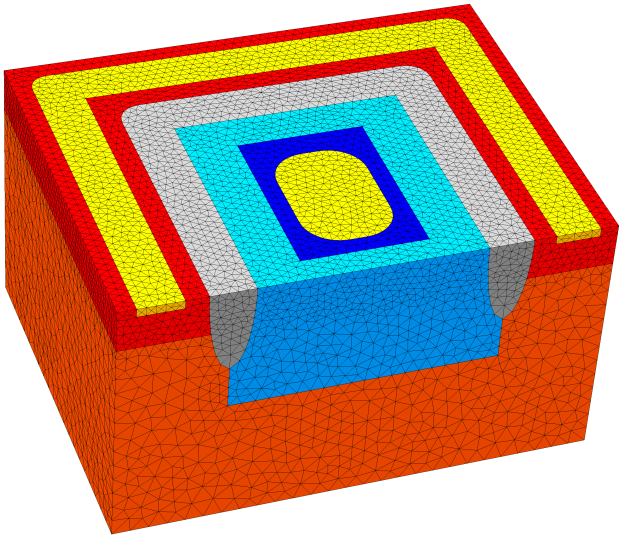
Parameter	Value	Description
τ	2 fs	Time step between MC iterations
t_{tot}	200 ps	Total simulated time
N_e^{init}	200 000	Number of initial superelectrons
N_h^{init}	409 885	Number of initial superholes
$\frac{N_{\text{real}}}{N_{\text{sup}}}$	1.4	Real particles per superparticle
Δt_P	2 fs	Time step between field updates
N_n	22 267	Number of mesh nodes

correspond well with the discussion of APD potential behavior in section 5.3, with a clear potential gradient in the N- region and close to constant potentials in other regions. These results also indicate that the Poisson solution and the newly implemented particle-mesh interactions work as expected for curved surfaces.

Program verification should be continued by analyzing stability for several complex 3D models with the same approach presented in previous sections. For full program validation we require thorough analysis against experimental values. This is the first priority for future development, briefly considered in the next chapter.

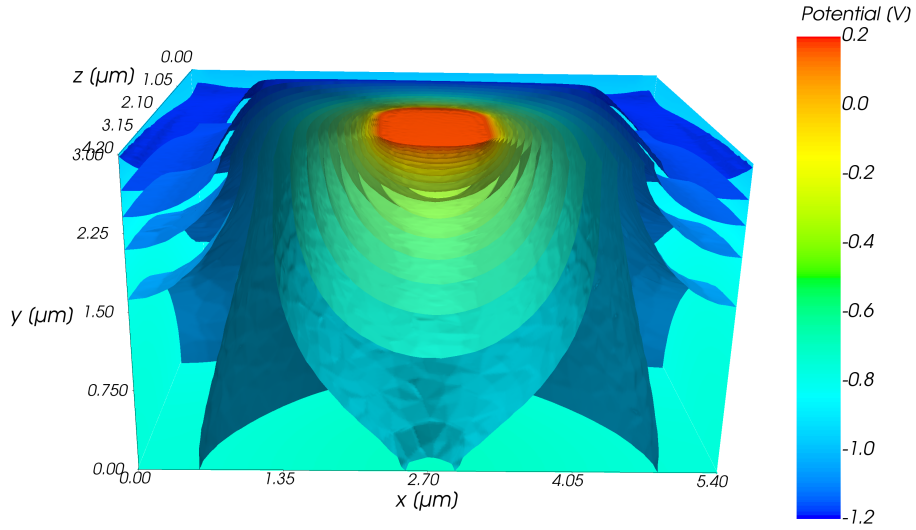


(a)

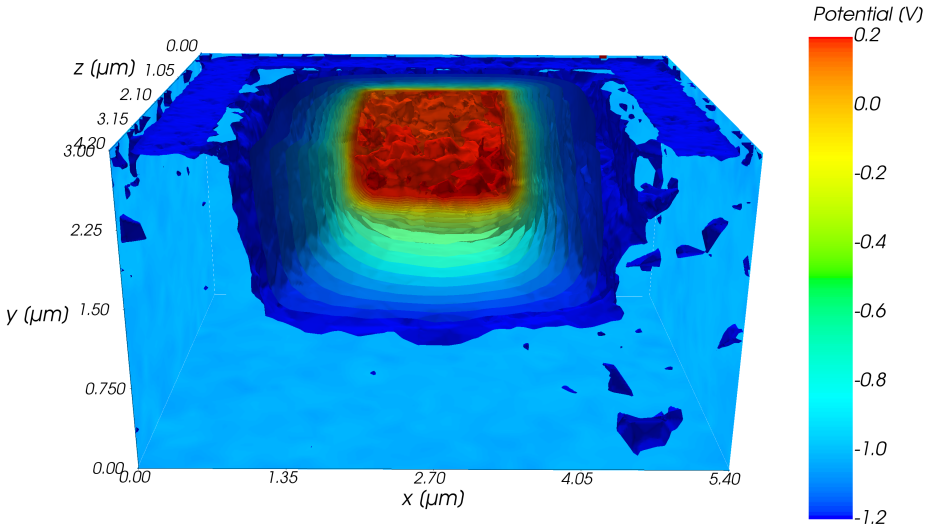


(b)

Figure 5.9: Geometry drawing (a) and mesh (b) for an APD model with an isolating groove colored grey. The P- region is colored orange, P+ red, N- light blue, N+ dark blue, and contact regions are yellow. Outer device dimensions are $5.4\,\mu\text{m}$ x $3.0\,\mu\text{m}$ x $4.2\,\mu\text{m}$.



(a)



(b)

Figure 5.10: Contour plots of the electric potential of the "complex" 3D APD model at simulation times (a) $t = 0$ and (b) $t = 100\,\text{ps}$.

6 | Further Work

MonteFFI is now capable of accurately simulating particle movement and Poisson solution for complex 3D geometries as well as flight and scattering with a full-band representation. To make the simulator more robust, the first steps should be to make a more realistic Ohmic contact representation and eliminate self-forces as covered in previous chapters. More importantly however, we need to fully develop missing components for realistic simulations to further analyze the correctness and applicability of MonteFFI compared to other simulators. Next, it is recommended to improve computational performance. Recommended paths to achieve these objectives are presented in the following paragraphs.

Field Solution with High Doping Density Very high doping densities make computationally costly because high mesh refinement and short timesteps are needed for stability. If doping levels are very high, simulations of even nanoscale devices can become too computationally expensive to simulate with a linear Poisson solution.

Palestri et al. has mostly opted for a non-linear Poisson solver for studying high density transistors [28]. Non-linear solvers can be stable for much larger time steps and grid spacing, and are in some cases unconditionally stable. However, they are not well suited for studying transient device behavior. If FFI is mainly looking to make frozen biases for non-self-consistent simulation, this type of solver should be considered.

Another possibility is to use higher order finite element solutions to provide higher resolution of the electric potential without decreasing grid spacing. Spatial stability criteria might also be relaxed for higher order approximations since the criteria considered herein have only been derived for linear approximations of the electric potential. However, no treatment of higher order elements applied to MC simulation was found in existing literature.

Parallel Poisson Solution Parallelization is a straightforward approach for increasing execution speed of the Poisson solver, which can provide massive speed-ups with access to the right hardware. A recommended parallelization approach is to use domain decomposition as covered in [66], demonstrating superlinear speedup for up to at least 32 CPU cores. For the Fortran implementation we can use standardized parallelization APIs for shared-memory systems using OpenMP [51] and/or

distributed-memory systems using OpenMPI [67]. With parallelization and access to a computing cluster, the Poisson problem can be solved many times faster compared to the current solver.

Efficient k-space Interpolation Full band interpolation of rates and energy is by far the most computationally expensive part of bulk simulation and a significant part of self-consistent simulation. The accuracy and efficiency of interpolation is highly dependent on the number of cells in the k-space discretization. Wessner [50] presents various structured and unstructured grid representations that greatly reduce the number of cells in k-space required to represent the energy band, which can be used to reduce the memory and CPU costs of full-band MC simulation. Moreover, the CMC method has the potential of greatly reducing CPU costs, but comes at a high memory typically proportional to the number of k-space cells [68, 69]. With a more memory efficient k-space discretization, the CMC method might improve the efficiency of flight and scattering by an order of magnitude or more.

CMT APD Physics Accurate simulation of CMT APDs developed at FFI will require further development of specialized physics components, particularly impact ionization mechanisms [62] and dark currents [63]. Compositional grading of alloy fractions is also an important factor for some APD models. To model the impact of alloy gradients on the electric potential, we can include a correction on top of the potential similar to previous versions of FFI-MCS [61]. Alternatively, a more direct and robust approach is to specify the correct alloy fraction in the simulation model. Vallone et al. [70] present a generally applicable approach, which divides the domain into many small device regions which each have a constant alloy fraction. This approach can be straightforwardly implemented with a script due to the source code independent device discretization used in MonteFFI.

Comparison to Other Simulation Programs With necessary physics modules included, we can compare MonteFFI to similar simulation programs. The 3D tensor-grid MC simulator by Vallone et al. [71, 72] has been successful at the simulation of a large range of devices, including IR CMT APDs similar to FFI's devices of interest. A comparison of this simulator with MonteFFI can reveal strengths and weaknesses with the 3D SCMC approach for APD simulation in particular and micrometer-scale device simulation in general.

7 | Conclusions

A self-consistent full-band Monte Carlo simulator using unstructured grids called MonteFFI was created during this work. Its applicability was demonstrated by successfully reaching a steady-state for a bias simulation of a CMT APD device with dimensions $30\,\mu\mathrm{m}$ x $10\,\mu\mathrm{m}$ x $1\,\mu\mathrm{m}$ in 37 hours. While noise in the potential was significant, this can be reduced by increasing the number of particles and number of mesh nodes. Hence, the MC and Poisson framework researchers at FFI need to simulate full-scale APDs in 3D is in place, but will require long execution times without further improvements.

I have addressed general usability issues of the 3D SCMC approach by presenting a selection of suitable development tools, a modularized program structure, and implementation of particle-mesh interactions for unstructured grids. The new development framework provides automated model generation, user-friendly configuration of simulation scenarios, and makes it easy to add and remove program features. Hence the user does not have to be an expert in numerics and Fortran programming to effectively make new device models and run simulations.

The simulation parameters corresponded well with criteria specific to MCS, showing that stable simulations can be achieved with much lower resolution than parts of the MC research community may be aware of. The potential range of application of the 3D FE SCMC method is vast, and well worth continued exploration. The superlinear relation between doping density and execution time can make 3D simulation too costly for large doping densities. However, we still have several possibilities to drastically improve execution time, such as parallelization and non-linear Poisson solution. Straightforward incorporation of various physical mechanisms has been central to the Monte Carlo method's success in providing new insights into semiconductor devices for decades. For the same reasons, the improved program framework of FFI-MCS with MonteFFI is likely to be a worthwhile investment for years to come.

Bibliography

[1] M. Aldegunde, A. J. García-Loureiro, and K. Kalna. "3D Finite Element Monte Carlo Simulations of Multigate Nanoscale Transistors". In: *IEEE Transactions on Electron Devices* 60.5 (2013), pp. 1561–1567. DOI: 10.1109/TED.2013.2253465.

Cited on pages 1, 41

[2] M. A. Elmessary, D. Nagy, M. Aldegunde, N. Seoane, G. Indalecio, J. Lindberg, et al. "Scaling/LER Study of Si GAA Nanowire FET using 3D Finite Element Monte Carlo Simulations". In: *Solid-State Electronics* 128 (2017), pp. 17–24. DOI: 10.1016/j.sse.2016.10.018.

Cited on page 1

[3] L. Donetti, C. Sampedro, F. G. Ruiz, A. Godoy, and F. Gámiz. "Three-Dimensional Multi-Subband Simulation of Scaled FinFETs". In: *2017 47th European Solid-State Device Research Conference (ESSDERC)*. 2017, pp. 180–183. DOI: 10.1109/ESSDERC.2017.8066621.

Cited on page 1

[4] L. Donetti, C. Sampedro, F.G. Ruiz, A. Godoy, and F. Gamiz. "A Thorough Study of Si Nanowire FETs with 3D Multi-Subband Ensemble Monte Carlo Simulations". In: *Solid-State Electronics* 159 (2019), pp. 19–25. DOI: 10.1016/j.sse.2019.03.044.

Cited on page 1

[5] A. K. Storebø, D. Goldar, and T. Brudevoll. "Simulation of Infrared Avalanche Photodiodes from First Principles". In: *Infrared Technology and Applications XLIII*. Ed. by B. F. Andresen, Gabor F. Fulop, Charles M. Hanson, John Lester Miller, and Paul R. Norton. Vol. 10177. International Society for Optics and Photonics. SPIE, 2017, pp. 276–285. DOI: 10.1117/12.2262473.

Cited on pages 2, 44, 47

[6] C. N. Kirkemo. "Monte Carlo Simulation of PN-Junctions". MA thesis. University of Oslo, 2011. URL: http://urn.nb.no/URN:NBN:no-29691.

Cited on pages 2, 21, 47

[7] J. J. Harang. "Implementation of Maxwell Equation Solver in Full-Band Monte Carlo Transport Simulators". Project Thesis. 2015.

Cited on pages 2, 21

[8] D. K. Åsen. "Self-Force Reduced Finite Element Poisson Solvers for Monte Carlo Particle Transport Simulators". MA thesis. Norwegian University of Science and Technology, 2016. URL: http://hdl.handle.net/11250/2418022.

Cited on pages 2, 21, 22, 47

[9] S. N. Fatnes. "A Three-Dimensional Finite Element Poisson Solver for Monte Carlo Particle Simulators". MA thesis. Norwegian University of Science and Technology, 2018. URL: http://hdl.handle.net/11250/2567223.

Cited on pages 2, 3, 12, 17, 18, 21, 22, 24, 26, 27, 32, 39, 41, 47

[10] E. Bellotti, H. Wen, S. Dominici, and A. L. Glasmann. "A Comparative Study of Carrier Lifetimes in ESWIR and MWIR Materials: HgCdTe, InGaAs, InAsSb, and GeSn (Conference Presentation)". In: *Infrared Technology and Applications XLIII*. Ed. by Bjørn F. Andresen, Gabor F. Fulop, Charles M. Hanson, John Lester Miller, and Paul R. Norton. Vol. 10177. International Society for Optics and Photonics. SPIE, 2017, pp. 312–312. DOI: 10.1117/12.2265894.

Cited on page 3

[11] H. Wen. "Advanced Numerical Modeling of Semiconductor Material Properties and their Device Performances". PhD thesis. Boston University, 2016. URL: https://open.bu.edu/handle/2144/17068.

Cited on page 3

[12] A. Rutle, H. Wang, R. Bye, and O. Osen. "Scalable And User-Friendly Simulation". In: *Proceedings - 29th European Conference on Modelling and Simulation, ECMS 2015*. 2015. DOI: 10.7148/2015-0164.

Cited on page 3

[13] S. S. Sawilowsky. "You Think You've Got the Trivials?" In: *Journal of Modern Applied Statistical Methods* 2.1 (2003), pp. 218–225. DOI: 10.22237/jmasm/1051748460.

Cited on page 5

[14] H. Kosina, M. Nedjalkov, and S. Selberherr. "Theory of the Monte Carlo Method for Semiconductor Device Simulation". In: *IEEE Transactions on Electron Devices* 47.10 (2000), pp. 1898–1908. DOI: 10.1109/16.870569.

Cited on page 5

[15] P. Lugli and C. Jacoboni. "Monte Carlo Simulation of Semiconductor Devices: A Critical Review". In: *ESSDERC '87: 17th European Solid State Device Research Conference*. 1987, pp. 97–101. URL: https://ieeexplore.ieee.org/servlet/opac?punumber=5436380.

Cited on page 5

[16] D. Vasileska, D. Mamaluy, H. R. Khan, K. Raleva, and S. Goodnick. "Semiconductor Device Modeling". In: *Journal of Computational and Theoretical Nanoscience* 5.6 (2008), pp. 999–1030. DOI: 10.1166/jctn.2008.2538.

Cited on pages 5, 8, 28

[17] D. Vasileska, S. M. Goodnick, and G. Klimeck. *Computational Electronics; Semiclassical and Quantum Device Modeling and Simulation*. 1st. Taylor & Francis Group, 2010. DOI: 10.1201/b13776.

Cited on pages 5–7, 14

[18] C. Jungemann and B. Meinerzhagen. *Hierarchical Device Simulation: The Monte-Carlo Perspective*. Springer, Vienna, 2003. DOI: 10.1007/978-3-7091-6086-2.

Cited on pages 5, 6, 9, 10, 14

[19] C. Moglestue. *Monte Carlo Simulation of Semiconductor Devices*. Chapman & Hall, London, 1993. DOI: 10.1007/978-94-015-8133-2.

Cited on pages 5, 7

[20] C. Jacoboni and L. Reggiani. *The Monte Carlo Method for Semiconductor Device Simulation*. Springer Vienna, 1989. DOI: 10.1007/978-3-7091-6963-6.

Cited on pages 5, 12

[21] R. M. Yorston. "Free-Flight Time Generation in the Monte Carlo Simulation of Carrier Transport in Semiconductors". In: *Journal of Computational Physics* 64.1 (1986), pp. 177–194. DOI: 10.1016/0021-9991(86)90024-0.

Cited on pages 7, 28, 30

[22] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles*. USA: Taylor & Francis, Inc., 1988.

Cited on pages 9, 13, 39

[23] J. Jin. *The Finite Element Method in Electromagnetics*. John Wiley & Sons, New Jersey, 2014.

Cited on page 12

[24] A. Quarteroni. *Numerical Models for Differential Problems*. Springer, Milano, 2014. DOI: 10.1007/978-88-470-5522-3.

Cited on page 12

[25] S. D. Mobbs. "Semiconductor Device Modelling". In: ed. by C. M. Snowden. Springer, London, 1989. Chap. Numerical Techniques - The Finite Element Method. DOI: 10.1007/978-1-4471-1033-0_4.

Cited on page 12

[26] M. Lundstrom. *Fundamentals of Carrier Transport*. 2nd ed. Cambridge University Press, 2000. DOI: 10.1017/CBO9780511618611.

Cited on pages 13, 14, 42

[27] P. W. Rambo and J. Denavit. "Time Stability of Monte Carlo Device Simulation". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 12.11 (1993), pp. 1734–1741. DOI: 10.1109/43.248084.

Cited on page 13

[28] P. Palestri, N. Barin, D. Esseni, and C. Fiegna. "Stability of Self-Consistent Monte Carlo Simulations: Effects of the Grid Size and of the Coupling Scheme". In: *IEEE Transactions on Electron Devices* 53.6 (2006), pp. 1433–1442. DOI: 10.1109/TED.2006.874758.

Cited on pages 14, 62

[29] M. Aldegunde and K. Kalna. "Energy Conserving, Self-Force Free Monte Carlo Simulations of Semiconductor Devices on Unstructured Meshes". In: *Computer Physics Communications* 189 (2015), pp. 31–36. DOI: 10.1016/j.cpc.2014.11.020.

Cited on page 15

[30] G. U. Jensen, B. Lund, T. A. Fjeldly, and M. Shur. "Monte Carlo Simulation of Semiconductor Devices". In: *Computer Physics Communications* 67.1 (1991), pp. 1–61. DOI: 10.1016/0010-4655(91)90220-F.

Cited on page 16

[31] X. Gonze, B. Amadon, G. Antonius, F. Arnardi, L. Baguet, J. Beuken, et al. "The ABINIT Project: Impact, Environment and Recent Developments". In: *Computer Physics Communications* 248 (2020), p. 107042. DOI: 10.1016/j.cpc.2019.107042.

Cited on page 16

[32] P. Blaha, K. Schwarz, F. Tran, R. Laskowski, G. K. H. Madsen, and L. D. Marks. "WIEN2k: An APW+lo Program for Calculating the Properties of Solids". In: *The Journal of Chemical Physics* 152.7 (2020), p. 074101. DOI: 10.1063/1.5143061.

Cited on page 16

[33] E. Halvorsen. "Numerical Calculation of Valence Band Structure and Hole Scattering Rates in GaAs". MA thesis. Norwegian University of Science and Technology, 1991.
Cited on pages 16, 21

[34] C. Geuzaine and J. Remacle. "Gmsh: A 3-D Finite Element Mesh Generator with Built-In Pre- and Post-Processing Facilities". In: *International Journal for Numerical Methods in Engineering* 79.11 (2009), pp. 1309–1331. DOI: 10.1002/nme.2579.
Cited on pages 16, 22

[35] B. Karlsen. "Carrier Scattering Rates in Zincblende Structure Semiconductors Derived From 14x14 kp and Ab Initio Pseudopotential Methods". MA thesis. Norwegian University of Science and Technology, 2013. URL: http://hdl.handle.net/11250/247104.
Cited on pages 16, 21

[36] Ø. Olsen. "Construction of a Transport Kernel for an Ensemble Monte Carlo Simulator". MA thesis. Norwegian University of Science and Technology, 2009. URL: http://hdl.handle.net/11250/246279.
Cited on page 21

[37] O. C. Norum. "Monte Carlo Simulation of Semiconductors - Program Structure and Physical Phenomena". MA thesis. Norwegian University of Science and Technology, 2009. URL: http://hdl.handle.net/11250/246281.
Cited on page 21

[38] Ø. Skåring. "Ultrashort Relaxation Dynamics in Laser Excited Semiconductors". MA thesis. Norwegian University of Science and Technology, 2010.
Cited on page 21

[39] A. J. V. Vestby. "Calculation of Terminal Currents in Single Photon Excited Avalanche Photodiodes". MA thesis. Norwegian University of Science and Technology, 2012. URL: http://hdl.handle.net/11250/246818.
Cited on page 21

[40] K. V. Falck. "Ensemble Averaged and Single Particle Auger Lifetimes in Zincblende Structure Semiconductors". MA thesis. Norwegian University of Science and Technology, 2013. URL: http://hdl.handle.net/11250/247121.
Cited on page 21

[41] T. S. Bergslid. "Implementing a Full-Band Monte Carlo Model for Zincblende Structure Semiconductors". MA thesis. Norwegian University of Science and Technology, 2013. URL: http://hdl.handle.net/11250/247128.
Cited on page 21

[42] J. Selvåg. "High Precision, Full Potential Electronic Transport Simulator: Implementation and First Results". MA thesis. Norwegian University of Science and Technology, 2014. URL: http://hdl.handle.net/11250/247368.

Cited on pages 21, 28, 45, 54

[43] T. Chirac. "Monte Carlo Simulation of Photoconductive Terahertz Sources in Mercury Cadmium Telluride". MA thesis. Norwegian University of Science and Technology, 2016. URL: http://hdl.handle.net/11250/2394152.

Cited on pages 21, 47

[44] D. Goldar. "Calculation of Wavefunction Overlaps in First Principles Electronic Structure Codes". MA thesis. Norwegian University of Science and Technology, 2017. URL: http://hdl.handle.net/11250/2461490.

Cited on page 21

[45] M. Haug. "Schrödinger-Poisson and Nonequilibrium Green's Function Methods Applied to Layered Semiconductor Devices". MA thesis. Norwegian University of Science and Technology, 2018. URL: http://hdl.handle.net/11250/2562316.

Cited on page 21

[46] M. Estensen. "Simulation of a Mercury Cadmium Telluride Avalanche Photodiode". MA thesis. Norwegian University of Science and Technology, 2019. URL: http://hdl.handle.net/11250/2611911.

Cited on pages 21, 24

[47] A. Bolstad. "Optimization of a Particle-Based Semiconductor Device Simulator: Self-Consistent Ensemble Monte Carlo Method". Project thesis. 2019.

Cited on pages 21, 25, 28, 50

[48] M. Aldegunde, J. J. Pombo, and A.J. García-Loureiro. "Modified Octree Mesh Generation for Manhattan Type Structures With Narrow Layers Applied to Semiconductor Devices". In: *International Journal of Numerical Modelling: Electronic Networks, Devices and Fields* 19.6 (2006), pp. 473–489. DOI: 10.1002/jnm.623.

Cited on page 22

[49] N. Seoane, D. Nagy, G. Indalecio, G. Espiñeira, K. Kalna, and A. García-Loureiro. "A Multi-Method Simulation Toolbox to Study Performance and Variability of Nanowire FETs". In: *Materials* 12.15.2391 (2019). DOI: 10.3390/ma12152391.

Cited on page 22

[50] W. Wessner. “Mesh Refinement Techniques for TCAD Tools”. PhD thesis. Vienna University of Technology, 2006. DOI: https://www.iue.tuwien.ac.at/phd/wessner/.
Cited on pages 22, 63

[51] L. Dagum and R. Menon. “"OpenMP: An Industry Standard API for Shared-Memory Programming"”. In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55.
Cited on pages 23, 62

[52] L. F. Williams. “A Modification to the Half-Interval Search (Binary Search) Method”. In: *Proceedings of the 14th Annual Southeast Regional Conference*. ACM-SE 14. Birmingham, Alabama: Association for Computing Machinery, 1976, pp. 95–101. DOI: 10.1145/503561.503582.
Cited on page 37

[53] W. Hörmann, J. Leydold, and G. Derflinger. *Automatic Nonuniform Random Variate Generation*. Springer-Verlag Berlin Heidelberg, 2004. DOI: 10.1007/978-3-662-05946-3.
Cited on page 37

[54] R. Y. Rubinstein and D. P. Kroese. *Simulation and the Monte Carlo method*. 3rd ed. John Wiley & Sons, 2016.
Cited on page 37

[55] C. Grimme. *Picking a Uniformly Random Point from an Arbitrary Simplex*. Tech. rep. Technical report, Information Systems and Statistics, Munster U., Germany, 2015. DOI: 10.13140/RG.2.1.3807.6968.
Cited on page 37

[56] T. González and D. Pardo. “Physical Models of Ohmic Contact for Monte Carlo Device Simulation”. In: *Solid-State Electronics* 39.4 (1996), pp. 555–562. DOI: 10.1016/0038-1101(95)00188-3.
Cited on pages 39, 40, 42

[57] *OpenMC - Handling Surface Crossings*. Accessed: 2020-05-25. Last updated: 2019-10-03. URL: https://docs.openmc.org/en/stable/methods/geometry.html#handling-surface-crossings.
Cited on page 39

[58] P. K. Romano, N. E. Horelik, B. R. Herman, A. G. Nelson, B. Forget, and K. Smith. “OpenMC: A State-of-the-Art Monte Carlo Code for Research and Development”. In: *Annals of Nuclear Energy* 82 (2015), pp. 90–97. DOI: 10.1016/j.anucene.2014.07.048.
Cited on page 39

[59] R. Chordá, J.A. Blasco, and N. Fueyo. "An Efficient Particle-Locating Algorithm for Application in Arbitrary 2D and 3D Grids". In: *International Journal of Multiphase Flow* 28.9 (2002), pp. 1565–1580. DOI: 10.1016/S0301-9322(02)00045-9.

Cited on page 39

[60] W. H. Press. *Numerical Recipes, the Art of Scientific Computing*. 3. ed. New York: Cambridge University Press, 2007.

Cited on page 40

[61] A. K. Storebø and T. Brudevoll. "Modeling of a Back-Illuminated HgCdTe MWIR Avalanche Photodiode with Alloy Gradients". In: *Journal of Physics: Conference Series* 647 (2015), p. 012051. DOI: 10.1088/1742-6596/647/1/012051.

Cited on pages 44, 45, 47, 63

[62] S. Derelle, S. Bernhardt, R. Hardar, J. Primot, J. Deschamps, and J. Rothman. "A Monte Carlo Study of $Hg_{0.7}Cd_{0.3}Te$ e-APD". In: *IEEE Transactions on Electron Devices* 56.4 (2009), pp. 569–577.

Cited on pages 44, 63

[63] A. Singh, V. Srivastav, and R. Pal. "HgCdTe Avalanche Photodiodes: A Review". In: *Optics & Laser Technology* 43.7 (2011), pp. 1358–1370. DOI: 10.1016/j.optlastec.2011.03.009.

Cited on pages 46, 63

[64] A. K. Storebø, A. Bolstad, T. Brudevoll, E. Selvig, R. W. Hansen, T. Lorentzen, and R. Haakenaasen. "Simulated Transit-Time Limited Response and External Resistance Effects in a MCT Avalanche Photodiode under High-Intensity Nanosecond Irradiation". Submitted to *Semiconductor Science and Technology*. 2020.

Cited on page 52

[65] M. Aldegunde, A. J. García-Loureiro, A. Martinez, and K. Kalna. "Tetrahedral Elements in Self-Consistent Parallel 3D Monte Carlo Simulations of MOSFETs". In: *Journal of Computational Electronics* 7.3 (2008), pp. 201–204. DOI: 10.1007/s10825-008-0241-3.

Cited on pages 56, 58

[66] M. Aldegunde, A. J. García-Loureiro, and K. Kalna. "Development of a 3D Parallel Finite Element Monte Carlo Simulator for Nano-MOSFETs". In: *Large-Scale Scientific Computing*. Ed. by Ivan Lirkov, Svetozar Margenov, and Jerzy Waśniewski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 115–122. DOI: 10.1007/978-3-540-78827-0_11.

Cited on page 62

[67] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, et al. "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation". In: *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, 2004, pp. 97–104.

Cited on page 63

[68] M. Saraniti and S. M. Goodnick. "Hybrid Fullband Cellular Automaton/Monte Carlo Approach for Fast Simulation of Charge Transport in Semiconductors". In: *IEEE Transactions on Electron Devices* 47.10 (2000), pp. 1909–1916. DOI: 10.1109/16.870571.

Cited on page 63

[69] B. Popescu, D. Popescu, M. Saraniti, and P. Lugli. "Full-Band 3-D Monte Carlo Simulation of InAs Nanowires and High Frequency Analysis". In: *IEEE Transactions on Electron Devices* 62.6 (2015), pp. 1848–1854.

Cited on page 63

[70] M. Vallone, M. Goano, F. Bertazzi, G. Ghione, S. Hanna, D. Eich, and H. Figgemeier. "FDTD Simulation of Compositionally Graded HgCdTe Photodetectors". In: *Infrared Physics & Technology* 97 (2019), pp. 203–209. DOI: 10.1016/j.infrared.2018.12.041.

Cited on page 63

[71] M. Vallone, M. Goano, F. Bertazzi, G. Ghione, R. Wollrab, and J. Ziegler. "Modeling Photocurrent Spectra of Single-Color and Dual-Band HgCdTe Photodetectors: Is 3D Simulation Unavoidable?" In: *Journal of Electronic Materials* 43.8 (2014), pp. 3070–3076. DOI: 10.1007/s11664-014-3252-9.

Cited on page 63

[72] M. Vallone, M. Mandurrino, M. Goano, F. Bertazzi, G. Ghione, W. Schirmacher, et al. "Numerical Modeling of SRH and Tunneling Mechanisms in High-Operating-Temperature MWIR HgCdTe Photodetectors". In: *Journal of Electronic Materials* 44.9 (2015), pp. 3056–3063. DOI: 10.1007/s11664-015-3767-8.

Cited on page 63

Appendix

A Program Guides

A.1 Installation Guide

This guide should cover all steps necessary to get MonteFFI running on a new computer.
Note: MonteFFI currently only supports Linux operating systems.

A.1.1 Prerequisites

You need the following programs installed on your system:

- Git (and a GitHub account to contribute)
- GNU Make
- MKL (installation covered later in this guide)
- GMSH (only required for 3D FEM program)
- Python 3 (required for fort_depend and run plotting scripts)
- fort_depend (requires pip which comes with python3)

Git, Make, GMSH and Python can be directly installed using the apt (or apt-get) command on Ubuntu and Debian systems without any further configuration:

```
sudo apt install git make gmsh python3
```

fort_depend can be installed using python's package manager pip:

```
python3 -m pip install --user fortdepend
```

Fortdepend on GitHub: [https://github.com/ZedThree/fort_depend.py]
As of late, installing MKL is much easier than before for Ubuntu and Debian systems, as they are available through the apt package manager. To install a version of MKL which works with MonteFFI, follow the guide at the intel website: [https://software.intel.com/en-us/articles/installing-intel-free-libs-and-python-apt-repo]. It is recommended to install MKL to a folder such as /opt/intel/mkl. This

folder, called MKLROOT, needs to be specified in the compile script (Makefile) described in the next section.
NOTE: There is a default intel-mkl in apt, but it is not compatible with intel's MKL link line advisor since it is not installed with MKL's default library structure (which the FFIMCS linking procedure is based on). I.e. don't use it unless you are prepared to change the Makefile's MKL linking script.

A.1.2 Compilation

Note: This section assumes the user has basic knowledge of how to compile Fortran programs in Bash. It is also advised to learn the basics of how a GNU Make works.
The program is compiled using GNU Make, which makes it easy to compile a large project because it automatically sorts out dependencies between source files. First off we show how to invoke make, and cover the details later.
In order to invoke make you need to be in the main folder where there is a file called `Makefile`, which is where all compilation settings and build rules for the project are written. To start of you can simply write `make` or `make help` which both output information about the main targets to use.

Main targets

1. Creating the dependencies file. `bash make depend` This command creates the Makefile.dep file, which contains the information about which file each file in the project depends on.

2. Compiling the source code. When compiling the program we need to (preferably statically) link MKL to it. This is done automatically as long as you have specified the correct MKLROOT path inside the `Makefile`. Now you should be able to write

   ```
   make all
   ```

   which installs all programs in the FFIMCS repository.

3. Cleaning up. To delete all build files (created by `make all`), write

   ```
   make clean
   ```

- If you only want to compile one of the programs, you can specify them directly, e.g. `bash make MCFEM` to only compile the MCFEM program.

Options

There are several options for which flags to use during compilation. These are invoked after specifying the target, e.g. `make <target> <OPTION>=<value>`. The main options are (with default value in parantheses):

- FC(=gfortran) : Specifies compiler. Values: ifort, gfortran.
- OPTIM(=1) : Use aggressive compiler optimizations. Values: 0, 1
- DEBUG(=0) : Add debugging symbols and check for out-of-bounds and floating point arithmetic errors. Values: 0, 1
- OPENMP(=1) : Use OpenMP parallelization. Values: 0, 1
- PROFILE(=0) : Use Gprof to make a profile of the program's execution/CPU time/cycles. Values: 0, 1

Notes

The Makefile is well documented, and should remain that way. It should be clear for future developers what each line does, as the Makefile code can be quite unintelligible.
If you add new source code files, or add new dependencies between files, you need to call `make depend`.
If you add a new program (PROGRAM, END PROGRAM), this needs to be added to the list `PROGS` inside the Makefile.
As long as you put new source code in the src folder, everything else is taken care of automatically. If you add external libraries etc., you will need to change the Makefile yourself.
The command `make depend` invokes fortdepend. If this isn't working, you can always create the Makefile.dep by hand by inspecting the source code, but this can be quite tedious. An alternative more dated package is makedepf90. Currently, fortdepend is used because it is much easier to use and more stable than other packages found (2020-04-18).

A.2 User Guide

This is a general user guide for MonteFFI, and a work in progress. It provides information targeted at helping FFI researchers effectively operate the program.
To install the program, see the installation guide (prerequisite to this guide).
Other documents extending the information provided here include:

- Server user guide for cluster computing on IDUN.
- Developer guide, style guide and a to-do-list for more information on how you can contribute to this project.

The latest student theses written about the program should also be read in order to better understand the program's capabilities and limitations.

A.2.1 Quick start: Run the default test case

After successful compilation (`make all`) you should be able to run a default program by entering the following commands:

```
cd archive
./unpack.sh
cd ../cases/default
./geoToMsh.sh
cd ../..
bin/MCFEM cases/default/default.cfg
```

The simulation can take a few hours to complete. However, if the program manages to enter the simulation loop after a few minutes it is likely that the program is working as expected.

A.2.2 Settings: The .cfg file

This part considers how to specify the main settings of MonteFFI. For instructions on how to implement new settings, see [https://github.com/jannisteunissen/config_fortran] or see examples in the code.

Syntax

In the .cfg file you will find the following syntax:

```
setting1 = 1
setting2 = string1

[group]
    groupSetting1 = 2.345D12
    groupSetting2 = heyho

setting3len = 4
setting3 = an array of strings
```

As you see, there are no type specifiers. The type is specified inside the Fortran program. When initializing arrays, you only need to separate entries with a space. The config file reading program used in MonteFFI can read all basic types and arrays of these types except COMPLEX.

Command line usage

As seen in the "Quick start", the .cfg (config) file is parsed to the program directly in the command line. You can also parse multiple files, as long as both files do not contain the same entries. To override a setting for this run only (not altering the .cfg file), write

```
program file.cfg -setting=newVal
```

With a setting under a group, this becomes

```
program file.cfg -groupName%setting=newVal
```

You can do this with multiple settings one after another. This can be very useful for running many simulations with slightly different settings.

Region settings

`regionCount` specifies how many regions to include. It is assumed that tags start at 1 and increase incrementally. Hence, if you specify `regionCount = 5` you need to have 5 physical regions with tags 1, 2, 3, 4 and 5.
`regionNames` specifies group names for each region in the config file. The regions should be in sorted order to match the physical tags in the mesh file. For example, if `regionNames = iso P N` it is assumed that settings under the group [iso] apply to the region with physical tag 1, [P] applies to region 2 and [N] applies to region 3. You should have as many region names as you specify for `regionCount`.

`regionType` is a variable that needs to be set for every region group. Currently supported types are normal, contact and isolation. Isolation regions do not require specifying any other setting. For both normal and contact regions we require `impurityType`, which are either "donor" or "acceptor", and `impurityDensity` which is specified as float (e.g. 1.D20). Contact regions also require specifying the potential `pot` in volts, and a `surfaceTag` which is the physical tag of the surface where Dirichlet conditions are applied.

"Core" settings

Under the group tag [core] (needs a better name), you specify the parameters regarding the MC kernel.

- `timeStepDelta` specifies the duration of each time increment.
- `simsteps` specifies the number of simulation steps.
- `esize` specifies the number of superelectrons. `hsize` is automatically set such that the superhole charge matches the charge of superelectrons.
- `loadParticles` specifies whether or not resume a previous simulation. If you set it to 1, you also need to specify `efile` to load electrons from and `hfile` to load holes from under the tag [load_particles].
- `isPoissonOn` specifies whether to run self-consistent or non-self-consistent simulations.
- `poicall` (integer) is the frequency of field updates as a multiple of `timeStepDelta`.
- `isPolarOptical`, `isAcoustic` , and similar "is"-values specifies which scattering mechanisms to include (F or T).

FEM settings

Under the tag [fem] you specify settings for the finite element solver.

- `GMSH_file` needs to be the path to the .msh file relative to the directory you execute the program from.
- `elem_order` specifies which polynomial degree the mesh has. Values can be either 1 or 2, although 2 has not been properly tested since Fatnes's thesis.
- `loadAssembly` specifies whether to load assembled matrices from file or start from scratch (1 or 0).
- `stiff_file` and `stiffBC_file` are the file paths to store or load assembly matrices in depending on the value of `loadAssembly`

A.2.3 Mesh generation with Gmsh

Here a brief introduction to the most important features of Gmsh required for creating a mesh for MonteFFI.
Assuming you have Gmsh installed, start the program to create a new script with

```
gmsh /path/to/new/gmsh/script.geo
```

You should now have the Gmsh GUI in front of you. Next, open the .geo script in a text-editor (this can be done through the Gmsh GUI by clicking Geometry -> Edit script) and write and save the following lines

```
// Export mesh with gmsh file format 2.2
Mesh.MshFileVersion = 2.2;
// Use the OpenCASCADE geometry kernel (for boolean operations)
SetFactory("OpenCASCADE");
```

Whenever we switch between the GUI and the text editor we need to reload the script. To reload the script in the GUI, click Geometry -> Reload script.

Drawing geometry

Lets consider a simple PN-diode. To draw the geometry, we will start by creating two boxes put next to each other. In the GUI, click on Geometry -> Elementary entity -> Add -> Box. The parameters should be self-explanatory. Press ‘e’ to add a box, and ‘q’ to exit. Open the script in your editor, and you should have something like this:

```
Box(1) = {-0.5, 0, 0, 1, 1, 1};
Box(2) = {0.5, 0, 0, 1, 1, 1};
```

What may come as surprising is that if you were to generate the mesh now you would end up with two separate meshes on the intersecting surface between the two boxes. To fix this, we need to somehow combine the surface of each box.
First we turn on volume visibility. This can be done by clicking Tools -> options in the top toolbar, and then selecting Geometry and checking the box for Volumes. Now you should see two colored dots at the center of each box.
Next click Geometry -> Elementary entity -> Boolean -> Fragments. Make sure to mark both object and tool for deletion. Now click on one of the volume dots and press ‘e’, and then click on the other volume dot and press ‘e’ again. Press ‘q’ to quit.
Now you should have no overlapping surfaces. The generated command in your script should look like this:

```
BooleanFragments{ Volume{1}; Delete; }{ Volume{2}; Delete; }
```

Now we have all the geometry we need for this model.
At this point you may be thinking it is easier to write the script manually. For this simple geometry, it probably is. However, for more complex geometry you will have to switch back and forth to keep track of everything.

Mesh Field

There are many ways to control the mesh refinement in Gmsh. Presented here is a simple method which can be applied to any simulation.
We start by making two box fields. To do this, click Mesh -> Size fields. In the pop-up window, click New -> Box. Then specify the same dimensions as the geometrical boxes. VIn specifies the edge length of the elements inside the box, and VOut is the edge length of elements outside the box. You should end up with two box fields in your script, something like this:

```
Field[1] = Box;
Field[1].VIn = 0.1;
Field[1].VOut = 10;
Field[1].XMax = 0.5;
Field[1].XMin = -0.5;
Field[1].YMax = 1;
Field[1].ZMax = 1;

Field[2] = Box;
Field[2].VIn = 0.2;
Field[2].VOut = 10;
Field[2].XMin = 0.5;
Field[2].XMax = 1.5;
Field[2].YMax = 1;
Field[2].ZMax = 1;
```

Min and max xyz-values are automatically zero unless specified otherwise.
Next we define a Min-field, which uses the minimum value of the edge length specified by other fields. This is done by entering the fields menu again (click Mesh -> Size fields), and then selecting New -> Min. Add field 1 and 2 (integer ID for each box field, seperated by comma) to the FieldsList. Click “Set as background field”, and then click apply. The following lines should now appear in your script:

```
Field[3] = Min;
Field[3].FieldsList = {1, 2};
Background Field = 3;
```

Now we can generate a mesh by simply clicking Mesh -> 3D. You may or may not see the mesh yet. To toggle mesh visibility and change mesh colors, go to Tools -> Options, and then Mesh -> Visibility/Color. However, the mesh does not contain the necessary physical tags covered next.

Physical tags

Specifying physical tags is very straightforward. Click Geometry -> Physical groups -> Add -> Volume. Give each region a name and a tag starting from 1 (the other will be 2). Check that this is in your script:

```
Physical Volume("P", 1) = {1};
Physical Volume("N", 2) = {2};
```

This should hopefully be sufficient for you to understand how to generate a basic mesh. Next you need to learn how to specify contacts, which is best understood by simply studying the .geo file of the default APD test case.

Generating the mesh

Once you are happy with your .geo file, write a command similar to the following to generate a mesh:

```
gmsh geometry.geo -3 -order 1 -o mesh.msh -format "msh2"
```

It is advisible to make a simple script file or alias instead of having to remeber such commands.

A.3 Cluster Computing Guide

Following this step by step guide, you should be able to run MonteFFI on IDUN. It is expected that you have already read the installation guide beforehand.
MonteFFI works on IDUN without any extra installations, since Gfortran, Ifort and MKL are all preinstalled. IDUN supports many other packages as well, e.g. Python and Matlab for your plotting needs. However, generation of the mesh files with Gmsh must be done locally (IDUN does not have Gmsh).
For more information see the IDUN homepage [https://www.hpc.ntnu.no/display/hpc/Idun+Cluster].

A.3.1 Getting started

First you need to get access to IDUN, e.g. by following the steps here: [https://www.hpc.ntnu.no/display/hpc/How+to+get+access+to+Idun]. You need to be on campus or use VPN to connect to IDUN.
Now copy MonteFFI over to IDUN and log in to one of the login-nodes. The "Getting Started on Idun" guide, [https://www.hpc.ntnu.no/display/hpc/Getting+Started+on+Idun], shows you how to do this.
Now you need to compile the MonteFFI program. Since MonteFFI requires MKL, we need the MKLROOT path. To find this you need to follow a few steps using the CLI on the cluster:
First write

```
module spider imkl
```

This will show you the available versions of MKL on IDUN. Choose any version, e.g. the newest one, and write

```
module spider imkl/<version>
```

This will show you the modules which MKL depends on. Load all these modules as listed by writing

```
module load <module>/<version>
```

before you load the MKL module by writing

```
module load imkl/<version>
```

Now you should get a path by writing

```
echo $MKLROOT
```

Copy this path and replace the string to the right of MKLROOT in the compilation script.
In order to compile with ifort, you need to load it first (due to licensing reasons, not necessary for gfortran):

```
module load ifort
```

Now you should be able to successfully compile MonteFFI.

A.3.2 Scheduling a job

To run MonteFFI on IDUN, the compiled executable must be added to the Slurm Queue. The following shows how to queue a Slurm job for executing MonteFFI on one of IDUN's nodes.
Note: Setting `OMP_NUM_THREADS` is not necessary on IDUN, since it is automatically set equal to the number of reserved cores.
Create a slurm-job file, e.g. job-fbmc.slurm, that is a script specifying what your job should do. Queue the job by writing

```
sbatch <slurm job file name>
```

which will queue a job that will start executing your program once enough resources are available. The following example shows how to set up the job file:

```
#!/bin/sh

#SBATCH --partition=CPUQ
#SBATCH --time=01-02:03:04
#SBATCH --nodes=1
#SBATCH --mem=12345
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=20
#SBATCH --job-name=MonteFFI
#SBATCH --output=MonteFFI.out
#SBATCH --mail-user=username@stud.ntnu.no
#SBATCH --mail-type=ALL
#SBATCH --exclude=compute-3-0-[1-8],\
#SBATCH           compute-4-0-[1-19],\
#SBATCH           compute-5-0-[1-5,17-23,25,27]

cd mc-fullband
stdbuf -o0 -e0 bin/MonteFFI cases/<case name>/settings.cfg
```

Almost all parts of the above script are essential for correct execution. Lines starting with `#SBATCH` are job settings processed by the job queueing system Slurm. Lets go through each setting one by one:

- `--partition` specifies what type of nodes should be used. WORKQ on IDUN is for jobs that do not require a GPU, and will select any available node that satisfies the other resource requirements.
- `--time` specifies the maximum wall time before aborting the job if it has not finished yet. Format days-hours:minutes:seconds.
- `--nodes` specifies the number of nodes to allocate for the job.
- `--mem` specifies the memory (in MB) to allocate per node, which is the total memory when only using 1 node.
- `--ntasks-per-node` How many tasks to run per node (for MonteFFI there is only one task, not sure if this option needs to be set.)
- `--cpus-per-task` sets how many CPUs to reserve per task. Since we only have one task, this is how many CPU cores we reserve for MonteFFI.
- `--job-name` sets a name for the job that others can see, e.g. when using the `squeue` command.
- `--output` sets the name of the file that stdout (write to screen) is stored in.
- `--mail-user` is the mail address to send updates about the job. E.g. when the job starts and when it has finished.
- `--mail-type` specifies what updates to receive. A useful tip is to make a filter for mail with subject "root" if you choose to receive all mail. Otherwise, your inbox will quickly be filled up with emails if you run many jobs. You typically get 2 mails per job.
- `--exclude` specifies what nodes not to use. In the example, all nodes except compute-2-0-[1-27] have been excluded, guaranteeing that one of the compute-2-0-[1-27] nodes will be used.

A few words on the "non-sbatch" lines should be useful to you as well:

- `#!/bin/sh` indicates that this file is a bash script.
- `cd mc-fullband` is used here to move into the working directory where the program needs to be called from. The initial path of the script is where you call it from. For the above script to work, this job must be queued from the same folder as mc-fullband is in.
- `stdbuf -o0 -e0` ensures that the stdout (-o0) and stderr (-e0) are not buffered for the following command. This ensures the output file is written to continuously. Otherwise, the output file might not be written to before the program has finished running. Using this command is especially useful when debugging and/or running long simulations.

B | Program Files

This appendix chapter contains selected program files created during this work.

B.1 Gmsh Script

```
//// Settings

// Export mesh with gmsh file format 2.2
Mesh.MshFileVersion = 2.2;
// Use the OpenCASCADE geometry kernel (required for boolean operations)
SetFactory(             );


//// Parameters

// Device dimensions
lx = 32;
ly = 11;
lz = 1;

// Isolation layer thickness
liso = 1;

// Contact height
dy = 0.1;

// P+
ppx = 3;
ppy = 1;

// N+
npx = 10;
npy = 1;

// N-
nx = 18;
ny = 5;

// P-
px = lx - 2*liso;
py = ly - liso;

```

```
//// Generate geometry

// Left P+
Box(1) = {liso, ly-ppy, 0, ppx, ppy, lz};
// Left contact
Box(2) = {liso, ly-dy, 0, ppx, dy, lz};

// Right P+
Box(3) = {lx-liso-ppx, ly-ppy, 0, ppx, ppy, lz};
// Right contact
Box(4) = {lx-liso-ppx, ly-dy, 0, ppx, dy, lz};

// N+
Box(5) = {(lx-npx)/2, ly-npy, 0, npx, npy, lz};
// N+ contact
Box(6) = {(lx-npx)/2, ly-dy, 0, npx, dy, lz};

// N-
Box(7) = {(lx-nx)/2, ly-ny, 0, nx, ny, lz};
// P-
Box(8) = {liso, liso, 0, 30, 10, lz};
// Iso
Box(9) = {0, 0, 0, 32, 11, lz};



//// Specify mesh parameters

// Extra padding
lextra = 0.5;

// Grid spacing
hiso = 0.75;
hp = 0.4;
hn = 0.2;
hpp = 0.15;
hnp = 0.12;

hpn = 0.15;
hpnp = 0.1;

Field[1] = Box;
Field[1].VIn = hp;
Field[1].VOut = hiso;
Field[1].XMin = liso;
Field[1].XMax = lx-liso;
Field[1].YMin = liso;
Field[1].YMax = ly;
Field[1].ZMin = 0;
Field[1].ZMax = lz;

Field[2] = Box;
Field[2].VIn = hn;
Field[2].VOut = hiso;
Field[2].XMax = (lx+nx)/2 + lextra;
Field[2].XMin = (lx-nx)/2 - lextra;
Field[2].YMax = ly;
```

```
Field[2].YMin = ly - ny - lextra;
Field[2].ZMin = 0;
Field[2].ZMax = lz;

Field[3] = Box;
Field[3].VIn = hpp;
Field[3].VOut = hiso;
Field[3].XMin = liso;
Field[3].XMax = liso + ppx + lextra;
Field[3].YMin = ly - ppy - lextra;
Field[3].YMax = ly;
Field[3].ZMax = lz;

Field[4] = Box;
Field[4].VIn = hpp;
Field[4].VOut = hiso;
Field[4].XMin = lx - liso - ppx - lextra;
Field[4].XMax = lx - liso;
Field[4].YMin = ly - ppy - lextra;
Field[4].YMax = ly;
Field[4].ZMin = 0;
Field[4].ZMax = lz;

Field[5] = Box;
Field[5].VIn = hnp;
Field[5].VOut = hiso;
Field[5].XMax = (lx+npx)/2 + lextra/2;
Field[5].XMin = (lx-npx)/2 - lextra/2;
Field[5].YMax = ly;
Field[5].YMin = ly - npy - lextra/2;
Field[5].ZMin = 0;
Field[5].ZMax = lz;

// P- to N-
Field[7] = Distance;
Field[7].NNodesByEdge = 40;
Field[7].FacesList = {88, 89, 93};
Field[8] = Threshold;
Field[8].IField = 7;
Field[8].DistMin = 0;
Field[8].DistMax = 1;
Field[8].LcMin = hpnp;
Field[8].LcMax = hiso;

// N- to N+
Field[9] = Distance;
Field[9].FacesList = {55, 59, 60};
Field[10] = Threshold;
Field[10].DistMin = 0;
Field[10].DistMax = 1;
Field[10].IField = 9;
Field[10].LcMin = hnp;
Field[10].LcMax = hiso;

// Apply min background field
Field[6] = Min;
Field[6].FieldsList = {1, 2, 3, 4, 5, 8, 10};
```

```
152 Background Field = 6;
153
154
155
156
157 //// GUI generated parameters. Not worth the time replacing with variables
158
159 // Seperate contact regions:
160 // Left P+
161 BooleanFragments{ Volume{5}; Delete; }{ Volume{6}; Delete; }
162 // Right P+
163 BooleanFragments{ Volume{1}; Delete; }{ Volume{2}; Delete; }
164 // N+
165 BooleanFragments{ Volume{3}; Delete; }{ Volume{4}; Delete; }
166
167 // Remove N+ and contact from N-
168 BooleanFragments{ Volume{7}; Delete; }{ Volume{10}; Volume{6}; Delete; }
169 // Remove all regions from P-
170 BooleanFragments{ Volume{8}; Delete; }{ Volume{13}; Volume{10}; Volume{6}; Volume{11}; Volume{2};
        Volume{12}; Volume{4}; Delete; }
171 // Remove all regions from Iso
172 BooleanFragments{ Volume{9}; Delete; }{ Volume{14}; Volume{13}; Volume{6}; Volume{10};
        Volume{11}; Volume{2}; Volume{12}; Volume{4}; Delete; }
173
174 // Iso
175 Physical Volume(1) = {15};
176 // P-
177 Physical Volume(2) = {14};
178 // P+ left
179 Physical Volume(3) = {11};
180 // P+ right
181 Physical Volume(4) = {12};
182 // N-
183 Physical Volume(5) = {13};
184 // N+
185 Physical Volume(6) = {10};
186 // P+ contact left
187 Physical Volume(7) = {2};
188 Physical Surface(10) = {105};
189 // P+ contact right
190 Physical Volume(8) = {4};
191 Physical Surface(11) = {109};
192 // N+ contact
193 Physical Volume(9) = {6};
194 Physical Surface(12) = {63};
```

B.2 Configuration File

```
regionCount = 9
#               1    2      3      4      5      6     7     8     9
regionNames = iso pminus pplusL pplusR nminus nplus cntLP cntRP cntN

[core]
   # Length of timestep:
   timeStepDelta = 2.D-15
   # Simulation steps:
   simsteps = 50000
   # Number of initial electrons:
   esize = 50000
   # Which particles to simulate (1: electrons, 2: holes, 3: both):
   simtype = 3
   # Load particle states from file:
   loadParticles = 1
   # Update electric potential (F:off/T:on):
   isPoissonOn = T
   # E-field update time step (multiple of basic timestep):
   poicall = 1

   # Scattering mechanisms

   # Polar optical scattering:
   isPolarOptical = T
   # Acoustic scattering:
   isAcoustic = T
   # Ionized impurity scattering:
   isIonizedImpurity = T
   # Nonpolar scattering:
   isNonpolar = T
   # Intervalley scattering:
   isIntervalley = T
   # Alloy scattering:
   isAlloy = T
   # Impact ionization scattering:
   isImpactionization = F

   # Apply alloy effect on electric field (on/off):
   isAlloyGradField = F


# Files to load if loadParticles==1 :
[load_particles]
   efile = data/electrons11.dat
   hfile = data/holes11.dat


# FEM settings:
[fem]
   elem_order = 1
   GMSH_file  = cases/test3/mesh.msh
   loadAssembly = 1 # 0: start from scratch, 1: load existing matrices
   stiff_file = cases/test3/bulk_matrix.dat
   stiffBC_file = cases/test3/boundary_matrix.dat
```

```
55
56
57 # Region specifications
58 [iso]
59    regionType = isolation
60
61 [pminus]
62    regionType = normal
63    impurityType = acceptor
64    impurityDensity = 1.D22
65
66 [pplusL]
67    regionType = normal
68    impurityType = acceptor
69    impurityDensity = 2.D22
70
71 [pplusR]
72    regionType = normal
73    impurityType = acceptor
74    impurityDensity = 2.D22
75
76 [nminus]
77    regionType = normal
78    impurityType = donor
79    impurityDensity = 5.D20
80
81 [nplus]
82    regionType = normal
83    impurityType = donor
84    impurityDensity = 2.D23
85
86 [cntLP]
87    regionType = contact
88    impurityType = acceptor
89    impurityDensity = 2.D22
90    surfaceTag = 10
91    pot   = -7.0
92
93 [cntRP]
94    regionType = contact
95    impurityType = acceptor
96    impurityDensity = 2.D22
97    surfaceTag = 11
98    pot   = -7.0
99
100 [cntN]
101    regionType = contact
102    impurityType = donor
103    impurityDensity = 2.D23
104    surfaceTag = 12
105    pot   = 0.0
```

B.3 Makefile

```
# ##############################################################################
# "GNU Make" file for MonteFFI (FFI-MCS)
#
# Author: Andreas Bolstad
# Date: 2020-05-18
#
# Coverage (verified to work with):
# Gfortran 6.3.0 on Debian 10
# Ifort 19.0.1.144 on CentOS Linux release 7.4.1708
#
# If you haven't used make before I would suggest getting a hold of
# "Managing Projects with GNU Make" by Robert Mecklenburg
# which provides a thorough but to-the-point introduction.
# ##############################################################################


### Contents
# 1. Short guide
# 2. Settings
# 3. Build rules


#################################
####### SHORT SYNTAX GUIDE #######
# Syntax for build rules:
# targets: prerequisites
#     rules
#
# Example:
# fileA.o: file1.o file2.o
#     gfortran -Wall -J modules -c fileA.f90 -o fileA.o
#
# Variable assignment:
# ":=" assigned in order (as expected in normal scripts)
# "=" assigned at runtime (always the latest available value)
#
# Automatic variables for make:
# $@ Filename of the target.
# $< Filename of the first prerequisite.
# $? Filenames of all prerequisites that have been updated more recently than the target.
# $^ Filenames of all prerequisites (no duplicates)
# $+ Filenames of all prerequisites with duplicates
# $* Stem of target filename (without its suffix, e.g. main.o -> main)
# $(<symbol>D) Only directories
# $(<symbol>F) Only files
#
# Print to console:
# $(info <text to print>)
#



#########################
```

```
55 ####### SETTINGS #########
56
57 ### Name of all programs
58 PROGS = MonteFFI HalvorMenu
59 ## 3D FE FB MC simulator
60 # MonteFFI
61 ## Program with menu to use KPBAND and SCRATE
62 # HalvorMenu
63
64
65 ### File locations
66 SRC_ROOT = src
67 # Generate list of all source directories automatically:
68 SRC_DIRS = $(shell find $(SRC_ROOT) -type d)
69 # Example:
70 # SRC_DIRS = src/constants src/utilities src/core
71
72 # Source files (every file in every directory)
73 SRC_FILES = $(shell find $(SRC_DIRS) -name *.f90)
74
75 # Build output directories
76 MOD_DIR := modules
77 OBJ_DIR := build
78 BIN_DIR := bin
79 OUT_DIRS := $(MOD_DIR) $(OBJ_DIR) $(BIN_DIR)
80
81 # Simulation data output directory
82 DATA_DIR := data
83
84
85 ### Sensible make settings and turning off magic (unexpected) functionality
86 # Make directory (and any superdirectory recursively) if it does not exist already.
87 MKDIR := mkdir -p
88 # Remove everything specified (Be careful!)
89 RM := rm -rf
90 # Do not use implicit suffixes rules. (We make our own rules...)
91 .SUFFIXES:
92 # Delete all unfinished files if program crashes. (Ought to be default, but isn't...)
93 .DELETE_ON_ERROR:
94 # Use the bash shell!
95 SHELL := bash
96 # Only use one shell. (E.g. allows us to use bash variables and loops)
97 .ONESHELL:
98 # Warn if dereferenced variable is undefined
99 MAKEFLAGS += --warn-undefined-variables
100 # No magic! Only do as you're told Make...
101 MAKEFLAGS += --no-builtin-rules
102
103
104 ### Compilation settings. RHS applies unless specified otherwise in the CLI
105 # Use the GNU Fortran compiler by default
106 FC    = gfortran
107 # Debugging off by default
108 DEBUG = 0
109 # Optimizations on by default
110 OPTIM = 1
111 # Profiling (Gprof) off by default
```

```
112 PROFILE = 0
113 # OpenMP parallelization on by default
114 OPENMP = 1
115
116 # Where to find the MKL library (Require lapack77 routine ZHEEV used in
        halvorsen/halvorrates.f90.)
117 # AB laptop
118 MKLROOT=/opt/intel/mkl
119 # IDUN cluster 2020-02-26
120 # MKLROOT=/share/apps/software/imkl/2019.5.281-iimpi-2019b/mkl
121
122 ## Compilation string of MKL. Should probably be done differently, but this works..
123 ## Currently works for both gfortran and ifort (2020-04-03)
124
125
126 ### Compilation flags
127 # See docs/compilerFlags.md for a description of each flag
128
129 # Gfortran
130 GFORTRAN_COMMON_FLAGS := -std=f2008 -ffree-line-length-none -Wall -pedantic -Wextra -Wconversion
        -Wsurprising
131 GFORTRAN_DEBUG_FLAGS := -g -fbacktrace -fcheck=all -ffpe-trap=invalid,zero,overflow,underflow
132 GFORTRAN_OPTIM_FLAGS := -O3 -march=native
133 GFORTRAN_PROFILE_FLAGS := -g -pg
134 GFORTRAN_OPENMP_FLAGS := -fopenmp
135 GFORTRAN_MOD_FLAG := -J
136 # Using -Ofast is 10% faster than O3, but is it safe?
137
138 ## String for linking MKL.
139 ## Generated with the online Intel MKL Link Line Advisor
140 GFORTRAN_MKL = -Wl,--start-group \
141                       ${MKLROOT}/lib/intel64/libmkl_gf_ilp64.a \
142                       ${MKLROOT}/lib/intel64/libmkl_sequential.a \
143                       ${MKLROOT}/lib/intel64/libmkl_core.a \
144                       -Wl,--end-group \
145                       -lpthread -lm -ldl \
146                       -fdefault-integer-8 -fdefault-real-8 -m64 -I${MKLROOT}/include
147
148
149 # Ifort
150 IFORT_COMMON_FLAGS := -standard-semantics -warn all # Note: "-standard-semantics" includes
        "-assume byterecl"
151 IFORT_DEBUG_FLAGS := -g -traceback -check all -fpe0 # Maybe add O0 if your output is cryptic
152 IFORT_OPTIM_FLAGS := -fast # Short for: -ipo -O3 -no-prec-div -static -fp-model fast=2 -xHost
153 IFORT_PROFILE_FLAGS := -g -p
154 IFORT_OPENMP_FLAGS := -qopenmp
155 IFORT_MOD_FLAG := -module
156
157 INTEL_MKL = -Wl,--start-group \
158                     ${MKLROOT}/lib/intel64/libmkl_intel_ilp64.a \
159                     ${MKLROOT}/lib/intel64/libmkl_sequential.a \
160                     ${MKLROOT}/lib/intel64/libmkl_core.a \
161                     -Wl,--end-group \
162                     -lpthread -lm -ldl \
163                     -i8 -r8 -I${MKLROOT}/include
164
165
```

```
166
167 # Selection of flags (gfortran/ifort)
168 ifeq ($(FC), gfortran)
169   COMMON_FLAGS = $(GFORTRAN_COMMON_FLAGS)
170   DEBUG_FLAGS = $(GFORTRAN_DEBUG_FLAGS)
171   OPTIM_FLAGS = $(GFORTRAN_OPTIM_FLAGS)
172   PROFILE_FLAGS = $(GFORTRAN_PROFILE_FLAGS)
173   OPENMP_FLAGS = $(GFORTRAN_OPENMP_FLAGS)
174   MOD_FLAG   = $(GFORTRAN_MOD_FLAG)
175   MKL        = $(GFORTRAN_MKL)
176 else ifeq ($(FC), ifort)
177   COMMON_FLAGS = $(IFORT_COMMON_FLAGS)
178   DEBUG_FLAGS = $(IFORT_DEBUG_FLAGS)
179   OPTIM_FLAGS = $(IFORT_OPTIM_FLAGS)
180   PROFILE_FLAGS = $(IFORT_PROFILE_FLAGS)
181   OPENMP_FLAGS = $(IFORT_OPENMP_FLAGS)
182   MOD_FLAG   = $(IFORT_MOD_FLAG)
183   MKL        = $(INTEL_MKL)
184 else
185   $(error $(FC)                                    )
186 endif
187
188 # Add flags (specified in settings)
189 FFLAGS = $(COMMON_FLAGS)
190 ifeq ($(DEBUG), 1)
191   FFLAGS += $(DEBUG_FLAGS)
192 endif
193 ifeq ($(OPTIM), 1)
194   FFLAGS += $(OPTIM_FLAGS)
195 endif
196 ifeq ($(PROFILE), 1)
197   FFLAGS += $(PROFILE_FLAGS)
198 endif
199 ifeq ($(OPENMP), 1)
200   FFLAGS += $(OPENMP_FLAGS)
201 endif
202
203 ### Flags for final linking (hence "LD") all objects into an executable
204 LDFLAGS = $(FFLAGS) $(MKL)
205
206
207 ### Where make should look for each type of file
208 vpath %.f90 $(SRC_DIRS)
209 vpath %.mod $(MOD_DIR)
210 vpath %.o $(OBJ_DIR)
211
212
213
214
215 ############################
216 ####### BUILD RULES #######
217 # About:
218 # .PHONY targets are not built (i.e. don't create a file "all", just execute the command.)
219
220 ### Help function.
221 # How to use:
222 # Add "##" after a target to make a help message.
```

```
# Write "##@<category>" instead to also specify a category for the help message.
HELP_FUNC = \
   %help; \
   while(<>) { \
      if(/^([a-z0-9_-]+):.*\#\#(?:@(\w+))?\s(.*)$$/) { \
         push(@{$$help{$$2}}, [$$1, $$3]); \
      } \
   }; \
   print                            ; \
   for ( sort keys %help ) { \
      print         ; \
      printf(             , $$_->[0], $$_->[1]) for @{$$help{$$_}}; \
      print     ; \
   }


### Utility targets
.PHONY: help
help: ##@utils Show this help. See Makefile for more info about each target.
      @perl -e '$(HELP_FUNC)' $(MAKEFILE_LIST)

.PHONY: tags
tags:  ##@utils Create tags. Extremely useful for navigating the source code.
      ctags -R --langmap=fortran:.f90 $(SRC_ROOT)
# Requires a ctags program to use. I strongly recommend using universal-ctags.
# Install by writing:
# > sudo apt install universal-ctags
# Tags can be used in both emacs and vim (and most other IDEs and editors I think)


### Main targets
.PHONY: all
all: dirs $(PROGS) ##@main Compile everything and create all executables
      $(info )
      $(info Installation complete!)

.PHONY: clean
clean: ##@main Delete all build files
      $(RM) $(OUT_DIRS)

.PHONY: depend
depend: ##@main Create dependencies with fortdepend => Makefile.dep
      fortdepend -w -o Makefile.dep -i ISO_FORTRAN_ENV -f $(SRC_FILES)

# Create the dependency file using fortdepend which can be installed writing:
# > pip3 install --user fortdepend
# See installation guide at: https://pypi.org/project/fortdepend/

# Don't want to use fortdepend or it isn't working?
# You can also write the dependency file manually if you prefer, but this is quite cumbersome.
# An alternative auto-dependency program which has been around longer is makedepf90,
# which can be installed by writing
# > sudo apt install makedepf90
# Haven't gotten it to run without error on MCFEM yet... -AB 2020-04-04


### Targets of "all"
```

```
280
281 # Create necessary directories for building
282 .PHONY: dirs
283 dirs:
284       $(MKDIR) $(OUT_DIRS) $(DATA_DIR)
285
286 # Rule to apply for all programs
287 $(PROGS):
288       $(FC) $(addprefix $(OBJ_DIR)/, $(^F)) -o $(addprefix $(BIN_DIR)/, $@) $(LDFLAGS)
289       $(info Linking program     :)
290
291 #--- Explanation of messy part "$(addprefix $(OBJ_DIR)/, $(^F))":
292 # Purpose: Get the path of every object file needed to compile MCFEM
293 # Details:
294 # "addprefix" concatenates LHS to the beginning of each element of RHS.
295 # "$(^F)" gets each prerequisite without its path, only the filename.
296 # The prerequisites of MCFEM are specified in Makefile.dep (autogenerated).
297 # Q&A:
298 # Why not use "$^"? Because it will return file.o for first run, but build/file.o on second run
        (._.)
299 # No clearer way of doing this? I have tried to find better alternatives, but so far no luck.
300
301
302 ### Suffix rules (rule is added to any target %.<suffix>)
303 %.o: %.f90
304       $(FC) $(FFLAGS) $(MOD_FLAG) $(MOD_DIR) -c $< -o $(addprefix $(OBJ_DIR)/, $@)
305
306
307 ### Dependencies (Targets for each source code file: which files to compile before others.)
308 include Makefile.dep
309 # See phony target "depend" for automatic generation of this file
```