ABSTRACT

COSTOLANSKI, ANNE STEFANIE. Numerical Simulation of Resonant Tunneling Devices Described by the Wigner-Poisson Equations. (Under the direction of Carl T. Kelley.)

Double barrier resonant tunneling diodes (RTDs) have been studied in detail for over 30 years due to the interesting physical characteristics they display as well as their potential to be used in high-speed electronic devices. One of the main models that properly takes quantum tunneling effects into account in simulating RTD behavior is the Wigner-Poisson formulation. While several previous versions of the Wigner-Poisson model have been implemented, this dissertation describes a more efficient parallel version that scales well, produces solutions with a higher degree of numerical accuracy, and decreases simulation run times significantly.

The new model is written in C++ to more easily incorporate Sandia National Laboratories' Trilinos software, which provides flexible parallel data structures and numerous highly efficient solvers. Other numerical methods used to improve upon previous results include the use of non-uniform grids, higher order numerical methods, and the inclusion of analytic solutions where possible.

Current-voltage (I-V) curves for the steady-state version of the Wigner equation were compared to previous results to ensure the accuracy of the model and to measure performance improvement. Various convergence properties of the I-V curves and the Wigner function were analyzed, and a new study of time-dependent Wigner simulations was performed using fine grids to analyze the existence of current oscillations. Improvements to the Wigner model were evaluated using strong and weak scaling studies on both the steady state and time dependent versions of the model.

Numerical Simulation of Resonant Tunneling Devices Described by the Wigner-Poisson Equations

by

Anne Stefanie Costolanski

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Applied Mathematics

Raleigh, North Carolina

2013

APPROVED BY:

Ralph Smith	Alina Chertock
Gary Howell	Mansoor Haider

Carl T. Kelley
Chair of Advisory Committee

BIOGRAPHY

Born in Washington, D.C., Anne spent her elementary school years in D.C. and high school in McLean, VA. Her undergraduate institution was Virginia Polytechnic Institute, where she majored in Mathematics and minored in Physics, Computer Science, and French.

After a decade long career in finance (during which she earned a Certificate in Accounting from the University of Virginia), Anne decided to pursue her true passion – Math – and returned to graduate school at George Mason University. She earned a Masters degree part-time while still employed, and finished her graduate education as a full-time student at North Carolina State University.

ACKNOWLEDGEMENTS

This material is based upon work supported by, or in part by, the U. S. Army Research Laboratory and the U. S. Army Research Office under contract/grant number W911NF-0710112.

This work was also supported by the Army Research Office under grant W911NF-0910159; by the National Science Foundation under Grant DMS-0707220; and by the NanoRTD LLC. I also appreciate the support of Sandia National Laboratories via their summer internship program at the Computer Science Research Institute.

TABLE OF CONTENTS

LIST OF TABLES . . . . . **vii**

LIST OF FIGURES . . . . . **viii**

Chapter 1 Introduction to Resonant Tunneling Diodes **1**
1.1 Background 1
1.2 Resonant Tunneling Diodes 1
1.3 Applications 3
1.4 Physical Description 3
1.5 Negative Differential Resistance 4
1.6 Fabrication 6

Chapter 2 Modeling of Resonant Tunneling Diodes **9**
2.1 Approaches to modeling an RTD 9
2.2 Basic Quantum Mechanics 10
2.3 Density Matrix Formulation 12
2.4 The Wigner Distribution Function 13
2.5 The Wigner-Poisson Equations 17
2.6 Previous work with Wigner-Poisson 19

Chapter 3 Mathematical Theory **22**
3.1 Stability Theory for Ordinary Differential Equations 22
3.2 Bifurcation Theory 23
3.2.1 Turning point bifurcation 23
3.2.2 Hopf bifurcations 24
3.2.3 Hysteresis 26
3.3 Numerical Methods 28
3.3.1 Newton's Method 28
3.3.2 GMRES 30
3.3.3 Preconditioning 31
3.3.4 Continuation 31
3.3.5 Pseudo arclength continuation 33

Chapter 4 Discretization of the Wigner-Poisson formulation **35**
4.1 Domain discretization 35
4.2 Discretization of the Equations 37

Chapter 5 C++ Implementation **44**
5.1 Object Oriented Programming 44
5.2 Trilinos 45
5.3 Trilinos packages 46

5.3.1 Teuchos . . . 46
5.3.2 Epetra . . . 47
5.3.3 Amesos . . . 47
5.3.4 AztecOO . . . 47
5.3.5 NOX . . . 48
5.3.6 LOCA . . . 48
5.4 Alglib software . . . 49
5.5 Data structures in C++ code . . . 49
5.6 C++ Class Descriptions . . . 52
5.6.1 Data and Structure Setup . . . 52
5.6.2 Pre-computed Terms . . . 53
5.6.3 Wigner function terms . . . 54
5.7 Solution Process using Trilinos . . . 55
5.8 Parallelizing the Wigner function vector . . . 56

Chapter 6 Results using the C++ Code . . . 59
6.1 Comparison to Previous Models . . . 59
6.2 Run Time Analysis . . . 61
6.3 Parallel performance . . . 62
6.4 Convergence of the IV curve . . . 66
6.5 Wigner function analysis . . . 68
6.6 Time dependent Wigner simulations . . . 76
6.6.1 Fixed momentum space results . . . 77
6.6.2 Variable momentum space results . . . 81
6.7 Von Neumann Analysis . . . 87
6.7.1 Elimination of the scattering term . . . 88
6.7.2 Elimination of the potential term . . . 89
6.7.3 Von Neumann stability analysis for Backward Euler . . . 90
6.7.4 Von Neumann stability analysis for BDF-2 . . . 92
6.7.5 Von Neumann stability analysis for Crank-Nicolson . . . 97

Chapter 7 Scalability . . . 100
7.1 RTD Code Improvements . . . 100
7.1.1 Wigner Equation Changes . . . 101
7.1.2 Sparse Matrix Interpolation . . . 101
7.1.3 BLAS . . . 102
7.1.4 Optimization of Interpolation Routine . . . 103
7.1.5 Improvements to Data Layout . . . 105
7.2 Final Scaling Results . . . 105
7.2.1 Strong Scaling . . . 106
7.2.2 Weak Scaling . . . 107
7.2.3 Time Dependence . . . 113
7.3 Future Work . . . 115

7.3.1 Longer Devices 116

REFERENCES **119**

Appendices **125**
Appendix A COMPILING TRILINOS 126
A.1 Trilinos and Cmake 126
A.2 Environmental Variables 127
A.3 Configuring Trilinos 128
A.4 Make and make install 131
A.5 Compiling the RTD code to run with Trilinos 131
A.6 Submitting the RTDCode in the NCSU HPC environment 134
Appendix B DETAILS OF THE C++ DATA-DRIVEN STRUCTS 136
Appendix C Incorporating Trilinos within the RTD code 141
C.1 Parallel Processing 142
C.2 Reference Counted Pointers 142
C.3 Trilinos header files 143
C.4 Problem class and Problem Interface class 144
C.5 NOX setup 145
C.6 LOCA setup 149

LIST OF TABLES

Table 6.1 FORTRAN vs. C++ run time results 61
Table 6.2 Uniform vs. non-uniform C++ run time results 62
Table 6.3 Strong scaling study with coarse grid 64
Table 6.4 Strong scaling study with fine grid 65
Table 6.5 Strong scaling study using FORTRAN model 66
Table 6.6 Convergence results for the Wigner function at $V = 0.248$ 73

Table 7.1 Profiling statistics for the parallel RTD code 102
Table 7.2 Profiling statistics for the parallel RTD code after BLAS-3 implementation and data layout improvements 104
Table 7.3 Profiling statistics for the parallel RTD code after the change to Interpolation matrix structure 104
Table 7.4 Strong scaling study using ultra fine grid of $Nk = 4096$, $Nx = 2049$. . . . 106
Table 7.5 Weak scaling study for various Nk and Nx. 110
Table 7.6 Weak scaling profiling statistics 112
Table 7.7 Weak scaling profiling statistics, continued 113
Table 7.8 Time dependent strong scaling studies for the fine grid of $Nk = 4096$, $Nx = 2049$. 114
Table 7.9 Weak scaling study for the time dependent Wigner code keeping the total number of non-uniform grid points (per core) constant. 115

LIST OF FIGURES

Figure 1.1 Sample material parameters and device structure for a two barrier resonant tunneling diode. 4
Figure 1.2 Energy band profiles of a double barrier RTD at different bias states . . 5
Figure 1.3 Example of negative differential resistance. 7

Figure 3.1 Bifurcation diagram for the turning point bifurcation in equation (3.4). From [1]. 24
Figure 3.2 Supercritical Hopf bifurcation for equation 3.5 with $\mathcal{F}(\lambda, r^2) = \lambda - r^2$. From [1]. 25
Figure 3.3 Subcritical Hopf bifurcation for equation 3.5 with $\mathcal{F}(\lambda, r^2) = -(r^2 - c^2) + c^2 + \lambda$ and $c > 0$. From [1]. 26
Figure 3.4 Bifurcation diagram and hysteresis loop for equation (3.7) showing the steady state solutions as a function of c. From [1]. 27

Figure 4.1 Zero bias Wigner distribution for a 550 Å device. 36
Figure 4.2 Example of nonuniform grid. 38

Figure 5.1 Flowchart of the `computeF` function. 56
Figure 5.2 Sample non-uniform mesh using $Nk = 256$, $Nx = 257$. 57

Figure 6.1 Comparison of I-V curves using the FORTRAN, MATLAB, and C++ models. 60
Figure 6.2 Comparison of I-V curves using multiple cores for a $Nk = 2048$, $Nx = 1025$ discretization of the solution space. 63
Figure 6.3 IV curves for two fixed momentum meshes and multiple spatial meshes, which shows the IV curve does not depend on the size of the spatial mesh 67
Figure 6.4 Grid convergence for the IV curve using multiple momentum meshes . . 67
Figure 6.5 Close up of the grid convergence for fine momentum meshes 68
Figure 6.6 The Wigner distribution function at V = 0. 69
Figure 6.7 The Wigner distribution function at V = 0.248. 70
Figure 6.8 Comparison of Wigner distributions at $V = 0.248$ for $Nx = 257$ vs. $Nx = 1025$ (using $Nk = 2048$ for both). 71
Figure 6.9 Comparison of Wigner distributions at $V = 0.248$ for $Nk = 2048$ vs. $Nk = 8192$ (using $Nx = 513$ for both). 72
Figure 6.10 Magnifications of the head-on view of the Wigner distribution with the interior region ($|k| < 0.0005$) eliminated 74
Figure 6.11 Magnification of the side view of the Wigner distribution with the interior region ($|k| < 0.0005$) eliminated 75
Figure 6.12 Time dependence: Oscillations in the current for certain voltage values using an Nx = 86, Nk = 72 uniform grid and the FORTRAN model. From [2]. 76

Figure 6.13 Time dependence: Coarse grid oscillations for $Nk = 256$, $Nx = 257$. . . 78
Figure 6.14 Time dependence: Coarse grid oscillations for $Nk = 512$, $Nx = 513$. . . 79
Figure 6.15 Time dependence: Fine grid of $Nk = 1024$, $Nx = 1025$ shows no solution oscillations. 80
Figure 6.16 Time dependence: Fine grid of $Nk = 2048$, $Nx = 1025$ shows no solution oscillations. 80
Figure 6.17 Time dependent results for variable momentum range $|k| \leq 0.75$: $Nx = 257$, $Nk = 512$ 82
Figure 6.18 Time dependent results for variable momentum range $|k| \leq 1.10$: $Nx = 385$, $Nk = 1024$ 83
Figure 6.19 Time dependent results for variable momentum range $|k| \leq 1.65$: $Nx = 577$, $Nk = 2048$ 83
Figure 6.20 Time dependent results for variable momentum range $|k| \leq 2.50$: $Nx = 865$, $Nk = 4096$ 84
Figure 6.21 IV curves using $Nx = 265$, $Nk =$ various with a variable momentum domain 85
Figure 6.22 IV curves using various values for Nx and $Nk = 10000$ with a variable momentum domain 85
Figure 6.23 Convergence results using a variable momentum domain 86

Figure 7.1 Steady-state IV curves for three different device lengths. 117

Chapter 1

Introduction to Resonant Tunneling Diodes

1.1 Background

Over the past few decades, electronic devices have become smaller, and the functional demands placed on them greater. The need for efficient micro and nanoscale components to power these devices is immediate, and a variety of research on simulating these ultra-small devices has been undertaken. One such nanoscale device that has the potential to be used as a power source for devices on the micro and nanoscales is the resonant tunneling diode (RTD).

1.2 Resonant Tunneling Diodes

Resonant tunneling diodes (RTDs) are nanoscale semiconductor devices which were first proposed by Tsu and Esaki in 1973 [3]. They predicted a phenomenon known as negative differential resistance (NDR), in which the current through a tunneling barrier reaches a local maximum when the injected carriers achieve certain resonant energies. In 1974, Chang et al. fabricated the first RTD device that demonstrated evidence of negative differential resistance [4]. A decade

later, in 1983, Sollner et al. improved upon previously achieved results [5], and research on RTDs increased.

Numerous studies have been performed using a standard two barrier RTD, and thus the general features of the current-voltage (I-V) curves generated by these RTDs have been well known for some time. Two of the key features that make them stand out from other semi-conductor devices are their high speed operation and their ability to produce negative differential resistance [6].

Because tunneling is a very fast phenomenon, the RTD is among the fastest devices ever made. It can be shown by theoretical analysis that the time taken to switch from its current peak to valley, or visa versa, can be less than 1 ps. It has been demonstrated that as a mixer it can detect radiation up to 2.5 THz, and as an oscillator it can generate 700 GHz signals, with maximum operational oscillation frequency projected to be over 1 THz [7]. Their practical demonstration as a THz regime power source [8] motivated a great deal of research during the last two decades by a variety of authors [9, 10, 11, 12, 13, 14, 15, 16, 17].

Due to the multiple current peaks produced by an RTD, they can be used to build efficient devices that can perform more complex functions with a single device. The RTD has recently been explored for use as a replacement to conventional devices, such as analog-to-digital converters [18, 19], multi-valued memories [20], and flip-flops [21]. The RTD also serves as a building block for other three-terminal devices, such as the resonant-tunneling bipolar transistor and the resonant-tunneling hot-electron transistor, and has been incorporated in structures to study hot-electron spectroscopy [7]. The most significant problem RTDs face is that they are two-terminal devices, and thus good isolation between the input and output is difficult to achieve [6]. When RTDs are integrated with other three-terminal devices, isolation can be achieved, but speed is reduced. In order to eliminate this flaw in three-terminal devices, optical signals would have to be used to control the device, and thus far no device with this capability has yet been designed [6].

1.3 Applications

In addition to being useful for predicting RTD behavior and studying the instability processes within RTDs, this model has immediate relevance to other types of high frequency devices, such as RTD relaxation oscillators [22, 23, 24]. This same basic technique is already being developed for application to the study of spin-dependent transport in magnetic semiconductor systems [25, 26]. Furthermore, since RTDs are now used prolifically throughout nearly all electronic application areas, this simulation tool will be useful towards the study of: RTD-enhanced sensor devices [27]; RTD-based integrated data-processing logic circuits [28]; and Hybrid-RTD optical telecommunications technology [29].

1.4 Physical Description

Resonant tunneling diodes are generally composed of two different semiconductor materials, one with a large energy band gap and one with a smaller energy band gap, that are incorporated into the device so that resonant tunneling is achieved. Most RTDs that have been studied have a two-barrier structure that is composed in the following fashion:

- At each end of the device, a large region of a narrow energy band gap material which is infused with dopants
- Next to the dopant regions, a thin layer of the same narrow energy band gap material which is not doped (aka spacer regions)
- In the interior of the device, thin layers (aka barriers) of a larger energy band gap material, separated from each other by thin layers of the undoped narrow energy band gap material (the regions between barriers are called wells)

See figure 1.1 for an example layout for a two barrier RTD.

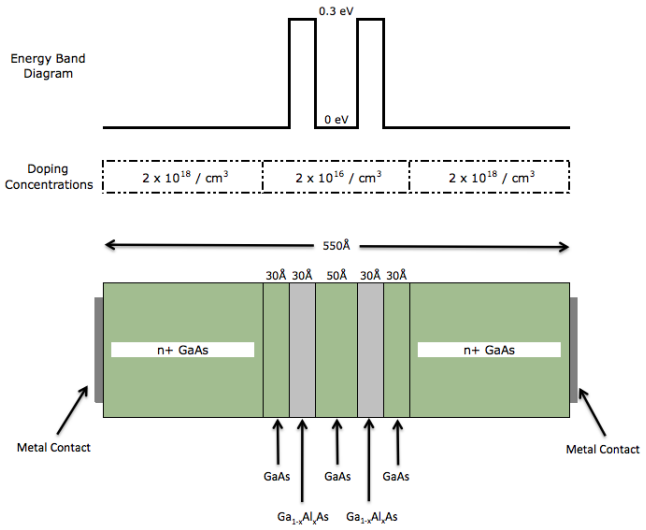

Figure 1.1: Sample material parameters and device structure for a two barrier resonant tunneling diode.

In order to facilitate current flow, the doped regions on either end of the device are generally large in size in comparison to the barrier, spacer, and well regions. In a typical RTD, the quantum well thickness might be around $50\mathring{A}$, and the barrier layers can range from 15 to $50\mathring{A}$ [7, 30]. The spacer regions, which are generally small and on the scale of the size of the barriers, ensure that dopants do not diffuse to the barrier layers. Bias is then applied across the device to induce current flow.

1.5 Negative Differential Resistance

The physics behind the behavior of a double barrier RTD is similar to that for a standard quantum well, where quantized energy states exist inside the area of the quantum well, but energies in the emitter and collector regions have a more continuous profile (as long as the energy is above the conduction band minimum). As an external voltage V is applied across the device, the effect is to lower the energy requirement to electron flow. Thus the right hand

side (collector region) of the energy band diagram shifts downward to indicate the reduction in minimum required energy. This is shown in figure 1.5, which depicts the energy band diagrams at four different voltage levels [6].

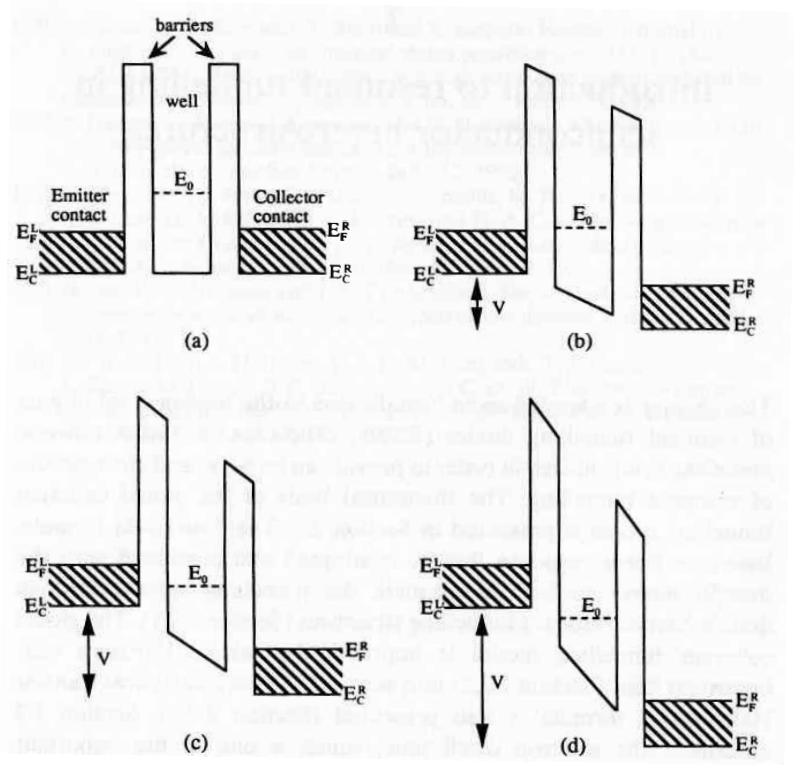

Figure 1.2: Energy band profiles of a double barrier resonant tunneling diode at different bias states: figure (a) is at zero bias, (b) threshold bias, (c) resonance, and (d) off-resonance. E_F^L and E_F^R represent the Fermi energies at the left and right (emitter and collector) ends of the device, and E_C^L and E_C^R represent the bottom of the conduction band in the left and right regions. From [6].

When no voltage is applied across the device, the energy band diagram looks like that shown in figure 1.5(a), with the first bound energy state in the quantum well area (labeled E_0 in the figures) above the conduction and Fermi energies. Since an electron must have an energy equivalent to E_0 in order to move into the quantum region and most energies are below that

level (between E_C^L and E_F^L in the diagram), there is minimal current flow.

However, once voltage is applied to the device, the first bound energy state in the quantum well is decreased due to the decrease in the minimum conduction band energy on the right (collector) side of the device, which allows current to begin to flow. Figure 1.5(b) depicts the energy band diagram when the bound state energy lines up with the Fermi energy on the emitter side of the device, which indicates the threshold of when current begins to flow more easily. Figure 1.5(c) shows the energy band diagram at resonance, which is when the maximum current is allowed to flow because the bound state energy E_0 is mid-way between the conduction band edge E_C^L and the Fermi level on the emitter side E_F^L. However, if too much voltage is applied across the device and the bound state energy no longer falls between the conduction band edge and the Fermi level, electron flow is impeded and the current drops dramatically. This is the case in figure 1.5(d).

The rapid drop in current which results from the transition from the energy state of figure (c) to that for figure (d) is called negative differential resistance. The current-voltage relationship is depicted in figure 1.3 [30].

1.6 Fabrication

Because RTDs are built of thin semiconductor layers with widths on the order of angstroms (Å), fabricating these devices is more difficult than standard macroscopic devices. In addition, because the principles of quantum mechanics dominate the physics at the nanoscale, measuring certain device characteristics is difficult and can change the resulting output. Thus, modeling these devices helps to more accurately predict device performance.

However, due to the complexity of the equations that describe electron interaction at the nanoscale, simulation models use approximations to predict the behavior of the device, as well as idealized material parameters in their calculations. Thus, when fabricating devices, high-

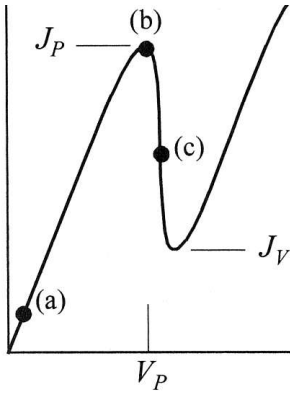

Figure 1.3: Example of negative differential resistance. Point (a) in the figure indicates a position near the beginning of electron flow; point (b) is at resonance; and point (c) shows a position in the area of negative differential resistance. J_P and V_P indicate the current and voltage, respectively, at maximum current flow (resonance); and J_V indicates the minimum current flow. From [30].

purity elemental sources should be used, or a large percentage of impurities may result in the material that can alter device performance enough to render the model useless.

Therefore, the parameters used in the model must reflect the use of materials that can closely replicate the device's predicted performance. For molecular sources that are group III semiconductors, Gallium (Ga) is one of the main materials used in fabricating RTDs due to the availability of 99.999999% (8N) pure Ga. Other materials, such as Indium (In) can also be used, although the available purity level is somewhat less at 99.99999% (7N). For the complimentary group V source, 7N arsenic (As) and 7N antimony (Sb) are available, although Sb can condense under certain conditions, which complicates crystal growth [31]. Due to its predominance in RTD fabrication, GaAs is the assumed semiconductor material used in this work.

Another potential fabrication problem lies with the doping levels of the contact regions. n-type doping can be achieved using extremely high-purity silicon; p-type doping is more difficult due to the lack of high-purity materials [31]. Thus most RTDs (including those modeled in this

work) have n-type doping in their structure.

Chapter 2

Modeling of Resonant Tunneling Diodes

2.1 Approaches to modeling an RTD

Since the behavior of an RTD is governed by quantum mechanics rather than classical physics, the standard drift-diffusion model [32] cannot be applied in calculating the current in the device. Several different methods have been proposed for modeling an RTD, which involve solving either the Schrödinger or Wigner equation at the quantum level, or adding quantum correction terms to classical macroscopic models. Several of the latter type have been proposed: the Density-Gradient (DG) model [33, 34], which uses the drift-diffusion model corrected with the Bohm potential; the Schrödinger-Poisson Drift-Diffusion model (SPDD) [35, 36], which takes into account the discrete spectrum of energy states for the electrons inside the expression of the density; and an entropic Quantum Drift Diffusion Model (eQDD), which was derived from a moment closure approach and extended to the context of quantum mechanics [37].

However, the standard approach that incorporates quantum effects throughout the derivation is the Wigner-Poisson model. The Wigner equation was first used by Frensley to model a

resonant tunneling diode in 1987 [8]. Since then, others have improved on the model by incorporating the Poisson equation and solving the system of equations self-consistently [38], and this model has been the foundation of much recent work [14, 12, 11, 15, 16, 17, 39, 13, 40, 2, 41, 42, 10, 43].

The Wigner function can be derived from the equation of motion of the nonequilibrium Green's function, as shown by Buot and Jensen [9]. To begin the derivation of Wigner's equation, we will start by introducing basic quantum mechanical principles that are necessary for the derivation.

2.2 Basic Quantum Mechanics

Assume a particle is in position $x(t)$ where t represents time. Quantum mechanics has five postulates (from [44, 45, 46]) that describe the behavior of the particle:

1. The state of a physical system is represented by a wavefunction $\psi(x(t))$ belonging to a complex Hilbert space $\mathcal{H}$. The wavefunction ψ and its first derivative are finite, continuous, and single valued.

2. Every observable physical quantity $\mathcal{A}$ is represented by a linear Hermitian operator A acting in $\mathcal{H}$. The operators in coordinate representation are:

   - Position $X \quad \Longrightarrow \quad x$
   - Momentum $P \quad \Longrightarrow \quad -i\hbar\frac{\partial}{\partial x}$
   - Energy $E \quad \Longrightarrow \quad i\hbar\frac{\partial}{\partial t}$

   where h is Planck's constant and $\hbar = \frac{h}{2\pi}$.

3. Let $\{a_1, a_2, \ldots a_n\}$ be the eigenvalues of the operator A corresponding to the eigenfunctions $\{\psi_1, \psi_2, \ldots \psi_n\}$. Then measurement of a physical quantity $\mathcal{A}$ will result in a real eigenvalue

a_i corresponding to the operator A. The expected value of the operator A when the system is in state ψ is:

$$\langle A \rangle = \int_{-\infty}^{\infty} \psi^* A \psi dx \tag{2.1}$$

where ψ is a linear combination of the eigenfunctions of A, and ψ^* represents the complex conjugate of the wavefunction ψ.

4. When the physical quantity $\mathcal{A}$ is measured, the probability $d\mathcal{P}(\alpha)$ of finding a result between α and $\alpha + d\alpha$

$$d\mathcal{P}(\alpha) = |\psi(\alpha)|^2 d\alpha \tag{2.2}$$

and the wave function is normalized to 1 such that

$$\int_{-\infty}^{\infty} |\psi(x)|^2 dx = 1. \tag{2.3}$$

5. Time evolution of the system is governed by the Schrödinger equation:

$$i\hbar \frac{d\psi}{dt} = H\psi \tag{2.4}$$

with

$$H = -\frac{\hbar^2}{2m^*} \frac{d^2}{dx^2} + U(x) \tag{2.5}$$

where m^* is the effective electron mass, h = Planck's constant, $\hbar = \frac{h}{2\pi}$, and $U(x)$ is the potential energy function.

The above are postulates for a single electron system, and are the foundation of quantum mechanical theory. However, to model a system involving a large number of particles (such as for a nanoscale device), changes must be made to the equations in order to capture the full dynamics of the system. Thus we will use the density matrix formulation, which describes the statistical state of a quantum system and uses the solutions to equation 2.4 as a basis to describe

the state of the system.

2.3 Density Matrix Formulation

Let $\psi(x)$ be an ensemble of wavefunctions $\psi_j(x)$ that are solutions to the one particle Schrödinger equation. Given a quantum mechanical system in thermal equilibrium, the density matrix ρ can be expressed as [6]:

$$\rho(x, x', t) = \sum_k \psi_k(x)\psi_k^*(x')f(k) \tag{2.6}$$

where x, x' are two positions in coordinate space and $f(k)$ is the Fermi-Dirac distribution function, which gives the probability that an electron will be in a given state. We are interested in how the system evolves over time, so the time derivative of the density operator is

$$\frac{\partial \rho(x, x', t)}{\partial t} = \frac{\partial}{\partial t}\left\{\sum_k \psi_k(x)\psi_k^*(x')f(k)\right\} \tag{2.7}$$

$$= \sum_k f(k)\frac{\partial}{\partial t}\left\{\psi_k(x)\psi_k^*(x')\right\} \tag{2.8}$$

since we are at thermal equilibrium (i.e., no time dependence for the Fermi-Dirac function). Using equations (2.4) and (2.5), we have

$$\frac{\partial}{\partial t}\left\{\psi_k(x)\psi_k^*(x')\right\} = \frac{\partial\psi_k(x)}{\partial t}\psi_k^*(x') + \psi_k(x)\frac{\partial\psi_k^*(x')}{\partial t} \tag{2.9}$$

$$= \frac{1}{i\hbar}H\psi_k(x)\psi_k^*(x') + \psi_k(x)\frac{1}{-i\hbar}H\psi_k^*(x') \tag{2.10}$$

$$= \frac{1}{i\hbar}\left\{\left[\frac{-\hbar^2}{2m^*}\frac{\partial^2\psi_k(x)}{\partial x^2} + U(x)\psi_k(x)\right]\psi_k^*(x') + \right.$$
$$\left. -\psi_k(x)\left[\frac{-\hbar^2}{2m^*}\frac{\partial^2\psi_k^*(x')}{\partial x'^2} + U(x')\psi_k^*(x')\right]\right\} \tag{2.11}$$

$$= \frac{i\hbar}{2m^*}\left[\frac{\partial^2\psi_k(x)}{\partial x^2}\psi_k^*(x') - \psi_k(x)\frac{\partial^2\psi_k^*(x')}{\partial x'^2}\right] +$$
$$+\frac{i}{\hbar}\left[U(x') - U(x)\right]\psi_k(x)\psi_k^*(x') \tag{2.12}$$

$$= \left\{\frac{i\hbar}{2m^*}\left[\frac{\partial^2}{\partial x^2} - \frac{\partial^2}{\partial x'^2}\right] + \frac{i}{\hbar}\left[U(x') - U(x)\right]\right\}\psi_k(x)\psi_k^*(x') \tag{2.13}$$

so that

$$\frac{\partial\rho(x,x',t)}{\partial t} = \left\{\frac{i\hbar}{2m^*}\left[\frac{\partial^2}{\partial x^2} - \frac{\partial^2}{\partial x'^2}\right] + \frac{i}{\hbar}\left[U(x') - U(x)\right]\right\}\rho(x,x',t) \tag{2.14}$$

2.4 The Wigner Distribution Function

The Wigner distribution function f represents the distribution of electrons in the device, and can be obtained from the density matrix [8] by a change of coordinates from (x, x') to (y, z) with

$$y = \frac{1}{2}(x + x') \qquad \text{and} \qquad z = x - x' \tag{2.15}$$

$$\Longrightarrow x = y + \frac{1}{2}z \qquad \text{and} \qquad x' = y - \frac{1}{2}z \tag{2.16}$$

where y represents the classical position variable, and the classical momentum variable is the Fourier transform of z. So

$$f(y,k,t) = \int_{-\infty}^{\infty} e^{-ikz}\rho(y + \frac{1}{2}z, y - \frac{1}{2}z, t)dz \tag{2.17}$$

To determine the time evolution of the Wigner function, we have

$$\frac{\partial f(y,k,t)}{\partial t} = \frac{\partial}{\partial t}\left\{\int_{-\infty}^{\infty} e^{-ikz}\rho(y+\frac{1}{2}z, y-\frac{1}{2}z, t)dz\right\} \tag{2.18}$$

$$= \int_{-\infty}^{\infty} e^{-ikz}\frac{\partial\rho(y+\frac{1}{2}z, y-\frac{1}{2}z, t)}{\partial t}dz \tag{2.19}$$

To complete the change of coordinates for equation (2.14), we use

$$\frac{\partial}{\partial x} = \frac{\partial y}{\partial x}\frac{\partial}{\partial y} + \frac{\partial z}{\partial x}\frac{\partial}{\partial z} = \frac{1}{2}\frac{\partial}{\partial y} + \frac{\partial}{\partial z} \tag{2.20}$$

$$\frac{\partial}{\partial x'} = \frac{\partial y}{\partial x'}\frac{\partial}{\partial y} + \frac{\partial z}{\partial x'}\frac{\partial}{\partial z} = \frac{1}{2}\frac{\partial}{\partial y} - \frac{\partial}{\partial z} \tag{2.21}$$

to rewrite the spatial derivatives:

$$\frac{\partial^2}{\partial x^2} - \frac{\partial^2}{\partial x'^2} = \left(\frac{\partial}{\partial x} - \frac{\partial}{\partial x'}\right)\left(\frac{\partial}{\partial x} + \frac{\partial}{\partial x'}\right) = \left(2\frac{\partial}{\partial y}\right)\left(\frac{\partial}{\partial z}\right) = 2\frac{\partial^2}{\partial y\partial z}. \tag{2.22}$$

Thus equation (2.19) becomes

$$\frac{\partial f(y,k,t)}{\partial t} = \int_{-\infty}^{\infty} e^{-ikz}\left\{\frac{i\hbar}{2m^*}\left[2\frac{\partial^2}{\partial y\partial z}\right] + \frac{i}{\hbar}\left[U(y-\frac{1}{2}z) - U(y+\frac{1}{2}z)\right]\right\} \tag{2.23}$$

$$\times\ \rho(y+\frac{1}{2}z, y-\frac{1}{2}z, t)dz \tag{2.24}$$

$$= F_1(y,z,t) + F_2(y,z,t) \tag{2.25}$$

where

$$F_1(y,z,t) = \frac{i\hbar}{m^*}\int_{-\infty}^{\infty} e^{-ikz}\frac{\partial^2\rho(y+\frac{1}{2}z, y-\frac{1}{2}z, t)}{\partial y\partial z}dz \tag{2.26}$$

$$F_2(y,z,t) = \frac{i}{\hbar}\int_{-\infty}^{\infty} e^{-ikz}\left[U(y-\frac{1}{2}z) - U(y+\frac{1}{2}z)\right]\rho(y+\frac{1}{2}z, y-\frac{1}{2}z, t)dz. \tag{2.27}$$

First, consider the F_1 term:

$$F_1(y,z,t) = \frac{i\hbar}{m^*}\frac{\partial}{\partial y}\left\{\int_{-\infty}^{\infty} e^{-ikz}\frac{\partial\rho(y+\frac{1}{2}z, y-\frac{1}{2}z, t)}{\partial z}dz\right\} \tag{2.28}$$

and

$$e^{-ikz}\frac{\partial\rho(y+\frac{1}{2}z, y-\frac{1}{2}z, t)}{\partial z} = \frac{\partial}{\partial z}\left\{e^{-ikz}\rho(y+\frac{1}{2}z, y-\frac{1}{2}z, t)\right\} + ike^{-ikz}\rho(y+\frac{1}{2}z, y-\frac{1}{2}z, t) \tag{2.29}$$

so

$$\begin{aligned} F_1(y,z,t) &= \frac{i\hbar}{m^*}\frac{\partial}{\partial y}\left\{\int_{-\infty}^{\infty}\frac{\partial}{\partial z}\left[e^{-ikz}\rho(y+\frac{1}{2}z, y-\frac{1}{2}z, t)\right]dz + \right. & (2.30)\\ &\qquad \left. + \int_{-\infty}^{\infty} ike^{-ikz}\rho(y+\frac{1}{2}z, y-\frac{1}{2}z, t)dz\right\} & (2.31)\\ &= \frac{i\hbar}{m^*}\frac{\partial}{\partial y}\left\{\int_{-\infty}^{\infty}\frac{\partial}{\partial z}\left[e^{-ikz}\rho(y+\frac{1}{2}z, y-\frac{1}{2}z, t)\right]dz + ikf(y,k,t)\right\} & (2.32)\end{aligned}$$

by equation (2.17). Then

$$\begin{aligned}\int_{-\infty}^{\infty}\frac{\partial}{\partial z}\left[e^{-ikz}\rho(y+\frac{1}{2}z, y-\frac{1}{2}z, t)\right]dz &= \lim_{z\to\infty} e^{-ikz}\rho(y+\frac{1}{2}z, y-\frac{1}{2}z, t) & (2.33)\\ &\quad - \lim_{z\to-\infty} e^{-ikz}\rho(y+\frac{1}{2}z, y-\frac{1}{2}z, t) & (2.34)\end{aligned}$$

Since

$$\begin{aligned}\lim_{x\to\pm\infty}\psi(x)\to 0, \qquad \lim_{z\to\pm\infty} e^{-ikz}\rho(y+\frac{1}{2}z, y-\frac{1}{2}z, t) = 0 & (2.35)\\ \implies \int_{-\infty}^{\infty}\frac{\partial}{\partial z}\left[e^{-ikz}\rho(y+\frac{1}{2}z, y-\frac{1}{2}z, t)\right]dz = 0 & (2.36)\end{aligned}$$

and thus

$$F_1(y,z,t) = -\frac{\hbar k}{m^*}\frac{\partial f(y,k,t)}{\partial y} \tag{2.37}$$

Next, consider the F_2 term. The inverse Fourier transform of equation (2.17) is

$$\rho(y+\frac{1}{2}z, y-\frac{1}{2}z, t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} e^{ikz} f(y,k,t)dk \tag{2.38}$$

so

$$\begin{aligned} F_2(y,z,t) &= \frac{i}{\hbar}\int_{-\infty}^{\infty} e^{-ikz}\left[U(y-\frac{1}{2}z) - U(y+\frac{1}{2}z)\right]\frac{1}{2\pi}\int_{-\infty}^{\infty} e^{ik'z} f(y,k',t)dk'dz & (2.39)\\ &= \frac{i}{h}\int_{-\infty}^{\infty} f(y,k',t)dk'\int_{-\infty}^{\infty} e^{-i(k-k')z}\left[U(y-\frac{1}{2}z) - U(y+\frac{1}{2}z)\right]dz & (2.40)\end{aligned}$$

Notice that for the integral over z, we can break it into two integrals, one for $(-\infty, 0)$ and the other for $[0,\infty)$. For the first integral, we can change the limits of integration to

$$\int_{-\infty}^{0} e^{-i(k-k')z}\left[U(y-\frac{1}{2}z) - U(y+\frac{1}{2}z)\right]dz = \int_{0}^{\infty} e^{i(k-k')z}\left[U(y+\frac{1}{2}z) - U(y-\frac{1}{2}z)\right]dz \tag{2.41}$$

so then for the full integral, we have

$$\begin{aligned} &\int_{-\infty}^{\infty} e^{-i(k-k')z}\left[U(y-\frac{1}{2}z) - U(y+\frac{1}{2}z)\right]dz & (2.42)\\ &\quad= \int_{0}^{\infty}\left[e^{i(k-k')z} - e^{-i(k-k')z}\right]\left[U(y+\frac{1}{2}z) - U(y-\frac{1}{2}z)\right]dz & (2.43)\\ &\quad= \int_{0}^{\infty} 2i\sin\left[\left(k-k'\right)z\right]\left[U(y+\frac{1}{2}z) - U(y-\frac{1}{2}z)\right]dz & (2.44)\end{aligned}$$

Therefore

$$\begin{aligned} F_2(y,k,t) &= \frac{i}{h}\int_{-\infty}^{\infty} f(y,k',t)dk'\int_{0}^{\infty} 2i\sin(z(k-k'))\left[U(y+\frac{1}{2}z) - U(y-\frac{1}{2}z)\right]dz & (2.45)\\ &= -\frac{2}{h}\int_{-\infty}^{\infty} f(y,k',t)dk'\int_{0}^{\infty} \sin(z(k-k'))\left[U(y+\frac{1}{2}z) - U(y-\frac{1}{2}z)\right]dz & (2.46)\\ &= -\frac{4}{h}\int_{-\infty}^{\infty} f(y,k',t)dk'\int_{0}^{\infty} \sin(2z(k-k'))\left[U(y+z) - U(y-z)\right]dz & (2.47)\end{aligned}$$

So the completed derivation of the Wigner distribution function is

$$\frac{\partial f(y,k,t)}{\partial t} = -\frac{\hbar k}{m^*}\frac{\partial f(y,k,t)}{\partial y} - \frac{4}{h}\int_{-\infty}^{\infty} f(y,k',t)dk' \int_0^{\infty} \sin(2z(k-k'))\left[U(y-z) - U(y+z)\right] dz. \tag{2.48}$$

However, interactions between electrons in the device are not taken into account as part of the derivation, so an additional term needs to be incorporated into the Wigner equation to properly account for these interactions. This can be done using the relaxation time approximation, so that a time derivative term $\frac{\partial f}{\partial t}|_{coll}$ is included on the right hand side specifically for collision processes.

2.5 The Wigner-Poisson Equations

Thus, the Wigner equation is written as

$$\frac{\partial f(y,k,t)}{\partial t} = K(f) + P(f) + S(f) \tag{2.49}$$

where $K(f)$ represents the effects due to kinetic energy on the distribution function f, $P(f)$ represents the potential energy contribution to the system, and $S(f)$ the electron-phonon collision effects. The kinetic energy term is given by

$$K(f) = -\frac{hk}{2\pi m^*}\frac{\partial f}{\partial y}. \tag{2.50}$$

where h is Planck's constant and m^* is the electron effective mass. The potential energy term is defined as

$$P(f) = -\frac{4}{h}\int_{-\infty}^{\infty} f(y,k',t)T(y,k-k')dk' \tag{2.51}$$

with

$$T(y,k-k') = \int_0^{\frac{L_c}{2}} \left[U(y+z) - U(y-z)\right]\sin(2z(k-k'))dz \tag{2.52}$$

where $U(y)$ is the potential energy function and L_c is the correlation length of the integral. The correlation length represents the maximum distance that one electron can feel the effects of another electron, and satisfies $L_c \leq L$. $U(y)$ can be written as

$$U(y) = \Delta_c(y) + u_p(y) \tag{2.53}$$

where $\Delta_c(y)$ represents the energy band function defined by the barriers and wells within the device and $u_p(y)$ represents the electrostatic potential found by the solution to Poisson's equation,

$$\frac{d^2 u_p(y)}{dy^2} = \frac{q^2}{\epsilon} \left[N_d(y) - n(y) \right]. \tag{2.54}$$

Here q is the charge on a electron, ϵ is the dielectric permittivity, and $N_d(y)$ is the concentration of ionized dopants. The electron density function, $n(y)$, is defined as

$$n(y) = \frac{1}{2\pi} \int_{-\infty}^{\infty} f(y,k) dk. \tag{2.55}$$

The boundary conditions for Poisson's equation are

$$u_p(0) = V_0, \qquad u_p(L) = V_L \tag{2.56}$$

where V_0 is the initial voltage at the left side of the device, and V_L is the amount of bias applied across the device. Traditionally $V_0 = 0$ and $V_L = -V$ with $V \geq 0$.

The third term in the Wigner equation, the scattering term $S(f)$, accounts for electron collision interactions in the device. Detailed scattering treatments create a heavy computational burden [47], so the lowest-order relaxation-time approximation is used in the model [11]. Thus the scattering term is defined as:

$$S(f) = \frac{1}{\tau} \left[\frac{\int_{-\infty}^{\infty} f(y,k,t) dk}{\int_{-\infty}^{\infty} f_0(y,k,t) dk} \cdot f_0(y,k) - f(y,k,t) \right] \tag{2.57}$$

where τ is the relaxation time of an electron in the device. $f_0(y,k)$ is the equilibrium Wigner distribution function, which is the solution to equation (2.49) with $S(f) = 0$ and no change in the bias voltage applied across the device; i.e., $V_L = V_0$.

Since the device to be modeled is of finite length, boundary conditions are imposed on the Wigner function. The boundary conditions represent the distribution of electrons emitted into the device from the reservoirs to which the device is attached [8]. Electrons can enter either from the left (at $y = 0$) with positive momentum, or from the right (at $y = L$) with negative momentum. The reservoirs are assumed to be at thermal equilibrium, so the electron distribution function is characterized by the thermal equilibrium distribution function of each reservoir [8]. Thus the boundary condition at $y = 0$ and $k > 0$ is:

$$f(0,k) = \frac{4\pi m^* k_B T}{h^2} \ln\left\{1 + \exp\left[-\frac{1}{k_B T}\left(\frac{h^2 k^2}{8\pi^2 m^*} - \mu_0\right)\right]\right\} \tag{2.58}$$

and for $y = L$ and $k < 0$:

$$f(L,k) = \frac{4\pi m^* k_B T}{h^2} \ln\left\{1 + \exp\left[-\frac{1}{k_B T}\left(\frac{h^2 k^2}{8\pi^2 m^*} - \mu_L\right)\right]\right\} \tag{2.59}$$

where k_B is Boltzmann's constant, T is the temperature, and μ_0 and μ_L are the chemical potential of the reservoirs at the corresponding ends of the device.

Finally, the current density in the device can be written as

$$j(x,t) = \frac{h}{2\pi m^*} \int_{-\infty}^{\infty} k f(x,k,t) dk. \tag{2.60}$$

2.6 Previous work with Wigner-Poisson

Previous work with the Wigner-Poisson model has been completed by a variety of authors[9, 8, 10, 11, 12, 13, 14, 15, 16, 17, 43]. However, with the exception of one version [43], the previous implementations have been limited by either (1) a coarse rendering of the domain

space by the chosen discretization in an effort to keep computation time to a minimum, which in turn decreases the accuracy of the solution, or (2) a fine grid discretization that places many grid points in the boundary regions where there is a negligible impact on the solution, which vastly increases computation time. These codes were written in FORTRAN, and the software utilized discretizations that did not allow much variety in the types of devices that were modeled, so modifications to the code were difficult. Upgrades to the FORTRAN versions included incorporating the LOCA [48] software from Sandia National Labs so that parallel computation could be used to decrease run times. However, while the upgrade allowed the use of finer meshes to refine the accuracy of the solution, it did not increase the ability to model longer and more complex device configurations.

In addition, while there has been some work on improving the numerical methods used to implement the Wigner function approach [49, 10], most of the previously published work has been focused on improving the formulation of the model [9] as well as studying the current oscillations present in the region of negative differential resistance [50, 11, 12]. Thus, most of the previous work has not made significant changes to the discretization of the equations, for which the numerical approximations had low order accuracy and some minor inaccurate computations.

Therefore, a more efficient quantum-mechanical electron transport code was developed in MATLAB [43] to simulate these devices. The MATLAB code required only one processor to duplicate the run times of the FORTRAN-LOCA version with 20 processors for short device lengths, due to the implementation of a nonuniform grid that significantly decreased computation time. In addition, the fourth order numerical methods were incorporated to produce more numerically accurate solutions. However, despite the advances made by the MATLAB code, simulating longer devices increased MATLAB run times significantly, and MATLAB's parallel computation toolbox is not well developed so parallel computation using MATLAB was not an option.

The work of this dissertation takes the MATLAB code a step further in creating an efficient Wigner-Poisson model to produce numerically accurate results with reasonable run times. Parallel computation must be incorporated into any efficient Wigner-Poisson model in order to broaden the abilities of the code to model longer and more complex devices. Thus, a new version of the Wigner-Poisson model, written in C++ and incorporating the highly efficient Trilinos software [51], has been developed to include parallel computation along with the improvements that were incorporated in the MATLAB version. This new version will be described in detail in future chapters.

Chapter 3

Mathematical Theory

3.1 Stability Theory for Ordinary Differential Equations

To determine the long term behavior of an RTD, we analyze the steady state solutions to the Wigner equation

$$\frac{\partial f(x,k,t)}{\partial t} = K(f) + P(f) + S(f) = W(f) \tag{3.1}$$

and determine their stability. To begin, we find vector(s) f^* for which $W(f^*) = 0$; the f^* are called equilibrium points. In a neighborhood of an equilibrium point f^*, Taylor's Theorem can be used to linearize equation 3.1 and provide a good first order approximation to the behavior of the nonlinear system [52]:

$$\frac{\partial f}{\partial t} \approx \frac{\partial W(f^*)}{\partial f} f \tag{3.2}$$

From linear stability theory, we know that the eigenvalues of the Jacobian at an equilibrium point f^* of equation 3.2 determine whether a solution nearby f^* will stay close to f^* as time evolves or diverge away from f^*. The criteria for stability is given in the following theorems:

Theorem 3.1. *An equilibrium point f^* of equation 3.2 is asymptotically stable if and only if all of the eigenvalues of $\frac{\partial W(f^*)}{\partial f}$ have negative real parts. [53]*

Theorem 3.2. *If one eigenvalue of* $\frac{\partial W(f^*)}{\partial f}$ *has a positive real part, then the equilibrium point* f^* *is unstable. [53]*

3.2 Bifurcation Theory

Since Poisson's equation involves the boundary condition $u_p(x = L) = -V$ where V is varied from 0 to V_{max}, we are solving a differential equation that looks more like

$$\frac{\partial f}{\partial t} = W(f, \lambda) \tag{3.3}$$

where λ is the voltage parameter. To better describe the possible dynamics of the system as the parameter changes, let's look at several examples.

3.2.1 Turning point bifurcation

The one-dimensional differential equation (from [1])

$$\dot{x} = c + x^2 \tag{3.4}$$

has two equilibrium points when $c < 0$ ($x_+ = +\sqrt{|c|}$ which is unstable, and $x_- = -\sqrt{|c|}$ which is stable); at $c = 0$, one equilibrium point; and none when $c > 0$. Thus, $c = 0$ is a bifurcation value:

Definition 3.1. *A bifurcation value is a specific value of a parameter for which the number of equilibrium points or the stability of the equilibrium points changes [54].*

A graphical representation for equation 3.4 is shown in figure 3.1, which is known as a saddle-node or turning point bifurcation [1, 52].

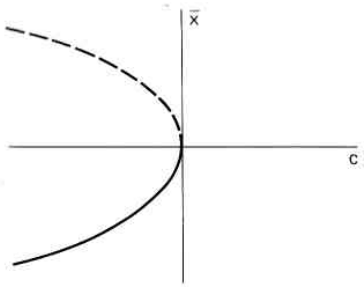

Figure 3.1: Bifurcation diagram for the turning point bifurcation in equation (3.4). From [1]. The set of stable equilibrium points is shown with a solid line, and the set of unstable equilibrium points as a dashed line.

3.2.2 Hopf bifurcations

Another type of bifurcation can be generated from the solutions to (from [1])

$$\begin{aligned} \dot{x_1} &= x_2 + \mathcal{F}(\lambda, r^2)x_1 \\ \dot{x_2} &= -x_1 + \mathcal{F}(\lambda, r^2)x_2 \end{aligned} \tag{3.5}$$

where $r^2 = x_1^2 + x_2^2$ and $\mathcal{F}(0,0) = 0$ so the origin is an equilibrium point [1]. The system can be rewritten in polar coordinates as

$$\begin{aligned} \dot{r} &= \mathcal{F}(\lambda, r^2)r \\ \dot{\theta} &= 1 \end{aligned} \tag{3.6}$$

For several specific forms of $\mathcal{F}$, periodic orbits can arise for certain values of λ and satisfy the criteria for a Hopf bifurcation:

Theorem 3.3. *Poincaré-Andronov-Hopf Theorem*

Let x_0 be an equilibrium point of equation 3.3 for all sufficiently small λ, and let $\alpha(\lambda) \pm i\beta(\lambda)$ denote a complex conjugate pair of eigenvalues of the Jacobian matrix $\frac{\partial F(x,\lambda)}{\partial x}$ which cross the

imaginary axis with nonzero speed (i.e., $\frac{\partial \alpha}{\partial \lambda} \neq 0$*). Then in a neighborhood* U *of* x_0 *and any* $\lambda_0 > 0$ *there exists* $\bar{\lambda}$ *with* $|\bar{\lambda}| < \lambda_0$ *such that equation 3.3 has a nontrivial periodic orbit in* U *[1].*

Let's look at two types of Hopf bifurcations, which are important in explaining the creation of periodic behavior: *supercritical* and *subcritical* Hopf bifurcations. For a supercritical Hopf bifurcation, a nontrivial stable periodic orbit is created at the bifurcation point, and there is a smooth transition from the equilibrium point to the periodic orbit [54]. If we set $\mathcal{F}(\lambda, r^2) = \lambda - r^2$ in equation 3.5, a supercritical bifurcation is present at $\lambda = 0$. See figure 3.2 for the bifurcation diagram [1].

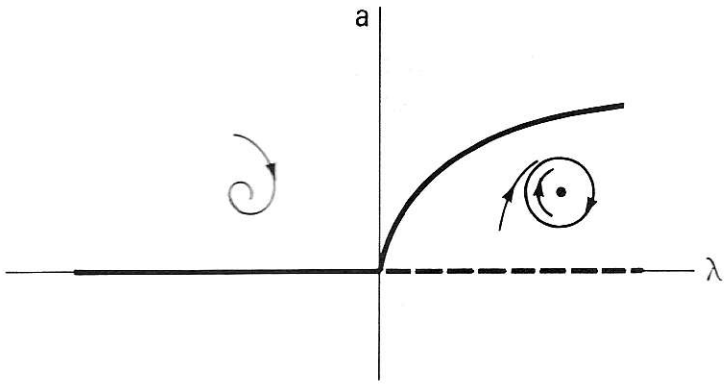

Figure 3.2: Supercritical Hopf bifurcation for equation 3.5 with $\mathcal{F}(\lambda, r^2) = \lambda - r^2$. For $\lambda \leq 0$, the equilibrium point $r = 0$ is asymptotically stable, but it becomes unstable for $\lambda > 0$ with the creation of an asymptotically stable periodic orbit. From [1].

A Hopf bifurcation is subcritical when a nontrivial unstable periodic orbit is created at the bifurcation point. If we set $\mathcal{F}(\lambda, r^2) = -(r^2 - c^2) + c^2 + \lambda$ with $c > 0$ in equation 3.5, a subcritical bifurcation is present at $\lambda = 0$. See figure 3.3 for the bifurcation diagram [1].

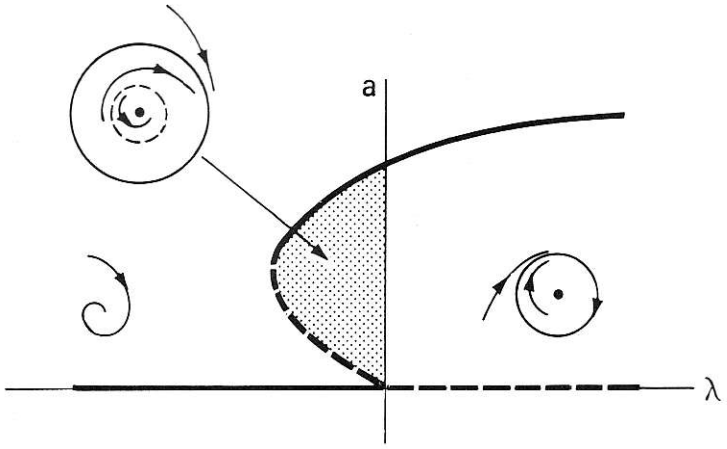

Figure 3.3: Subcritical Hopf bifurcation for equation 3.5 with $\mathcal{F}(\lambda, r^2) = -(r^2 - c^2) + c^2 + \lambda$ and $c > 0$. For $\lambda < 0$, the equilibrium point $r = 0$ is asymptotically stable. At $\lambda = 0$, an unstable nontrivial periodic orbit emerges, and for $-c^2 < \lambda < 0$, the unstable periodic orbit coexists with an asymptotically stable periodic orbit of larger amplitude. At $\lambda = -c^2$, the two periodic orbits merge and disappear. For $\lambda > 0$, the large amplitude periodic orbit continues to be asymptotically stable, and the equilibrium point $r = 0$ becomes unstable. From [1].

3.2.3 Hysteresis

Subcritical Hopf bifurcations can produce a phenomenon called hysteresis, which depends on the coexistence of two attractors at the same parameter value [54, 55]. This can be seen in figure 3.3 for the region $-c^2 < \lambda < 0$, since the equilibrium point $r = 0$ and the large amplitude periodic orbit are both asymptotically stable. To describe hysteresis in more detail, let's look at the one-dimensional equation (from [1])

$$\dot{x} = c + x - x^3 \tag{3.7}$$

Let $c^* = \frac{2}{3\sqrt{3}}$. For $c \leq -c^*$ and $c \geq +c^*$, there is one stable equilibrium point, but for $-c^* < c < +c^*$, there are three, with two (those with greater magnitude) stable and one unstable. The bifurcation diagram is shown in figure 3.4 on the left, with $-c^*$ a turning point in the second quadrant and $+c^*$ a turning point in the fourth quadrant.

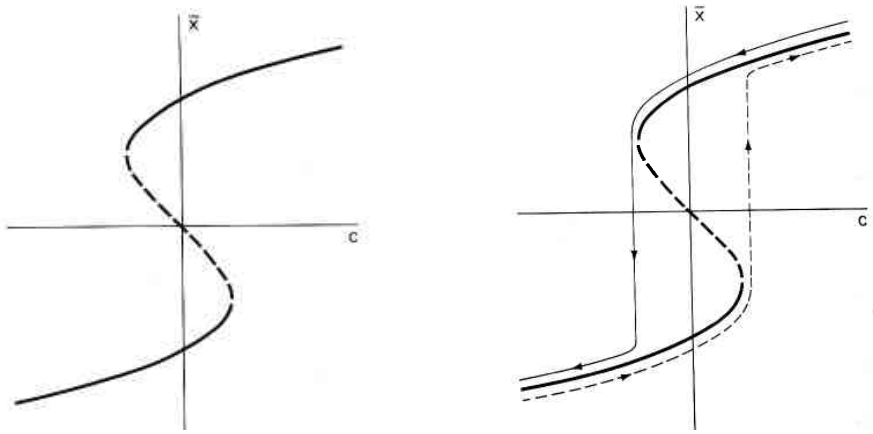

Figure 3.4: The figure on the left is the bifurcation diagram for equation (3.7) showing the steady state solutions as a function of c; the right side figure is the corresponding hysteresis loop. From [1].

To analyze how solutions to equation 3.7 behave, start by choosing $c < -c^*$ and a starting value x_0. After a long time, the system will be close to the equilibrium point on the stable solution branch corresponding to c, and as the value of c is increased up to $c = +c^*$, the system will stay close to the lower stable solution branch until $c = +c^*$. However, once $c > +c^*$, the system will seek the stable equilibrium states on the top solution branch, and will jump up to and stay close to the upper solution branch as c increases.

However, when x is started near an equilibrium state with $c > +c^*$, the system will stay close to the upper stable solution branch as c is decreased as long as $c > -c^*$. However, once $c < -c^*$, the system will immediately move down to the lower stable solution branch. This is behavior is called a hysteresis loop [1] and is depicted in the left picture of figure 3.4. Hysteresis-like behavior has been observed in numerical simulations for the current-voltage curves for standard two-barrier RTDs [56, 57, 12, 14], and hysteresis loops have been seen experimentally for RTDs as well [6].

3.3 Numerical Methods

Since the Wigner-Poisson equations are too complex to be solved analytically, numerical methods are used to approximate the solution. The domain is discretized and numerical approximations are used to replace the derivative and integral terms, which are described in detail in chapter 4. Thus, the Wigner and Poisson equations can be written as a system of equations based on function values at each discretized grid point and solved via computer simulations, with the accuracy of the approximate solution determined by the particular formulas used.

Once the system of equations is determined, additional numerical techniques are used to ensure the solution is not only numerically accurate but also yields a physically realistic solution within an acceptable amount of run time.

3.3.1 Newton's Method

To solve a nonlinear equation $W(x) = 0$, Newton's method can be used to find a root x^* [58, 59, 60] (provided one exists) by solving the iterative equation

$$x_{i+1} = x_i - W'(x_i)^{-1}W(x_i) \tag{3.8}$$

where $W'(x_i)$ is the Jacobian of W at x_i. However, the initial iterate x_0 must be close enough to the final solution x^* in order to assure convergence of the set $\{x_i\}$ to x^* [59, 60]. When Newton's method converges, it has a quadratic convergence rate [60], which is defined by

Definition 3.2. *Let $\{x_i\} \in \mathcal{R}^N$ and $x^* \in \mathcal{R}^N$. Then $x_i \to x^*$ q-quadratically if $x_i \to x^*$ and there exists a $C > 0$ such that*

$$||x_{i+1} - x^*|| \leq C||x_i - x^*||^2 \tag{3.9}$$

The requirements for convergence of Newton's method are stated in theorem 3.4 (from [60]):

Theorem 3.4. *Let $W : \Omega \to \mathcal{R}^N$, where $\Omega \in \mathcal{R}^N$. Assume*

1. *$W(x) = 0$ has a solution x^**
2. *$W' : \Omega \to \mathcal{R}^{N \times N}$ is Lipschitz continuous, and*
3. *$W'(x^*)$ is nonsingular.*

Then there is $\delta > 0$ such that if $||x^* - x_0|| < \delta$, the Newton iteration defined by equation 3.8 converges q-quadratically to x^*.

For the Wigner equation, the Jacobian is dense, which makes computing $W'(x_i)^{-1}$ computationally burdensome, especially as the grids are refined and the size of the solution vector increases. Thus, we use an inexact Newton method [61, 60, 59] to compute the x_{i+1}, which redefines the equation to be solved as

$$||W'(x_i)s_i + W(x_i)|| \leq \eta_i ||W(x_i)|| \tag{3.10}$$

where η_i is a forcing term that controls the size of the relative residual, and s is the step, which can be computed using an iterative linear solver. Convergence rates for inexact Newton methods are not as fast as Newton's Method, and are linear at worst and superlinear at best [60], as defined below:

Definition 3.3. *Let $\{x_i\} \in \mathcal{R}^N$ and $x^* \in \mathcal{R}^N$. Then*

1. *$x_i \to x^*$ q-linearly if there exists $C \in (0, 1)$ such that*

$$||x_{i+1} - x^*|| \leq C||x_i - x^*|| \tag{3.11}$$

   for i large.

2. *$x_i \to x^*$ q-superlinearly if*

$$\lim_{n \to \infty} \frac{||x_{i+1} - x^*||}{||x_i - x^*||} \to 0 \tag{3.12}$$

The convergence requirements for inexact-Newton methods require the same assumptions as for Newton's Method, but require constraints on the forcing terms η_i (from [60]):

Theorem 3.5. *Let $W : \Omega \to \mathcal{R}^N$, where $\Omega \in \mathcal{R}^N$. Assume*

1. *$W(x) = 0$ has a solution x^**
2. *$W' : \Omega \to \mathcal{R}^{N \times N}$ is Lipschitz continuous, and*
3. *$W'(x^*)$ is nonsingular.*

Then there exist $\delta > 0$ and $\bar{\eta}$ such that if $||x^* - x_0|| < \delta$ and $\{\eta_i\} \in [0, \bar{\eta}]$, then the set $\{x_i\}$ with $x_{i+1} = x_i + s_i$ where s_i satisfies the inexact Newton equation 3.10 converges q-linearly to x^*, and q-superlinearly if $\eta_i \to 0$.

3.3.2 GMRES

The GMRES (Generalized Minimum Residual) method [62], a Krylov subspace method, is chosen as the linear solver for the inexact Newton method. Krylov subspace methods solve the linear equation $Ax = b$ by minimizing the error at each iteration i over the subspace $\mathcal{K}_i$, defined as

$$\mathcal{K}_i = \text{span}(r_0, Ar_0, \ldots, A^{i-1}r_0) \tag{3.13}$$

where r_0 denotes the residual $r_0 = b - Ax_0$ [60, 59]. GMRES chooses the x_i so that the Euclidean norm of the residual, $||b - Ax||_2$, is minimized at each step i for $x_i \in x_0 + \mathcal{K}_i$. Krylov methods do not require the use of an iteration matrix [60], which decreases memory requirements and allows for finer grids to be used in RTD simulation runs.

One of the important properties of GMRES is that it will find the exact solution x^* and terminate in at most N iterations, where N is the size of the solution space[60, 58]. However, if N is excessively large, storage of the Krylov subspace can become burdensome, so an alternative

method named GMRES(m) is based on restarting the GMRES computations after m steps, with the initial vector for the restart being the last computed vector, x_m [60, 58].

3.3.3 Preconditioning

Convergence of the solution can be accelerated by preconditioning the equation, which involves introducing a new matrix M whose inverse approximates that of the matrix A and allows $M^{-1}A$ to be relatively well-conditioned [63]. There are two types of preconditioning: left preconditioning and right preconditioning, where left preconditioning applies the matrix M^{-1} to the left side of the standard linear equation $Ax = b$ and solves

$$M^{-1}Ax = M^{-1}b. \tag{3.14}$$

The termination criteria for GMRES changes since we are now minimizing

$$||M^{-1}b - M^{-1}Ax||_2 \tag{3.15}$$

over $x \in x_0 + \mathcal{K}_i$ [60, 2]. The C++ Wigner-Poisson implementation applies left preconditioning. Right preconditioning solves

$$AM^{-1}y = b \quad \text{with } y = Mx \tag{3.16}$$

which leaves the termination criteria for GMRES the same as with the original system [60, 2].

3.3.4 Continuation

To compute the RTD current-voltage curve, an initial current is computed at $v = 0$, and then the voltage is increased up to a terminal value via a continuation method [64, 65, 48]. Continuation

methods solve a nonlinear equation of the form

$$W(x, v) = 0 \tag{3.17}$$

where x is an N-dimensional vector and v is the parameter to be changed along a solution path. Two main types of continuation methods are natural continuation and pseudo arclength continuation [65, 48]. Natural continuation methods find a steady state solution x_0 to equation 3.17 at an initial value of the parameter v_0, and then increment the parameter v successively and resolve equation 3.17 until either (1) the final parameter value v_{final} is reached, or (2) a steady state solution cannot be found for the next value of the parameter (i.e., x_i exists for v_i, but a solution cannot be found at $v_i + v_{min}$, where v_{min} is the smallest desired parameter increment) [48].

For natural continuation, the initial iterate x^0_{i+1} used in solving equation 3.17 at a parameter value v_{i+1} can be chosen in numerous ways. The simplest way is to choose $x^0_{i+1} = x_i^{final}$ at v_i [48, 65], but other predictor methods [65] may be used to make approximations of the trajectory of the solution path. One such method is a tangent predictor, which incorporates the solution to

$$W'(x_i, v_i)\frac{\partial x}{\partial v} = -\frac{\partial W}{\partial v} \tag{3.18}$$

where $\frac{\partial W}{\partial v}$ is computed using a forward difference formula

$$\frac{\partial W}{\partial v} = \frac{W(x_i, v_i + \delta) - W(x_i, v_i)}{\delta} \tag{3.19}$$

so that the initial iterate at v_{i+1} becomes $x^0_{i+1} = x_i^{final} + (v_{i+1} - v_i)\frac{\partial x}{\partial v}$ [48].

3.3.5 Pseudo arclength continuation

However, when there are turning points in a solution path (such as those described in section 3.2.3), using a natural continuation method will fail close to a turning point because the Jacobian becomes singular [65, 48]. Previous work with the Wigner-Poisson formulation [12, 11, 14, 2, 43] has demonstrated hysteresis-like behavior in the IV curve, and thus pseudo arclength continuation must be used to choose the voltage steps and solve the Wigner equation.

Pseudo arclength continuation augments the system of equations by including an arclength parameter equation, which prevents the Jacobian from becoming singular in the area of a turning point [65]. Thus the parameter that determines the continuation increments becomes the arclength step s rather than the voltage parameter v. The new system becomes

$$\begin{aligned} \bar{W}(x(s), v(s)) &= 0 \\ \mu(x(s), v(s), s) &= 0 \end{aligned} \tag{3.20}$$

where $\mu(x(s), v(s), s)$ is the arclength equation, defined by

$$\mu(x(s), v(s), s) = (x - x_i)\frac{\partial x_i}{\partial s} + (v - v_i)\frac{\partial v_i}{\partial s} - \Delta s = 0. \tag{3.21}$$

Equation 3.21 represents the equation of the plane that is perpendicular to the tangent line through (x_i, v_i) at a distance Δs from (x_i, v_i), and will intersection the solution path provided Δs is reasonably sized and the curvature of the path is not too severe [65]. So in place of equation 3.8 used by Newton's method, the new iterative equation is

$$\begin{bmatrix} \frac{\partial \bar{W}}{\partial x} & \frac{\partial \bar{W}}{\partial v} \\ \left(\frac{\partial x}{\partial s}\right)^T & \frac{\partial v}{\partial s} \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta v \end{bmatrix} = - \begin{bmatrix} \bar{W} \\ \mu \end{bmatrix}. \tag{3.22}$$

Note that natural continuation only required the initial iterate x_{i+1}^0 to be chosen, whereas

pseudo arclength requires an initial guess for both x^0_{i+1} and v^0_{i+1}, which are chosen based on the value of the parameter step s [65, 48].

To find an initial guess for x^0_{i+1} and v^0_{i+1}, the bordering algorithm [65, 48] solves two interim equations

$$W'(x,v)\cdot a = -W(x,v) \tag{3.23}$$

$$W'(x,v)\cdot b = -\frac{\partial W(x,v)}{\partial v} \tag{3.24}$$

to yield the updated initial values

$$\Delta v = -\left(\mu + \frac{\partial x}{\partial s}\cdot a\right) / \left(\frac{\partial v}{\partial s} + \frac{\partial x}{\partial s}\cdot b\right) \tag{3.25}$$

$$v_{i+1} = v_i + \Delta v \tag{3.26}$$

$$x_{i+1} = x_i + a + \Delta v \cdot b \tag{3.27}$$

Once these have been chosen, equations 3.20 are resolved until (1) the final parameter value v_{final} is reached or surpassed (in which case, a final solve at v_{final} is computed), or (2) a steady state solution cannot be found for (x_{i+1}, v_{i+1}) using the smallest arclength step s_{min} allowed [48].

Chapter 4

Discretization of the Wigner-Poisson formulation

To numerically solve the Wigner-Poisson model, the domain is discretized and finite difference methods are applied to the Wigner and Poisson equations, as has been done in previous work [9, 12, 11, 15, 16, 17, 39, 13, 40, 2, 43].

4.1 Domain discretization

First, the momentum domain is truncated from $(-\infty, \infty)$ to $(-K_{max}, K_{max})$ with K_{max} chosen such that for $|k| > K_{max}$, $f(x, k, t) \approx 0$ for all x, t. Earlier work tied the value of K_{max} to how the spatial domain was discretized [40, 8]; however, later work has used $K_{max} = 0.25$ Å^{-1} as a best approximation [2, 43]. Unless otherwise stated, for the simulations discussed in this dissertation, the fixed value of $K_{max} = 0.25$ will be used. The spatial and momentum domains are then discretized into either a uniform or nonuniform mesh, with Nx denoting the number of grid points used to compute a uniform spatial mesh and Nk the number of grid points for a uniform momentum mesh.

To determine how to discretize the domain, we look at a sample Wigner distribution function (see figure 4.1). Note that for a large number of grid points in the domain, especially for $|k| \geq 0.15\mathring{A}^{-1}$, the Wigner function $f \approx 0$. Since we are interested in how f behaves away from $f \approx 0$, a large percentage of the calculations (that involve grid points in this range) used to compute the solution to the Wigner equation provides no meaningful information.

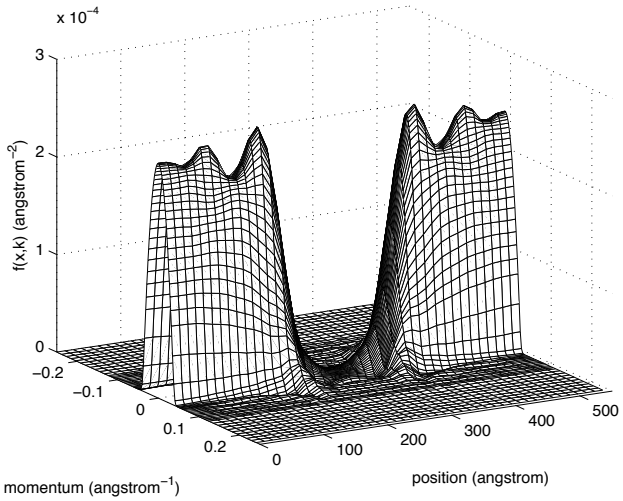

Figure 4.1: Zero bias Wigner distribution for a 550 Å device.

Therefore, while uniform grids have been used in a variety of previous work [12, 11, 15, 16, 17, 39, 13, 40, 2], implementing appropriately chosen nonuniform grids has been shown to reduce simulation run times [43]. Thus for this work, the domain is broken into several regions symmetric about the $k = 0$ line, and nonuniform grids are implemented in both the spatial and momentum dimensions. To ensure the nonuniform grid does not compromise the quality of the simulation results, the grid points are concentrated near the zero momentum axis where there is more variability in the Wigner function, and the density of the grid points is reduced from

one region to the next as one moves further from the $k = 0$ line.

In the interior region around the zero momentum axis, the grid points are chosen to avoid using $k = 0$ where there would be a singularity. This forces Nk to be even based on the quadrature rules used to discretize the integrals in the Wigner equation. In this innermost region, the mesh spacing is that for a uniform mesh; i.e., $\Delta k = \frac{2K_{max}}{Nk}$. For each successive region, the grid points are thinned out in the momentum domain by doubling the size of the momentum mesh width, so that $\Delta k(\text{exterior region}) = 2 \times \Delta k(\text{interior region})$.

For the spatial grid, we use the endpoints as grid points, which allows the number of mesh points to be odd based on quadrature rules. The spatial mesh in the innermost region around $k = 0$ is defined as $\Delta x = \frac{L}{Nx-1}$ (i.e., equivalent to a uniform grid), and the grid points in successive regions away from the $k = 0$ line are again thinned out by powers of 2, so that $\Delta x(\text{exterior region}) = 2 \times \Delta x(\text{interior region})$.

The type of mesh used is illustrated in Figure 4.2.

4.2 Discretization of the Equations

To discretize the Wigner and Poisson equations, we use a variety of finite difference methods. Note that for the methods described, Δx and Δk represent the sizes of the spatial and momentum meshes respectively, and they may have different values in different regions of the non-uniform grid, as described in the previous section.

The kinetic term, $K(f)$, is approximated using an upwinding scheme due to the one-sided nature of the boundary conditions for the Wigner function. The fourth order upwinding approximation for the first derivative [43] is:

$$K(f_{ij}) \approx \begin{cases} -\frac{hk_j}{2\pi m^*}\left(\frac{25f_{ij}\ -48f_{i-1,j}\ +36f_{i-2,j}\ -16f_{i-3,j}\ +3f_{i-4,j}}{12\Delta x}\right) & , \quad k_j > 0 \\ -\frac{hk_j}{2\pi m^*}\left(\frac{-25f_{ij}\ +48f_{i+1,j}\ -36f_{i+2,j}\ +16f_{i+3,j}\ -3f_{i+4,j}}{12\Delta x}\right) & , \quad k_j < 0. \end{cases} \tag{4.1}$$

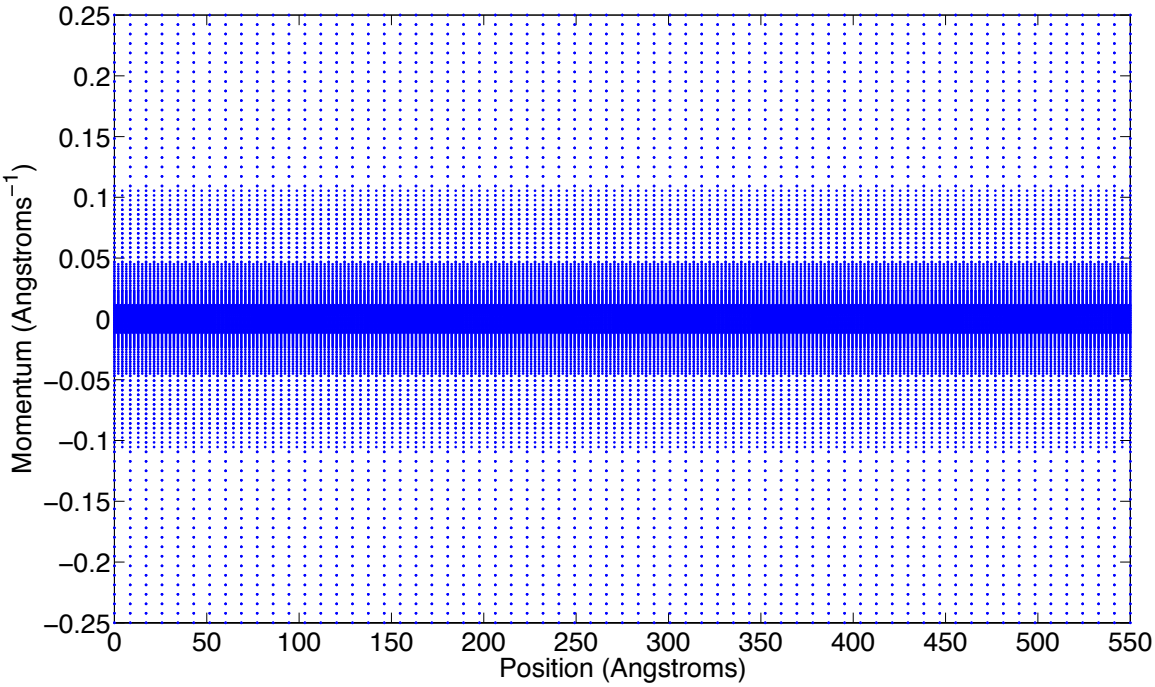

Figure 4.2: Example of nonuniform grid.

where Δx depends on the size of the spatial mesh at each value of k. However, close to the boundaries at $x = 0$ and $x = L$, decreasing order upwinding approximations are used due to the lack of available grid points. We believe this approximation to be acceptable since there is very little variation in the Wigner solution near the boundaries. The lower order schemes near the boundaries are, for $k < 0$:

$$K(f_{Nx-3,j}) = -\frac{hk_j}{2\pi m^*}\left(\frac{2f_{Nx,j} - 9f_{Nx-1,j} + 18f_{Nx-2,j} - 11f_{Nx-3,j}}{6\Delta x}\right) \tag{4.2}$$

$$K(f_{Nx-2,j}) = -\frac{hk_j}{2\pi m^*}\left(\frac{-f_{Nx,j} + 4f_{Nx-1,j} - 3f_{Nx-2,j}}{2\Delta x}\right) \tag{4.3}$$

$$K(f_{Nx-1,j}) = -\frac{hk_j}{2\pi m^*}\left(\frac{f_{Nx,j} - f_{Nx-1,j}}{\Delta x}\right) \tag{4.4}$$

and for $k > 0$:

$$K(f_{2,j}) = -\frac{hk_j}{2\pi m^*}\left(\frac{f_{2,j} - f_{1,j}}{\Delta x}\right) \tag{4.5}$$

$$K(f_{3,j}) = -\frac{hk_j}{2\pi m^*}\left(\frac{3f_{3,j} - 4f_{2,j} + f_{1,j}}{2\Delta x}\right) \tag{4.6}$$

$$K(f_{4,j}) = -\frac{hk_j}{2\pi m^*}\left(\frac{11f_{4,j} - 18f_{3,j} + 9f_{2,j} - 2f_{1,j}}{6\Delta x}\right) \tag{4.7}$$

The potential $P(f)$ term is discretized using Newton-Cotes quadrature rules [63, 64, 58]:

$$P(f_{ij}) \approx -\frac{4}{h}\sum_{j'=1}^{N_k} f_{ij'}T(x_i, k_j - k_{j'})w_{j'} \tag{4.8}$$

where the $w_{j'}$ are the weights associated with the quadrature rule in each region. In the interior

region about $k = 0$, the composite midpoint rule is used, where

$$w_{j'} = \begin{cases} \Delta k & \text{for j}' = 2, 3, ..., m_r - 1, \\ \frac{\Delta k}{2} & \text{for j}' = 1, m_r \end{cases} \tag{4.9}$$

where Δk represents the smallest momentum mesh size (equivalent to a uniform grid), m_r is the number of grid points in the mesh region, and m_r is even. Away from the $k = 0$ line, however, the grid points in the region(s) can be chosen so that a fourth order composite Simpson's rule can be implemented:

$$w_{j'} = \begin{cases} \frac{4\Delta k}{3} & \text{for j}' = 2, 4, ..., m_r - 1, \\ \frac{2\Delta k}{3} & \text{for j}' = 3, 5, ..., m_r - 2, \\ \frac{\Delta k}{3} & \text{for j}' = 1, m_r \end{cases} \tag{4.10}$$

where Δk represents the size of the momentum mesh in each region, m_r is again the number of grid points in the region, and in this case, m_r is odd.

To compute $T(x_i, k_j - k_{j'})$, the integral can be split into two pieces and the solutions to each piece summed together:

$$T^a(x_i, k_j - k_{j'}) = \int_0^{\frac{L_c}{2}} [\Delta_c(x_i + y) - \Delta_c(x_i - y)] \sin(2x_i(k_j - k_{j'}))dy \tag{4.11}$$

and

$$T^b(x_i, k_j - k_{j'}) = \int_0^{\frac{L_c}{2}} [u_p(x + y) - u_p(x - y)] \sin(2x(k_j - k_{j'}))dy \tag{4.12}$$

Since the energy band function $\Delta_c(x) = 0$ for a large portion of the x domain, the integral in equation (4.11) can be computed analytically, which decreases computation time and increases the accuracy of the solution. For equation (4.12), the $T^b(x_i, k_j - k_{j'})$ term is computed by dividing the integral into two pieces, one from 0 to x_{N_c} (where N_c is the closest odd numbered grid point just below L_c using a uniform grid) and another from x_{N_c} to L_c. Composite Simpson's rule can be used to compute the integral from $[0, x_{N_c}]$, but computation of the integral from

$[x_{N_c}, L_c]$ is more complicated since L_c may not correspond to a grid point. Thus, modifications are made to the weights between $N_c - 1$ and $N_c + 2$ using fourth order Taylor expansions to compute the integral from x_{N_c} to L_c. So equation (4.12) is approximated as

$$T^b(x_i, k_j - k_{j'}) \approx \sum_{i'=1}^{N_c+1} [u_p(x_i + x_{i'}) - u_p(x_i - x_{i'})] \sin(2x_{i'}(k_j - k_{j'}))w_{i'} \tag{4.13}$$

where the $w_{i'}$ are the modified composite Simpson's rule weights

$$w_{i'} = \begin{cases} \frac{\Delta x}{3} & \text{for i}' = 1 \\ \frac{4\Delta x}{3} & \text{for i}' = 2, 4, ..., N_c - 2 \\ \frac{2\Delta x}{3} & \text{for i}' = 3, 5, ..., N_c - 3 \\ \frac{2\Delta x}{3} + W_1 & \text{for i}' = N_c - 1 \\ \frac{4\Delta x}{3} + W_2 & \text{for i}' = N_c \\ \frac{\Delta x}{3} + W_3 & \text{for i}' = N_c + 1 \\ W_4 & \text{for i}' = N_c + 2. \end{cases} \tag{4.14}$$

with Δx equal to that for a uniform grid, and W_l, $l = 1, 2, 3, 4$ the additional fourth order weighting terms for the approximation to the integral over $[x_{N_c}, L_c]$.

The scattering term $S(f)$ can be discretized using composite Simpson's rule as

$$S(f_{ij}) \approx \frac{1}{\tau} \left[\frac{f_0(x_i, k_j)}{\sum_{j'=1}^{N_k} f_0(x_i, k_{j'}) w_{j'}} \sum_{j'=1}^{N_k} f_{ij'} w_{j'} - f_{ij} \right] \tag{4.15}$$

where the $w_{j'}$ are the standard weights listed in equations (4.9) and (4.10) for the $P(f)$ term.

To solve Poisson's equation (equation 2.54), we split it into two pieces,

$$\frac{\partial^2 u_p^a(x,t)}{\partial x^2} = \frac{q^2}{\epsilon} N_d(x) \tag{4.16}$$

and

$$\frac{\partial^2 u_p^b(x,t)}{\partial x^2} = -\frac{q^2}{\epsilon} n(x,t), \tag{4.17}$$

solve equation (4.16) analytically and equation (4.17) using a finite difference approximation, and then sum the two solutions so that

$$u_p(x) = u_p^a(x) + u_p^b(x). \tag{4.18}$$

This method not only saves computational time but produces a more accurate solution.

To solve equation 4.16 analytically, we note that for the standard two barrier symmetric device,

$$N_d(x) = \begin{cases} Nd & : \quad x < x_d \ \text{ or } \ x > L - x_d \\ 0 & : \quad \text{otherwise} \end{cases} \tag{4.19}$$

where Nd is the doping density on each end of the device, and x_d is the length of the doping region. Setting the boundary conditions for equation 4.16 as

$$u_p^a(0) = 0 \qquad \text{and} \qquad u_p^a(L) = 0, \tag{4.20}$$

we can compute this portion of the electrostatic potential $u_p(x)$ as

$$u_p^a(x) = Nd \times \begin{cases} \frac{x^2}{2} - x_d x & : \quad x < x_d \\ -\frac{x_d^2}{2} & : \quad x_d \le x \le L - x_d \\ \frac{(L-x)^2}{2} - x_d \cdot (L - x) & : \quad L - x_d < x \end{cases} \tag{4.21}$$

To solve equation (4.17), a fourth order center difference formula is used:

$$\frac{-u_p^b(x_{i-2}) + 16u_p^b(x_{i-1}) - 30u_p^b(x_i) + 16u_p^b(x_{i+1}) - u_p^b(x_{i+2})}{12(\Delta x)^2} = -\frac{q^2}{\epsilon} n(x_i) \tag{4.22}$$

where Δx is equal to that for a uniform grid and $3 \leq i \leq Nx - 2$. At the end points, one-sided fourth order methods are used with $u_p^b(x_1) = 0$ and $u_p^b(x_n) = -V$. The electron density $n(x)$ is approximated using composite Simpson's rule:

$$n(x_i) \approx \frac{1}{2\pi} \sum_{j=1}^{N_k} f_{ij} w_j \tag{4.23}$$

where the weights are as listed in equations (4.9) and (4.10).

Finally, the current density $j(x)$ can be approximated using composite Simpson's rule:

$$j(x_i) \approx \frac{h}{2\pi m^*} \sum_{j=1}^{N_k} k_j f_{ij} w_j \tag{4.24}$$

where the weights are again the standard Simpson's rule weights from equations (4.9) and (4.10).

Chapter 5

C++ Implementation

5.1 Object Oriented Programming

> A programming language serves two related purposes: it provides a vehicle for the programmer to specify actions to be executed, and it provides a set of concepts for the programmer to use when thinking about what can be done. The first purpose ideally requires a language that is "close to the machine" so that all important aspects of a machine are handled simply and efficiently in a way that is reasonably obvious to the programmer. The C language was primarily designed with this in mind. The second purpose ideally requires a language that is "close to the problem to be solved" so that the concepts of a solution can be expressed directly and concisely. The facilities added to C to create C++ were primarily designed with this in mind.
>
> – Bjarne Stroustrup, *The C++ Programming Language* [66]

C++ is a highly desirable language in which to program because it combines the efficiency of the C language with object-oriented programming. Object-oriented programming places its emphasis on the data in a program, rather than on the calculations/manipulations involved, by

allowing programmers to define abstract data types that can be manipulated.

It is for these reasons that Trilinos [51], the main software used by the Wigner-Poisson code, was written in C++: to take advantage of the object-oriented structure that allows abstract data types to be defined and used efficiently.

5.2 Trilinos

The Trilinos software, developed by Sandia National Laboratories, was designed to solve large scale complex science and engineering problems. It utilizes established libraries such as the BLAS and LAPACK as well as more recently developed software that incorporates parallel architectures, such as PETSc (used by Trilinos' ML package for multi-level preconditioning); Metis/ParMetis (used by Zoltan, a dynamic load balancing toolkit); SuperLU (used by Amesos, the direct sparse solver package); and Aztec (used by AztecOO, the Krylov solver package) [51, 67].

The first public release, Trilinos 3.1, was in September 2003 with eight basic packages. The functionalities at that time were a single linear system solver, a non-linear solver, and multigrid preconditioning [51]. Also included was the major advantage of Trilinos - the parallel data structures that easily integrate into a user's code without explicitly using a parallel programming language. It immediately won two major awards, a 2004 R&D 100 Award, given out annually by R&D Magazine to recognize the "100 most technologically significant products introduced in the past year" [68]. It also won one of two HPC Software Challenge Awards as part of the Technical Program at SC2004. (Per the call for submissions: "This HPC Software Challenge will honor participants working to improve the productivity of HPC software developers and the quality of HPC software.")

Since its initial release, the Trilinos software has been (and continues to be) upgraded and expanded numerous times, to include additional capabilities as needs dictated. The version of

Trilinos used in this implementation of the Wigner-Poisson model was Trilinos 10.6.0, which was released in September 2010 and included 48 packages and 13 major capability areas; the most recent release is version 11.0.3, released October 25, 2012, and includes 54 packages.

The main reasons for Trilinos' use in this application are its flexible parallel data structures and numerous highly efficient solver packages. The nonlinear solver package in Trilinos, NOX, provides a Jacobian-Free Newton-Krylov method, which saves memory and computation time by eliminating calculation of the dense Jacobian for the Wigner equation. This is crucial for the fine meshes and longer device lengths which we hope to simulate in this work.

Another reason Trilinos was chosen is its ability to properly handle the hysteretic effects present in the Wigner solution [56, 57]. Trilinos' continuation package, LOCA, provides a pseudo arclength continuation option which is used to compute the Wigner function around the turning points in the solution.

5.3 Trilinos packages

The Trilinos packages used by the C++ Wigner-Poisson code are Teuchos, Epetra, Amesos, AztecOO, NOX and LOCA.

The primary packages incorporated in the Wigner-Poisson model were Teuchos (reference counted pointers), Epetra (data structures), Amesos (direct solvers for sparse matrices), NOX (non-linear solvers), and LOCA (continuation methods). AztecOO functionality (linear Krylov solvers) was also incorporated via NOX.

5.3.1 Teuchos

Teuchos provides a variety of tools to be used by programmers for developing objects within the Trilinos framework [51]. The main use of Teuchos in the C++ Wigner-Poisson code was the reference counted pointer (RCP), which is used to create objects and allocate memory.

RCPs provide automatic memory deallocation when the object associated with an RCP is no longer active, which makes memory usage more efficient. In addition, both NOX and LOCA use Teuchos parameter lists to set up non-linear functions for computation.

5.3.2 Epetra

Epetra is the core linear algebra package upon which several other packages are based, including Amesos, NOX, and LOCA. The Epetra data structures are designed to keep track of how the elements of a particular object are spread across processors, so that parallel computations can be performed efficiently using BLAS and LAPACK routines [51]. These structures are also used by other Trilinos packages to simplify cross-processor communication. Thus, most of the vector and matrix elements used by the C++ model to compute the various elements of the Wigner-Poisson equations were defined using Epetra data structures.

5.3.3 Amesos

The goal of the Amesos package is to enable the user to easily interface a code using Epetra objects with efficient direct solver libraries developed outside Trilinos [69]. Amesos provides a variety of sparse direct solvers, written for both serial and parallel implementations. In the C++ model, Poisson's equation was solved in Amesos using a KLU factorization, since the matrix is sparse and nearly symmetric. In addition, a serial implementation proved more efficient in decreasing run times due to the relatively smaller problem size.

5.3.4 AztecOO

AztecOO is the Krylov solver package used within NOX to calculate the linear system portion of the inexact Newton solve. AztecOO provides several Krylov methods, including two conjugate gradient methods and two GMRES methods [70]. It also provides a matrix-free mechanism that allows users to bypass computing and storing a Jacobian directly. In this case, finite difference

methods are used to approximate the Jacobian [71]. To evaluate the Jacobian-vector product $W'(x)s$ from equation 3.10, a forward differencing method is used

$$W'(x)s = \frac{F(x + \delta s) - F(x)}{\delta} \tag{5.1}$$

where δ is chosen to appropriately scale the step s [60]. The approximation requires an additional Wigner function evaluation at $(x + \delta s)$ but reduces storage requirements from a matrix of size N^2 to a vector of size N.

5.3.5 NOX

NOX is the non-linear solver package used by the C++ code to solve for the initial Wigner distribution $f_0(x, k)$, and it is used by LOCA to solve the non-linear equation at each voltage step. NOX uses Newton's method to solve non-linear equations [71], and provides a variety of choices in the implementation. Globalization techniques such as line search and trust region methods are available, as well as exact and inexact Newton methods [71]. In addition, NOX provides an option to evaluate the Jacobian itself using finite differencing methods, as well as a matrix-free implementation via AztecOO.

5.3.6 LOCA

The continuation package LOCA is built using the non-linear solver package NOX and provides both natural and arc length continuation methods, as well as the ability to locate and track several types of bifurcations [48]. For the Wigner-Poisson model, hysteretic effects are present in the I-V curve [56, 57], so LOCA uses an arc length continuation method to choose the voltage values and compute the corresponding Wigner functions.

5.4 Alglib software

The other software package incorporated into the Wigner-Poisson C++ code was the Alglib package [72], an open-source numerical analysis library that supports several programming languages, including C++. The Alglib interpolation routine is incorporated into the C++ model in order to handle conversions from the non-uniform spatial grid to a uniform spatial grid. It was chosen due to its ease of implementation and ability to be compiled across multiple Unix and Linux platforms [72].

5.5 Data structures in C++ code

A variety of data structures were used within the C++ code. In general, standard C++ arrays were used for a series of elements that were temporary to a particular class; but for arrays of elements that stored information that was important to computation of the Wigner function (such as weights, the barrier and doping profiles, the Wigner function itself, etc.), the Epetra array structure `Epetra_Vector` was used.

`Epetra_Vectors` have a variety of functions that can be applied to them, such as addition, scalar multiplication, dot products, etc., which have been written to take advantage of BLAS and LAPACK routines. In addition, parallel communication is handled automatically within the routines without any user manipulation, which makes the use of `Epetra_Vectors` very advantageous. Examples in the C++ code of the implementation of `Epetra_Vectors`:

- the parallelized Wigner function
- the serial electrostatic potential vector
- the weights used for both the X-convolution and the K-convolution
- the values of the discretized momentum mesh

- the current density vector

`Epetra_Vectors` are built on `Epetra_Maps`, which are indexed arrays of elements. They allow elements in one object that is created with an `Epetra_Map` to be linked with corresponding elements in another object. For example, numerous `Epetra_Maps` were created, including one to parallelize the Wigner vector over x and another to parallelize the Wigner vector over k, since different portions of the Wigner equation require different discretizations. Examples of `Epetra_Maps`:

- the momentum grid points
- the momentum grid points (parallelized across processors)
- the full Wigner function
- the Wigner function (parallelized across processors over k space)
- the Wigner function (parallelized across processors over x space)

Another data structure that is critical to the successful parallelization of the Wigner function is the `Epetra_Import`, which uses `Epetra_Maps` to copy corresponding data from one `Epetra_Vector` to another. This was critical when applying the preconditioner, since for efficient parallel performance, all x values (for a particular k value) must be on the same processor (i.e., parallelized over k); however, the calculations immediately preceding application of the preconditioner are discretized over x due to parallel efficiencies. Thus, the `Epetra_Import` structure allowed the correct element in one discretization to be copied efficiently to the other discretization. Examples of `Epetra_Imports`:

- the transfer of elements from the parallelized electron density function to the full electron density function

- the transfer of elements from the parallelized Wigner function (over x space) to the parallelized Wigner function (over k space)

- the transfer of elements from the parallelized interpolated Wigner function (over k space) to the parallelized interpolated Wigner function (over x space)

To solve the Poisson equation, a matrix class called an `Epetra_CRSMatrix` was used to hold the sparse matrix. The other matrices involved in computation of the Wigner function were dense matrices that required a user-defined type due to the parallel nature in which they were implemented. Thus, a special class was created called a `localMatrixBase` that is based on the standard C++ <vector> class, but is more flexible than existing standard C++ data structures and integrates well with Epetra objects. Examples of this user-defined class are:

- the matrix that holds the values of $\sin(2x_i(k_j - k_{j'}))$, which is pre-computed at the start of the C++ model

- the matrix that holds the values of the portion of the X-convolution: $T(x_i, k_j - k_{j'}) = \int_0^{\frac{L_c}{2}} [\Delta_c(x_i + y) - \Delta_c(x_i - y)] \sin(2x_i(k_j - k_{j'}))dy$, which is pre-computed at the start of the C++ model

- the matrix that holds the values of the other half of the X-convolution: $T(x_i, k_j - k_{j'}) = \int_0^{\frac{L_c}{2}} [u_p(x+y) - u_p(x-y)] \sin(2x(k_j - k_{j'}))dy$, which is computed with each iteration since u_p changes as the Wigner function f changes

And finally, numerous Teuchos parameter lists are used by both NOX and LOCA to store the parameter values to be used (when default values are not adequate). Many Teuchos reference counted pointers (RCPs) are also used, such as the interfaces that set up the computation of the non-linear equation, the matrix-free operator and Jacobian types used by NOX, the status tests to check for convergence, etc.

5.6 C++ Class Descriptions

To use C++ for the Wigner model, numerous classes were created, each representing a different piece of the Wigner-Poisson equations. Most classes have a `compute` function to calculate either the appropriate action on the Wigner function f or to compute terms that will be used later in calculating f. Most of the classes also have a `print` function that prints out the resulting vector or matrix from the `compute` function.

5.6.1 Data and Structure Setup

The first classes that are implemented are those that set up the barrier and doping structures, and one to compute the discretization (and parallelization) of the momentum and spatial meshes. The `Barrier` class adds additional flexibility to the structure of the devices modeled, as does the `Doping` class. Previous versions of the Wigner-Poisson code allowed at most three barrier regions [40] and three doping regions (two on either end of the device that were doped at the same level, and a middle region that was virtually undoped), and forced the energy band profile to be piecewise constant. The upgrades to the device structure are:

- There can be an unlimited number of barrier regions (to improve on the three barrier structures from [40])
- The energy band profile at each barrier can be either constant, linear (having either a negative or positive slop), or quadratic (based on improvements requested [73]).
- There can be an unlimited number of doped regions, with each having different doping levels.

The `Barrier` class creates the barrier profile that is used in the solution of the potential term $P(f)$. The `compute` function calculates the relative energy band profile, $\Delta_c(x)$, at each spatial grid point, which is then used in equation (4.13). The `Doping` class creates the doping

profile that is used in the solution to Poisson's equation. The `compute` function calculates the doping profile $N_d(x)$ at each spatial grid point, which is used in the right hand side of Poisson's equation, equation 4.22.

The `VectorComp` class computes the non-uniform k grid, the corresponding non-uniform x grid, and the quadrature weights for both the integral over x and the integral over k. There are multiple functions within the class: `Xregions` calculates the maximum number of x regions based on the number of (uniform) x grid points; `compute_Nk` calculates new number of (non-uniform) k grid points; `compute_xkpts` calculates the new non-uniform grid; `compute_kvec` calculates the values of the k grid points and the associated weights; and `computeXweights` calculates the values of the weights for the spatial integral (based on a uniform x grid).

5.6.2 Pre-computed Terms

The next set of classes calculate terms that do not depend on the Wigner function f, and thus can be computed and stored for use by other classes. The `compute` function of the `BCMethod` computes the vector of boundary condition values that is used to set up the initial guess for the Wigner distribution function when $V = 0$. This vector is also by the preconditioner class. The vector of boundary conditions is of length Nk since there is only one unique boundary condition value per k grid point, so the `loadBCs` function copies the appropriate boundary condition value to the initial value vector for the Wigner distribution function.

The `SineMatrix` class computes the values of $\sin(2x_i(k_j - k_{j'}))$ for each value of x_i and $k_j - k_{j'}$ via the `compute` function. This is pre-computed, as is the matrix in the `TcMethod` class that stores the values for the integral $T_c(x_i, k_j - k_{j'}) = \int_0^{\frac{L_c}{2}} [\Delta_c(x_i + y) - \Delta_c(x_i - y)] \sin(2x_i(k_j - k_{j'}))dy$. It is calculated via the `compute` function of the `TcMethod` class.

A special class, the `InterpMethod`, is used on rare occasions to interpolate $f0$ (the initial Wigner distribution function) from an existing coarse uniform grid. While this is not used for

short devices (e.g., $L = 550\mathring{A}$), we anticipate it will be necessary to model longer device lengths.

5.6.3 Wigner function terms

The next classes created were those to calculate specific portions of the Wigner distribution function.

The `kinetic4Method` class computes the action of the kinetic operator on the Wigner vector, which is done via the `compute` function. The `kinetic5Method` class applies the preconditioner to the sum of the three terms in the Wigner equation (equation 2.49) also via a `compute` function.

There are numerous pieces that make up the computation for the Wigner Potential term, and the classes that are involved are:

- `FIntMethod`
- `Poisson1Method`
- `TpMethod`
- `potentialMethod`

The `FIntMethod` calculates the integral $\int_{-K_{max}}^{K_{max}} f(x,k)dk$. This class is used to compute the electron density $n(x)$, and the two integrals involved in the Scattering term (one involving the integral of f, and the other involving the integral of $f0$, as per equation 2.57).

The `poisson1Method` class has an `operator` function to create the sparse matrix associated with the finite difference approximations in equation (4.22). The associated `compute` function solves the linear system $Ax = b$ where A is the sparse matrix and b is the right hand side term that includes the doping profile $N_d(x)$ and the electron density $n(x)$.

The `compute` function in the `TpMethod` class calculates the values of the $T(x, k-k')$ integral involving the electrostatic potential $u_p(x)$, and finally, the `potentialMethod` class has a `compute`

function to add all the components of the $T(x, k - k')$ integral together and calculate the k integral via equation (4.8) for a given f.

The final two computational classes are `InterpFMethod` and `CurrentMethod`. The `InterpFMethod` class uses the Alglib software in the `compute` function to calculate the interpolated Wigner function f at every uniform grid x value for a given k value. While all of the `compute` functions in the classes mentioned above (starting with `kinetic4Method`) are executed multiple times to find the Wigner function for a particular value of the voltage V, the `CurrentMethod` class is called only once, when the Wigner function f has converged at a particular value of the bias voltage.

5.7 Solution Process using Trilinos

To solve the system in Trilinos, two additional classes are created to set up and solve the Wigner equation as in equation 2.49, the `f0Problem` and `wpProblem` classes. Both have a `computeF` function that computes each element of the Wigner equation, feeds it into the next piece, and finally applies the preconditioner to calculate the residual, which will be used by NOX. For the `f0Problem` class, the `printSolution` function outputs the initial Wigner distribution function $f0$ once it has converged via NOX; for the `wpProblem` class, the `printSolution` function calls the `CurrentMethod` class to calculate and output the current. The output files from `f0Problem` and `wpProblem` are `f0.dat` and `iv.dat`, respectively.

The next classes that are mandated by NOX are the `ProblemInterface` classes required as part of the NOX framework. Thus, the `f0ProblemInterface` and `wpProblemInterface` classes are created to call the `computeF` functions in each of the corresponding `Problem` classes. In addition to their own `computeF` functions, each `ProblemInterface` class has a `printSolution` function which calls the `printSolution` function in the corresponding `Problem` class; and finally, the `wpProblemInterface` class has a `setParameters` function which manages the voltage

parameter V that is fed into the `wpProblem` class. See appendix C for more details.

See figure 5.1 for a flowchart of the process.

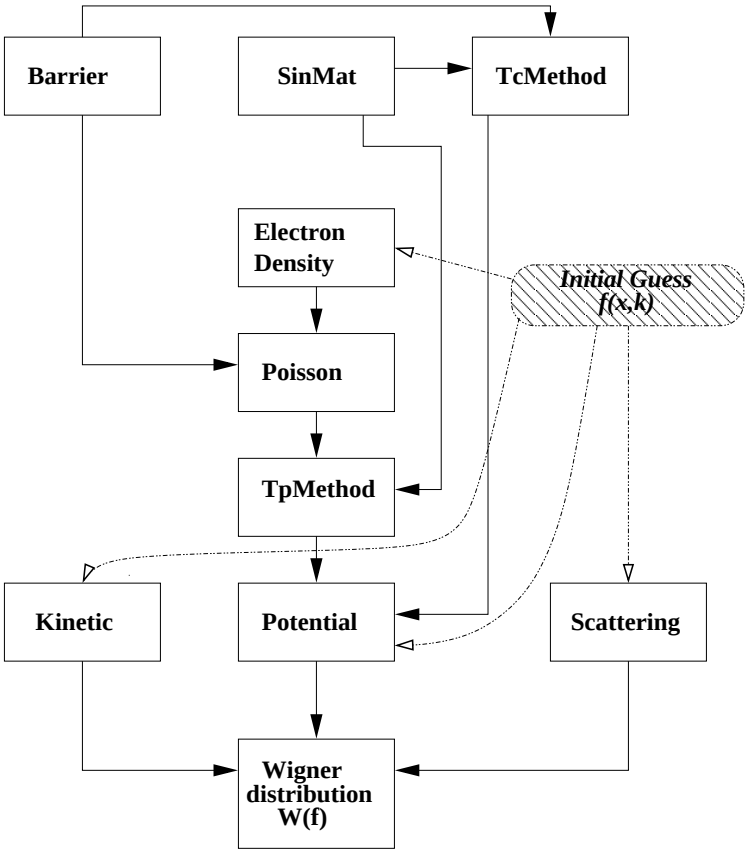

Figure 5.1: Flowchart of the `computeF` function.

5.8 Parallelizing the Wigner function vector

For a non-uniform mesh in both space and momentum, the number of x grid points corresponding to each value of k (and visa versa) will change depending on the value of k (or x). Figure 5.2

demonstrates this for $Nk = 256$, $Nx = 257$, with each blue dot representing a grid point in the mesh.

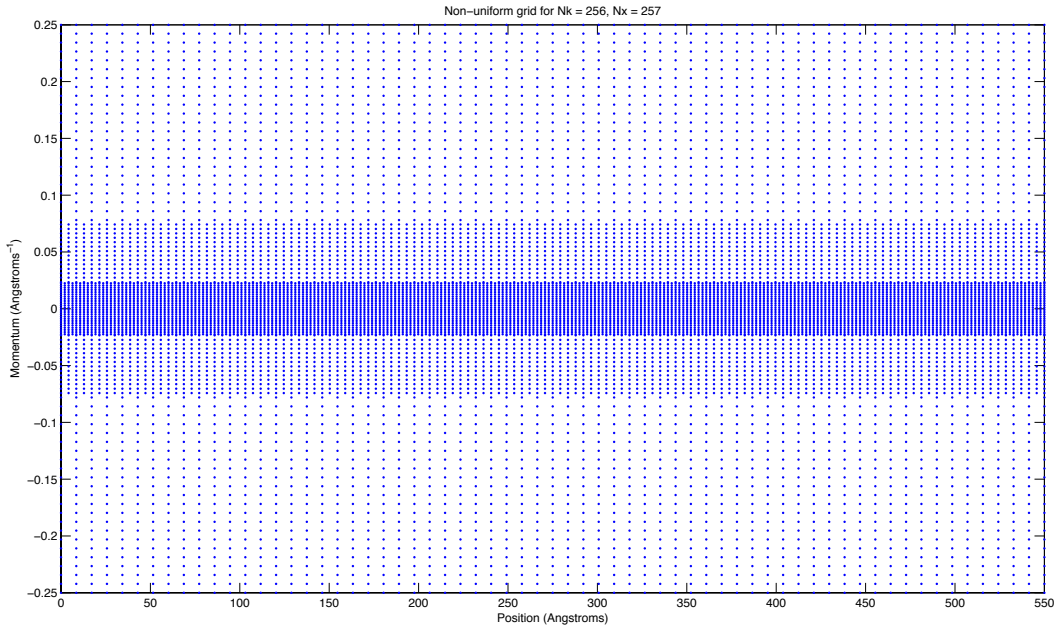

Figure 5.2: Sample non-uniform mesh using $Nk = 256$, $Nx = 257$.

Thus, to properly parallelize the Wigner vector, care must be taken to balance the number of calculations (i.e., the number of grid points) assigned to each processor. To calculate the kinetic and potential terms, the Wigner vector is parallelized across values of k, and for the preconditioner, it is parallelized across x. (Some of the internal portions of the potential term, such as the integral term over space, are also parallelized across x.) Thus, parallel allocations are necessary across both dimensions.

To partition f efficiently, first an average number of total grid points per processor is calculated by dividing the total number of grid points by the number of processors. This average

is then used to partition the Wigner function over the k and x grids.

To partition the Wigner function over the momentum space, the number of spatial grid points in the non-uniform mesh for each k value (starting at the first k value) is summed until the total exceeds the average. Then the number of k grid points (to be assigned to the first processor) is either rounded up or down depending on how close the total number of grid points is to the average. The process is then started over with the second processor, until all grid points have been assigned to each processor. (A similar method is employed to distribute the Wigner vector over the x grid.) This rounding method, based on the total grid points in the solution domain (rather than just the number of k or x grid points), ensures that the number of calculations will be as balanced as possible across processors.

And finally, since different portions of the Wigner equation require different parallelizations of the Wigner function f, conversions from one parallel form of f to another are required. However, the speed of cross-processor communication can be quite slow compared to calculation speeds, so conversions from one grid parallelization to another are minimized when computing $W(f) = 0$. In addition, the distributed Wigner vector f over k is used by NOX and LOCA to compute each successive f iterate, which makes the Wigner-Poisson C++ code more computationally efficient and scalable than previous versions.

Chapter 6

Results using the C++ Code

To test the performance of the C++ code, a variety of simulation runs were performed to test the accuracy, speed, and scalability of the new model. In addition, numerous studies have been done to investigate the impact of different mesh sizes on the steady state IV curve, the Wigner distribution function, and time dependent current-time plots. These simulation runs were performed on a 240 core cluster at the NCSU High Performance Computing lab. The cluster is composed of 30 dual-processor Intel Xeon E5520 blades, with 4 cores per processor, a clock speed of 2.27 GHz, QPI speed of 5.86 GT/s, max memory bandwidth of 25.6 GB/s, 8.2 GB of cache, and 144 GB total memory. Infiniband switches are used to connect the blades, and GNU/Linux version 2.6.18-164.el5 is used along with Intel compilers (version 10.1.022) for optimum performance. Trilinos version 10.6.0 was incorporated into the C++ code along with Alglib version 3.6.0.

6.1 Comparison to Previous Models

To ensure the accuracy of the C++ model, the steady state I-V curves for the FORTRAN and MATLAB models were compared against the C++ model to determine how closely aligned the C++ results are to the previous versions. The agreement between the three sets of results is

very close, especially between MATLAB and C++, both of which use fourth order numerical methods (whereas the FORTRAN version uses second order methods). This can be seen in figure 6.1.

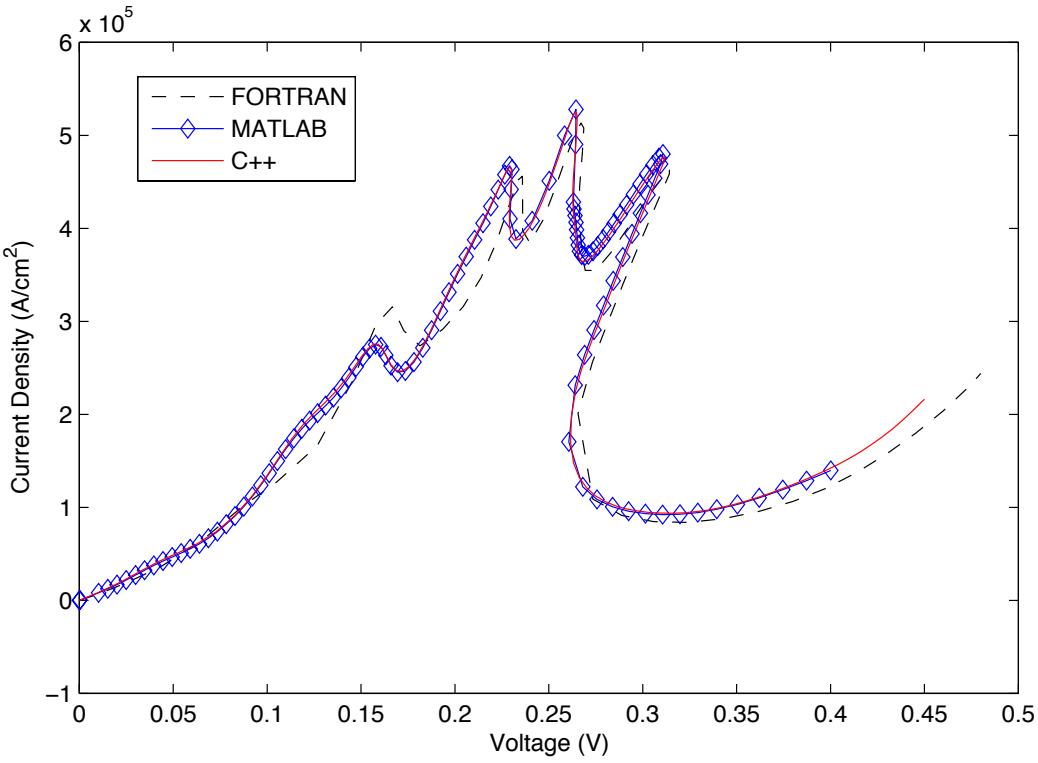

Figure 6.1: Comparison of I-V curves using the FORTRAN, MATLAB, and C++ models.

The grid used in the FORTRAN code in figure 6.1 is a uniform grid in both space and momentum with 2,048 momentum grid points ($Nk = 2048$) and 513 spatial grid points ($Nx = 513$). The C++ and the MATLAB codes use non-uniform meshes that represent the uniform grid with $Nk = 2048$ and $Nx = 513$. In the tables and figures throughout this chapter and the next, only the uniform grid numbers will be given, but the meshes used for the C++

Table 6.1: Comparison of run times for the FORTRAN vs. non-uniform C++ models

FORTRAN vs C++

No. of Cores	*Grid* Nk	Nx	*Time (hr:min)* FORT	C++	*% Reduction* Time	Grid Pts
20	512	513	17:17	2:52	83.4	90.4

and MATLAB simulations will be those for a corresponding non-uniform mesh.

6.2 Run Time Analysis

Simulation run times for the non-uniform C++ code were compared against those for FORTRAN, MATLAB, and the uniform grid C++ model to ensure that simulation run times were significantly improved. When compared against the parallel FORTRAN model, run times for the C++ model decreased significantly for a similar number of cores due to the incorporation of non-uniform grids (see table 6.1 for an example). The MATLAB code is a highly efficient serial code, and while run times were lower for the non-uniform MATLAB code (on the order of $\approx$ 15 hours) than for the serial non-uniform C++ code ($>$ 50 hours), the ability to increase the number of cores allows the C++ code to return results more quickly by choosing an appropriate number of cores.

Comparing run times for the uniform and non-uniform C++ code shows that although the incorporation of a non-uniform spatial grid requires interpolation for the Wigner vector not required with a uniform grid, the decreased number of computations more than offsets the addition of the interpolation. The reduction in timings shown in table 6.2 for different mesh sizes demonstrate these results are not grid-dependent.

Table 6.2: Comparison of run times for the uniform C++ model vs. the non-uniform C++ model

Uniform vs Non-uniform C++

No. of	*Grid*		*Time (hr:min)*		*% Reduction*	
Cores	Nk	Nx	Unif	NonU	Time	Grid Pts
48	512	513	5:39	1:44	69.3	90.4
24	256	257	1:11	0:22	69.0	85.2

6.3 Parallel performance

To check that the C++ Wigner-Poisson model was handling parallelization properly, numerous simulation runs were performed with identical parameters and differing numbers of cores. Figure 6.2 shows that the number of cores used does not make a difference to the solution curve, and thus the parallel implementation was handled correctly.

A strong scaling study was also performed to determine how well the parallel C++ code performs against the serial version. Scaling is measured by computing the speedup of a code using Amdahl's Law

$$\text{speedup} = \frac{1}{(1-P) + \frac{P}{N}} \tag{6.1}$$

where N is the number of cores (assuming 1 core as a base case) and P is the portion of the program that can be made parallel (equivalently, $1 - P$ is the portion of the code that is serial). Strong scalability fixes the problem size and then increases the number of cores, and compares the run times using Amdahl's Law. Ideal performance keeps the serial portion $(1 - P)$ minimized and maximizes the speedup. Another measure used in judging parallel performance is the efficiency of a code, which is a comparison of the run time for 1 core versus run times for multiple (N) cores:

$$\text{efficiency} = \frac{\text{Run time(1 core)}}{N \times \text{Run time(N cores)}} \times 100\% \tag{6.2}$$

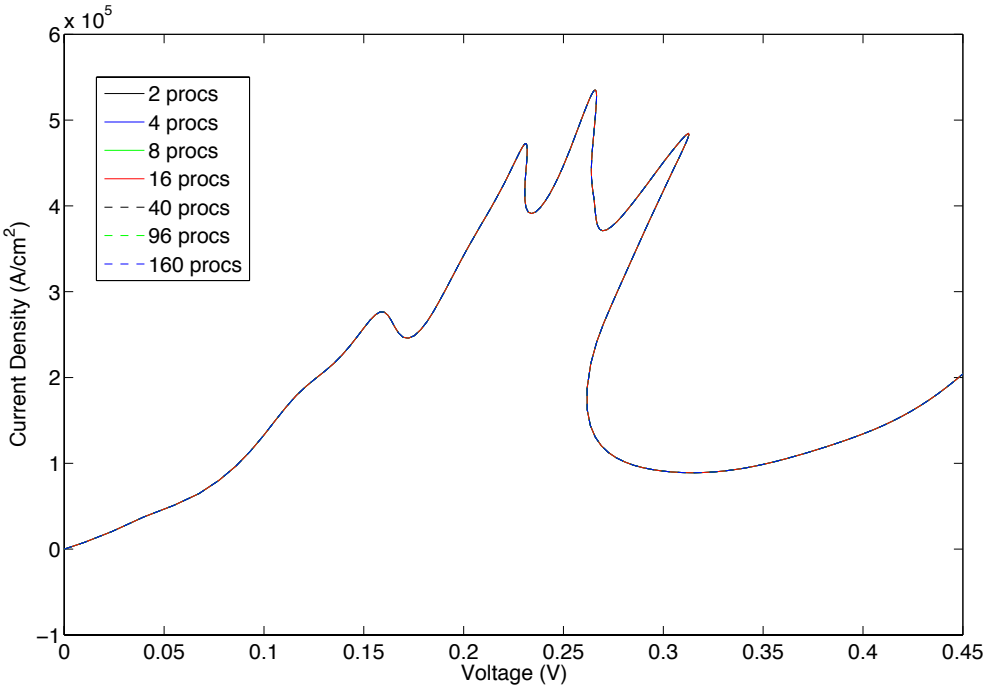

Figure 6.2: Comparison of I-V curves using multiple cores for a $Nk = 2048$, $Nx = 1025$ discretization of the solution space.

and is related to Amdahl's Law by

$$\text{speedup} = N \times \text{efficiency}. \tag{6.3}$$

The study shown in table 6.3 demonstrates the strong scalability of the C++ code for simulation runs using a fine mesh of $Nx = 513$, $Nk = 512$ on the NCSU cluster with the number of cores increased from 2 up to 64. Two cores are used as a base case for these scaling studies to accurately compare the C++ results against the previous parallel FORTRAN results.

The computation time listed for the C++ code is the total time to compute the continuation run from $0.0V$ to $0.45V$, and does not include initialization time or the computation of the initial

Wigner distribution $f0$. The efficiencies calculated from the simulation runs show very good adherence to Amdahl's law, with a maximum speedup of just under 30.

Table 6.3: Strong scaling study using a coarse grid of $Nk = 512$, $Nx = 513$ and variable number of cores

No. of Cores	Run Time (hr:min)	% efficiency	speedup	% serial
2	23:45	100.0	2.00	
4	12:23	95.9	3.84	4.28
6	8:13	96.3	5.78	1.89
8	6:14	95.3	7.62	1.66
10	5:10	91.9	9.19	2.19
12	4:18	92.1	11.05	1.74
16	3:18	90.0	14.39	1.59
20	2:52	82.8	16.57	2.30
24	2:24	92.5	19.79	1.93
32	2:00	74.2	23.75	2.32
40	1:48	66.0	26.39	2.71
48	1:49	54.5	26.15	2.63
64	1:43	43.2	27.67	4.24

Simulation runs were also performed using a slightly finer grid ($Nx = 513$, $Nk = 2048$) with the results listed in table 6.4. Amdahl's law is again adhered to, and we find that the speedup tops out around 55 for the larger problem. However, superlinear speedup is noted when $N \leq 24$ cores, which most likely indicates that the size of the problem exceeds efficient memory capabilities of the blades, and that larger numbers of cores should be used for very fine meshes.

Table 6.4: Strong scaling study using a fine grid of $Nk = 2048$, $Nx = 513$ and variable number of cores

No. of Cores	Run Time (hr:min)	% efficiency	speedup	% serial
2	135:19	100.0	2.00	
4	68:26	98.9	3.95	1.15
6	42:50	105.3	6.32	-2.52
8	33:40	100.5	8.04	-0.16
12	21:12	106.4	12.77	-1.20
16	16:25	103.0	16.49	-0.42
24	11:11	100.8	24.20	-0.08
32	9:03	93.5	29.90	0.47
48	6:38	85.0	40.80	0.77
64	5:55	71.5	45.74	1.29
96	4:59	56.6	54.31	1.63

Table 6.5 shows the parallel performance results for the older FORTRAN code, where the simulation runs were performed on a Linux cluster at Sandia National Laboratories, and calculated the clock time to take 5 continuation steps from $V = 0.2093$ to $V = 0.2293$ [2]. The times listed were for runs performed by Matthew Lasater for his dissertation work [2]; the % efficiency, speedup, and the % serial were calculated using the methods cited above.

Comparing the non-uniform C++ scalability for a fine mesh ($Nk = 2048$, $Nx = 513$) against that for the FORTRAN code (which uses the same $Nk = 2048$, $Nx = 513$ with a uniform grid), it is clear that the C++ code uses parallelization more efficiently than the FORTRAN code, due to the lower percentages of serial code (approximately 2% for C++ vs. approximately 4% for FORTRAN) and higher maximum speedup factors (approx. 30 vs. approximately 55). The reason for the improvement of the C++ code over the FORTRAN code is due to the preconditioner; in the Fortran code, the preconditioner is applied serially after the Wigner equation is solved, whereas the C++ code parallelizes the preconditioner calculation, which improves scalability and decreases run times.

Table 6.5: Parallel performance results using FORTRAN code, data from [2]

No. of Cores	Run Time (hr:min)	% efficiency	speedup	% serial
2	9,120.61	100.0	2.00	
4	4,904.46	93.0	3.72	7.55
8	3,422.43	66.6	5.33	16.70
12	1,925.05	79.0	9.48	5.33
16	1,581.53	72.1	11.53	5.53
24	1,171.00	64.9	15.58	4.92
32	966.06	59.0	18.88	4.63
40	908.92	50.2	20.07	5.23
48	771.91	49.2	23.63	4.48
56	712.25	45.7	25.61	4.39
64	667.62	42.7	27.32	4.33
72	662.24	38.3	27.54	4.61
80	641.39	35.6	28.44	4.65

6.4 Convergence of the IV curve

To determine whether the meshes used to discretize the domain converge to a single solution, numerous simulation runs using increasingly fine mesh spacing were run. From the simulation run results, we determined that the size of the spatial mesh did not change the shape of the IV curve, but the size of the momentum mesh did. As shown in figure 6.3, the IV cures are virtually indistinguishable for differing spatial meshes (when the momentum mesh is held constant), but the IV curve changes shape and converges toward a single curve as the momentum mesh is refined (see figures 6.4 and 6.5.)

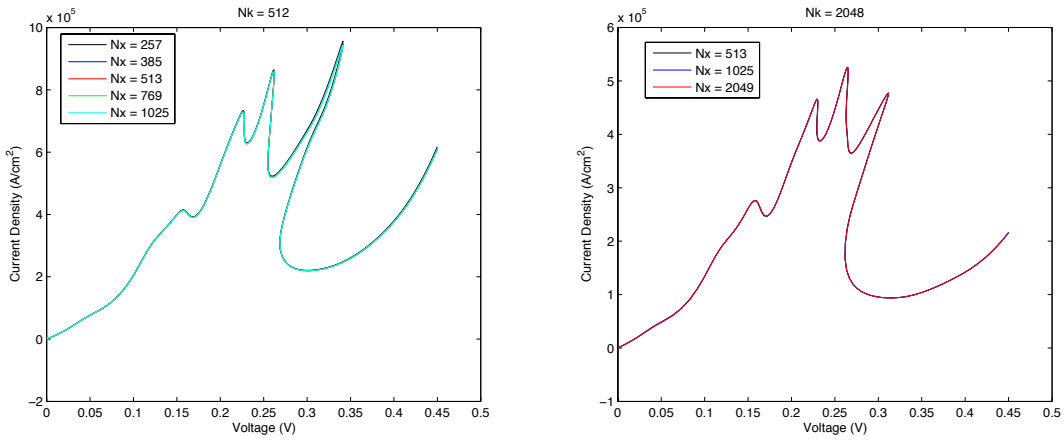

Figure 6.3: The shape of the IV curve does not depend on the spatial mesh, regardless of the size of the momentum mesh. The figure on the left uses a coarser momentum mesh of $Nk = 512$, and the right side figure a very fine momentum mesh of $Nk = 2048$.

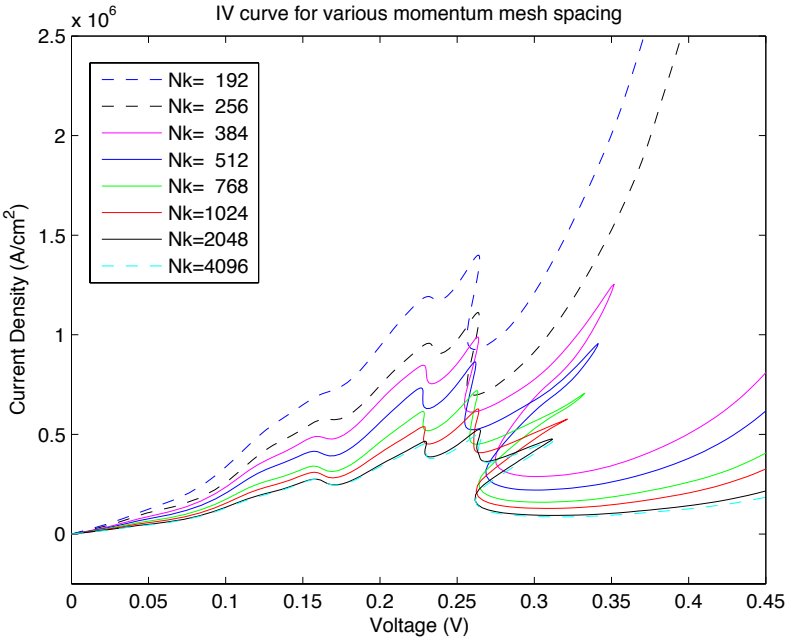

Figure 6.4: The graphs above shows convergence of the IV curve as the number of momentum grid points increases.

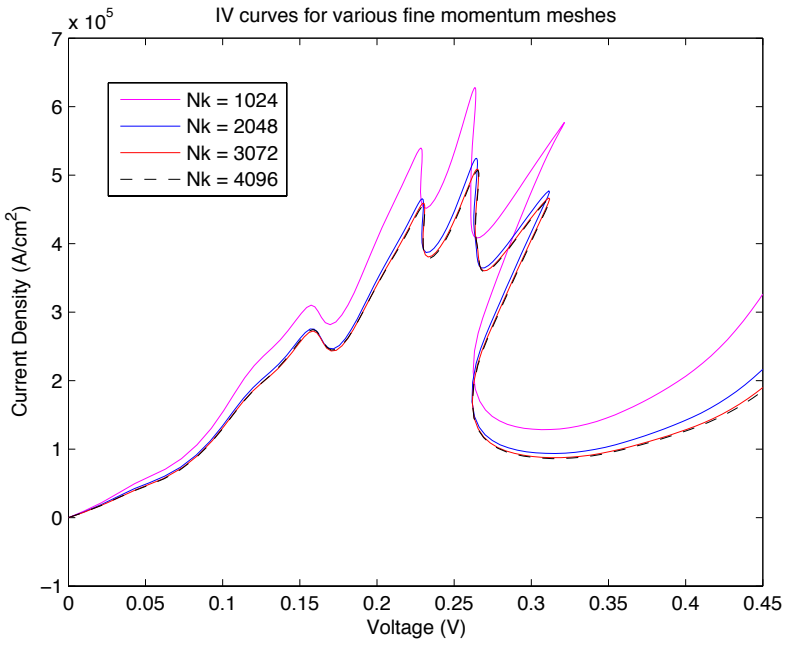

Figure 6.5: A close up of the IV curves for the finest momentum meshes, with the spatial mesh fixed at Nx = 513.

6.5 Wigner function analysis

We also looked at the Wigner function to determine if the mesh sizes had any impact on the shape of the distribution. The Wigner function has a very symmetric shape at $V = 0$ (see figure 6.6), but as soon as voltage is applied, the Wigner function begins to develop "fins" in the center of the distribution (around $k = 0$) that increase in height and change shape as the voltage is increased (and decreased) throughout the continuation process. See figure 6.7 for an example.

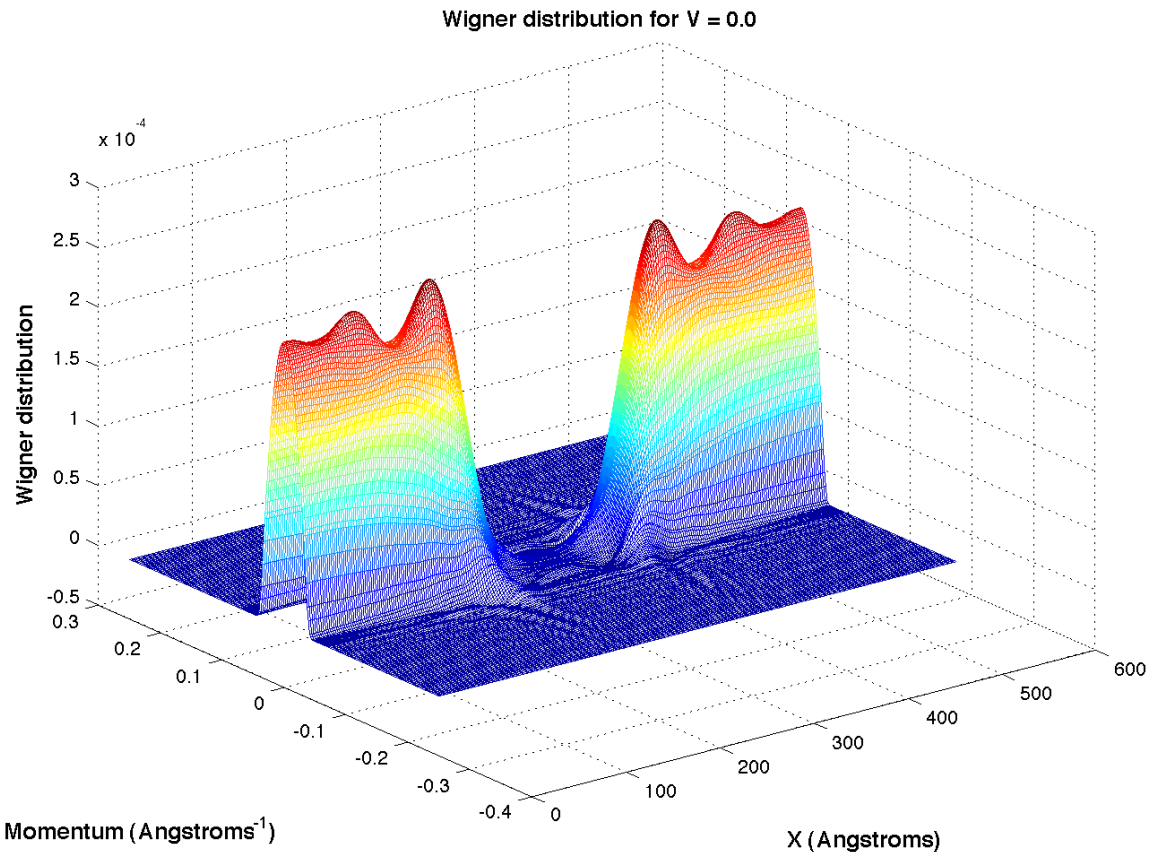

Figure 6.6: The Wigner distribution function at V = 0.

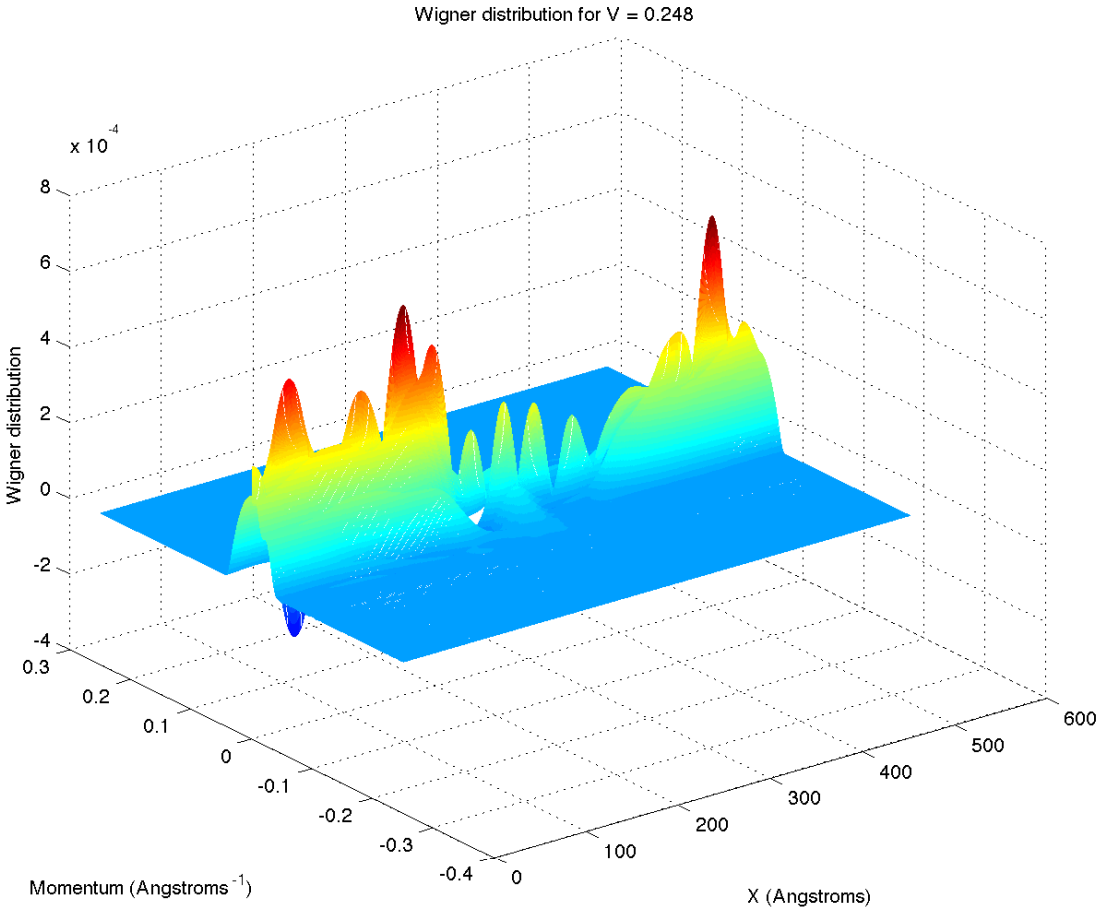

Figure 6.7: The Wigner distribution function at V = 0.248.

To determine whether the size of the meshes changed the shape of the distribution, I captured distribution data for a variety of mesh sizes at $V = 0.248$. We determined that the size of the spatial mesh has no impact on the shape of the Wigner distribution (e.g., the shape or height of the fins in the Wigner function). See figure 6.8.

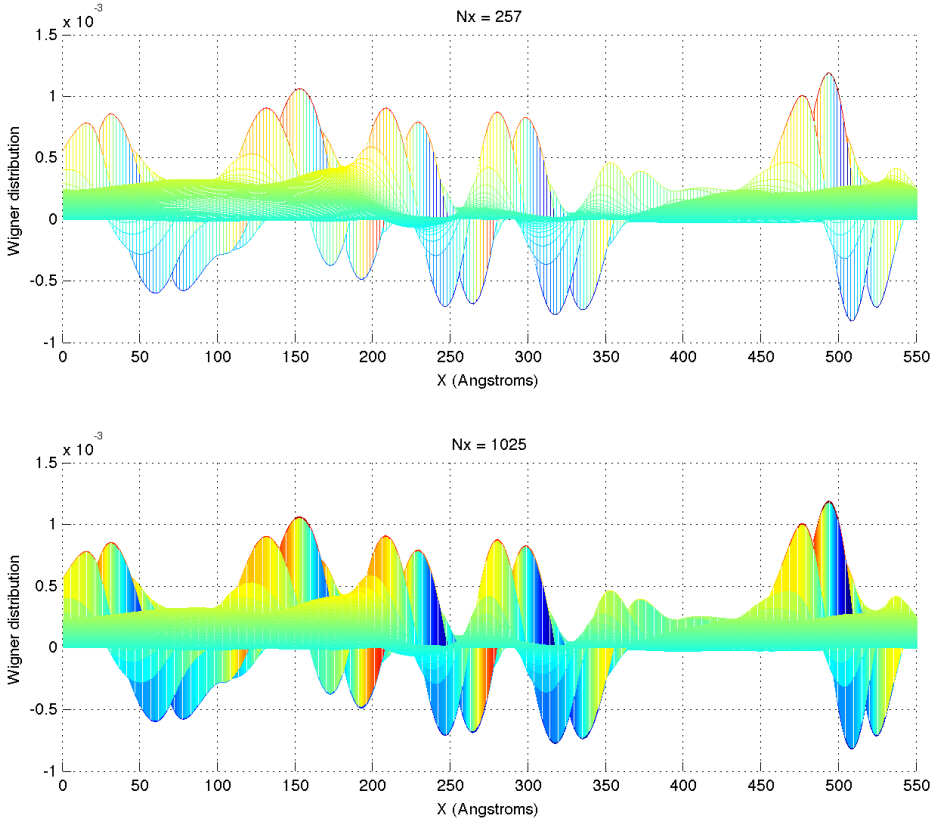

Figure 6.8: Comparison of Wigner distributions at $V = 0.248$ for $Nx = 257$ vs. $Nx = 1025$ (using $Nk = 2048$ for both).

However, the shape of the fins in the Wigner distribution function does change as the size of the momentum mesh is decreased – the fins merge and increase in height. See figure 6.9.

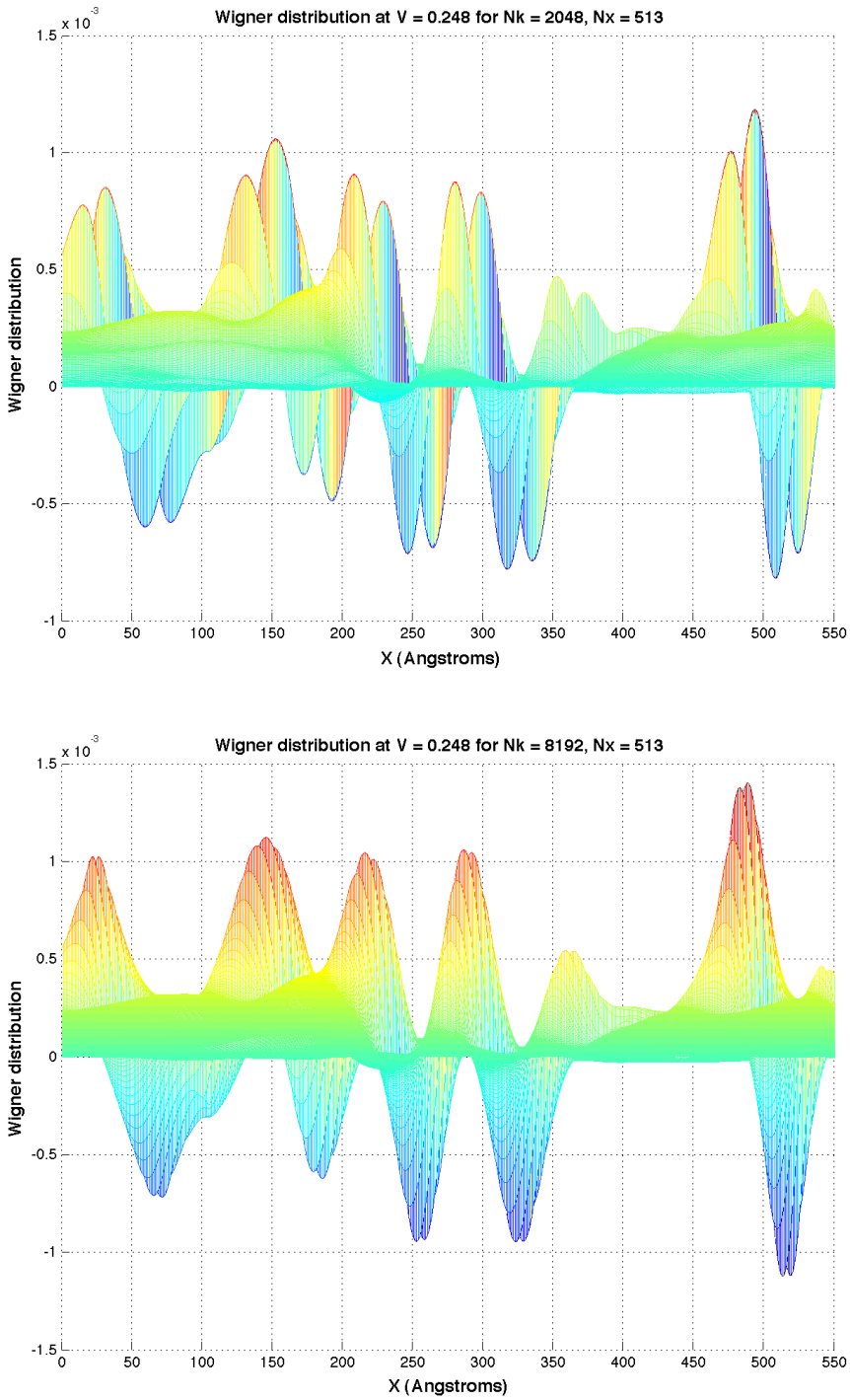

Figure 6.9: Comparison of Wigner distributions at $V = 0.248$ for $Nk = 2048$ vs. $Nk = 8192$ (using $Nx = 513$ for both).

To determine if the fins were just a numerical artifact of the discretization or if they actually had real volume, we studied the Wigner function just outside of the discontinuity at the $k = 0$ line to determine if the Wigner function is converging as the momentum grids are refined. To do this, we interpolated each grid to the next finer grid and calculated the difference using the max norm between the actual to interpolated grids for both (a) the full momentum region and (b) the momentum region excluding a small portion close to the singularity at $k = 0$. Table 6.6 shows the results.

While the differences between the interpolated and actual grids didn't decrease that much for the full momentum space, for the momentum space that excluded a small region around the discontinuity at $k = 0$ ($|k| < 0.0005$), the differences decreased more than quadratically with each grid refinement after the first. The height and volume of the fins can also be seen when the interior region around $k = 0$ ($|k| < 0.0005$) is eliminated in figures 6.10 and 6.11.

Thus we conclude that the Wigner function is converging as $Nk \to \infty$ to a function with "fins" that have actual volume, and are not a result solely of the numerical discretizations.

Table 6.6: Convergence results for the Wigner function at $V = 0.248$

	all k's		$\mathbf{\lvert k \rvert > 0.0005}$	
Coarse → Fine	Difference	D(n-1)/D(n)	Difference	D(n-1)/D(n)
512 → 1024	4.36e-04		4.18e-04	
1024 → 2048	4.36e-04	1.00	2.02e-04	2.07
2048 → 4096	3.55e-04	1.23	2.98e-05	6.77
4096 → 8192	1.46e-04	2.44	2.80e-06	10.63

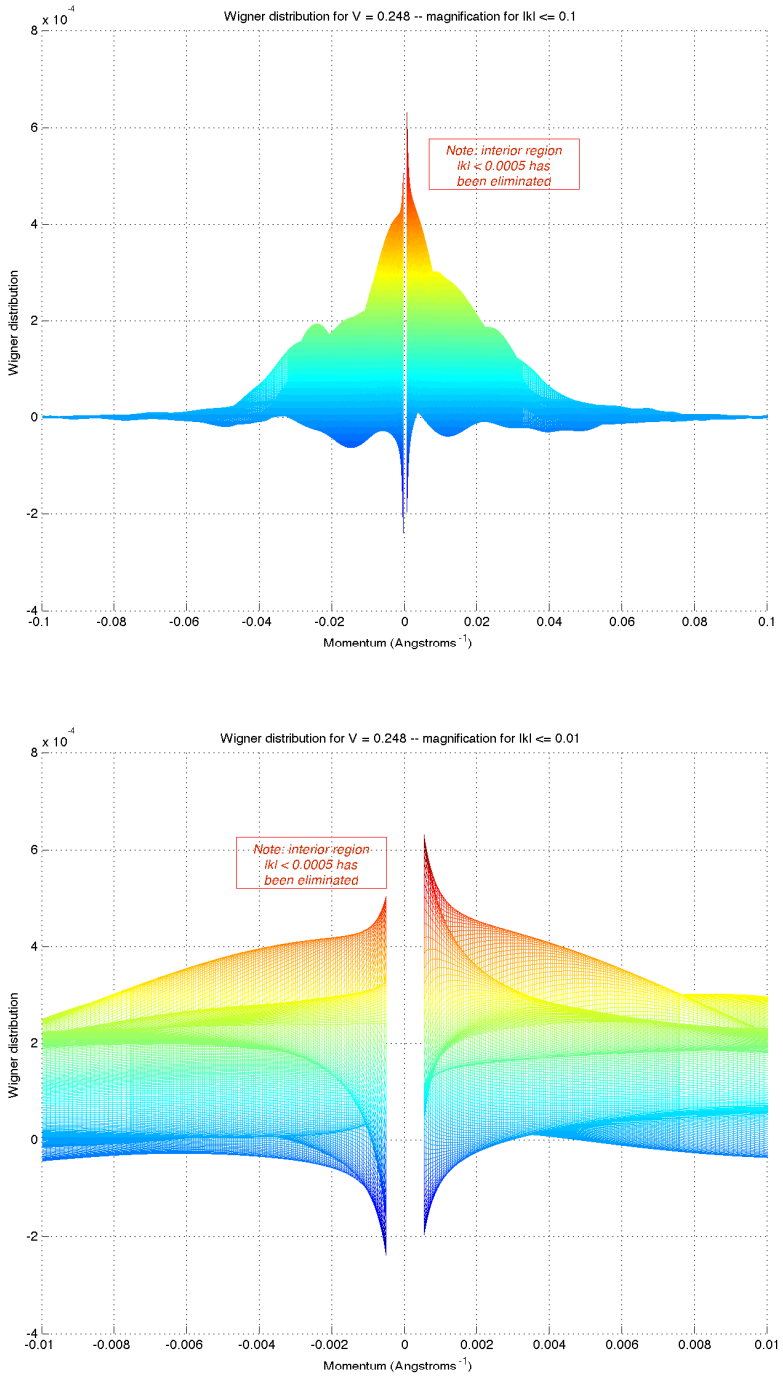

Figure 6.10: Magnifications of the Wigner distribution: the top figure shows the distribution function for $|k| \leq 0.1$, and the bottom figure shows $|k| \leq 0.01$. The interior region $|k| < 0.0005$ has been excluded in both figures.

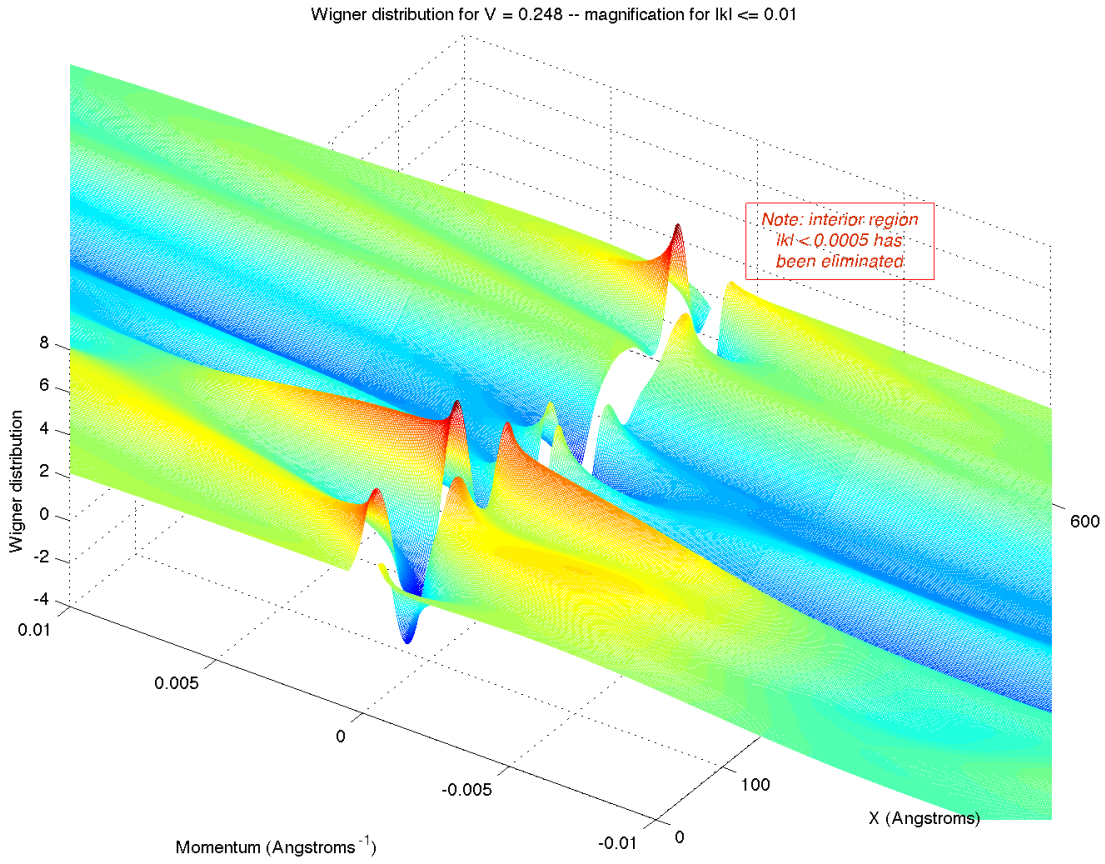

Figure 6.11: Another close up view of the Wigner distribution using Nk = 8192, Nx = 513 at V = 0.248, with the interior region $|k| < 0.0005$ excluded.

6.6 Time dependent Wigner simulations

Previous studies on the Wigner-Poisson formulation found oscillations in the time dependent solutions for very coarse meshes [2, 11], but failed to find similar oscillations as the grids were refined. Since the C++ model is more numerically accurate and faster than the previous models, time dependence was revisited to determine if oscillatory solutions existed under grid refinement.

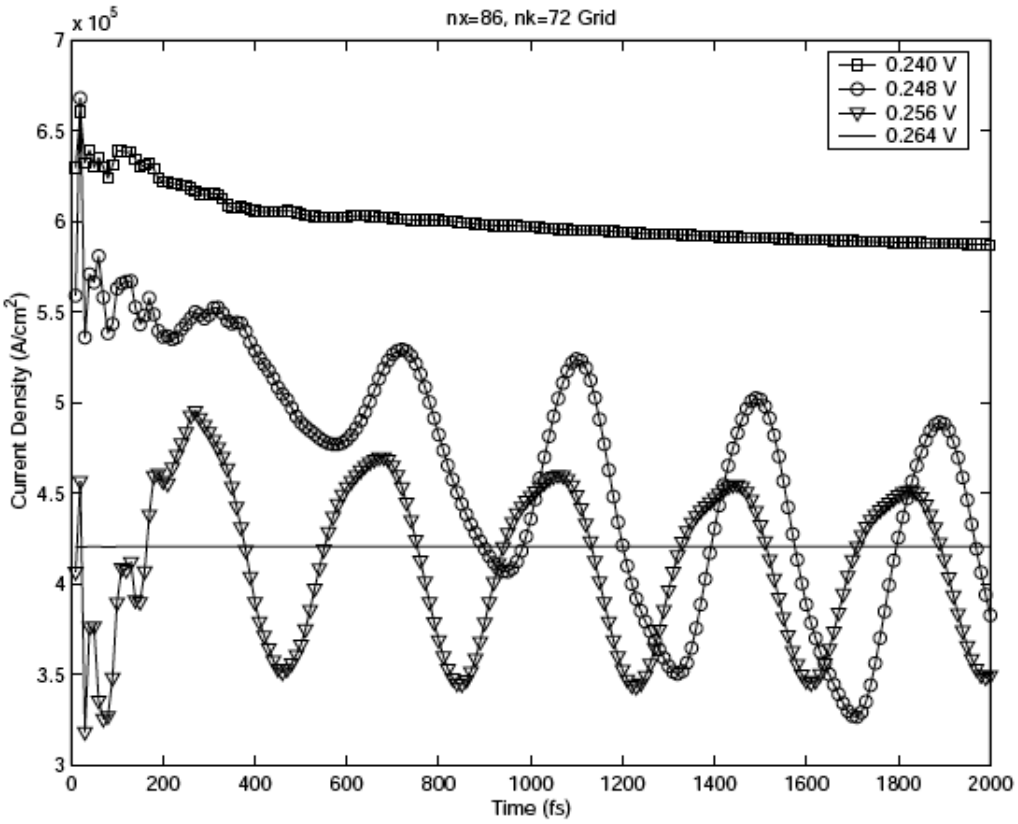

Figure 6.12: Time dependence: Oscillations in the current for certain voltage values using an Nx = 86, Nk = 72 uniform grid and the FORTRAN model. From [2].

6.6.1 Fixed momentum space results

The original goal was to duplicate the results obtained using the original FORTRAN model in 2005 [2] (see figure 6.12), which used a coarse grid of $Nk = 72$ and $Nx = 86$ and a fixed momentum space of $[-0.25, 0.25]$. Originally, I planned to use the time integration package Rythmos (part of the Trilinos software), but was unable to do so since Rythmos does not have a matrix-free implicit implementation (matrix-free is needed due to the denseness of the Jacobian, and an implicit implementation is due to the stiffness of the Wigner-Poisson equations). Thus, we chose to discretize the time derivative $\frac{\partial f}{\partial t}$ using backward difference formulas, which could be implemented by incorporating Trilinos' non-linear solver, NOX. Backward Euler was used to find the first time step, and BDF2 for all others.

It was also necessary to find an appropriately sized time step Δt that would produce a convergent solution whenever the mesh sizes were changed, so to choose Δt efficiently, Von Neumann analysis was used to determine the stability of the discretized Wigner equation. The results of this analysis (to be discussed in section 6.7) showed that the time step for both Backward Euler and BDF2 had to satisfy

$$\Delta t \ \geq \ C \ \frac{\Delta x}{\Delta k} \tag{6.4}$$

where C is a constant determined by the discretization of $\frac{\partial f}{\partial t}$, and Δx and Δk are the smallest mesh sizes for the spatial and momentum dimensions respectively. Thus, a momentum mesh Δk could be chosen first, and then an appropriately sized Δt and Δx determined to produce a convergent solution.

I began the time dependent simulations by using a coarse mesh of $Nk = 256$ and $Nx = 257$, and found oscillations in the current for various voltage values (from $V = 0.248$ to $V = 0.256$, with small amplitude oscillations at $V = 0.264$ that appear to be damping out extremely slowly). See figure 6.13. However, when the grid was refined to $Nk = 512$, $Nx = 513$, the range of voltage values where oscillations appeared shrank so that they were only present at $V = 0.264$. See figure 6.14.

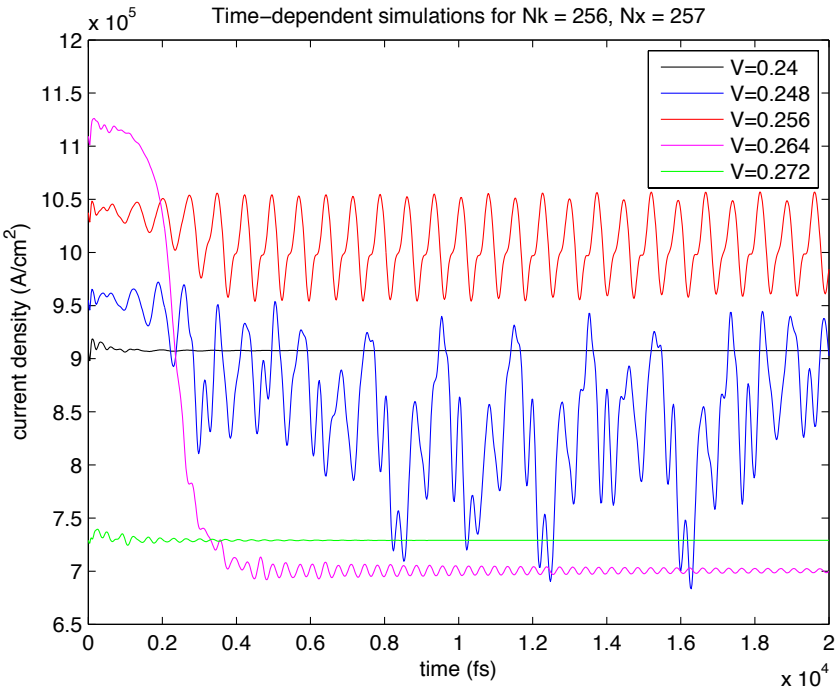

Figure 6.13: Coarse grid oscillations for $Nk = 256$, $Nx = 257$. Oscillations appear at a couple voltage values (V = 0.248 and V = 0.256), with small amplitude damping oscillations at $V = 0.264$. The time step used was $\Delta t = 20fs$.

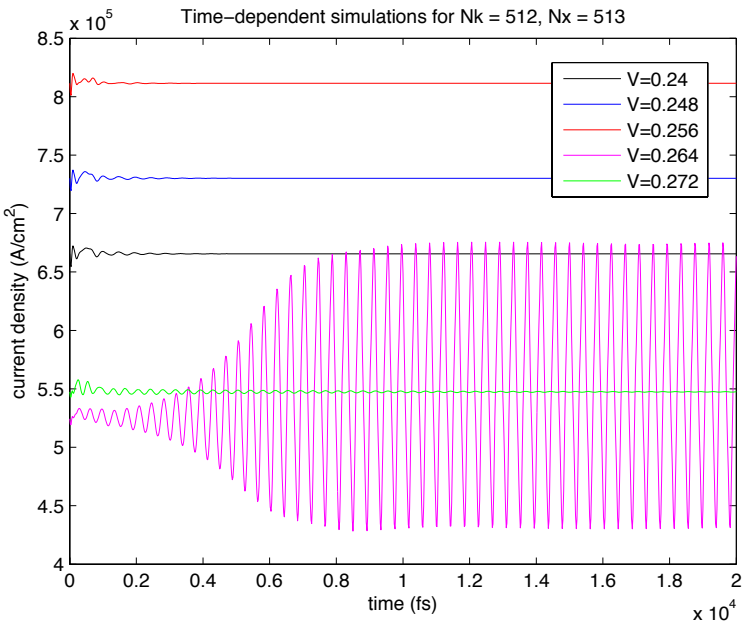

Figure 6.14: Oscillations appear only for one voltage value ($V = 0.264$) for a slightly finer grid using $Nk = 512$, $Nx = 513$.The time step used was $\Delta t = 20fs$.

As the grids are refined even further, evidence of oscillations disappears. Since oscillations appeared only around $V = 0.264$ for the $Nk = 512$, $Nx = 513$ grid, I refined the grid again to $Nk = 1024$, $Nx = 1025$ and focused the simulation runs in the voltage range of $0.264 \leq V \leq 0.276$. However, the only oscillations present damped out after a short period of time. See figure 6.15. To complete the study, I refined the grid one last time to $Nk = 2048$, $Nx = 1025$, and sampled the full range of voltage values, from $0.248 \leq V \leq 0.31$, and found no oscillations at any voltage value, as per figure 6.16. Thus we concluded that the oscillations are present for certain voltage values for coarse grids, but disappear under grid refinement.

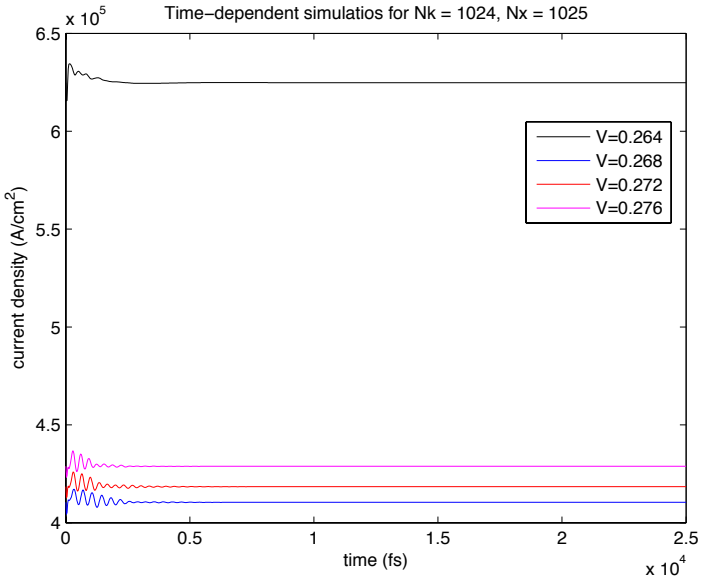

Figure 6.15: Time dependent simulations were focused around the voltage values $0.264 \leq V \leq 0.276$ with a further grid refinement to $Nk = 1024$, $Nx = 1025$, $\Delta t = 20fs$, but no oscillations were found.

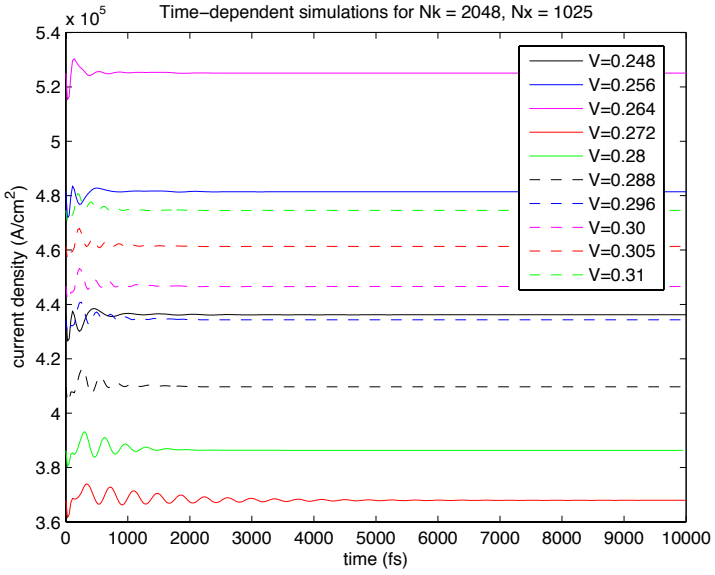

Figure 6.16: A final grid refinement to $Nk = 2048$, $Nx = 1025$ with $\Delta t = 27.5fs$ did not produce oscillations for any sampled voltage value in the full range of $0.248 \leq V \leq 0.31$

6.6.2 Variable momentum space results

It was noticed while reviewing the results of time-dependent simulations that fixing the size of the momentum domain is inconsistent with the definition of the momentum range given in the original papers that described the Wigner-Poisson model [9, 8], so we re-examined the time-dependent problem using the relation

$$\Delta k = \frac{\pi}{Nk\Delta x} \tag{6.5}$$

which forces the momentum range to be

$$k \in [-\frac{\pi(Nx-1)}{2L}, \frac{\pi(Nx-1)}{2L}] = [-L_k, L_k] \tag{6.6}$$

where L is the length of the device and Nx is the total number of grid points used for a uniform mesh. The goal was again to determine whether oscillations existed under grid refinement.

To determine appropriate time steps, we combined the results of the Von Neumann analysis (see section 6.7 with the new momentum range equation to determine new conditions on the mesh sizes. The conditions were:

$$\Delta x \;\rightarrow\; 0 \implies \Delta x = \frac{\pi}{L_k} \rightarrow 0 \implies L_k \rightarrow \infty \tag{6.7}$$

$$\Delta k \;\rightarrow\; 0 \implies \Delta k = \frac{L_k}{Nk} \rightarrow 0 \implies Nk \rightarrow \infty \tag{6.8}$$

$$\Delta t \;\rightarrow\; 0 \implies \Delta t \geq C\frac{\Delta x}{\Delta k} = \frac{C\pi/L_k}{L_k/Nk} = \frac{C\pi Nk}{L_k^2} \rightarrow 0 \tag{6.9}$$

To satisfy all conditions, we choose to increase Nk by a factor of 2, and increase L_k by a factor of 1.5 so each successive grid would be $Nx(fine) = 1.5 \times Nx(coarse)$ and $Nk(fine) = 2 \times Nk(coarse)$.

The results of this investigation were similar to those for the fixed momentum range: for coarse meshes, we found oscillations for a wide range of voltage values (see figures 6.17 and 6.18), but as the grid was refined, the voltage range at which oscillatory solutions were found diminished and shifted from previous voltage ranges (see figures 6.19 and 6.20). However, the corresponding IV curves were not as physically realistic as the ones produced by the fixed momentum range (see the plots in the lower right hand corner of each figure).

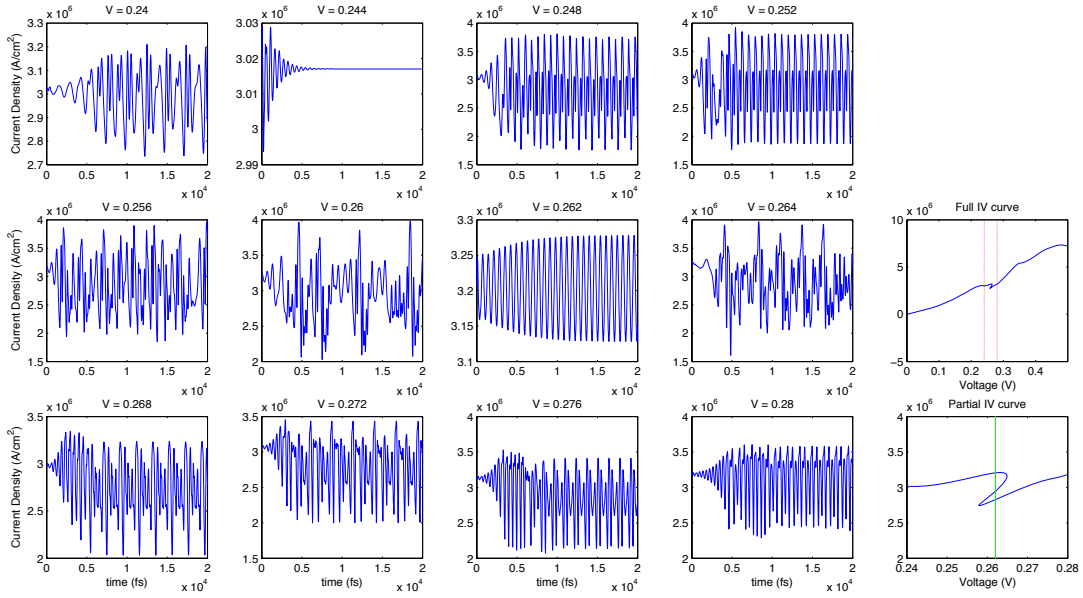

Figure 6.17: Simulation results for variable momentum range, using coarse grids of $Nx = 257,\ Nk = 512,\ k_{max} = 0.75, \Delta t = 20fs$.

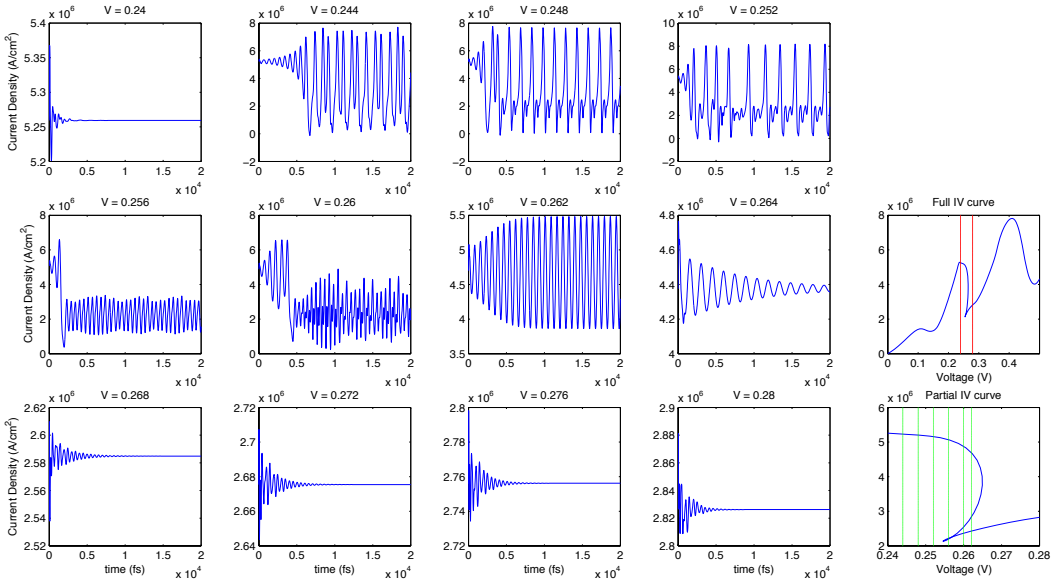

Figure 6.18: Simulation results for variable momentum range, using coarse grids of $Nx = 385,\ Nk = 1024,\ k_{max} = 1.10, \Delta t = 20fs$.

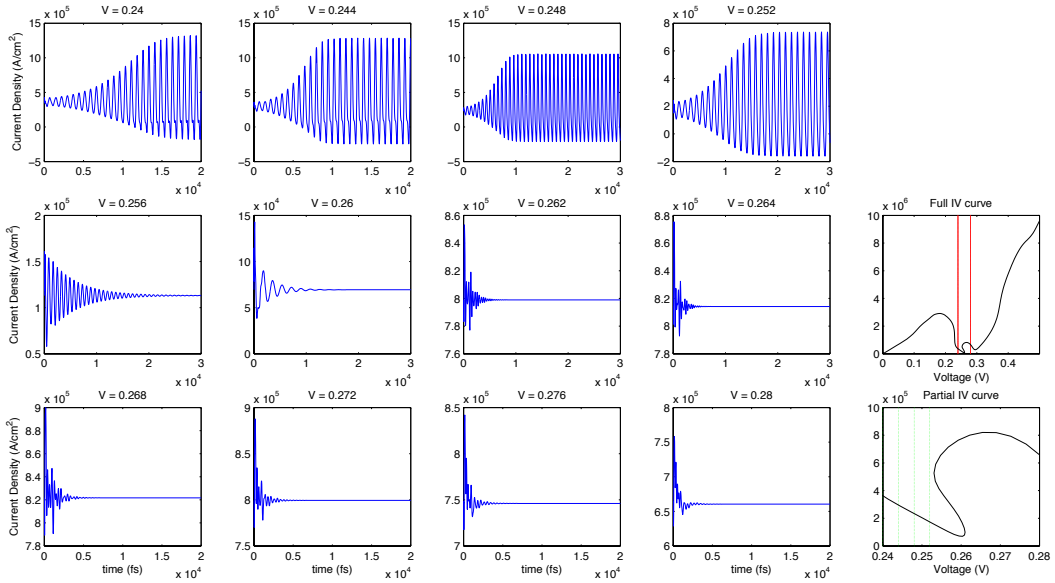

Figure 6.19: Simulation results for variable momentum range, using fine grids of $Nx = 577,\ Nk = 2048,\ k_{max} = 1.65, \Delta t = 20fs$.

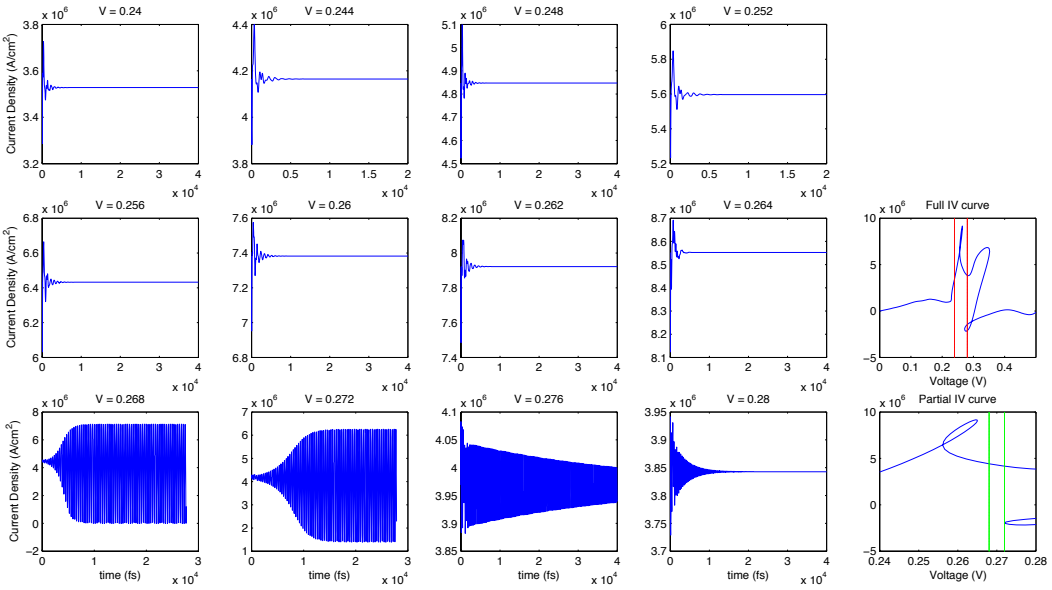

Figure 6.20: Simulation results for variable momentum range, using fine grids of $Nx = 865,\ Nk = 4096,\ k_{max} = 2.50, \Delta t = 20fs$

Since the IV curves produced by the variable momentum space did not look physically realistic, we analyzed numerous grid spacings to determine if we could find a set of meshes for which the IV curves converged. First I held the spatial grid fixed at $Nx = 265$ and allowed the momentum mesh to change, but could not find a set of momentum grid size for which the IV curve converged (see figure 6.21).

Next I fixed the momentum grid and changed the size of the spatial mesh (which changed the size of the momentum domain), but I again found that I could not find a set of meshes for which the IV curve converged (see figure 6.22).

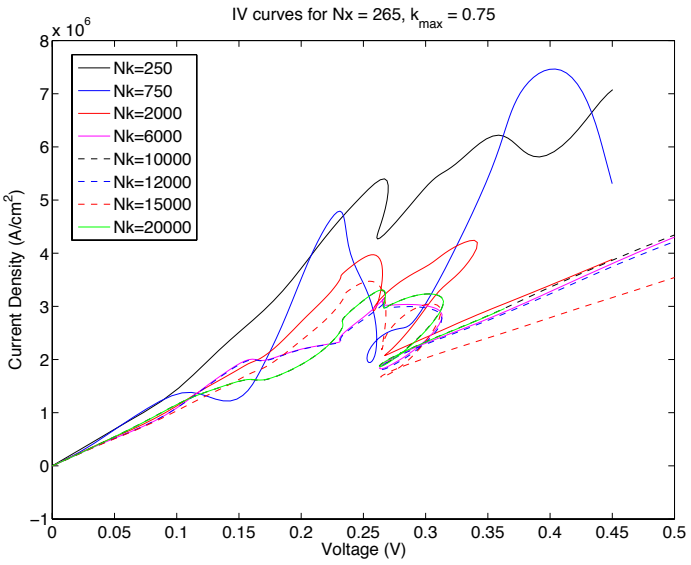

Figure 6.21: IV curves using Nx = 265, Nk = various with a variable momentum domain. The $Nk = 10000$ and $Nk = 20000$ curves are identical, as are $Nk = 6000$ and $Nk = 12000$ (but different from $Nk = 10000$); and $Nk = 15000$ is completely distinct from either of the other sets of curves.

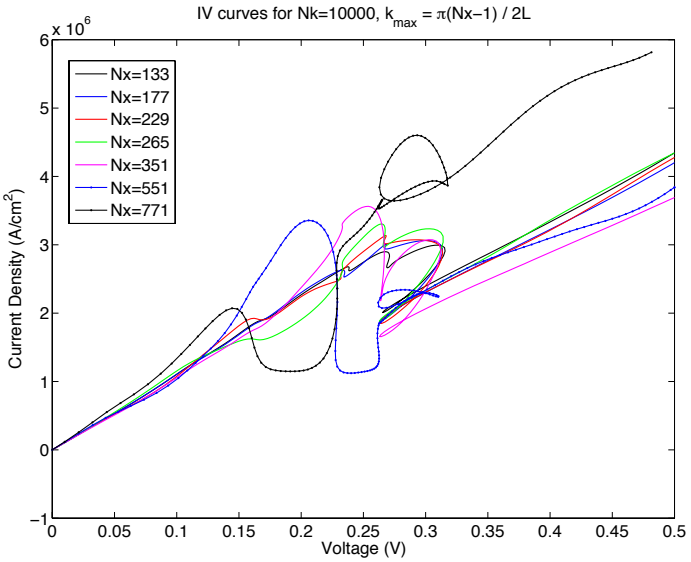

Figure 6.22: IV curves using various values for Nx and $Nk = 10000$. For $Nk \leq 229$, the curves are similar in shape, but they develop larger loops and veer away from the initial curves as $Nx > 351$.

The only way I was able to find a set of spatial and momentum meshes that produced a converged IV curve was to fix size of the momentum domain and then decrease the spatial mesh used to do simulation runs. When I did this, I was able to get the IV curve to converge to a single solution, which matched previous IV curves using a fixed momentum space of $[-0.25, 0.25]$. See figure 6.23 for an example.

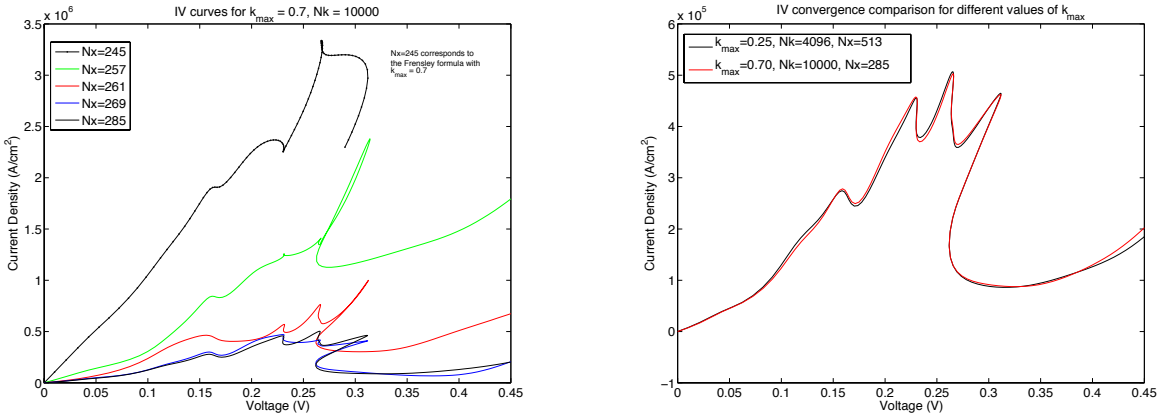

Figure 6.23: The figure on the left fixes L_k at 0.70, and then the size of the spatial mesh is decreased until convergence is seen. The figure on the right is a comparison between the converged IV curves for $L_k = 0.25$ (previous work) and the curve using $Nx = 285$ and $L_k = 0.70$ (from the figure on the left).

Since we cannot get physically realistic results using the variable momentum domain, we concluded that the momentum domain should be fixed to get reasonable results from our simulation runs.

6.7 Von Neumann Analysis

Von Neumann analysis is generally used to determine stability criteria for constant coefficient linear PDEs [74]. Since the Wigner equation is a nonlinear integro-differential equation, we applied Von Neumann analysis cautiously, hoping to come up with a result that could be used as a guideline in determining the time step Δt for our time dependent simulation runs.

The Wigner equation is

$$\frac{\partial f(x,k,t)}{\partial t} = K(f) + P(f) + S(f) \tag{6.10}$$

where

$$K(f) = -\frac{\hbar k}{m^*}\frac{\partial f(x,k)}{\partial x} \tag{6.11}$$

$$P(f) = -\frac{4}{h}\int_{-\infty}^{\infty} f(x,k')dk' \int_0^{\infty} [U(x+y) - U(x-y)]\sin(2y(k-k'))dy \tag{6.12}$$

and

$$S(f) = \frac{1}{\tau}\left(\frac{\int_{-\infty}^{\infty} f(x,k')dk'}{\int_{-\infty}^{\infty} f_0(x,k')dk'} \cdot f_0(x,k) - f(x,k)\right) \tag{6.13}$$

Applying 4th order finite difference approximations, the terms on the right hand side become

$$K(f(x_m,k_j,t_n)) = \frac{\hbar|k_m|}{m^*}\left(\frac{25f_{m,j} - 48f_{m\pm1,j} + 36f_{m\pm2,j} - 16f_{m\pm3,j} + 3f_{m\pm4,j}}{12\Delta x}\right) \tag{6.14}$$

$$P(f(x_m,k_j,t_n)) = -\frac{4}{h}\sum_{j'=1}^{Nk} f_{m,j'}w_{j'} \sum_{m'=1}^{Nc+1} [U(x_m + x_{m'}) - U(x_m - x_{m'})] \times \sin(2x_{m'}(k_j - k_{j'}))w_{m'} \tag{6.15}$$

and

$$S(f(x_m,k_j,t_n)) = \frac{1}{\tau}\left(\frac{\sum_{j'=1}^{Nk} f_{m,j'}w_{j'}}{\sum_{j'=1}^{Nk} f^0_{m,j'}w_{j'}} \cdot f^0_{m,j} - f_{m,j}\right) \tag{6.16}$$

where the weight terms are (at worst): $w_{j'} \sim O(\Delta k)$ and $w_{m'} \sim O((\Delta x)^2)$ and $f^0_{m,j} = f(x_m, k_j, t = 0)$ at V=0.

Von Neumann analysis assumes that we can write the solution $f(x, k, t)$ as

$$f(x, k, t) = g(\xi, \eta) e^{i(\xi x + \eta k)} \tag{6.17}$$

where $g(\xi, \eta)$ is an amplification factor, with $\xi, \eta \in \mathcal{Z}$. The discretized version becomes

$$f(x_m, k_j, t_n) = f^n_{m,j} = g(\xi, \eta)^n e^{i\xi m\Delta x + i\eta j \Delta k} \tag{6.18}$$

6.7.1 Elimination of the scattering term

Substituting the discretized Von Neumann function into the Wigner equation, we can immediately eliminate the scattering term from the analysis:

$$\begin{aligned}
\mathbf{S}(\mathbf{f^n_{m,j}}) &= \frac{1}{\tau}\left(\frac{\sum_{j'=1}^{Nk} f^n_{m,j'} w_{j'}}{\sum_{j'=1}^{Nk} f^0_{m,j'} w_{j'}} \cdot f^0_{m,j} - f^n_{m,j}\right) && (6.19)\\
&= \frac{1}{\tau}\left(\frac{\sum_{j'=1}^{Nk} g(\xi,\eta)^n e^{i\xi m\Delta x + i\eta j'\Delta k} w_{j'}}{\sum_{j'=1}^{Nk} e^{i\xi m\Delta x + i\eta j'\Delta k} w_{j'}} \cdot e^{i\xi m\Delta x + i\eta j\Delta k} - g(\xi,\eta)^n e^{i\xi m\Delta x + i\eta j\Delta k}\right) && (6.20)\\
&= \frac{1}{\tau}\left(\frac{g(\xi,\eta)^n \sum_{j'=1}^{Nk} e^{i\xi m\Delta x + i\eta j'\Delta k} w_{j'}}{\sum_{j'=1}^{Nk} e^{i\xi m\Delta x + i\eta j'\Delta k} w_{j'}} \cdot e^{i\xi m\Delta x + i\eta j\Delta k} - g(\xi,\eta)^n e^{i\xi m\Delta x + i\eta j\Delta k}\right) && (6.21)\\
&= \frac{1}{\tau}\left(g(\xi,\eta)^n \cdot e^{i\xi m\Delta x + i\eta j\Delta k} - g(\xi,\eta)^n e^{i\xi m\Delta x + i\eta j\Delta k}\right) = 0 && (6.22)
\end{aligned}$$

So the scattering term has no effect on the stability.

6.7.2 Elimination of the potential term

By analyzing the effect of the potential term on the stability of the Wigner problem, we can show it only plays a minor role:

$$\mathbf{P}(\mathbf{f^n_{mj}}) = -\frac{4}{h}\sum_{j'=1}^{Nk} f_{m,j'} w_{j'} \sum_{m'=1}^{Nc+1} [U(x_m + x_{m'}) - U(x_m - x_{m'})] \sin(2x_{m'}(k_j - k_{j'}))w_{m'} \quad (6.23)$$

$$\sim -\frac{4}{h}\sum_{j'=1}^{Nk} f_{m,j'} \cdot O(\Delta k) \sum_{m'=1}^{Nc+1} [U(x_m + x_{m'}) - U(x_m - x_{m'})] \times \quad (6.24)$$
$$\sin(2x_{m'}(k_j - k_{j'})) \cdot O((\Delta x)^2)$$

Now

$$-\frac{4}{h}\sum_{j'=1}^{Nk} f_{m,j'} \cdot O(\Delta k) \sim O(1) \quad (6.25)$$

and

$$\sum_{m'=1}^{Nc+1} [U(x_m + x_{m'}) - U(x_m - x_{m'})] \sin(2x_{m'}(k_j - k_{j'})) \cdot O((\Delta x)^2) \sim O(\Delta x) \quad (6.26)$$

so

$$\mathbf{P}(\mathbf{f^n_{mj}}) \sim O(1) \cdot O(\Delta x) = O(\Delta x) \quad (6.27)$$

Thus the potential term contributes an effect proportional to the size of the spatial mesh. The kinetic term, however, is $O(1/\Delta x)$:

$$\mathbf{K}(\mathbf{f}(\mathbf{x_m}, \mathbf{k_j})) = \frac{\hbar|k_m|}{m^*}\left(\frac{25f_{m,j} - 48f_{m\pm1,j} + 36f_{m\pm2,j} - 16f_{m\pm3,j} + 3f_{m\pm4,j}}{12\Delta x}\right) \quad (6.28)$$

$$\sim \frac{\hbar|k_m|}{m^*} \cdot O(\frac{1}{\Delta x}) \quad (6.29)$$

$$(6.30)$$

Compared to the kinetic term, the potential term's contribution to the stability of the Wigner equation is minor. Thus, we can drop the potential term from consideration, and focus only on the kinetic term. So our stability analysis is reduced to that for a constant coefficient

PDE:

$$\frac{\partial f}{\partial t} = \frac{\hbar|k_m|}{m^*}\left(\frac{25f_{m,j} - 48f_{m\pm1,j} + 36f_{m\pm2,j} - 16f_{m\pm3,j} + 3f_{m\pm4,j}}{12\Delta x}\right) \tag{6.31}$$

where $\frac{\partial f}{\partial t}$ will be discretized using either the first order Backward Euler method or the second order BDF2 method.

6.7.3 Von Neumann stability analysis for Backward Euler

Using Backward Euler to approximate the time derivative and applying the discretized Von Neumann solution, we have

$$\begin{aligned}
\frac{f_{m,j}^{n+1} - f_{m,j}^{n}}{\Delta t} &= \frac{\hbar|k_m|}{m*}\left(\frac{25f_{m,j}^{n+1} - 48f_{m\pm1,j}^{n+1} + 36f_{m\pm2,j}^{n+1}}{12\Delta x}\right. \\
&\qquad \left.\frac{-16f_{m\pm3,j}^{n+1} + 3f_{m\pm4,j}^{n+1}}{12\Delta x}\right) \qquad (6.32)\\
\frac{(g^{n+1} - g^n)e^{i\xi m\Delta x + i\eta j\Delta k}}{\Delta t} &= \frac{\hbar|k_m|}{12m^*\Delta x}\cdot g^{n+1}e^{i\xi m\Delta x + i\eta j\Delta k}\times \qquad (6.33)\\
&\qquad \left(25 - 48e^{\pm i\xi\Delta x} + 36e^{\pm 2i\xi\Delta x} - 16e^{\pm 3i\xi\Delta x} + 3e^{\pm 4i\xi\Delta x}\right)\\
g^{n+1} - g^n &= \frac{\hbar|k_m|\Delta t}{12m^*\Delta x}\cdot g^{n+1}\times \qquad (6.34)\\
&\qquad \left(25 - 48e^{\pm i\xi\Delta x} + 36e^{\pm 2i\xi\Delta x} - 16e^{\pm 3i\xi\Delta x} + 3e^{\pm 4i\xi\Delta x}\right)\\
g - 1 &= \frac{\hbar|k_m|\Delta t}{12m^*\Delta x}\cdot g\,(A + Bi) \qquad (6.35)
\end{aligned}$$

where

$$A = 25 - 48\cos(\xi\Delta x) + 36\cos(2\xi\Delta x) - 16\cos(3\xi\Delta x) + 3\cos(4\xi\Delta x) \tag{6.36}$$

$$B = \mp 48\sin(\xi\Delta x) \pm 36\sin(2\xi\Delta x) \mp 16\sin(3\xi\Delta x) \pm 3\sin(4\xi\Delta x) \tag{6.37}$$

Then

$$g\left(1-\frac{\hbar|k_m|\Delta t}{12m^*\Delta x}(A+Bi)\right)=1 \quad \Longrightarrow \quad g=\left(1-\frac{\hbar|k_m|\Delta t}{12m^*\Delta x}(A+Bi)\right)^{-1} \tag{6.38}$$

For stability, we need the amplification factor $||g|| \leq 1$, so

$$||g||=\left|\left|\left(1-\frac{\hbar|k_m|\Delta t}{12m^*\Delta x}(A+Bi)\right)^{-1}\right|\right| \leq 1 \quad \Longrightarrow \quad \left|\left|1-\frac{\hbar|k_m|\Delta t}{12m^*\Delta x}(A+Bi)\right|\right| \geq 1 \tag{6.39}$$

Now

$$\begin{aligned}
\left|\left|1-\frac{\hbar|k_m|\Delta t}{12m^*\Delta x}(A+Bi)\right|\right| &= \left|\left|1-\frac{\hbar|k_m|\Delta t}{12m^*\Delta x}A+i\cdot\frac{\hbar|k_m|\Delta t}{12m^*\Delta x}B\right|\right| \geq 1 && (6.40)\\
\left(1-\frac{\hbar|k_m|\Delta t}{12m^*\Delta x}A\right)^2+\left(\frac{\hbar|k_m|\Delta t}{12m^*\Delta x}B\right)^2 &= 1-\frac{\hbar|k_m|\Delta t}{6m^*\Delta x}A && (6.41)\\
&\quad +\left(\frac{\hbar|k_m|\Delta t}{12m^*\Delta x}\right)^2\left(A^2+B^2\right)\geq 1 \\
\left(\frac{\hbar|k_m|\Delta t}{12m^*\Delta x}\right)^2\left(A^2+B^2\right) &\geq \frac{\hbar|k_m|\Delta t}{6m^*\Delta x}A && (6.42)\\
\frac{\hbar|k_m|\Delta t}{12m^*\Delta x} &\geq \frac{2A}{A^2+B^2} && (6.43)\\
\Delta t &\geq \frac{24m^*A}{\hbar(A^2+B^2)}\frac{\Delta x}{|k_m|} && (6.44)\\
\mathbf{\Delta t} &\geq \frac{48m^*A}{\hbar(A^2+B^2)}\frac{\Delta x}{\Delta k}\sim \mathbf{O}\left(\frac{\mathbf{\Delta x}}{\mathbf{\Delta k}}\right) && (6.45)
\end{aligned}$$

Thus, for Backward Euler, we can use the ratio of the spatial mesh over the momentum mesh as a guideline to choose an appropriate time step to yield a convergent solution.

6.7.4 Von Neumann stability analysis for BDF-2

Using BDF-2 to approximate the time derivative and applying the discretized Von Neumann solution, we have

$$\begin{aligned}
\frac{3f_{m,j}^{n+1} - 4f_{m,j}^{n} + f_{m,j}^{n-1}}{2\Delta t} &= \frac{\hbar|k_m|}{m*}\left(\frac{25f_{m,j}^{n+1} - 48f_{m\pm1,j}^{n+1} + 36f_{m\pm2,j}^{n+1}}{12\Delta x}\right. \\
&\qquad \left.\frac{-16f_{m\pm3,j}^{n+1} + 3f_{m\pm4,j}^{n+1}}{12\Delta x}\right) \qquad (6.46)\\
\frac{(3g^{n+1} - 4g^{n} + g^{n-1})e^{i\xi m\Delta x + i\eta j\Delta k}}{2\Delta t} &= \frac{\hbar|k_m|\ g^{n+1}}{12m^*\Delta x}e^{i\xi m\Delta x + i\eta j\Delta k} \times \qquad (6.47)\\
&\qquad \left(25 - 48e^{\pm i\xi\Delta x} + 36e^{\pm 2i\xi\Delta x} - 16e^{\pm 3i\xi\Delta x} + 3e^{\pm 4i\xi\Delta x}\right)\\
3g^{n+1} - 4g^{n} + g^{n-1} &= \frac{\hbar|k_m|\Delta t\ g^{n+1}}{6m^*\Delta x} \times \qquad (6.48)\\
&\qquad \left(25 - 48e^{\pm i\xi\Delta x} + 36e^{\pm 2i\xi\Delta x} - 16e^{\pm 3i\xi\Delta x} + 3e^{\pm 4i\xi\Delta x}\right)\\
3g^2 - 4g + 1 &= \frac{\hbar|k_m|\Delta t\ g^2}{6m^*\Delta x}\left(A + Bi\right) \qquad (6.49)\\
0 &= \left(3 - \frac{\hbar|k_m|\Delta t}{6m^*\Delta x}\left(A + Bi\right)\right)g^2 - 4g + 1 \qquad (6.50)
\end{aligned}$$

where A, B are defined as in equations 6.36 and 6.37 for Backward Euler. Using the quadratic equation to solve for g, we have

$$g = \frac{4 \pm \sqrt{16 - 4\left(3 - \frac{\hbar|k_m|\Delta t}{6m^*\Delta x}\left(A + Bi\right)\right)}}{2\left(3 - \frac{\hbar|k_m|\Delta t}{6m^*\Delta x}\left(A + Bi\right)\right)} = \frac{2 \pm \sqrt{1 + \frac{\hbar|k_m|\Delta t}{6m^*\Delta x}\left(A + Bi\right)}}{3 - \frac{\hbar|k_m|\Delta t}{6m^*\Delta x}\left(A + Bi\right)} \qquad (6.51)$$

We need $||g|| \leq 1$, so let $M = \frac{\hbar|k_m|\Delta t}{6m^*\Delta x}$, and

$$\begin{aligned}
||g|| = \left|\left|\frac{2 \pm \sqrt{1 + \frac{\hbar|k_m|\Delta t}{6m^*\Delta x}\left(A + Bi\right)}}{3 - \frac{\hbar|k_m|\Delta t}{6m^*\Delta x}\left(A + Bi\right)}\right|\right| &= \left|\left|\frac{2 \pm \sqrt{1 + M\left(A + Bi\right)}}{3 - M\left(A + Bi\right)}\right|\right| \leq 1 \qquad (6.52)\\
\implies \qquad ||2 \pm \sqrt{1 + MA + MBi}|| &\leq ||3 - MA - MBi|| \qquad (6.53)
\end{aligned}$$

The square root of an imaginary number is again imaginary, so let

$$p + qi = \sqrt{1 + MA + MBi} \tag{6.54}$$

where

$$p = \frac{1}{\sqrt{2}}\sqrt{\sqrt{(1+MA)^2 + (MB)^2} + 1 + MA} \tag{6.55}$$

$$q = \frac{sgn(b)}{\sqrt{2}}\sqrt{\sqrt{(1+MA)^2 + (MB)^2} - 1 - MA} \tag{6.56}$$

and

$$p^2 + q^2 = \sqrt{(1+MA)^2 + (MB)^2}. \tag{6.57}$$

Then

$$||2 \pm (p + qi)||^2 = (2 \pm p)^2 + q^2 = 4 \pm 4p + p^2 + q^2 \tag{6.58}$$

$$\leq ||3 - MA - MBi||^2 \tag{6.59}$$

$$= (3 - MA)^2 + (MB)^2 \tag{6.60}$$

and

$$\pm 4p \leq (3 - MA)^2 + (MB)^2 - 4 - (p^2 + q^2) \tag{6.61}$$

$$16p^2 \leq \left[(3 - MA)^2 + (MB)^2 - 4 - (p^2 + q^2)\right]^2 \tag{6.62}$$

$$8\left(\sqrt{(1+MA)^2 + (MB)^2} + 1 + MA\right) \leq \left[(3 - MA)^2 + (MB)^2 - 4\right]^2 + (p^2 + q^2)^2 \tag{6.63}$$
$$-2\left[(3 - MA)^2 + (MB)^2 - 4\right](p^2 + q^2)$$

$$8(p^2 + q^2) + 8(1 + MA) \leq \left[(3 - MA)^2 + (MB)^2 - 4\right]^2 \tag{6.64}$$
$$-2\left[(3 - MA)^2 + (MB)^2 - 4\right](p^2 + q^2)$$
$$+(1 + MA)^2 + (MB)^2$$

$$
\begin{aligned}
8(p^2+q^2)+8(1+MA) &\leq \left[(3-MA)^2+(MB)^2-4\right]^2 && (6.65)\\
&\quad -2\left[(3-MA)^2+(MB)^2\right](p^2+q^2)\\
&\quad +8(p^2+q^2)+(1+MA)^2+(MB)^2\\
2\left[(3-MA)^2+(MB)^2\right](p^2+q^2) &\leq \left[(3-MA)^2+(MB)^2-4\right]^2 && (6.66)\\
&\quad +(1+MA)^2+(MB)^2-8(1+MA)\\
&\leq \left[(3-MA)^2+(MB)^2\right]^2 && (6.67)\\
&\quad -8\left[(3-MA)^2+(MB)^2\right]\\
&\quad +16+(-16+9-6MA+(MA)^2+(MB)^2)\\
&\leq \left[(3-MA)^2+(MB)^2\right]^2 && (6.68)\\
&\quad -8\left[(3-MA)^2+(MB)^2\right]+(3-MA)^2+(MB)^2\\
&\leq \left[(3-MA)^2+(MB)^2\right]\times && (6.69)\\
&\quad \left[(3-MA)^2+(MB)^2-7\right]
\end{aligned}
$$

$$
\begin{aligned}
\implies \quad 2(p^2+q^2) &\leq \left[(3-MA)^2+(MB)^2-7\right] && (6.70)\\
4\left[(1+MA)^2+(MB)^2\right] &\leq \left[(3-MA)^2+(MB)^2-7\right]^2 && (6.71)\\
0 &\leq \left[(3-MA)^2+(MB)^2-7\right]^2-4\left[(1+MA)^2+(MB)^2\right] && (6.72)\\
&\leq \left[-6MA+(MA)^2+(MB)^2+2\right]^2 && (6.73)\\
&\quad -4[1+8MA-6MA+(MA)^2+(MB)^2]\\
&\leq \left[-6MA+(MA)^2+(MB)^2\right]^2 && (6.74)\\
&\quad +4\left[-6MA+(MA)^2+(MB)^2\right]\\
&\quad +4-4[-6MA+(MA)^2+(MB)^2]-4-32MA\\
&\leq [-6MA+M^2(A^2+B^2)]^2-32MA && (6.75)
\end{aligned}
$$

$$
\implies \quad (A^2+B^2)^2\cdot M^3-12A(A^2+B^2)\cdot M^2+36A^2\cdot M-32A \geq 0 \qquad (6.76)
$$

To determine for which values of M the above equation holds, we solve the cubic equation for M to find the roots of the equation. For the general cubic equation $F(M) = aM^3 + bM^2 + cM + d = 0$, the discriminant $\Delta = 18abcd - 4b^3d + b^2c^2 - 4ac^3 - 27a^2d^2$ determines how many real roots exist. For equation 6.76,

$$\begin{aligned}
\Delta &= 18(A^2+B^2)^2(-12A[A^2+B^2])(36A^2)(-32A) - 4(-12A[A^2+B^2])^3(-32A) \qquad (6.77)\\
&\quad +(-12A[A^2+B^2])^2(36A^2)^2 - 4(A^2+B^2)^2(36A^2)^3 - 27(A^2+B^2)^4(-32A)^2\\
&= -27,648\ A^2B^2(A^2+B^2)^3\ <\ 0\ \forall\ A,B \text{ (since A, B} \neq 0 \text{ at the same time)} \qquad (6.78)
\end{aligned}$$

Since the determinant is negative, there only exists one real root, which is

$$M^* = \frac{4A}{A^2+B^2} + \frac{2A^{1/3}}{A^2+B^2}\left[(\sqrt{A^2+B^2}-B)^{2/3} + (\sqrt{A^2+B^2}+B)^{2/3}\right] \qquad (6.79)$$

To determine whether M^* is a minimum or maximum, we look at the derivative of the cubic equation at M^*:

$$\begin{aligned}
F'(M^*) &= 3a(M^*)^2 + 2bM^* + c \qquad (6.80)\\
&= 3(A^2+B^2)^2(M^*)^2 - 24A(A^2+B^2)M^* + 36A^2 \qquad (6.81)\\
&= 3\Big(16A^2 + 16A^{4/3}\left[(\sqrt{A^2+B^2}-B)^{2/3} + (\sqrt{A^2+B^2}+B)^{2/3}\right] \qquad (6.82)\\
&\quad +4A^{2/3}\left[(\sqrt{A^2+B^2}-B)^{2/3} + (\sqrt{A^2+B^2}+B)^{2/3}\right]^2\Big)\\
&\quad -24A\left(4A + 2A^{1/3}\left[(\sqrt{A^2+B^2}-B)^{2/3} + (\sqrt{A^2+B^2}+B)^{2/3}\right]\right) + 36A^2\\
&= -12A^2 + 12A^{2/3}\left[(\sqrt{A^2+B^2}-B)^{2/3} + (\sqrt{A^2+B^2}+B)^{2/3}\right]^2 \qquad (6.83)
\end{aligned}$$

To show the derivative is positive (and thus that M^* is a minimum), we need $(\sqrt{A^2+B^2}-B)^{2/3}+(\sqrt{A^2+B^2}+B)^{2/3} \geq A^{2/3}$:

$$\begin{aligned}
\left[(\sqrt{A^2+B^2}-B)^{2/3}+(\sqrt{A^2+B^2}+B)^{2/3}\right]^3 &= (\sqrt{A^2+B^2}-B)^2 && (6.84)\\
&\quad + (\sqrt{A^2+B^2}+B)^2 \\
&\quad +3\,(A^2+B^2-B^2)^{2/3}\times \\
&\qquad \left[(\sqrt{A^2+B^2}-B)^{2/3}\right. \\
&\qquad\qquad \left. + (\sqrt{A^2+B^2}+B)^{2/3}\right] \\
&= 2A^2+4B^2+3A^{4/3}\times && (6.85)\\
&\qquad \left[(\sqrt{A^2+B^2}-B)^{2/3}\right. \\
&\qquad\qquad \left. + (\sqrt{A^2+B^2}+B)^{2/3}\right] \\
&\geq 2A^2+4B^2+3A^{4/3}\times && (6.86)\\
&\qquad \left[\sqrt{A^2+B^2}+B\right]^{2/3} \\
&\geq 2A^2+3A^{4/3}\cdot A^{2/3}=5A^2 && (6.87)
\end{aligned}$$

$$\implies \quad (\sqrt{A^2+B^2}-B)^{2/3}+(\sqrt{A^2+B^2}+B)^{2/3} \geq \sqrt[3]{5}A^{2/3} \geq A^{2/3} \qquad (6.88)$$

So

$$\begin{aligned}
F'(M^*) &= -12A^2+12A^{2/3}\left[(\sqrt{A^2+B^2}-B)^{2/3}+(\sqrt{A^2+B^2}+B)^{2/3}\right]^2 && (6.89)\\
&\geq -12A^2+12A^{2/3}(\sqrt[3]{5}A^{2/3})^2=12(-1+\sqrt[3]{25})A^2 \;>\; 0 && (6.90)
\end{aligned}$$

And thus $(A^2+B^2)^2 \cdot M^3 - 12A(A^2+B^2) \cdot M^2 + 36A^2 \cdot M - 32A \geq 0$ is satisfied when $M \geq M^*$, so

$$M \geq M^* \tag{6.91}$$

$$\Longrightarrow \quad \frac{\hbar|k_m|\Delta t}{6m^*\Delta x} \geq M^* \tag{6.92}$$

$$\mathbf{\Delta t} \geq \frac{6m^*\Delta x}{\hbar|k_m|}M^* \sim O\left(\frac{\Delta x}{|k_m|}\right) \sim \mathbf{O}\left(\frac{\mathbf{\Delta x}}{\mathbf{\Delta k}}\right) \tag{6.93}$$

And we find again that choosing an appropriate time step to yield a convergent solution is dependent on the ratio of the size of the spatial mesh over the size of the momentum mesh.

6.7.5 Von Neumann stability analysis for Crank-Nicolson

For some of my early time dependent simulation runs, I tried using the Crank-Nicolson method to calculate the Wigner function for $t \geq t_2$, but could not get convergent results regardless of the time step and momentum / spatial meshes used. When we decided to use Von Neumann analysis to provide a guideline for choosing the time step for BDF2, I applied Von Neumann analysis to the Crank-Nicolson method and discovered that my experimental results were consistent with the result of the analysis, which was that Crank-Nicolson will not provide convergent solutions for the Wigner equation, regardless of the time step used.

Applying the discretized approximations for Crank-Nicolson to the reduced Wigner equation used for Von Neumann analysis (see equation 6.31):

$$\frac{f_{m,j}^{n+1} - f_{m,j}^{n}}{\Delta t} = \frac{1}{2}\left[W(f_{m,j}^{n+1}) + W(f_{m,j}^{n})\right] \tag{6.94}$$

$$= \frac{1}{2}\left[\frac{\hbar|k_m|}{m*}\left(\frac{25f_{m,j}^{n+1} - 48f_{m\pm1,j}^{n+1} + 36f_{m\pm2,j}^{n+1} - 16f_{m\pm3,j}^{n+1} + 3f_{m\pm4,j}^{n+1}}{12\Delta x}\right) + \right. \tag{6.95}$$

$$\left. \frac{\hbar|k_m|}{m*}\left(\frac{25f_{m,j}^{n} - 48f_{m\pm1,j}^{n} + 36f_{m\pm2,j}^{n} - 16f_{m\pm3,j}^{n} + 3f_{m\pm4,j}^{n}}{12\Delta x}\right)\right]$$

Substituting $g(\xi,\eta)^n e^{i\xi m\Delta x+i\eta j\Delta k}$ for $f^n_{m,j}$ leads to

$$\begin{aligned}
\frac{(g^{n+1}-g^n)e^{i\xi m\Delta x+i\eta j\Delta k}}{\Delta t} &= \frac{\hbar|k_m|}{24m^*\Delta x}\cdot(g^{n+1}+g^n)e^{i\xi m\Delta x+i\eta j\Delta k}\times \qquad (6.96)\\
&\quad \left(25-48e^{\pm i\xi\Delta x}+36e^{\pm 2i\xi\Delta x}-16e^{\pm 3i\xi\Delta x}+3e^{\pm 4i\xi\Delta x}\right)\\
g^{n+1}-g^n &= \frac{\hbar|k_m|\Delta t}{24m^*\Delta x}\cdot(g^{n+1}+g^n)\times \qquad (6.97)\\
&\quad \left(25-48e^{\pm i\xi\Delta x}+36e^{\pm 2i\xi\Delta x}-16e^{\pm 3i\xi\Delta x}+3e^{\pm 4i\xi\Delta x}\right)\\
g-1 &= \frac{\hbar|k_m|\Delta t}{24m^*\Delta x}\cdot(g+1)\,(A+Bi) \qquad (6.98)
\end{aligned}$$

where A, B are as defined in equations 6.36 and 6.37 for Backward Euler. Let $M=\frac{\hbar|k_m|}{24m^*\Delta x}$; then $M>0$ for all discretized values of $k_m, \Delta x$. So

$$\begin{aligned}
g-1 &= M\cdot(g+1)(A+Bi) \qquad (6.99)\\
g-Mg(A+Bi) &= 1+M(A+Bi) \qquad (6.100)\\
\implies g &= \frac{1+MA+MBi}{1-MA-MBi} \qquad (6.101)
\end{aligned}$$

We need $||g||\leq 1$, so

$$\begin{aligned}
||1+MA+MBi|| &\leq ||1-MA-MBi|| \qquad (6.102)\\
(1+MA)^2+(MB)^2 &\leq (1-MA)^2+(MB)^2 \qquad (6.103)\\
2MA &\leq -2MA \qquad (6.104)\\
\implies A &\leq 0 \qquad (6.105)
\end{aligned}$$

Per Von Neumann analysis, the above inequality must hold true for all values of A. But $A=25-48\cos(\xi\Delta x)+36\cos(2\xi\Delta x)-16\cos(3\xi\Delta x)+3\cos(4\xi\Delta x)$ where ξ is an arbitrary wave

number, and if $\xi = \frac{\pi}{\Delta x}$, then

$$A \;=\; 25 - 48\cos\pi + 36\cos 2\pi - 16\cos 3\pi + 3\cos 4\pi \;=\; 25 + 48 + 36 + 16 + 3 \;=\; 128 > 0.$$

So the amplification factor $g(\xi, \eta)$ does not comply with $||g|| \leq 1$ for all values of ξ, η, and thus Crank-Nicolson is not a convergent method for the Wigner equation.

Chapter 7

Scalability

7.1 RTD Code Improvements

To improve the performance of the code, I analyzed each piece of the Wigner-Poisson computation to determine how to improve both the scalability and the simulation run times. Improvements were made to the computational methods, data storage, and the distributed Wigner vectors. The result is a simulation model that decreases run times by over 90% and reaches maximum speedup using half the number of cores as the original code, although some scalability is sacrificed.

As mentioned in chapter 6, the scaling studies were performed on the NCSU HPC cluster using a set of 30 dual-processor Intel Xeon blades with Infiniband interconnects. Since we wanted to maximize the use of the available blades, I ran simulations for the scaling study using extremely fine grids that would reach maximum speedup at over 100 cores. The base number of cores was set to 8 (for efficiency and serial percentage computations using Amdahl's Law) to match the number of cores per blade, and each simulation run used all 8 cores per assigned blade.

7.1.1 Wigner Equation Changes

Initially, the numerical simulation was set up to solve the Wigner equation (equation 2.49) by calculating each term separately and then applying a preconditioner:

$$\mathbf{A}\left[\mathbf{K}f + P(f) + S_1(f) - \frac{1}{\tau}f\right] = 0 \tag{7.1}$$

Here $\mathbf{A}$ is the preconditioner matrix, which is the inverse of the kinetic matrix; $\mathbf{K}$ is the matrix representing the numerical approximation of the kinetic term; $P(f)$ is the potential term; and $S_1(f) = \frac{1}{\tau}\frac{\int f}{\int f_0}$. However, the Wigner equation can be rewritten to eliminate separate computation of the kinetic term, by rearranging terms as:

$$f + \left[\mathbf{K} - \frac{1}{\tau}\mathbf{I}\right]^{-1} \cdot [P(f) + S_1(f)] = 0 \tag{7.2}$$

Solving the Wigner equation in this form reduced run times by approximately 30%.

7.1.2 Sparse Matrix Interpolation

Next, the interpolation routine was changed to use a sparse matrix-vector product rather than an in-line computation at each grid point. The sparse matrix was pre-computed by determining the interpolation constants needed to produce each unit vector in a grid region and assigning the constants to the matrix so that the result of the sparse matrix with a non-uniform Wigner vector is the interpolated Wigner vector.

While these changes improved the scalability of the Wigner code, parallel profiling results revealed several areas that could significantly decrease run times. However, decreasing run times of highly parallel portions of the code would increase the serial percentage and lose some of the scalability improvements. Since the main goal of this work is to decrease simulation run times to allow finer grids to be used and longer devices to be modeled, we chose to implement these changes at the possible expense of scalability.

Table 7.1: Profiling statistics for each portion of the parallel RTD code using a grid of $Nk = 2048$, $Nx = 513$

DESCRIPTION *(in order of computation)*	*Avg run time (sec)* 8 cores	24 cores	% change
Import distributed non-unif vector to full vector	451.398	682.842	+51.3
Compute interpolation via sparse mat-vec	1,627.618	911.624	-44.0
Integrate (distributed) electron density	12.517	4.492	-64.1
Import distributed electron density to full vector	73.220	73.294	+0.1
Compute Poisson's equation	36.510	35.779	-2.0
Compute the inner potential integral (over x)	44,957.463	14,979.325	-66.7
Compute the outer potential integral (over k)	16,227.338	5,027.028	-69.0
Add the scattering term	9.350	3.119	-66.6
Flip the distributed (over x) vector to dist over k	1,203.569	1,182.423	-1.8
Apply the preconditioner	26.344	17.240	-34.6
Flip the vector back to distribution over x	17.418	29.684	+70.4
Add the original Wigner function to new solution	2.945	0.777	-73.6
Total for all Wigner computations	64,647.788	22,948.758	-64.5

The brown text signifies cross-processor communication.
The blue text indicates serial code.

7.1.3 BLAS

One of the most significant differences in run times was a change to the Tp matrix computation to incorporate a highly efficient routine from the BLAS library.

Profiling statistics on the parallel version of the Wigner code were obtained to determine where the bottlenecks were. Initial results pointed to one routine – the calculation of the Tp matrix – as the main culprit, since it was responsible for about 70% of total computation time (see table 7.1). Review of the Tp matrix calculation suggested a rewrite of the multiple nested loops into a matrix-matrix calculation. To increase the efficiency of the matrix-matrix computation, a BLAS routine was used.

The Basic Linear Algebra Subprograms (BLAS) library is a set of routines that provide highly optimized programs for computing basic vector and matrix operations [75, 76]. BLAS routines are divided into three levels: level 1 deals with vector-vector operations, level 2 with matrix-vector operations, and level 3 with matrix-matrix routines. Optimized level 3 BLAS routines can come close to peak machine speed [75], so reordering loops to use higher level BLAS routines (particularly BLAS-3 routines) can decrease run times significantly. Thus, the BLAS-3 routine DGEMM, an optimized matrix-matrix multiply routine, was incorporated within the Trilinos framework, which reduced total run times by 70-80%.

However, since the Tp matrix calculation was almost perfectly parallel and contributed significantly to overall run times, the incorporation of the optimized routine decreased the efficiency of the code (as computed via Amdahl's Law), which in turn decreased the number of cores needed to achieve maximum speedup. So some scalability was lost, but the significant decrease in run times justified the change.

7.1.4 Optimization of Interpolation Routine

Once the BLAS-3 routine was implemented, new profiling statistics were gathered to determine if there were other areas to target for improvement (see table 7.2.) The parallel profiling results showed that the interpolation routine (sparse matrix-vector multiply) was not performing well; as an example, see table 7.2, which shows a decrease in run time of only 40% (as cores are increased by a factor of 5) rather than the expected 80%.

In reviewing the structure of the data objects used in the interpolation routine, I realized that the distributed Wigner vector over x required significant cross-processor communication, and a change to a distributed Wigner vector over k should eliminate it. So the interpolation computation was updated to reflect the more efficient parallelization method, and new profiling results (shown in table 7.3) reflect a proper decrease in run times for the interpolation routine.

Table 7.2: Profiling statistics for each portion of the parallel RTD code after the implementation of the BLAS-3 routine and data layout improvements. Statistics gathered using a grid of $Nk = 2048$, $Nx = 513$

DESCRIPTION *(in order of computation)*	*Avg run time (sec)* 8 cores	24 cores	% change
Compute interpolation via sparse mat-vec	1,523.346	888.815	-41.7
Integrate (distributed) electron density	8.159	2.863	-64.9
Import distributed electron density to full vector	39.892	63.763	+59.8
Compute Poisson's equation	35.926	35.630	-0.8
Compute the inner potential integral (over x)	1,477.291	635.471	-57.0
Compute the outer potential integral (over k)	5,170.314	1,353.925	-73.8
Add the scattering term	5.726	2.123	-62.9
Flip the distributed vector to distribution over k	103.935	192.611	+85.3
Apply the preconditioner	22.353	7.565	-66.2
Flip the vector back to distribution over x	19.072	33.472	+75.5
Add the original Wigner function to new solution	3.007	0.853	-71.6
Total for all Wigner computations	8,409.233	3,217.277	-61.7

Table 7.3: Profiling statistics for each portion of the parallel RTD code after the change to the Interpolation matrix structure and additional data layout improvements. Statistics gathered using a grid of $Nk = 2048$, $Nx = 513$

DESCRIPTION *(in order of computation)*	*Avg run time (sec)* 8 cores	24 cores	% change
Compute interpolation via sparse mat-vec	176.383	65.498	-62.9
Multiply interpolated vector by weights for k integration	13.560	4.517	-66.8
Flip the distributed (over k) vector to distribution over x	281.763	216.883	-23.0
Integrate (distributed) electron density	2.282	0.725	-68.2
Import distributed electron density to full vector	7.218	33.695	+366.8
Compute Poisson's equation	33.685	32.507	-3.5
Compute the inner potential integral (over x)	1,397.260	599.199	-57.1
Compute the outer potential integral (over k)	3,160.695	876.767	-72.3
Flip the distributed (over x) vector to distribution over k	85.976	104.374	+21.4
Add the scattering term	4.280	1.676	-60.8
Apply the preconditioner	22.306	7.386	-66.9
Add the original Wigner function to new solution	2.614	0.751	-71.3
Total for all Wigner computations	5,187.054	1,934.689	-62.7

The brown text signifies cross-processor communication.
The blue text indicates serial code.

7.1.5 Improvements to Data Layout

While the changes to the solution method for the Wigner equation, the interpolation routine, and the Tp calculation made the most significant improvements to the run times, a variety of other changes to the structure of the data objects were also made to improve memory usage and reduce communication time.

One of these changes was to store Tp and Tc matrix data in a manner which allowed efficient cache usage. When one element is pulled into cache, nearby elements are also included, so optimal data layout will store elements in a sequence such that as many as possible will be used before they are flushed out of cache [75]. Thus the elements of the Tp and Tc matrices were reordered so that integrals over k would be more efficiently computed.

Similarly, the elements of the distributed Wigner vector over x were reordered to decrease cache misses when computing integrals over k. Loop unrolling was also incorporated into the integral computations to improve the use of cache and decrease run times.

Additional changes were made to decrease overall memory usage. This was accomplished by consolidating memory for data elements that were used in multiple areas of the code; eliminating data elements if another could be overwritten and used in its place; and passing all vectors and matrices by reference or pointer.

7.2 Final Scaling Results

As mentioned in chapter 6, measuring parallel performance is done using scaling studies, where the number of cores used for an application is increased and the results are compared against a base case. Two different methods are generally used, strong scaling and weak scaling.

7.2.1 Strong Scaling

One of the most common scaling studies is strong scaling, which fixes the total problem size and then increases the number of cores used. The previous C++ code achieved decent strong scalability results but run times were still on the order of hours for a fine grid (see table 6.4). The new C++ code reduces run times significantly, and cuts in half the number of cores necessary to achieve maximum speedup while keeping the serial percentage for strong scalability low.

As a comparison, for the fine grid of $Nk = 2048$, $Nx = 513$ used in table 6.4, the maximum speedup (and minimum run time) was achieved using 96 cores (with a run time of 5 hours); for the same grid, the maximum speedup was at 48 cores (with a run time of 34 minutes). Thus, finer grids can be used in simulation runs and still produce reasonable run times, which was not possible using the previous C++ code. One attempt with the old C++ code at using an ultra fine grid of $Nk = 4096$, $Nx = 4097$ and 128 cores took 224 hours to run; a simulation run using these same parameters with the new C++ code took only 6 hours.

Thus, to maximize the performance of the Wigner code in testing strong scalability, an ultra fine grid of $Nk = 4096$, $Nx = 2049$ was used, with 8 cores as a base (see table 7.4). The results show a serial percentage around 3.25% at maximum speedup along with significantly improved run times.

Table 7.4: Strong scaling study using ultra fine grid of $Nk = 4096$, $Nx = 2049$.

No. of Cores	Run times (hr:min)	% efficiency	Speedup	% Serial
8	31:22	100.0	8.00	
16	14:45	106.3	17.01	-5.95
32	6:47	115.6	36.99	-4.50
64	3:44	105.0	67.21	-0.68
128	2:55	67.2	86.03	3.25
192	3:04	42.6	81.83	5.85

The serial percentage of the new C++ code could be reduced from 3.25% to possibly as low as 0.5%, but it would require eliminating all of the memory improvements and the addition of the BLAS-3 routine which reduced run times significantly. The gains in run time performance far outweigh the loss to scalability, so the new C++ code exceeds the performance objectives even though it does not match the scalability goal.

7.2.2 Weak Scaling

Another form of parallel performance monitoring is weak scaling, which fixes the problem size on each core and increases the number of cores used. Weak scaling efficiency is measured by

$$\text{Efficiency} = \frac{T_1}{T_N} \times 100\% \tag{7.3}$$

where T_1 and T_N are run times for the base case and an increment of the base case respectively.

Ideal weak scaling can be achieved by eliminating (or severely minimizing) cross-processor communication, and by keeping the number of computations constant on each core. However, neither of these are realistic with the Wigner model. The issues are:

Cross-processor communication. In order to most efficiently handle the integrals over k and the differentials with respect to x, we need two different partitionings of the Wigner vector. This requires cross-processor communication to transpose back and forth between the two partitionings. In addition, a partitioned x vector is used to store the results of the electron density calculation, the results of which are then sent to all other processors in order to compute the Poisson term.

Thus, communication time per core increases (which degrades scaling performance) in a weak scaling study because each vector transposition (or transfer of electron density data) requires global communication of elements from one processor to all others. Access latency is fixed per processor, so communication time will increase in proportion with the total number

of processors used.

Fixed problem size per core. There are two main issues to keeping the problem size constant across cores: first, the non-uniform grid does not allow even distribution of elements across cores. Load balancing of the full non-uniform grid is employed in an attempt to minimize processor wait time, but equal balancing cannot be achieved; and the more zones are present in the full non-uniform grid, the more unbalanced the Wigner vector distributions can get.

The second issue is that the calculations in the Wigner equation depend on four parameters of which the user only directly determines two. The four parameters are: Nx, Nk, $Nk_{(\mathrm{non-uniform})}$ (the number of k grid points in the thinned out momentum grid; loosely but not directly proportional to Nk), and the total number of (x, k) grid points in the full non-uniform mesh (loosely but not directly proportional to $Nx \times Nk_{(\mathrm{non-uniform})}$). Each piece of the Wigner equation depends on a different combination of these four parameters, so determining a method for keeping the problem size constant is more difficult. To do this, I looked only at the major contributors to the run times.

Reviewing the data in table 7.3, there are four calculations that impact run times the most:

1. The interpolated Wigner vector over k – computation time is proportional to Nx, $Nk_{(\mathrm{non-uniform})}$, and the number of cores
2. The Poisson term – dependent only on Nx
3. The Tp matrix – computation time dependent on Nk, Nx, Nc (proportional to Nx), and the number of cores, and
4. The potential term – the most significant term, it contributes more to total run time than the previous three terms combined. Run time is proportional to the number elements of the non-uniform Wigner vector, as well as to $Nk_{(\mathrm{non-uniform})}$ and the number of cores.

The last two terms reveal the main problem – these terms require opposite parameter changes to achieve decent weak scaling. The Tp matrix calculation is proportional to $Nx^2 \times Nk$, so fixing Nx and increasing Nk (in proportion to the number of cores) will hold the problem size constant for this term. However, for the potential term, increasing Nk will increase both $Nk_{(\text{non}-\text{uniform})}$ and the number elements of the non-uniform Wigner vector. To hold the number of potential term calculations constant, we need to fix Nk (and thus $Nk_{(\text{non}-\text{uniform})}$) and increase Nx (in proportion to the number of cores). And since the potential calculation consumes the most run time, we should see our best weak scaling performance from this method.

The accompanying tables (see table 7.5) show a comparison of weak scaling performance for three scenarios: keeping Nk fixed and increasing Nx; keeping Nx fixed and increasing $Nk_{(\text{non}-\text{uniform})}$; and keeping the number of total grid points for the non-uniform mesh fixed (equates to increasing both Nx and Nk). As expected, the best results, which are still not particularly good, are from holding Nk fixed and increasing Nx.

Table 7.5: Weak scaling study for various Nk and Nx.

Nk constant, Nx variable

Nk		Nx	No. of	Tot non-u	Avg non-unif	Run time	Weak
Orig	Non-U		cores	grid pts	pts/proc	(hr:min)	effic %
2,048	612	513	8	92,646	11,581	1:43	100.0
2,048	612	1,025	16	184,678	11,542	2:01	85.1
2,048	612	2,049	32	368,742	11,523	2:51	60.2
2,048	612	4,097	64	736,870	11,514	4:21	39.5
2,048	612	8,193	128	1,473,126	11,509	7:45	22.2
2,048	612	12,289	192	2,209,382	11,507	39:09	4.4

Total number of non-uniform grid points (per core) constant

Nk		Nx	No. of	Tot non-u	Avg non-unif	Run time	Weak
Orig	Non-U		cores	grid pts	pts/proc	(hr:min)	effic %
992	310	993	8	100,750	12,594	1:36	100.0
1,456	446	1,457	16	201,738	12,609	2:12	72.7
2,144	642	2,145	32	402,106	12,566	3:11	50.3
3,072	914	3,073	64	801,938	12,530	4:55	32.5
4,432	1,306	4,433	128	1,606,798	12,553	7:21	21.8
5,456	1,606	5,457	192	2,419,978	12,604	10:12	15.7

$Nk_{\text{non-uniform}}$ variable, Nx constant

Nk		Nx	No. of	Tot non-u	Avg non-unif	Run time	Weak
Orig	Non-U		cores	grid pts	pts/proc	(hr:min)	effic %
1,102	340	1,025	8	110,676	13,835	1:46	100.0
2,278	680	1,025	16	201,640	12,603	2:19	76.3
4,610	1,360	1,025	32	389,200	12,163	3:36	49.1
9,310	2,720	1,025	64	753,568	11,775	6:59	25.3
18,682	5,440	1,025	128	1,491,008	11,649	17:28	10.1
28,058	8,160	1,025	192	2,226,912	11,599	31:37	5.6

To further demonstrate our inability to keep the problem size constant, we can review the profiling results for the individual Wigner terms. When Nk is held constant and Nx increased, the run time to compute theTp matrix increases while the run time to compute the potential integral over k stays about the same. However, for the opposite case when Nx is held constant and $Nk_{(\mathrm{non-uniform})}$ increased, there is a slight elevation in the computation time for the Tp matrix due to the dependence of the calculation on the Nk rather than $Nk_{(\mathrm{non-uniform})}$. However, the main increase in run time is due to the potential term, which increases significantly due to its dependence on $Nk_{(\mathrm{non-uniform})}$ as well as the size of the Tp matrix. See tables 7.6 and 7.7, which show profiling results for the elements on core zero for each weak scaling case. (Only results for core zero are shown because the allocation of elements in the non-uniform grid at the extreme ends of the grid should involve only the coarsest x grid, and thus make a better comparison as the total problem size is increased.)

Tables 7.6 and 7.7 also show increases for each of the terms involving communication between cores. (Note that 16 cores was used as a base for comparison since communication times for 8 cores involve only intra-blade communication and would be artificially lower.) The functions that transpose one distribution of the Wigner vector to the other (i.e., "flipping" the vector) increase proportionally with the number of cores, since the communication transfers an element on one core to another core. However, the function importing the distributed electron density vector to a full vector increases proportional to the square of the increase in cores because each element on every core must be communicated to every other core. These proportional increases are reflected for each of the weak scaling cases.

Table 7.6: Weak scaling profiling statistics for core number zero.

Nk = 2048, Nx variable

DESCRIPTION *(in order of computation)*	*Run time (sec)*		% change
Nx	1,025	8,193	699.3
Cores	16	128	700.0
Compute interpolation via sparse mat-vec	483.897	605.441	25.1
Multiply vector by weights for k integration	29.888	30.019	0.4
Flip the distributed (over k) vector to dist over x	129.451	1,249.710	865.4
Integrate (distributed) electron density	2.449	2.152	-12.1
Import distributed electron density to full vector	18.228	1,147.100	6,193.1
Compute Poisson's equation	59.499	435.583	632.1
Compute the inner potential integral (over x)	2,484.470	18,261.400	635.0
Compute the outer potential integral (over k)	2,714.460	2,647.260	-2.5
Flip the distributed (over x) vector to dist over k	338.459	2,373.270	601.2
Add the scattering term	4.332	10.209	135.6
Apply the preconditioner	21.106	20.636	-2.2
Add the original Wigner function to new solution	1.829	1.796	-1.8
Total for all Wigner computations	6,286.720	26,778.100	325.9

Total number of non-uniform grid points (per core) constant

DESCRIPTION *(in order of computation)*	*Run time (sec)*		% change
Nk	1,456	4,432	204.4
Nx	1,457	4,433	204.3
# non-U grid pts/core	12,609	12,553	-0.4
Cores	16	128	700.0
Compute interpolation via sparse mat-vec	552.242	708.836	28.4
Multiply vector by weights for k integration	34.349	35.980	4.7
Flip the distributed (over k) vector to dist over x	156.798	1,289.100	722.1
Integrate (distributed) electron density	2.304	2.845	23.5
Import distributed electron density to full vector	11.611	1,271.340	10,849.4
Compute Poisson's equation	84.907	241.826	184.8
Compute the inner potential integral (over x)	3,469.530	13,183.200	280.0
Compute the outer potential integral (over k)	2,149.190	7,151.830	232.8
Flip the distributed (over x) vector to dist over k	322.222	1,477.070	358.4
Add the scattering term	5.239	8.335	59.1
Apply the preconditioner	24.651	24.709	0.2
Add the original Wigner function to new solution	2.131	2.309	8.3
Total for all Wigner computations	6,813.880	25,390.900	272.6

The brown text signifies cross-processor communication.
The blue text indicates serial code.

Table 7.7: Weak scaling profiling statistics for core number zero.

$\mathbf{Nk_{non-uniform}}$ variable, Nx = 1025

DESCRIPTION *(in order of computation)*	*Run time (sec)*		% change
Nk	2,278	18,682	720.1
$Nk_{(\text{non-uniform})}$	680	5,440	700.0
Cores	16	128	700.0
Compute interpolation via sparse mat-vec	533.329	578.745	8.5
Multiply vector by weights for k integration	32.675	29.866	-8.6
Flip the distributed (over k) vector to dist over x	141.778	1,141.530	705.2
Integrate (distributed) electron density	2.775	3.919	41.2
Import distributed electron density to full vector	17.441	1,264.100	7,147.8
Compute Poisson's equation	59.572	59.158	-0.7
Compute the inner potential integral (over x)	2,748.520	5,946.050	116.3
Compute the outer potential integral (over k)	3,545.870	52,491.400	1,380.4
Flip the distributed (over x) vector to dist over k	262.712	909.555	246.2
Add the scattering term	4.593	4.698	2.3
Apply the preconditioner	23.156	20.996	-9.3
Add the original Wigner function to new solution	2.040	2.160	5.8
Total for all Wigner computations	7,373.100	62,445.700	746.9

The brown text signifies cross-processor communication.
The blue text indicates serial code.

7.2.3 Time Dependence

To complete the analysis on the Wigner-Poisson model, I ran strong scaling studies on the time dependent version of the Wigner-Poisson code. A very fine grid of $Nk = 4096$, $Nx = 2049$ was used to compare the results against those for the steady state problem, with $\Delta t = 20$ fs and $t_{max} = 40{,}000$ fs. Since it has been shown (see section 6.6.1) that oscillations disappear under grid refinement, I chose only one value of the voltage ($V = 0.248$) at which to test strong scaling, since simulation results will be very similar for any value of the voltage.

The results from table 7.8 show consistency between the steady state code and the time dependent version. A similar superlinear speedup to that for the steady state code (when the no. of cores ≤ 64) is present, although the maximum speedup and serial percentage is improved over the steady state version. This improvement is a result of the convergence of the Wigner vector to a single solution after a number of time steps (which is much less than the total number of time steps to be taken). After convergence, the number of computations requiring cross-processor communication time decreases, which improves the efficiency and the serial %.

Table 7.8: Time dependent strong scaling studies for the fine grid of $Nk = 4096$, $Nx = 2049$.

Voltage = 0.248, $\Delta t = 20$fs

No. of Cores	Run times (hr:min)	% efficiency	Speedup	% Serial
8	9:16	100.0	8.00	
16	4:12	110.3	17.65	-9.35
32	1:53	123.0	39.36	-6.24
64	0:55	126.4	80.87	-2.98
128	0:41	84.8	108.49	1.20
192	0:42	55.2	105.90	3.53

For time dependent weak scaling, the mesh sizes and time steps must be chosen appropriately based on the limitations derived using Von Neumann analysis (see section 6.7), and issues arise in attempting a weak scaling study due to the time step restrictions. For fixed Nx and increasing Nk, in order to get convergent solutions as Nk gets large, Δt must be chosen beyond the range of what seems physically reasonable. Similarly, for Nk fixed and Nx increasing, the time step restrictions require a base run using $Nk \approx Nx$ small and Δt small.

Thus the most appropriate time dependent weak scaling study is that which fixes the number of total grid points for the non-uniform mesh (equates to increasing both Nx and Nk at the same rate). This allows a fixed time step to be chosen and used throughout the scaling study. In this case, the time dependent Wigner solution converges after a fixed number of time steps, so the current computation plays a more significant role in scalability. Each piece of the current computation scales well because each data object depends only on Nx, Nk, and the number of cores, and thus the size of the data objects per core can be held fairly constant. However, more cross-processor communication is needed for the current computation than for the standard Wigner equation, so weak scalability is not improved over the steady state version. See table 7.9.

Table 7.9: Weak scaling study for the time dependent Wigner code keeping the total number of non-uniform grid points (per core) constant.

$$\mathrm{V} = 0.248,\ \Delta t = 20 \text{ fs}$$

Nk		Nx	No. of	Tot non-u	Avg non-unif	Run time	Weak
Orig	Non-U		cores	grid pts	pts/proc	(hr:min)	effic %
992	310	993	8	100,750	12,594	0:34	100.0
1,456	446	1,457	16	201,738	12,609	0:49	69.4
2,144	642	2,145	32	402,106	12,566	1:14	45.9
3,072	914	3,073	64	801,938	12,530	1:56	29.3
4,432	1,306	4,433	128	1,606,798	12,553	3:02	18.7
5,456	1,606	5,457	192	2,419,978	12,604	3:59	14.2

7.3 Future Work

While the new C++ Wigner-Poisson model is a vast improvement over the previous FORTRAN and MATLAB versions, there are a few improvements that could be tried to decrease run times

further. The first is to implement a method used in the MATLAB code – an FFT / inverse FFT combination (used in conjunction with the convolution theorem) to calculate the Wigner potential term. However, using an FFT would require uniform spacing between the elements of the Wigner vector used in the calculation, so changes would be needed to restructure the existing non-uniform Wigner vectors. Additional changes might be needed to the distribution of the elements across cores to ensure proper load balancing. And finally, it would eliminate the highly efficient BLAS-3 call, so the run time improvement may not be as significant as the traditional switch from an n^2 computation to a $n\log_2 n$ computation.

Another possible change to improve the speed of the code would be to rewrite some of the distributed Wigner vectors as `Epetra_MultiVectors` to take advantage of more efficient Trilinos commands. For example, the weight terms for the k integral are currently incorporated into the Wigner vector via multiplications within nested for loops. This could be replaced by writing the k weights vector as an `Epetra_Vector`, and then multiplying it by the Wigner vector if the Wigner vector was an `Epetra_MultiVector`. A similar calculation could be done with the Scattering term and the application of the $\int f_0$ term.

7.3.1 Longer Devices

Due to the significant improvements in the run times for the Wigner code, I have been able to obtain simulation results using longer device lengths. The standard RTD length used throughout this work is 550 Å, but figure 7.1 shows results for device lengths of 750 Å and 900 Å as well. A comparison of the I-V curves for the longer devices against that for the shorter device seems to indicate that the steady state I-V curve will smooth out as the length of the device increases. As discussed in [43], this supports the idea that the boundary conditions used for this simulation model (and previous models) may not be ideal.

In order to test the idea that the I-V curve smooths out as the device is lengthened, the preconditioner for the Wigner model could be improved to reduce the number of Krylovs per

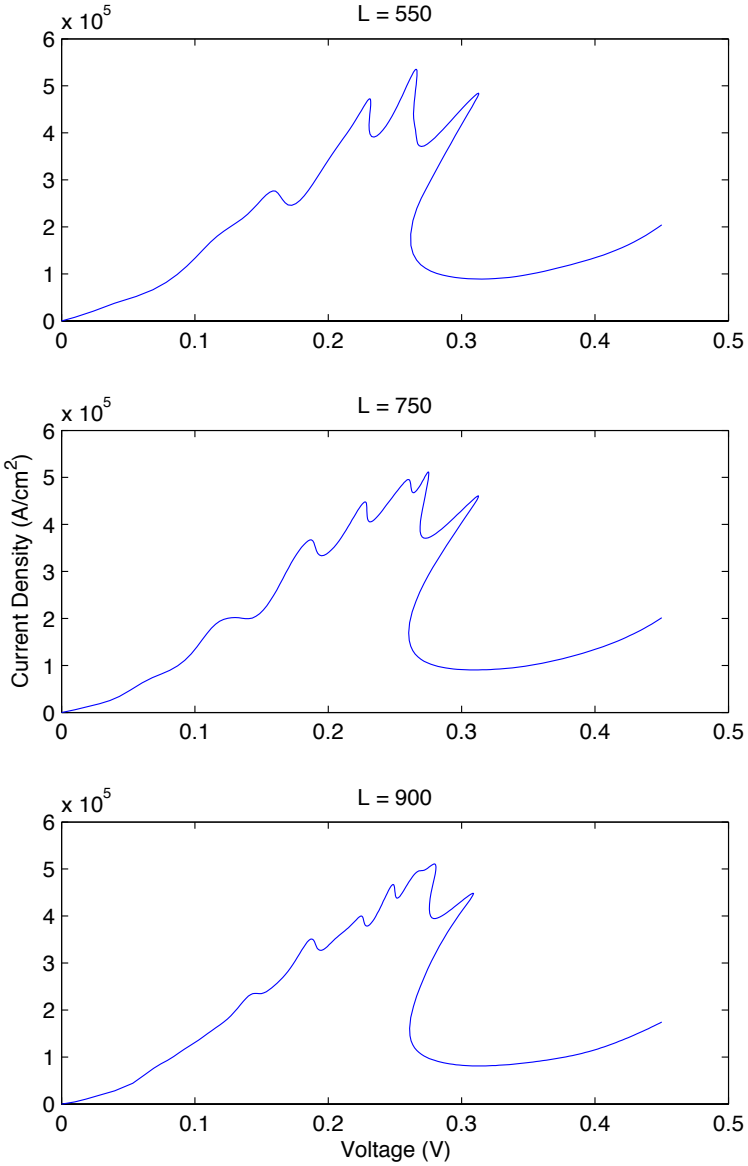

Figure 7.1: Steady-state IV curves for three different device lengths. The doping levels and barriers are held constant in the middle of the device, and the doped regions on each end are increased in length.

Newton. For the new Wigner-Poisson model using a standard 550 Å length, the number of Krylovs per Newton varies between 119 and 318. This increases to between 168 and 575 for a 750 Å device, and 203 to 841 for 900 Å. The computations associated with the additional Krylovs/Newton add to overall run times, which increase (for similar grids and numbers of cores) from approximately 2 hours for a 550 Å device to just under 4 hours for 750 Å and to over 6 hours for 900 Å. Improving the preconditioner would allow longer devices to be modeled more easily, and would also decrease run times for the standard 550 Å model.

REFERENCES

[1] J. K. Hale and H. Koçak. *Dynamics and Bifurcations*. Springer-Verlag, New York, 1991.

[2] M. S. Lasater. *Numerical Methods for the Wigner-Poisson Equations*. PhD thesis, North Carolina State University, Raleigh, North Carolina, 2005.

[3] R. Tsu and L. Esaki. Tunneling in a finite superlattice. *Applied Physics Letters*, 22:562–564, 1973.

[4] L.L. Chang, L. Esaki, and R. Tsu. Resonant tunneling in semiconductor double barriers. *Applied Physics Letters*, 24:593–595, 1974.

[5] T.C.L.G. Sollner, W.D. Goodhue, P.E. Tannenwald, C.D. Parker, and D.D. Peck. Resonant tunneling through quantum wells at frequencies up to 2.5thz. *Applied Physics Letters*, 43:588–590, 1983.

[6] H. Mizuta and T. Tanoue. *The Physics and Applications of Resonant Tunnelling Diodes*. Cambridge University Press, Cambridge, UK, 1995.

[7] K.K. Ng. *Complete Guide to Semiconductor Devices*. Wiley-Interscience, New York, NY, 2nd edition, 2002.

[8] W. R. Frensley. Wigner-function model of a resonant-tunneling semiconductor device. *Phys. Rev. B*, 36:1570–1580, 1987.

[9] F. A. Buot and K. L. Jensen. Lattice Weyl-Wigner formulation of exact many-body quantum-transport theory and applications to novel solid-state quantum-based devices. *Phys. Rev. B*, 42:9429–9457, 1990.

[10] M. S. Lasater, C. T. Kelley, A. G. Salinger, D. L. Woolard, G. Recine, and P. Zhao. Analysis of a scalable preconditioner for the wigner-poisson equations. *International Journal of Pure and Applied Mathematics*, 37, 2007.

[11] P. Zhao, H. L. Cui, and D. L. Woolard. Dynamical instabilities and I-V characteristics in resonant tunneling through double barrier quantum well systems. *Phys. Rev. B*, 63:75302, 2001.

[12] P. Zhao, H. L. Cui, D. L. Woolard, K. L. Jensen, and F. A. Buot. Simulation of resonant tunneling structures: Origin of the i-v hysteresis and plateau-like structure. *Journal of Applied Physics*, 87:1337–1349, 2000.

[13] P. Zhao, D. L. Woolard, M. S. Lasater, C. T. Kelley, and R. J. Trew. Terahertz-frequency quantum oscillator operating in the positive differential resistance region. In *Proceedings of SPIE, the International Society for Optical Engineering*, volume 5790, pages 289–300, 2005.

[14] P. Zhao, H. L. Cui, D. L. Woolard, K. L. Jensen, and F. A. Buot. Origin of hysteresis and plateau-like behavior of the i-v characteristics of resonant tunneling diodes. *International Journal of Modern Physics B*, 14(4):411–426, 2000.

[15] D. L. Woolard, P. Zhao, and H. L. Cui. Thz-frequency intrinsic oscillations in double-barrier quantum well systems. *Physica B*, 314:108–112, 2002.

[16] P. Zhao, D. L. Woolard, H. L. Cui, and N. J. M. Horing. Origin of intrinsic oscillations in double-barrier quantum well systems. *Physics Letters A*, 311:432–437, 2003.

[17] P. Zhao, D. L. Woolard, and H. L. Cui. Multisubband theory for the origination of intrinsic oscillations within double-barrier quantum well systems. *Physical Review B*, 67:085312, 2003.

[18] T. P. E. Broekaert, B. Brar, J. P. A. van der Wagt, A. C. Seabaugh, F. J. Morris, T. S. Moise, E. A. Beam, and G. A. Frazier. A monolithic 4-Bit 2-Gsps resonant tunneling analog-to-digital converter. *IEEE J. Solid-State Circuits*, 33:1342–1349, September 1998.

[19] S.-R. Li, P. Mazumder, and L. O. Chua. Cellular neural/nonlinear networks using resonant tunneling diode. In *Proc. 2004 4th IEEE Conf. on Nanotechnology*, pages 164–167, Piscataway, NJ, August 2004. IEEE Press.

[20] J. P. A. van der Wagt, A. C. Seabaugh, and E. A. Beam III. Rtd-hfet low standby power sramgain cell. *IEEE Electron Device Lett.*, 19:7–9, January 1998.

[21] K. Sano, K. Murata, T. Otsuji, T. Akeyoshi, N. Shimizu, and E. Sano. An 80-gb/s optoelectronic delayed flip-flop ic using resonant tunneling diodes and uni-traveling-carrier photodiode. *IEEE J. Solid-State Circuits*, 36:281–289, February 2001.

[22] H.L. Grubin and R.C. Buggeln. Rtd relaxation oscillations and the time-dependent wigner equation. *Physica B*, 314:117–122, 2002.

[23] H.L. Grubin and R.C. Buggeln. Rtd relaxation oscillations, the time dependent wigner equation and phase noise. *Journal of Computational Electronics*, 1:33–37, 2002.

[24] H.L. Grubin and R.C. Buggeln. Wigner function simulations of quantum device-circuit interactions. *International Journal of High Speed Electronics and Systems*, 13(4):1255–1286, 2003.

[25] H.L. Grubin and H.L. Cui. Spin dependent transport in quantum and classically configured devices. *International Journal of High Speed Electronics and Systems*, 16(2):639–658, 2006.

[26] H.L. Grubin. Spin dependent wigner function simulations of diluted magnetic semiconductor superlattices - b field tuning. *International Journal of High Speed Electronics and Systems*, 17(4):877–888, 2007.

[27] B. Zhang, J. Wang, C. Xue, W. Zhang, and J. Xiong. A gaas acoustic sensor with frequency output based on resonant tunneling diodes. *Sensors and Actuators A*, 139:42–46, 2007.

[28] H. Kim, S. Yeon, and K. Seo. High-speed and low-power non-return-to-zero delayed flip-flop circuit using resonant tunneling diode/high electron mobility transistor integration technology. *Japanese Journal of Applied Physics*, 46(4B):2300–2305, 2007.

[29] H.W. Li, B. E. Kardynał, P. See, A.J. Shields, P. Simmons, H.E. Beere, and D.A. Ritchie. Quantum dot resonant tunneling diode for telecommunication wavelength single photon detection. *Applied Physics Letters*, 91:073516, 2007.

[30] S.M. Sze and K.K. Ng. *Physics of Semiconductor Devices*. Wiley-Interscience, New York, New York, 3rd edition, 2007.

[31] N.G. Einspruch and W.R. Frensley. *Heterostructures and Quantum Devices*. Academic Press, San Diego, CA, 1994.

[32] A. Jüngel. *Transport Equations for Semiconductors*. Springer-Verlag, Berlin, Germany, 2009.

[33] M.G. Ancona and G.J. Iafrate. Quantum correction of the equation of state of an electron gas in a semiconductor. *Physical Review B*, 39, 1989.

[34] A. Jüngel and R. Pinnau. A positivity preserving numerical scheme for a fourth order parabolic equation. *SIAM Journal of Numerical Analysis*, 39, 2001.

[35] C. de Falco, E. Gatti, A.L. Lacaita, and R. Sacco. Quantum-corrected drift-diffusion models for transport in semiconductor devices. *Journal of Computational Physics*, 204, 2005.

[36] A. Pirovano, A. Lacaita, and A. Spinelli. Two-dimensional quantum effects in nanoscale mosfets. *IEEE Tran. Electr. Devi.*, 47, 2002.

[37] P. Degond, S. Gallego, and F. Méhats. Simulation of a resonant tunneling diode using an entropic quantum drift-diffusion model. *J. Comput. Electron*, 6:133–136, 2007.

[38] N. C. Kluksdahl, A. M. Kriman, D. K. Ferry, and C. Ringhofer. Self-consistent study of the resonant-tunneling diode. *Phys. Rev. B*, 39:7720–7735, 1989.

[39] P. Zhao, D. L. Woolard, B. L. Gelmont, and H.-L. Cui. Creation and quenching of interference-induced emitter-quantum wells within double-barrier tunneling structures. *Journal of Applied Physics*, 94(3):1833–1849, 2003.

[40] Gregory J. Recine. *Numerical Simulation of Quantum Electron Transport in Nanoscale Resonant Tunneling Structures*. PhD thesis, Stevens Institute of Technology, Hoboken, New Jersey, 2004.

[41] M. S. Lasater, C. T. Kelley, A. Salinger, P. Zhao, and D. L. Woolard. Simulating nanoscale devices. *International Journal of High Speed Electronics and Systems*, 16:677–690, 2006.

[42] M. S. Lasater, C. T. Kelley, A. Salinger, D. L. Woolard, and P. Zhao. Parallel parameter study of the Wigner-Poisson equations for RTDs. *Computers and Mathematics with Applications*, 51:1677–1688, 2006.

[43] A. S. Costolanski and C. T. Kelley. Efficient solution of the wigner-poisson equations for modeling resonant tunneling diodes. *IEEE Trans. on Nano.*, 9:708–715, 2010.

[44] R. Shankar. *Principles of Quantum Mechanics*. Plenum Press, New York, NY, 1980.

[45] C. Cohen-Tannoudji, B. Diu, and F. Laloë. *Quantum Mechanics*, volume 1. John Wiley & Sons, Inc., France, 1977.

[46] R. M. Kolbas. ECE 530 Physical Electronics Lecture Notes. North Carolina State University, December 2007.

[47] P. Borbone, M. Pascoli, R. Brunetti, A. Bertoni, and C. Jacoboni. Quantum transport of electrons in open nanostructures with the wigner-function formalism. *Phys. Rev. B*, 59:3060–3069, 1999.

[48] A. G. Salinger, N. M. Bou-Rabee, R. P. Pawlowski, E. D. Wilkes, E. A. Burroughs, R. B. Lehoucq, and L. A. Romero. Loca 1.0 library of continuation algorithms: Theory and implementation manual. Technical Report SAND2002-0396, Sandia National Laboratories, March 2002.

[49] B. A. Biegel and J. D. Plummer. Comparison of self-consistency iteration options for the wigner function method of quantum device simulation. *Physical Review B*, 54:8070–8082, 1996.

[50] K. L. Jensen and F. A. Buot. Numerical simulation of intrinsic bistability and high-frequency current oscillations in resonant tunneling structures. *Physical Review Letters*, 66(8):1078–1081, 1991.

[51] M. A. Heroux, R. A. Bartlett, V. E. Howle, R.J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[52] L. Perko. *Differential Equations and Dynamical Systems*. Springer, New York, 3rd edition, 2001.

[53] P. Waltman. *A Second Course in Elementary Differential Equations*. Academic Press, Inc., Orlando, FL, 1986.

[54] K. T. Alligood, T. D. Sauer, and J. A. Yorke. *Chaos: An Introduction to Dynamical Systems*. Springer, New York, NY, 1996.

[55] S. H. Strogatz. *Nonlinear Dynamics and Chaos*. Westview Press, Cambridge, MA, 1994.

[56] V.J. Goldman, D.C. Tsui, and J.E. Cunningham. Observation of intrinsic bistability in resonant-tunneling structures. *Physical Review Letters*, 58(12):1256–1259, 1987.

[57] T.C.L.G. Sollner. Comment on "observation of intrinsic bistability in resonant-tunneling structures". *Physical Review Letters*, 59(14):1622, 1987.

[58] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical Mathematics*. Springer-Verlag, Berlin, Germany, second edition, 2007.

[59] C. T. Kelley. *Solving Nonlinear Equations with Newton's Method*. Number 1 in Fundamentals of Algorithms. SIAM, Philadelphia, 2003.

[60] C. T. Kelley. *Iterative Methods for Linear and Nonlinear Equations*. Number 16 in Frontiers in Applied Mathematics. SIAM, Philadelphia, 1995.

[61] R.S. Dembo, S.C. Eisenstat, and T. Steihaug. Inexact newton methods. *SIAM J. Numer. Anal.*, 19:400–408, 1982.

[62] Y. Saad and M. Schultz. GMRES a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7:856–869, 1986.

[63] M. T. Heath. *Scientific Computing: An Introductory Survey*. McGraw-Hill, New York, NY, second edition, 2002.

[64] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer-Verlag, New York, NY, second edition, 1993.

[65] H. B. Keller. *Lectures on Numerical Methods in Bifurcation Problems*. Springer-Verlag, Heidelberg, Germany, 1987.

[66] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, Westford, MA, special edition, 2007.

[67] *Trilinos home page, http://trilinos.sandia.gov*.

[68] R&D 100 awards honor innovation at its best. (cover story). *R&D Magazine*, 46(9):26–66, September 2004.

[69] M. Sala, K. Stanley, and M. Heroux. Amesos: A set of general interfaces to sparse direct solver libraries. In et al B. Kågström, editor, *Applied Parallel Computing*, pages 976–985. Springer, 2007.

[70] M. A. Heroux. AztecOO user guide. Technical Report SAND2004-3796, Sandia National Laboratories, 2004. Updated August 2007.

[71] M. Sala, M. A. Heroux, D. M. Day, and J. M. Willenbring. Trilinos tutorial. Technical Report SAND2004-2189, Sandia National Laboratories, 2004. Updated July 2010, for Trilinos Release 10.12.

[72] S. Bochkanov and V. Bystritsky. *ALGLIB (http://www.alglib.net)*.

[73] *private communication between A. S. Costolanski and H. L. Grubin*.

[74] R. J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations.* SIAM, 2007.

[75] S. Goedecker and A. Hoisie. *Performance Optimization of Numerically Intensive Codes.* SIAM, Philadelphia, PA, 2001.

[76] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. on Math. Software*, 5:308–323, September 1979.

[77] *private communication between A. S. Costolanski and T. S. Coffey.*

[78] *private communication between A. S. Costolanski and A. G. Salinger.*

APPENDICES

Appendix A

COMPILING TRILINOS

A.1 Trilinos and Cmake

The Trilinos software can be downloaded from the Sandia website (go to http://trilinos.sandia.gov and find the link to the latest version.) Trilinos requires the Cmake build system, which can be downloaded from the Cmake website (http://www.cmake.org/cmake/resources/software.html). I installed the binary version (as recommended for casual users) specified for Linux on the NCSU HPC.

NOTE: At this moment, the latest version of Trilinos (version 10.12.2) does not compile cleanly on the NCSU HPC; thus, I am running Trilinos version 10.6.0. If compilation issues on your hardware platform are encountered that cannot be debugged, my suggestion is to try compiling previous versions of Trilinos until you find one that will compile easily on your hardware. The Trilinos modules that are incorporated in the RTD code are stable and have not been updated since prior to version 10. Thus, any version 10 Trilinos software should work with the RTD code, and possibly earlier versions as well.

A.2 Environmental Variables

For the NCSU HPC, the environmental variables you set will determine which set of hardware and software will be used. There are three configurations that I used for the simulation runs:

- InfiniBand network (via the `gto` queue on the NCSU HPC)
- Myrinet network (via the `aro` queue)
- Standard Ethernet network (via the `kelley` queue)

For each network, the best compiler to use is the Intel compiler. Gnu compilers can be used, but Intel compilers will run the RTD code more efficiently. The descriptions below detail how I set up the environmental variables to run on the NCSU HPC, and assume the use of the appropriate Intel compiler.

Environmental variables can be set either via (1) your `.tcshrc` file in your home directory, (2) a script that is sourced before you run your code, or (3) at the command line prior to running your code. The listing below describes my setup for each network type.

All three networks require

- `setenv LD_LIBRARY_PATH /usr/local/apps/acml/acml4.3.0/ifort64/lib`

LD_LIBRARY_PATH represents the Intel library that contains the BLAS and LAPACK libraries, which are required by Trilinos. I added this to my `.tcshrc` file so it is executed automatically when I log on.

InfiniBand Network

The InfiniBand network requires that a fair amount of memory be available for applications (default values are too low). This can be set (also in the `.tcshrc` file since it does not affect the other network compilations) using

- `setenv RLIMIT_MEMLOCK 100000`

Intel compilers for each network

Each network requires a different compiler script to be run. Prior to compiling Trilinos or running the RTD code, execute one of the following scripts:

- InfiniBand network: `add intel_mvapich`
- Myrinet network: `add intel64_mx`
- Ethernet network: `add intel64_hydra`

A.3 Configuring Trilinos

Once you have downloaded and untarred the appropriate Trilinos and Cmake packages and set up the proper environmental variables, create a directory called BUILD in the Trilinos base source directory (e.g., `/home/username/trilinos-10.XX.XX-Source`), and then create an empty file called `do-configure` within the BUILD directory.

Next, edit the do-configure file and include all appropriate commands necessary to configure Trilinos according to your specifications. The do-configure file will create the Makefile to generate all the code for Trilinos. Information on how to set up the configure file, and the options available for the do-configure file, is located in the documentation provided at http://trilinos.sandia.gov/Trilinos10CMakeQuickstart.txt; see section (C). A sample do-configure file for the Trilinos implementation used on the NCSU HPC for the RTD code is below.

```
rm CMakeCache.txt

/(cmake parent directory)/cmake-2.8.8-Linux-i386/bin/cmake -D
CMAKE_INSTALL_PREFIX:PATH=/(trilinos parent dir)/trilinos-10.6.0-Source/BUILD \
```

```
-D BLAS_LIBRARY_DIRS:PATH=/usr/local/apps/acml/acml4.3.0/ifort64/lib \
-D BLAS_LIBRARY_NAMES:STRING="acml;acml_mv" \
-D LAPACK_LIBRARY_DIRS:PATH=/usr/local/apps/acml/acml4.3.0/ifort64/lib \
-D LAPACK_LIBRARY_NAMES:STRING="acml;acml_mv" \
-D TPL_ENABLE_MPI:BOOL=ON \
-D MPI_EXEC_MAX_NUMPROCS:STRING=250 \
-D Trilinos_ENABLE_ALL_PACKAGES:BOOL=OFF \
-D Trilinos_ENABLE_ALL_OPTIONAL_PACKAGES:BOOL=OFF \
-D Trilinos_ENABLE_Teuchos:BOOL=ON \
-D Trilinos_ENABLE_Sacado:BOOL=ON \
-D Trilinos_ENABLE_Epetra:BOOL=ON \
-D Trilinos_ENABLE_EpetraExt:BOOL=ON \
-D Trilinos_ENABLE_Ifpack:BOOL=ON \
-D Trilinos_ENABLE_AztecOO:BOOL=ON \
-D Trilinos_ENABLE_Amesos:BOOL=ON \
-D Trilinos_ENABLE_Anasazi:BOOL=ON \
-D Trilinos_ENABLE_Belos:BOOL=ON \
-D Trilinos_ENABLE_ML:BOOL=ON \
-D Trilinos_ENABLE_NOX:BOOL=ON \
-D Trilinos_ENABLE_Stratimikos:BOOL=ON \
-D Trilinos_ENABLE_Thyra:BOOL=ON \
-D Trilinos_ENABLE_RTOp:BOOL=ON \
-D Trilinos_ENABLE_Rythmos:BOOL=ON \
-D Trilinos_ENABLE_Didasko:BOOL=ON \
-D Trilinos_ENABLE_Piro:BOOL=ON \
-D NOX_ENABLE_TESTS:BOOL=OFF \
-D Didasko_ENABLE_TESTS:BOOL=OFF \
-D Didasko_ENABLE_EXAMPLES:BOOL=OFF \
-D NOX_ENABLE_EXAMPLES:BOOL=OFF \
-D Piro_ENABLE_TESTS:BOOL=OFF \
-D Trilinos_ENABLE_EXAMPLES:BOOL=OFF \
-D Trilinos_ENABLE_TESTS:BOOL=OFF \
-D CMAKE_VERBOSE_MAKEFILE:BOOL=ON \
-D Trilinos_VERBOSE_CONFIGURE:BOOL=OFF \
-D CMAKE_BUILD_TYPE:STRING=RELEASE \
 ../
```

A breakdown of the do-configure file above:

- The first line finds the Cmake executable, and directs the output to be saved in the BUILD directory.

- The next 4 lines are unique to the NCSU HPC: lists the BLAS and LAPACK library files to be included in the Trilinos build.
- For parallel computing, TPL_ENABLE_MPI must be set to "ON". The default number of processors used when running MPI is 4, so MPI_EXEC_MAX_NUMPROCS must be set if more than 4 processors will be used.
- Trilinos 10.6.0 has 48 packages, according to the download website. Including all of these is not necessary, so only certain packages are enabled. To do this:
  - Trilinos_ENABLE_ALL_PACKAGES and

    Trilinos_ENABLE_ALL_OPTIONAL_PACKAGES are set to "OFF"
  - Only the necessary packages are enabled by setting

    Trilinos_ENABLE_<Package name> to "ON"
- The tests and examples do not need to be enabled for the RTD Code.
- The verbosity of both the "configure" and the "make" can be set to list either a minimum (OFF) or maximum (ON) of information.
- CMAKE_BUILD_TYPE = RELEASE builds an optimized version (optimized flags sent to compiler), whereas setting the ...TYPE = DEBUG sends default debug flags to the compiler.

Make the do-configure file executable (`chmod +x do-configure`) and then run the file. The configure will set up the Trilinos information needed to "make" Trilinos, such as the compilers to be used, additional libraries to include, etc. Some of the tests it performs may fail if it does not find certain files it expects, but this will not always kill the configure. A successful configure will show

```
-- Configuring done
-- Generating done
-- Build files have been written to: /(trilinos parent directory)/
 trilinos-10.6.0-Source/BUILD
```

if the configure completed correctly.

A.4 Make and make install

After the configure file has run and generated the Makefile for Trilinos, run the Makefile by typing `make` at the command line. As each package is built, each object file will be listed, along with a percentage for how much of your Trilinos compile has completed. This is the most common place for the Trilinos compile to fail. If it does, it will fail immediately after the file that caused the error, so it is easier to debug. Most likely the issue is with your compiler/software, and not Trilinos, such as if the `make` cannot find a particular command from a standard library, which may be due to the environmental variables that were set.

If the `make` completes without error, type `make install` at the command line to install the files in the proper place. Once this final step has completed, your Trilinos compile is done.

A.5 Compiling the RTD code to run with Trilinos

To build a Makefile for the RTD code to include Trilinos, a number of libraries that correspond with those listed in the original configure file must be included in the trilinos BUILD directory. These are included by listing the module name (in lower case) appended to `-l` and included as part of the `LIB_PATH`. Also, the RTD code uses the NOX module, which requires additional inclusions.

A sample Makefile for the RTD code is listed below. The top 10 lines set up variables and include necessary Trliinos and BLAS libraries. The remaining lines (starting with `make RTDCode.exe`) reflect the non-Trilinos files that are specific to the RTD code.

```
#!/bin/sh

TOPDIR=/(trilinos parent directory)/trilinos-10.6.0-Source
BLAS=/usr/local/apps/acml/acml4.3.0/ifort64/lib
INC_PATH= -I$(TOPDIR)/packages/nox/src-epetra -I$(TOPDIR)/BUILD/include
LIB_PATH= -L$(TOPDIR)/BUILD/lib -lepetra -lepetraext -lamesos -laztecoo -lnox
   -lnoxepetra -lloca -llocaepetra -lml -ltriutils -lteuchos -lanasazi
   -lanasaziepetra -lifpack -L$(BLAS) -lacml -lacml_mv -lgfortran
CXX=mpicxx
CXX_FLAGS= -O3

include $(TOPDIR)/BUILD/include/Makefile.export.NOX

all:
        make RTDCode.exe

RTDCode.exe: RTDCode.o vector_comp.o sinmat.o BCs.o interp.o interp_ele.o
  kinetic5.o kinetic4.o barrier.o doping.o Tc.o current.o Tp.o poisson1.o ap.o
  alglibinternal.o alglibmisc.o linalg.o solvers.o optimization.o
  specialfunctions.o integration.o interpolation.o RTDHelpers.o
  f_integral.o potential.o Problem_f0.o ProblemInterface_f0.o
  Problem_wp.o ProblemInterface_wp.o

<tab>   $(NOX_CXX_COMPILER) RTDCode.o vector_comp.o sinmat.o BCs.o interp.o
               interp_ele.o kinetic5.o kinetic4.o barrier.o doping.o Tc.o
               current.o Tp.o poisson1.o ap.o alglibinternal.o alglibmisc.o
               linalg.o solvers.o optimization.o specialfunctions.o
               integration.o interpolation.o RTDHelpers.o f_integral.o
               potential.o Problem_f0.o ProblemInterface_f0.o Problem_wp.o
               ProblemInterface_wp.o
               $(NOX_CXX_FLAGS) $(LIB_PATH) $(NOX_LIBRARIES)
               $(NOX_TPL_LIBRARIES) -o RTDCode.exe

RTDCode.o: RTDCode.cpp vector_comp.hpp sinmat.hpp BCs.hpp interp.hpp
  interp_ele.hpp kinetic5.hpp kinetic4.hpp barrier.hpp doping.hpp Tc.hpp
  current.hpp Tp.hpp poisson1.hpp ap.h alglibinternal.h alglibmisc.h linalg.h
  solvers.h optimization.h specialfunctions.h integration.h interpolation.h
```

```
  RTDHelpers.hpp RTDTypes.hpp RTDMatrix.hpp f_integral.hpp potential.hpp
  Problem_f0.hpp ProblemInterface_f0.hpp RTDMesh.hpp Problem_wp.hpp
  ProblemInterface_wp.hpp

<tab>   $(NOX_CXX_COMPILER) $(NOX_CXX_FLAGS) -c RTDCode.cpp $(INC_PATH)

interp.o: interp.cpp interp.hpp RTDMesh.hpp RTDTypes.hpp
<tab>   $(NOX_CXX_COMPILER) $(NOX_CXX_FLAGS) -c interp.cpp $(INC_PATH)
interp_ele.o: interp_ele.cpp interp_ele.hpp RTDMesh.hpp
<tab>   $(NOX_CXX_COMPILER) $(NOX_CXX_FLAGS) -c interp_ele.cpp $(INC_PATH)
vector_comp.o: vector_comp.cpp vector_comp.hpp RTDMesh.hpp
        $(NOX_CXX_COMPILER) $(NOX_CXX_FLAGS) -c vector_comp.cpp $(INC_PATH)
BCs.o: BCs.cpp BCs.hpp RTDTypes.hpp RTDMesh.hpp
        $(NOX_CXX_COMPILER) $(NOX_CXX_FLAGS) -c BCs.cpp $(INC_PATH)
.
.
.
.
Problem_f0.o: Problem_f0.cpp Problem_f0.hpp RTDHelpers.hpp poisson1.hpp Tp.hpp
  potential.hpp RTDMatrix.hpp RTDMesh.hpp kinetic5.hpp kinetic4.hpp
  f_integral.hpp
        $(NOX_CXX_COMPILER) $(NOX_CXX_FLAGS) -c Problem_f0.cpp $(INC_PATH)
ProblemInterface_f0.o: ProblemInterface_f0.cpp ProblemInterface_f0.hpp
  Problem_f0.hpp RTDMesh.hpp
        $(NOX_CXX_COMPILER) $(NOX_CXX_FLAGS) -c ProblemInterface_f0.cpp
          $(INC_PATH)
Problem_wp.o: Problem_wp.cpp Problem_wp.hpp RTDHelpers.hpp poisson1.hpp Tp.hpp
  potential.hpp RTDMatrix.hpp RTDMesh.hpp kinetic5.hpp kinetic4.hpp current.hpp
  f_integral.hpp
        $(NOX_CXX_COMPILER) $(NOX_CXX_FLAGS) -c Problem_wp.cpp $(INC_PATH)
ProblemInterface_wp.o: ProblemInterface_wp.cpp ProblemInterface_wp.hpp
  Problem_wp.hpp RTDMesh.hpp
        $(NOX_CXX_COMPILER) $(NOX_CXX_FLAGS) -c ProblemInterface_wp.cpp
          $(INC_PATH)

clean:
        rm *.o RTDCode.exe
```

The first four lines after `make RTDCode.exe` create the executable version of the RTD code (`RTDCode.exe`). To create the executable file, each object file for each class in the C++ code must be included. To create the object file for each class, the basic lines are

```
<Class>.o: <Class>.cpp <Class>.hpp {Other hpp files included in the <Class>
  header file}
        $(NOX_CXX_COMPILER) $(NOX_CXX_FLAGS) -c <Class>.cpp $(INC_PATH)
```

A.6 Submitting the RTDCode in the NCSU HPC environment

All jobs on the NCSU HPC should be run via the batch submission process and not via the command line. To submit a job using the RTD Code to the applicable queue (kelley, aro, or gto), the batch script must contain certain information specific to each queue.

A basic job submission script looks like:

```
#! /bin/csh
#BSUB -W <T>
#BSUB -n <N>
#BSUB -q <Q>
#BSUB -R "span[ptile=<P>]"
source <Intel compiler script>
mpiexec_<hardware abbreviation> ./<name of executable file>
#BSUB -o o.\%J
#BSUB -e e.\%J
```

where

- $< T >=$ the maximum time in minutes the job should run
- $< N >=$ the number of processors to be used
- $< Q >=$ the queue to be used: gto, aro, or kelley
- $< P >=$ the number of processes to allocate to each blade. For the gto queue, this should be 8; for aro and kelley, 2.
- $<$ *Intel compiler script* $>=$ the script to be used based on the queue used. They are:

  - gto: /usr/local/apps/env/intel_mvapich.csh
  - aro: /usr/local/apps/env/intel64_mx.csh
  - kelley: /usr/local/apps/env/intel64_hydra.csh

- < *hardware abbreviation* >= `mvapich` for the gto queue; `mx` for the aro queue; and `hydra` for the kelley queue

The last two lines specify the output and error file names – in this case, they are linked to the job number (%J = job number).

Thus, a sample job submission script looks like:

```
#! /bin/csh
#BSUB -W 40000
#BSUB -n 16
#BSUB -q gto
#BSUB -R "span[ptile=8]"
source /usr/local/apps/env/intel_mvapich.csh
mpiexec_mvapich ./RTDCode.exe
#BSUB -o o.%J
#BSUB -e e.%J
```

Appendix B

DETAILS OF THE C++ DATA-DRIVEN STRUCTS

The five main structs that are implemented in the code are:

- Constants - to hold well known/defined constants. These are populated from within `RTDCODE.cpp`, using values on the NIST website, at:

  http://physics.nist.gov/cuu/Constants/index.html.

  The data values in this struct are:

  - q, the charge on an electron
  - h, Planck's constant
  - k_B, Boltzmann's constant
  - π, the mathematical constant
  - ϵ_0, the dielectric permittivity constant

- MatProps - to hold properties of the materials used in the device. These constants are read in from the file `matprops.dat` via `RTDCODE.cpp`. The values in this struct are:

- m^*, the effective mass of an electron
- μ_0, the fermi energy at $x = 0$
- μ_L, the fermi energy at $x = L$
- ϵ_d, the dielectric permittivity proportional constant (material dependent)
- ϵ, the computed dielectric permittivity
- τ, the relaxation time

- DevProps - to hold the device-related properties. Most of these variables are read in from `devprops.dat` via the `RTDCODE.cpp`; however, some are calculated in `RTDCODE.cpp` (i.e., $Dx,\ Nc,\ Lc$, etc.) from the input values:
  - T, the temperature of the device
  - V_0, the bias at $x = 0$
  - V_L, the bias at $x = L$
  - K_{max} flag, a character that determines whether or not to use a fixed momentum space (or a variable one)
  - K_{max}, the value that is used to approximate the truncated upper limit of the dk integrals
  - L, the device length
  - Lc, the correlation length
  - k type, a character used to determine whether or not to implement a non-uniform momentum grid
  - x type, a character used to determine whether or not to implement a non-uniform spatial grid
  - Nc, the grid point immediately below (in value) the correlation length
  - Nx, the number of grid points in the spatial dimension

- Nk, the number of grid points in the momentum dimension
- Nk original, the number of grid points in the momentum dimension if a uniform grid were used
- Dx, the grid spacing in the spatial dimension, assuming a uniform grid (i.e., the smallest mesh size for a non-uniform mesh)
- Dk, the grid spacing in the momentum dimension, assuming a uniform grid (i.e., the smallest mesh size for a non-uniform mesh)

- Barrier class - to hold the barrier profile information. The constants are input via the `barrier.dat` file; the remainder is calculated in the Compute function of the Barrier class. The elements are:
  - Num: the number of barriers in the device
  - Startpos: the position (in Angstroms Å) in the device where the barriers begin
  - Length (array): the length of each barrier in the device (in Angstroms Å), with the number of data elements in the array is equal to Num
  - Well (array): the size of each well (in Angstroms Å), with the number of data elements in the array equal to Num-1
  - Type (array): the type of each barrier, where barriers can either be constant ('c'), linear ('l'), or quadratic ('q'). The number of data elements in the array is equal to Num. Certain assumptions are made about the linear and quadratic barriers:
    * For linear barriers, the slope can either be positive or negative, based on the starting ('bot') and ending ('top') values.
    * For quadratic barriers, a symmetric profile is assumed for the barrier, where the value of the barrier is the same at each end of the barrier ('bot'), and the value in the middle is the highest/lowest ('top').

  - Bot (array): the value of the barrier at the left hand side of the barrier. How it is used depends on the type of barrier:
    * This is the value used if the barrier is constant.
    * If the barrier is linear, it is the starting value at the left end of the barrier, and is used to compute the barrier profile to the end of the barrier.
    * If the barrier is quadratic, the left and right ends of the barrier are assumed to be equal to this value, and it is used to compute the rest of the barrier profile.
  - Top (array): an additional value used to determine the barrier profile for the linear and quadratic cases.
    * For a linear barrier, it is assumed to be the barrier value at the right end of the barrier. Together with Bot and Length, it is used to compute the barrier profile for the rest of the barrier.
    * For a quadratic barrier, it is assumed to be the value in the middle of the barrier (since the quadratic profile is assumed to be symmetric). It is used together with Bot and Length to compute the barrier profile for the rest of the barrier.
  - Profile (Epetra_Vector): computed using the input information, it holds the barrier value at each grid point along the device
  - Barr_symm: boolean variable holds whether the barrier is symmetric. If it is true, it is used to position the starting position of the barriers.

- Doping class - to hold the doping profile information. The constants are input via the `doping.dat` file; the remainder is calculated in the Compute function of the Doping class. The elements are:
  - Num: the number of doping regions
  - Length (array): the length of each doping region in the device (in Angstroms Å), with the number of data elements in the array is equal to Num

- Density (array): the value of the density in each doping region. Note that the density is assumed to be constant in each region.
- Profile (Epetra_Vector): holds the value of the doping density at each grid point along the device. Computed using the input information.
- Dop_symm: boolean variable holds whether the doping regions are symmetric. If true, it is used to determine the length (and positions) of the region in the middle of the device, which is generally undoped (or low doped).

Appendix C

Incorporating Trilinos within the RTD code

Several commands within the main RTD code are unique to a Trilinos environment. They include:

1. Setting up the appropriate Epetra objects for parallel programming;
2. Including a command to use Teuchos reference counted pointers;
3. Including appropriate Trilinos header files;
4. Setting up a Problem class and a Problem Interface class for NOX and LOCA; and
5. Including the proper parameter lists and objects for NOX and LOCA.

Most, but not all, of the information given here can be found in the Trilinos tutorial [71] and/or on the Trilinos webpage for the given packages (http://trilinos.sandia.gov/packages).

C.1 Parallel Processing

To utilize parallel processing within Trilinos, include the following lines at the top of the code prior to `int main( int argc, char **argv )` statement [71]:

```
#ifdef HAVE_MPI
#include "mpi.h"
#include "Epetra_MpiComm.h"
#else
#include "Epetra_SerialComm.h"
#endif
```

These lines will check whether the code is set to run in parallel, and if so, will allocate either parallel MPI Epetra objects or serial Epetra objects. Also, the following needs to be included at the very end of the code [71]:

```
#ifdef HAVE_MPI
  MPI_Finalize();
#endif
return(EXIT_SUCCESS);
```

to properly deallocate any MPI objects created for parallel processing.

C.2 Reference Counted Pointers

To include reference counted pointers from Teuchos (which are used by a variety of packages, and make memory management easier and more efficient), the line

```
using Teuchos::RCP;
```

must be included at the top of the code prior to the `int main( int argc, char **argv )` statement.

C.3 Trilinos header files

In addition to standard header files, Trilinos header files associated with the Trilinos objects used need to be included. When creating objects for which reference counted pointers will be used to allocate/deallocate memory, the header file

```
#include <Teuchos_RCP.hpp>
```

should be included, as well as the line

```
using Teuchos::RCP;
```

just after the include statement [77]. In the RTD code, numerous Epetra objects were used that required their corresponding Epetra header files. For most computational classes that used Epetra_Vectors (barrier class, kinetic class, etc.) to store the Wigner function or other vectors, the following statements were included:

```
#include <Epetra_ConfigDefs.h>
#include <Epetra_Map.h>
#include <Epetra_Comm.h>
#include <Epetra_Vector.h>
```

`Epetra_ConfigDefs.h` includes the standard files, functions, and definitions used by Epetra objects; `Epetra_Map.h` the various constructors used to distribute vectors/matrices across processors via the `Epetra_Map.h` class; `Epetra_Comm.h` the various functions associated with communication across processors; and `Epetra_Vector.h` the constructors and functions associated with the `Epetra_Vector.h` vector class.

Several of the RTD code classes used additional Trilinos objects, which required their associated header file. The PoissonMethod class used the sparse matrix object Epetra_CrsMatrix, so the additional file `<Epetra_CrsMatrix.h>` was included to set up the associated functions and constructors, and the RTDMesh struct set up Import objects to facilitate distribution and

accumulation of vector elements across processors, so the file `<Epetra_Import.h>` was included to define the import and export functions.

Finally, numerous header files are associated with NOX and LOCA that must be included in order to run those packages. In the Problem class, the following files were included in addition to the standard Epetra header files:

```
// Nox definition files
#include "NOX.H"
#include "NOX_Epetra_Interface_Required.H"
#include "NOX_Epetra_Interface_Jacobian.H"
#include "NOX_Epetra_LinearSystem_AztecOO.H"
#include "NOX_Epetra_Group.H"
// Loca definition files
#include "LOCA_Parameter_Vector.H"
```

and the following were included in the Problem Interface class:

```
// include all header files that are needed for Loca continuation
#include "Epetra_LinearProblem.h"
#include "AztecOO.h"
#include "LOCA.H"
#include "LOCA_GlobalData.H"
#include "LOCA_Epetra.H"
#include "NOX_Epetra_MatrixFree.H"
```

in order to set up the various objects and functions required for NOX and LOCA.

C.4 Problem class and Problem Interface class

The Problem Interface class is a mandatory class of NOX and LOCA, and sets up several functions used in computing the solution to $F(x) = 0$: for NOX, the main required function is `computeF`, which computes $y = F(x)$. There are several other functions that must be defined in the Problem Interface class (or else a compiler error will result), but need not be implemented to solve the nonlinear equation. In this case, the code should be written to throw an error [71]. The mandatory functions are: `computeJacobian`, `computePrecMatrix`,

and `computePreconditioner`. For LOCA, the same functions must be defined, as well as the `setParameters` function [78], which sets the value of the continuation parameter. An additional function is created in both the NOX and LOCA Problem Interfaces classes called `printSolution`, which prints user-defined information about the converged solution for each continuation value.

However, to reduce the complexity of the Problem Interface class and allow changes to be made to $F(x)$ more easily, the actual computations associated with $F(x)$ are defined in a Problem class [78], and the `computeF` function in the Problem Interface class calls a function in the Problem class to do the actual calculation. A similarly setup is involved with the `printSolution` function, which calls a function in the Problem class to do the current calculation and output the result. For the RTD code, a Problem class and a Problem Interface class are defined to solve for the initial Wigner distribution f_0 using $K(f) + P(f) = 0$, and another Problem class and Problem Interface class are defined to solve the full Wigner function $K(f) + P(f) + S(f) = 0$ for various voltages.

C.5 NOX setup

NOX and LOCA both have a variety of options associated with their use. To hold the information necessary to each package, a main parameter list is created [71], and the specific methods and parameters to be used are assigned via nested sublists.

NOX requires an initial guess for the Wigner distribution function [71], which is given as `fx_dist` below. The portion of code below specific to NOX is included in the main program once the constructors for the computational classes (e.g., barrier, kinetic, poisson, etc.) have been set up. Summary explanations for each piece of code are given in *italicized* text.

A main parameter list, nlParams, is created for NOX, along with its corresponding pointer, nlParamsPtr:

```
// Create the top level parameter list
Teuchos::RCP<Teuchos::ParameterList> nlParamsPtr = Teuchos::rcp(new Teuchos::
  ParameterList);

// Set up NOX info
Teuchos::ParameterList& nlParams = *(nlParamsPtr.get());
```

Sublists to hold the methods to be used and the parameter options related to the nonlinear solver:

```
// Set the nonlinear solver method
nlParams.set("Nonlinear Solver", "Line Search Based");

// Set the printing parameters in the "Printing" sublist
Teuchos::ParameterList& printParams = nlParams.sublist("Printing");
printParams.set("MyPID", Comm.MyPID());
printParams.set("Output Precision", 8);
printParams.set("Output Processor", 0);
printParams.set("Output Information",
NOX::Utils::OuterIteration +
NOX::Utils::OuterIterationStatusTest +
NOX::Utils::InnerIteration +
NOX::Utils::LinearSolverDetails +
NOX::Utils::Parameters +
NOX::Utils::Details +
NOX::Utils::Warning);

// NOX parameters - Sublist for line search
Teuchos::ParameterList& searchParams = nlParams.sublist("Line Search");
searchParams.set("Method", "Full Step");

// Sublist for direction
Teuchos::ParameterList& dirParams = nlParams.sublist("Direction");
dirParams.set("Method", "Newton");

Teuchos::ParameterList& newtonParams = dirParams.sublist("Newton");
newtonParams.set("Forcing Term Method", "Constant");

// Sublist for linear solver for the Newton method
Teuchos::ParameterList& lsParams = newtonParams.sublist("Linear Solver");
lsParams.set("Aztec Solver", "GMRES");
lsParams.set("Max Iterations",maxGMRES);
lsParams.set("Tolerance", 1e-6);
```

```
lsParams.set("Output Frequency", 25);
lsParams.set("Preconditioner", "None");
```

Objects created for NOX, including the initial guess fed into NOX as well as those related to the Jacobian-free method:

```
// Set up the problem interface
Teuchos::RCP<f0ProblemInterface> interface =
         Teuchos::rcp(new f0ProblemInterface(Problem,mesh,Comm) );

Teuchos::RCP<NOX::Epetra::Interface::Required> iReq = interface;

// Create the operator to hold the Matrix-free operator
Teuchos::RCP<Epetra_Operator> A;
Teuchos::RCP<NOX::Epetra::Interface::Jacobian> iJac;

// Need a NOX::Epetra::Vector for constructor
// This becomes the initial guess vector that is used for the nonlinear solves

//AGS Partition parallel
Epetra_Vector fx_dist(*mesh.xkMap_dist);
fx_dist.Import(fx, *mesh.xkImport_rev, Insert);

NOX::Epetra::Vector noxInitGuess(fx_dist, NOX::DeepCopy);

// Matrix Free application (Epetra Operator):
Teuchos::RCP<NOX::Epetra::MatrixFree> MF =
Teuchos::rcp(new NOX::Epetra::MatrixFree(printParams,interface,noxInitGuess));

A = MF;
iJac = MF;

// Build the linear system solver
Teuchos::RCP<NOX::Epetra::LinearSystemAztecOO> linSys =
     Teuchos::rcp(new NOX::Epetra::LinearSystemAztecOO(printParams, lsParams,
          iReq, iJac, A, noxInitGuess));

// Create the Group - must be NOX group
Teuchos::RCP<NOX::Epetra::Group> grpPtr =
     Teuchos::rcp(new NOX::Epetra::Group(printParams, iReq, noxInitGuess,
          linSys));
```

Set up the convergence criteria and solve the nonlinear system:

```
// Calculate the first F(x0) as a starting point.  This is only needed for
// certain status tests to ensure an initial residual (|r0|) is calculated
grpPtr->computeF();

// Set up the status tests to check for convergence
// Determines the error tolerance for the Newton solves
Teuchos::RCP<NOX::StatusTest::NormF> testNormF =
    Teuchos::rcp(new NOX::StatusTest::NormF(1.0e-10));
// Checks for the max number of nonlinear (Newton) iterations to be taken
Teuchos::RCP<NOX::StatusTest::MaxIters> testMaxIters =
    Teuchos::rcp(new NOX::StatusTest::MaxIters(maxNewtonIters));
// This combination of tests will be used by NOX to determine whether the step
// converged
Teuchos::RCP<NOX::StatusTest::Combo> combo =
    Teuchos::rcp(new NOX::StatusTest::Combo(NOX::StatusTest::Combo::OR,
         testNormF, testMaxIters));

// Create the solver
Teuchos::RCP<NOX::Solver::Generic> solver =
    NOX::Solver::buildSolver(grpPtr, combo, nlParamsPtr);

// Solve the nonlinear system
NOX::StatusTest::StatusType status = solver->solve();
```

Output the convergence results and the solution vector:

```
// Output whether the nonlinear solver converged
if( NOX::StatusTest::Converged  == status )
cout << "\n" << "-- NOX solver converged --" << "\n";
else
cout << "\n" << "-- NOX solver did not converge --" << "\n";

// Output the NOX parameter info
if( Comm.MyPID() == 0 ) {
cout << "\n" << "-- Parameter List From Solver --" << "\n";
solver->getList().print(cout);
}

// Get the Epetra_Vector with the final solution from the solver
```

```
const NOX::Epetra::Group & finalGroup =
    dynamic_cast<const NOX::Epetra::Group&>(solver->getSolutionGroup());
const Epetra_Vector & finalSolution =
(dynamic_cast<const NOX::Epetra::Vector&>(finalGroup.getX())).
     getEpetraVector();

//AGS Partition parallel
Epetra_Vector finalSolution_undist(*mesh.xkMap);
finalSolution_undist.Import(finalSolution, *mesh.xkImport, Insert);
```

C.6 LOCA setup

In the main program of the RTD code, once NOX has computed the initial Wigner distribution function via the code above, the continuation code for LOCA can be run. Setting up the parameter lists and objects for LOCA is similar to those for NOX, and includes a sublist specifically for NOX options. LOCA requires a continuation parameter [78] (in the sample RTD code below, the continuation parameter is `cparm2`, which represents the voltage and is initially set at $V = 0$) and an initial vector (`finalSolution`, which is the converged Wigner distribution function when $V = 0$).

ContParamList is the pointer to the top level parameter list for the continuation run, with locaContParamsList the sublist for all of the step size methods and parameters:

```
// Create the top level parameter list
Teuchos::RCP<Teuchos::ParameterList> ContParamList =
    Teuchos::rcp(new Teuchos::ParameterList);

// Create LOCA sublist
Teuchos::ParameterList& locaContParamsList = ContParamList->sublist("LOCA");

// Create the sublist for continuation and set the stepper parameters
Teuchos::ParameterList& stepperContList=locaContParamsList.sublist("Stepper");
stepperContList.set("Continuation Method", "Arc Length");// Default
stepperContList.set("Continuation Parameter", "cparm2");  // Must set
stepperContList.set("Initial Value", devprops.bias_start);  // Must set
```

```
stepperContList.set("Max Value", devprops.bias_end);    // Must set
stepperContList.set("Min Value", devprops.bias_start);    // Must set
stepperContList.set("Max Steps", maxContSteps);     // Should set
stepperContList.set("Max Nonlinear Iterations", maxNewtonIters); // Should set
stepperContList.set("Compute Eigenvalues",false);
stepperContList.set("Goal Arc Length Parameter Contribution",0.5);
stepperContList.set("Max Arc Length Parameter Contribution",0.7);
stepperContList.set("Initial Scale Factor",1.0);
stepperContList.set("Min Scale Factor",1.0e-8);
stepperContList.set("Min Tangent Factor",-1.0);
stepperContList.set("Tangent Factor Exponent",1.0);

stepperContList.set("Bordered Solver Method", "Householder");

// Create predictor sublist
Teuchos::ParameterList& predictorContList =
     locaContParamsList.sublist("Predictor");
predictorContList.set("Method", "Secant");               // Default

// Create step size sublist
Teuchos::ParameterList& stepSizeContList =
     locaContParamsList.sublist("Step Size");
stepSizeContList.set("Method", "Adaptive");             // Default
stepSizeContList.set("Initial Step Size", 0.01);   // Should set
stepSizeContList.set("Min Step Size", 1.0e-5);    // Should set
stepSizeContList.set("Max Step Size", 0.02);      // Should set
stepSizeContList.set("Aggressiveness", 0.5);
```

A sublist specifically for NOX is created, along with associated sublists specifying the nonlinear solver options:

```
// Set up NOX info
Teuchos::ParameterList& nlContParams = ContParamList->sublist("NOX");

// Set the nonlinear solver method
nlContParams.set("Nonlinear Solver", "Line Search Based");

// Set the printing parameters in the "Printing" sublist
Teuchos::ParameterList& printContParams = nlContParams.sublist("Printing");
printContParams.set("MyPID", Comm.MyPID());
printContParams.set("Output Precision", 5);
printContParams.set("Output Processor", 0);
```

```
printContParams.set("Output Information",
NOX::Utils::OuterIteration +
NOX::Utils::OuterIterationStatusTest +
NOX::Utils::InnerIteration +
NOX::Utils::LinearSolverDetails +
NOX::Utils::Parameters +
NOX::Utils::Details +
NOX::Utils::Warning +
NOX::Utils::StepperIteration +
NOX::Utils::StepperDetails +
NOX::Utils::StepperParameters);

// NOX parameters - Sublist for line search
Teuchos::ParameterList& searchContParams = nlContParams.sublist("Line Search");
searchContParams.set("Method", "Full Step");

// Sublist for direction
Teuchos::ParameterList& dirContParams = nlContParams.sublist("Direction");
dirContParams.set("Method", "Newton");

Teuchos::ParameterList& newtonContParams = dirContParams.sublist("Newton");
newtonContParams.set("Forcing Term Method", "Constant");

// Sublist for linear solver for the Newton method
Teuchos::ParameterList& lsContParams=newtonContParams.sublist("Linear Solver");
lsContParams.set("Aztec Solver", "GMRES");
lsContParams.set("Max Iterations", maxGMRES);
lsContParams.set("Tolerance", 1e-10);
lsContParams.set("Output Frequency", 25);
lsContParams.set("Preconditioner", "None");
```

Objects created for LOCA, including the continuation parameter, the initial guess fed into NOX, and those related to the Jacobian-free method:

```
// set up the continuation parameter vector
LOCA::ParameterVector pcont;
pcont.addParameter("cparm2",cparm2);

// Set up the problem interface
Teuchos::RCP<wpProblemInterface> Cont_interface =
     Teuchos::rcp(new wpProblemInterface(&ProblemCont, cparm2, mesh, outtest,
     Comm));
```

```
Teuchos::RCP<LOCA::Epetra::Interface::Required> iReq_Cont = Cont_interface;

// Create the operator to hold the Matrix-free operator
Teuchos::RCP<Epetra_Operator> A_Cont;
Teuchos::RCP<NOX::Epetra::Interface::Jacobian> iJac_Cont;

// Need a NOX::Epetra::Vector for constructor
// This becomes the initial guess vector that is used for the nonlinear solves

NOX::Epetra::Vector noxInitGuess_Cont(finalSolution, NOX::DeepCopy);

     // Matrix Free application (Epetra Operator):
     Teuchos::RCP<NOX::Epetra::MatrixFree> MF_Cont =
Teuchos::rcp(new NOX::Epetra::MatrixFree(printContParams, Cont_interface,
   noxInitGuess_Cont));
     A_Cont = MF_Cont;
     iJac_Cont = MF_Cont;

// Build the linear system solver
Teuchos::RCP<NOX::Epetra::LinearSystemAztecOO> linSys_Cont =
     Teuchos::rcp(new NOX::Epetra::LinearSystemAztecOO(printContParams,
     lsContParams, iReq_Cont, iJac_Cont, A_Cont, noxInitGuess_Cont));

// Create the Loca (continuation) vector
NOX::Epetra::Vector locaSoln_Cont(noxInitGuess_Cont);

// Create Epetra Factory
Teuchos::RCP<LOCA::Abstract::Factory> epetraFactory_Cont =
    Teuchos::rcp(new LOCA::Epetra::Factory);

// Create global data object
Teuchos::RCP<LOCA::GlobalData> globalData_Cont =
    LOCA::createGlobalData(ContParamList, epetraFactory_Cont);

// Create the Group - must be LOCA group
Teuchos::RCP<LOCA::Epetra::Group> grpPtr_Cont =
     Teuchos::rcp(new LOCA::Epetra::Group(globalData_Cont, printContParams,
 iReq_Cont, locaSoln_Cont,
 linSys_Cont, pcont));
```

Set up the convergence criteria and run the continuation:

```
// Calculate the first F(x0) as a starting point.  This is only needed for
// certain status tests to ensure an initial residual (|r0|) is calculated
grpPtr_Cont->computeF();

// Set up the status tests to check for convergence
// Determines the error tolerance for the Newton solves
Teuchos::RCP<NOX::StatusTest::Generic> testNormF_Cont =
Teuchos::rcp(new NOX::StatusTest::NormF(1.0e-10));

// Sets the max number of nonlinear (Newton) iterations that will be taken.
//  If this is not already set, it will default to '20'
Teuchos::RCP<NOX::StatusTest::MaxIters> testMaxIters_Cont =
     Teuchos::rcp(new NOX::StatusTest::
    MaxIters(stepperContList.get("Max Nonlinear Iterations", 5)));

// This combination of tests will be used by NOX to determine whether the
// step converged
Teuchos::RCP<NOX::StatusTest::Combo> combo_Cont =
     Teuchos::rcp(new NOX::StatusTest::Combo(NOX::StatusTest::Combo::OR,
testNormF_Cont, testMaxIters_Cont));

// Create the stepper
LOCA::Stepper stepper_Cont(globalData_Cont, grpPtr_Cont, combo_Cont,
     ContParamList);
LOCA::Abstract::Iterator::IteratorStatus status_Cont = stepper_Cont.run();
```

Output the convergence results and deallocate memory for LOCA Objects when the continuation is complete:

```
// Check if the stepper completed
if  (status_Cont == LOCA::Abstract::Iterator::Finished)
globalData_Cont->locaUtils->out() << "\nAll tests passed!" << endl;
else
if (globalData_Cont->locaUtils->isPrintType(NOX::Utils::Error))
globalData_Cont->locaUtils->out() << "\nStepper failed to converge!"  << endl;

// Output the stepper parameter list info
if (globalData_Cont->locaUtils->isPrintType(NOX::Utils::StepperParameters))
```

```
{
globalData_Cont->locaUtils->out() << endl << "Final Parameters" << endl
<< "******************" << endl;
stepper_Cont.getList()->print(globalData_Cont->locaUtils->out());
globalData_Cont->locaUtils->out() << endl;
}

// Make sure all processors are done
Comm.Barrier();

// Deallocate memory
LOCA::destroyGlobalData(globalData_Cont);
```